UNIVERSIDADE DO ESTADO DA BAHIA

DEPARTAMENTO DE CIÊNCIAS EXATAS E DA TERRA

CURSO DE GRADUAÇÃO EM SISTEMAS DE INFORMAÇÃO

EDUARDO IKEDA SEGER

APLICAÇÃO DE SELF-QUERYING PARA FILTRAGEM DINÂMICA DE METADADOS EM SISTEMAS DE BUSCA HÍBRIDO ACADÊMICO

SALVADOR, BAHIA, BRASIL

2025

EDUARDO IKEDA SEGER

APLICAÇÃO DE SELF-QUERYING PARA FILTRAGEM DINÂMICA DE METADADOS EM SISTEMAS DE BUSCA HÍBRIDO ACADÊMICOS

Monografia apresentada ao curso de Sistemas de Informação do Departamento de Ciências Exatas e da Terra da Universidade do Estado da Bahia - UNEB, como requisito à obtenção do grau de bacharel em Sistemas de Informação. Área de Concentração: Engenharia de Software.

Orientador: Eduardo Manuel de Freitas Jorge

SALVADOR, BAHIA, BRASIL

2025

EDUARDO IKEDA SEGER

APLICAÇÃO DE SELF-QUERYING PARA FILTRAGEM DINÂMICA DE METADADOS EM SISTEMAS DE BUSCA HÍBRIDO ACADÊMICOS

Monografia apresentada ao curso de Sistemas de Informação do Departamento de Ciências Exatas e da Terra da Universidade do Estado da Bahia - UNEB, como requisito à obtenção do grau de bacharel em Sistemas de Informação. Área de Concentração: Engenharia de Software.

Aprovado em: .

BANCA EXAMINADORA

Prof. Dr. Eduardo Manuel de Freitas Jorge
Orientador

Prof. Dr. Ana Patrícia Fontes Magalhães Mascarenhas
Examinador interno (DCET-1/UNEB)

Prof. Dr. Hugo Saba Pereira Cardoso
Examinador interno (DCET-1/UNEB)

RESUMO

Os sistemas de busca acadêmica tradicionais, que implementam filtros no fluxo da busca, frequentemente enfrentam desafios significativos de manutenibilidade e evolução, principalmente devido à lógica de filtragem de metadados ser rigidamente acoplada ao código fonte. Este trabalho investiga e propõe uma arquitetura de software que utiliza *Large Language Models* (LLMs) com a técnica de *self-querying* para desacoplar essa lógica. A abordagem permite a realização de uma busca híbrida e a tradução de consultas em linguagem natural, que incluem filtros complexos, em requisições estruturadas para bancos de dados vetoriais de forma dinâmica e autônoma. Adotando a metodologia *Design Science Research*, foi desenvolvido um artefato computacional que realiza buscas de publicações acadêmicas em uma base de dados de pesquisadores vinculados ao Grupo de pesquisa: Núcleo de Pesquisa Aplicada e Inovação. Para validação, foi feita a comparação entre o artefato desenvolvido e o sistema do Sistema de Mapeamento de Competências Científicas (SIMCC) por meio de duas etapas. A primeira é uma análise quantitativa por métricas como: linhas de código e complexidade ciclomática e a análise qualitativa verificando questões como a coesão e acoplamento dos métodos e classes envolvidas na busca desses dois sistemas. Na segunda parte, foi realizada uma simulação de 2 cenários de implementação para verificar como ambos os sistemas se comportam diante de manutenções e evoluções. Os resultados da validação evidenciaram que a arquitetura *self-query* conseguiu promover maior manutenibilidade no sistema sem adicionar complexidade ciclomática durante as simulações, contribuindo com um modelo de software mais sustentável e adaptável para sistemas acadêmicos de recuperação de informação.

Palavras-chave: *Large Language Models*. Manutenibilidade de *Software*. Arquitetura de *Software*. Recuperação de Informação.

ABSTRACT

Traditional academic search systems that implement in-stream search filters often face significant maintainability and evolution challenges, primarily because the metadata filtering logic is rigidly coupled with the source code. This work investigates and proposes a software architecture that utilizes Large Language Models (LLMs) with the self-querying technique to decouple this logic. The approach enables hybrid search and the translation of natural language queries, including complex filters, into structured requests for vector databases dynamically and autonomously. Adopting the Design Science Research methodology, a computational artifact was developed to search for academic publications in a database of researchers affiliated with the *Núcleo de Pesquisa Aplicada e Inovação* (NPAI - Center for applied research and innovation) Research Group. For validation, a two-stage comparison was conducted between the developed artifact and the existing *Sistema de Mapeamento de Competências Científicas* (SIMCC - Scientific Competency Mapping System) system. The first stage involved a quantitative analysis using metrics such as lines of code and cyclomatic complexity, and a qualitative analysis examining the cohesion and coupling of the methods and classes involved in the search functionality of both systems. In the second stage, a simulation of two implementation scenarios was conducted to observe how both systems behave when facing maintenance and evolution tasks. The validation results demonstrated that the self-querying architecture successfully enhanced system maintainability without increasing cyclomatic complexity during the simulations, contributing to a more sustainable and adaptable software model for academic information retrieval systems.

Keywords: Large Language Models. Software Maintainability. Software Architecture. Information Retrieval.

LISTA DE ILUSTRAÇÕES

LISTA DE QUADROS

SUMÁRIO

1. INTRODUÇÃO

Grandes volumes de dados multimídia são gerados e armazenados em bancos de dados, como repositórios acadêmicos, sites e sistemas corporativos, apresentando um crescimento próximo ao exponencial (Khan; Rashid; Khan, 2024). Para que esses dados se tornem úteis, é essencial que possam ser recuperados ou descobertos de forma eficiente por meio de sistemas de Recuperação de Informação (RI), como os motores de busca. Isso ocorre por meio de consultas que retornem resultados com relevância à requisição do usuário e ao contexto das perguntas. Vale ressaltar, que os usuários desses sistemas possuem diferentes níveis de entendimento do assunto que é pesquisado, realizando pesquisas exploratórias sobre o assunto, o que pode afetar a relevância dos resultados obtidos, devida a possibilidade de consultas ambíguas por desconhecimento da área pesquisada (Khan; Rashid; Khan, 2024).

Nos sistemas de RI, historicamente, existem as abordagens tradicionais que se apoiam em realizar uma recuperação esparsa, se baseando na busca por correspondência ou similaridade das palavras-chave extraídas da *query* do usuário ou de uma expansão da *query* do usuário, ainda ignorando o contexto semântico (Li et al., *[S.d.]*; Xie et al., 2023) . Essa abordagem faz o uso de métodos como *Boolean Retrieval* e *Best Match* 25 (BM25) que são métodos para realizar consultas por termos exatos e ranqueamento por relevância dos documentos retornados respectivamente. A primeira mais frequentemente usada em revisões sistemáticas em trabalhos acadêmicos.

Para aprimorar a compreensão da consulta realizada pelo usuário, novas estratégias estão sendo aplicadas como a busca semântica (Rozsa et al., 2019). Nesta, o objetivo é decifrar o significado da questão, capturando nuances e relações que não seriam evidentes apenas com uma análise superficial de termos isolados. Existem diferentes propostas que apoiam a estratégia de busca semântica. Uma delas consiste na utilização do Processamento de Linguagem Natural (PLN) que é aplicada à *query* do usuário. Esta proposta inclui formas de separar palavras relevantes que indiquem o contexto buscado. Outra abordagem presente é a utilização de *embedding* para converter consultas e documentos em vetores comparáveis. Essa disposição espacial dos vetores de n-dimensões confere significado semântico a eles, facilitando a identificação de relações de sentido e a busca por similaridade. (Barnard, 2023; Xie et al., 2023).

Dentro do aspecto da busca semântica temos algumas abordagens que são aprimoradas pela Inteligência Artificial (IA) Generativa pelo uso de *Large Language Models* (LLMs).

Existem modelos como o BERT e a família de versões do GPT que utilizam encoders bidirecionais para gerar *embeddings*, o que melhora a compreensão do contexto em textos não rotulados e torna as buscas mais precisas. (BATISTA, 2024; Li et al., *[S.d.]*). Outro método emergente é o *self-querying retrieval* que, usando modelos de IA Generativa, podem realizar buscas semânticas conjugadas fazendo filtragem dinâmica dos dados por metadados, identificando na consulta os aspectos semânticos e de filtragem requeridos pelo usuário.

A partir disso, é notável os avanços que a busca semântica e os LLMs trouxeram para o campo da RI por meio da capacidade de compreender a intenção e o contexto da consulta em linguagem natural, porém essas abordagens resolvem parcialmente o problema. Isso se deve ao fato de que, durante o processo de busca, o usuário ainda é obrigado a realizar a pesquisa em duas etapas: a formulação da consulta e depois a aplicação dos filtros para conseguir refinar os resultados. Dessa forma, o sistema não é capaz de realizar de maneira autônoma a separação dos filtros solicitados, que fazem parte dos metadados, e a parte semântica da *query* do usuário.

O desafio de recuperar informações de forma híbrida e intuitiva é particularmente relevante e presente no meio acadêmico. Nesse contexto, existem duas vertentes principais. A primeira, mais exploratória, permite que o pesquisador ou estudante, sem domínio da área, seja auxiliado por buscas semanticamente enriquecidas sendo feitas em linguagem natural. A segunda é a necessidade de filtragem de resultados e um retorno preciso dos termos pesquisados para realização de revisões sistemáticas.

No meio acadêmico, a base de dados pode ser definida como um repositório digital que preserva, organiza e viabiliza o acesso à produção científica de uma comunidade de pesquisa. A título de exemplo, destacam-se bases de dados de grande escala como a *Web of Science* e a *Scopus* que realizam busca exata, além de iniciativas como o *Open Alex* e o *Google Scholar*, que visam expandir o acesso por meio da indexação de outras bases e também um viés mais exploratório com retorno semântico dos termos pesquisados. Elas são ferramentas indispensáveis para a pesquisa, mas frequentemente dependem de interfaces de busca que exigem do usuário a habilidade de traduzir sua necessidade informacional em consultas de palavras-chave e filtros manuais, uma limitação que este trabalho busca endereçar.

De modo geral, as bases acadêmicas são caracterizadas por uma dualidade: elas contêm um vasto volume de texto não estruturado (o conteúdo semântico das publicações) e,

ao mesmo tempo, são ricas em metadados estruturados e de alto valor, como autor, ano de publicação, área de conhecimento, periódico e métricas de impacto (a exemplo do Qualis).

No geral, sistemas de busca acadêmica tradicionais, a exemplo do Sistema de Mapeamento de Competências Científicas ("Sistema de Mapeamento de Competências - SIMC | Nit Uesc", *[S.d.]*) utilizado como referência e *baseline* neste trabalho, frequentemente implementam regras de filtragem de metadados diretamente na lógica de aplicação. Esta arquitetura, embora funcional em primeiro momento, apresenta desafios significativos de manutenibilidade, pois qualquer alteração nos requisitos de filtragem exige modificação, compilação e reimplantação do sistema. Isso pode aumentar o risco de introdução de erros e o custo de evolução do software. Neste contexto, essa pesquisa discute a seguinte questão: de que forma a implementação de uma arquitetura de *self-querying retrieval* com LLMs, para a filtragem de metadados em sistemas de busca acadêmica, impacta a manutenibilidade e a evolução do software ao desacoplar a lógica de filtragem do código-fonte?

Para alcançar os resultados esperados, este trabalho foi guiado por um objetivo geral claro: propor uma solução de recuperação de informação baseada em IA Generativa, utilizando técnicas de *self-querying* para filtragem dinâmica de metadados em repositórios acadêmicos, e avaliar seu impacto na manutenibilidade e evolução do sistema. Para isso, os seguintes objetivos específicos foram realizados: primeiro, será especificada a arquitetura computacional da solução; em segundo lugar, um protótipo funcional será implementado e aplicado sobre a base de dados do grupo de pesquisa: Núcleo de Pesquisa Aplicada e Inovação (NPAI); e, por fim, foram realizados testes experimentais com o protótipo para validar a eficácia da recuperação semântica e o impacto da abordagem *self-querying* no em relação à manutenibilidade e evolução do sistema.

Do ponto de vista científico e tecnológico, o projeto é relevante por propor uma arquitetura de busca que integra tecnologias emergentes. O uso de IA Generativa e *self-querying* para a filtragem dos resultados obtidos da união de uma busca sintática e semântica em um domínio de alta especificidade, como o acadêmico. Através de uma prova de conceito, foi realizada não apenas uma busca semântica e filtragem dos metadados proporcionada pelo uso dessas ferramentas, mas uma avaliação do impacto gerado na manutenção e evolução desses sistemas ao desacoplar a lógica de filtragem do restante código.

No âmbito acadêmico, a pesquisa responde diretamente a lacuna prática e teórica identificada em trabalhos anteriores, como o de João Café Batista (2024), desenvolvida no mesmo contexto. O referido trabalho apontou, em suas propostas de trabalhos futuros, a

limitação da solução de *self-querying* elaborada que não era capaz de integrar o resultado da busca semântica com a busca híbrida desenvolvida no trabalho (BATISTA, 2024), um desafio que este projeto visa solucionar. Ainda, sob a perspectiva social, o projeto contribui para a democratização do acesso à informação. Ao promover a inclusão informacional, a ferramenta desenvolvida permite que diversos públicos tenham acesso de forma mais intuitiva e eficaz com acervos de pesquisa, reduzindo a necessidade de conhecimento prévio da área para exploração de novos conhecimentos, reforçando assim seu papel social.

2. FUNDAMENTAÇÃO TEÓRICA

Este capítulo apresenta os pilares teóricos que fundamentam a pesquisa, explorando conceitos essenciais de manutenção e evolução de *software*, recuperação de informação e o panorama dos sistemas de busca acadêmicos. Em seguida, serão abordados os temas que embasam a solução proposta: Elementos do Processamento de Linguagem Natural (PLN), incluindo seus fundamentos e componentes essenciais; Princípios de *Embedding*, para a compreensão da representação numérica de palavras e frases para fins de análise; Busca Léxica e Busca Semântica, distinguindo e aplicando essas duas abordagens na Busca Híbrida; Avanços da Inteligência Artificial Generativa, apresentando um panorama das inovações recentes em IA generativa e as técnicas de otimização como engenharia de *prompt*, RAG (*Retrieval-Augmented Generation*) e *fine-tuning*; e *self-querying*, com uma explicação detalhada deste conceito, que é um elemento central da proposta.

2.1. Manutenção e evolução de *software*

No ciclo de vida da engenharia de *software*, a fase de manutenção é, reconhecidamente, a que consome a maior fração de esforço e recursos. Estudos indicam que entre 60% e 80% de todo o esforço despendido em um *software* ocorre após sua entrega inicial ao cliente (Pressman; Maxim, 2021).

Ao contrário da percepção comum relacionada à produção de *software*, a fase pós entrega ao cliente não se resume à correções de problemas como *bugs*, no caso da manutenção corretiva. Um *software* de valor está em constante mudança e evolução, exigindo que ele seja alterado e aperfeiçoado para incluir novos requisitos de negócio ou até se adaptar a novos dispositivos ou plataformas que acabaram surgindo no futuro. Devido a imprevisibilidade de uma mudança, a manutenibilidade se torna um dos principais indicadores de qualidade de um projeto. Um código com boa capacidade de sofrer manutenção traz uma maior facilidade em futuras modificações e possivelmente reduzindo custos de implementá-las.

Para avaliar objetivamente a manutenibilidade, a engenharia de *software* utiliza um conjunto de métricas de produto, que quantificam características do código-fonte e do projeto. Para este trabalho, as métricas foram divididas em quantitativas e qualitativo-conceituais.

2.1.1. Métricas quantitativas

Métricas quantitativas oferecem uma medida numérica inicial e direta para o entendimento do sistema, auxiliando a visualização geral do tamanho e da complexidade do código-fonte. Normalmente, essas métricas devem ser vistas em conjuntos para maior assertividade na tomada de decisão.

Dentre as métricas existentes, serão utilizadas as seguintes para a etapa de validação: a primeira, linhas de código: é uma métrica orientada ao tamanho do programa. Nessa métrica é feita a contabilização do número de linhas no código fonte para medir o esforço e oferecer uma visão geral da escala do *software* sendo amplamente usada na literatura (Pressman; Maxim, 2021). A segunda, complexidade ciclomática (CC): é uma medida da complexidade estrutural do software que foi desenvolvida por Thomas McCabe. Ela é derivada do teste do caminho básico e quantifica o número de caminhos lógicos independentes através de um módulo de código, indicando que possui muitas estruturas condicionais (*if, while, for, try-catch*) tornando difícil de entender, testar e manter (Pressman; Maxim, 2021).

O cálculo da complexidade pode ser realizado pela fórmula:

$$CC = A - N + 2$$

Onde A é o número de arestas (fluxos de controle) e N é o número de nós (blocos de código) no grafo.

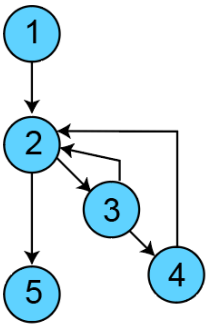
Para contabilizar a complexidade ciclomática de um método ou fluxo de código, uma das formas é a elaboração de um grafo como pode ser visto na Figura 1. O grafo representa de forma simplificada os fluxos que o código vai executar. As arestas são os fluxos de controle, enquanto os nós são blocos de códigos. Quando é aplicada a fórmula de CC mencionada anteriormente, obtém-se o valor 3. Ele representa a quantidade de caminhos independentes do código que é gerado pelas condicionais do *loop* e do *if*.

Figura 1: Exemplo de código e construção do grafo para contabilizar a complexidade ciclomática.

Código

```python
def calcular_soma_positivos(lista_numeros):
    """
    Soma apenas os números positivos em uma lista.
    """
    # Nó 1: Ponto de Entrada
    soma = 0
    # Nó 2: Decisão do loop com 1 condicional
    for numero in lista_numeros:
        # Nó 3: Decisão do 'if'
        if numero > 0:
            # Nó 4: Bloco TRUE do 'if'
            soma += numero
    # Nó 5: Ponto de Saída
    return soma
```

Grafo resultante



$$CC = 6 - 5 + 2 = 3$$

Autor: Elaborada pelo autor.

2.1.2. Métricas qualitativas

Estas métricas avaliam a qualidade do projeto arquitetural, focando em como os módulos interagem e quais responsabilidades eles possuem. Elas são a base do conceito da independência funcional, que diz que módulos de software devem ser o mais independentes e coesos possível (Pressman; Maxim, 2021).

- Acoplamento (*Coupling*): O acoplamento é a medida do nível de interdependência entre diferentes módulos de *software*. Em um sistema com alto acoplamento, uma alteração em um módulo (Módulo A) exige, invariavelmente, alterações em outros módulos (Módulo B, C) que dele dependem (Pressman; Maxim, 2021). Isso acaba tornando a evolução do software cara e arriscada. O objetivo de um bom projeto é o baixo acoplamento, onde os módulos se comunicam através de interfaces bem definidas, sem conhecer detalhes internos uns dos outros.
- Coesão (*Cohesion*): A coesão refere-se ao grau em que as responsabilidades e tarefas dentro de um único módulo estão relacionadas e focadas. Um módulo com alta coesão realiza uma única função bem definida. Um módulo com baixa coesão realiza múltiplas tarefas não relacionadas. A baixa coesão aumenta a complexidade de manutenção, pois encontrar e alterar uma lógica específica torna-se difícil.

A aplicação conjunta destas métricas permite uma avaliação mais robusta da manutenibilidade, pois o acoplamento e a coesão das classes impacta na criação de testes unitários e de integração.

2.2. Repositórios acadêmicos

As instituições de ensino e pesquisa como as universidades, em sua essência, são grandes geradoras de conhecimento. A preservação e a disseminação deste conhecimento são de suma importância para que ela atenda seu papel social. Atuando como uma base de conhecimento, um repositório acadêmico robusto garante que as informações geradas por pesquisadores e alunos estejam organizadas e permanentemente acessíveis, para que sejam recuperadas de forma efetiva, servindo como um pilar para a gestão do conhecimento institucional (Subramanian et al., 2025).

No Brasil, a Plataforma *Lattes* pode ser compreendida como um dos mais vastos repositórios de dados acadêmicos. Ela combina um grande volume de dados estruturados de professores e pesquisadores, como nomes, filiações, datas, com uma grande quantidade de texto livre (não estruturado) encontrados nos resumos e detalhamento das produções acadêmicas (BATISTA, 2024). Sendo assim, a tarefa de recuperação desses documentos não pode ser feita realizando simples consultas de bancos de dados, necessitando a aplicação de técnicas que trabalham com coleções de dados em sistemas de recuperação de informação.

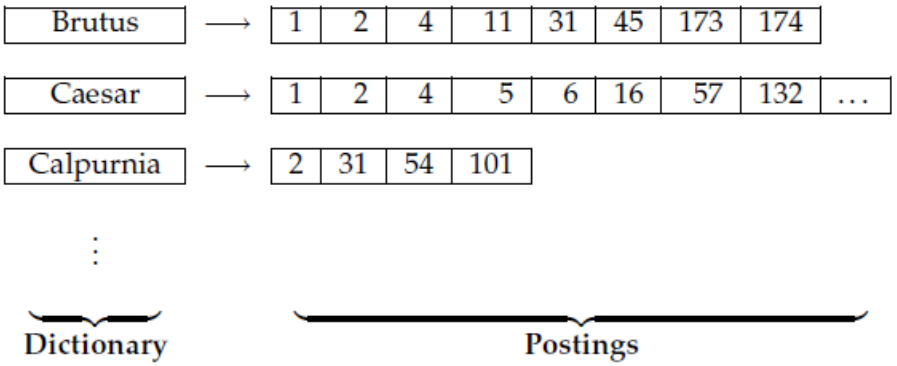
2.3. Recuperação de Informação

A Recuperação de Informação (RI) é uma área voltada para lidar com dados não estruturados, como textos, páginas *web* e documentos diversos, incluindo produções científicas. Ela se relaciona com a procura de materiais (normalmente documentos) que são de natureza não estruturada (textos) que satisfaçam a necessidade de informação a partir de uma grande coleção (que está armazenada em computadores) (Manning; Raghavan; Schuetze, 2009). Dessa forma, não é possível fazer a aplicação direta de consultas, como em bases de dados, para encontrar informações relevantes dentro desses textos livres precisando fazer o uso de outras técnicas para a obtenção de resultados.

O desafio principal na recuperação de informação em grandes coleções de documentos reside na necessidade de se processar textos e documentos não estruturados. Uma abordagem inicial e ingênua para encontrar um termo seria realizar uma varredura sequencial, comparando a palavra buscada com cada palavra de cada documento. Embora funcional para volumes de dados muito pequenos, este método é computacionalmente inviável e não escalável para os domínios de larga escala, como a *web* ou bases de dados acadêmicas.

Para superar essa limitação, uma das otimizações mais importantes para os sistemas de busca é a estrutura de dados conhecida como índice invertido (*inverted index*). Conforme descrito por Manning, Raghavan e Schütze (2009), em vez de percorrer os documentos, o índice mapeia cada termo do vocabulário ao conjunto de documentos em que ele aparece (Figura 2), permitindo acesso rápido a essa lista. Fundamentado nesta estrutura eficiente, o Modelo Booleano foi estabelecido como o primeiro modelo formal de recuperação de informação, permitindo não apenas a busca rápida por um termo, mas a combinação de múltiplos termos através de operadores lógicos (*AND, OR, NOT*) para formular consultas complexas (Manning; Raghavan; Schuetze, 2009). Como, uma pesquisa por termos que combinam múltiplas operações: *"("Machine Learning" OR "ML") AND ("Robotics") NOT ("Games" OR "Gaming")".*

Figura 2: Representação do índice invertido. Na esquerda o termo pesquisado, à direita o documento apontado.



Autor: (Manning; Raghavan; Schuetze, 2009)

Essa abordagem apresenta algumas limitações devido ao modo rígido que determinava se um documento era ou não relevante pela presença de termos, o que motivou o desenvolvimento de outros modelos mais sofisticados que pudessem ranquear os resultados. Dentre eles, destaca-se o modelo vetorial.

O modelo vetorial, tanto a consulta do usuário quanto os documentos são representados como vetores em um espaço n-dimensional, onde cada dimensão corresponde a um termo do vocabulário (Manning; Raghavan; Schuetze, 2009). Como exemplo, na Quadro 1, temos os documentos e uma representação de um vetor de n-dimensões que contém os termos de cada um dos documentos para serem comparados posteriormente.

Quadro 1: Primeiro, o exemplo de uma coleção de documentos. Em segundo, temos o vetor n-dimensional para representar os documentos e uma query.

Documento	Texto livre
1	"Gato preto"
2	"Cachorro preto"

Vetor n-dimensões
[“Gato”, “Cachorro”, “preto”]

Autor: Elaborada pelo autor

Para representar a importância de cada termo em um vetor é preciso dar a cada termo um valor que é calculado por esquemas de ponderação. O TF-IDF (*Term Frequency-Inverse Document Frequency*), por exemplo, faz o cálculo do valor equilibrando a frequência desse termo em documentos com a raridade de um termo que podem ser obtidos pelas fórmulas:

$$TF(t,d) = \frac{contagem\ de\ t\ em\ d}{total\ de\ termos\ em\ d}$$

$$IDF(t) = ln_{10} \frac{número\ total\ de\ documentos}{número\ documentos\ com\ termo\ t}$$

Onde: t - Termo analisado, d - documento

No Quadro 2, é demonstrado os valores resultantes do TF-IDF para os documentos citados anteriormente na Quadro 1 de forma simplificada, observe que nesse caso as palavras presentes em ambos os documentos acabam não tendo relevância para uma busca pois terminam com o valor 0.

Quadro 2: Exemplo de aplicação do TF-IDF para criar vetores usando os documentos do Quadro 1.

Termos	Doc 1: “Gato Preto”	Doc 2: “Cachorro preto”	Query: “Gato preto”
"gato" (TF)	1/2 = 0.5	0/2	1/2 = 0.5
"preto" (TF)	0/2	1/2 = 0.5	0
“Cachorro”	1/2 = 0.5	1/2 = 0.5	1/2 = 0.5

Documento	"Gato" (IDF≈0.693)	"Cachorro" (IDF≈0.693)	"preto" (IDF=0)
D1	0.5×0.693 = 0.3465	0×0.693=0	0.5×0=0
D2	0 × 0.176 = 0	0.1667 × 0.176 ≈ 0.0294	0.5×0=0
Query	0.5×0.693=0.3465	0×0.693=0	0.5×0=0

Vetor D1	[0.3465, 0, 0]
Vetor D2	[0, 0.3465, 0]
Vetor Query	[0.3465, 0, 0]

Autor: Elaborada pelo autor

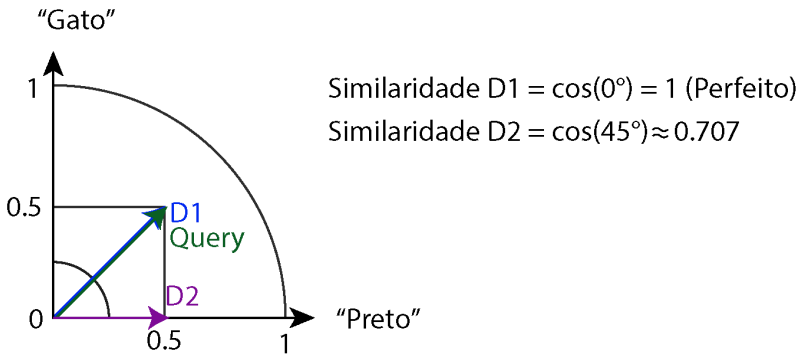
Depois de fazer a construção de vetores para representar consultas e documentos, é preciso comparar esses vetores para determinar a relevância de um documento para aquela busca. Para se realizar comparações entre documentos, textos ou consultas, do ponto de vista computacional, pode ser feito o cálculo da similaridade de cossenos de modo que quanto maior o valor obtido, mais similares são os vetores em relação à sua orientação (BATISTA, 2024; Manning; Raghavan; Schuetze, 2009) e, consequentemente, os documentos que são representados por eles.

A similaridade de cossenos é calculada pela seguinte fórmula, onde d1 e d2 são documentos e V representam os vetores deles:

$$Similaridade(d1, d2) = \frac{V(d1) \cdot V(d1)}{|V(d1)|\,|V(d1)|}$$

Ao calcular o cosseno do ângulo entre esses vetores, obtém-se um valor: quanto mais próximo de 1 ele estiver, maior será a proximidade semântica do documento em relação à sua *query*. Ainda, pode-se usar o resultado desse cálculo como uma forma de pontuação do documento para uma dada consulta do usuário (Manning; Raghavan; Schuetze, 2009). A Figura 3 mostra os vetores que representam visualmente cada documento calculado anteriormente no Quadro 2, assim, pode-se observar que o documento 1 é totalmente similar a *query* enquanto D2 possui uma menor proximidade.

Figura 3: Exemplo visual da aplicação da similaridade de cossenos considerando apenas a frequência de termos "Gato" e "Preto" para melhor visualização do ângulo.



Fonte: Elaborada pelo autor

A influência desse modelo é duradoura: seus princípios de representação vetorial e cálculo de similaridade ainda fundamentam técnicas modernas de busca semântica com embeddings, como exemplificado por João Café Batista (2024) que realizou o desenvolvimento de um artefato computacional que realiza buscas híbridas usando *Word Embedding* em uma base de dados acadêmica.

2.4. Processamento de Linguagem Natural

O Processamento de Linguagem Natural (PLN), ou *Natural Language Processing* (NLP), é um subcampo da Inteligência Artificial e da Ciência da Computação cujo objetivo é capacitar as máquinas a compreender, interpretar e gerar a linguagem humana de forma significativa. Conforme apontado por Vieira e Lopes (2010), o tratamento computacional da linguagem é essencial para lidar com o volume crescente de informações textuais, transformando dados não estruturados em representações que um computador possa processar (BATISTA, 2024). Para que essa transformação ocorra, um conjunto de etapas sequenciais, conhecido como *pipeline* de pré-processamento, é aplicado ao texto bruto.

O pré-processamento é uma fase fundamental que visa limpar e padronizar o texto, reduzindo a complexidade e a variabilidade dos dados. As etapas mais comuns, conforme detalhado por Manning, Raghavan e Schütze (2009), incluem:

- **Tokenização**: O primeiro passo consiste em segmentar o texto contínuo em unidades menores, chamadas *tokens*. Geralmente, esses *tokens* são palavras, mas também podem ser sentenças ou pontuações, servindo como os blocos de construção para as análises subsequentes (Manning; Raghavan; Schuetze, 2009)

- **Normalização**: Após a tokenização, aplica-se um conjunto de técnicas para uniformizar os tokens:
    - **Remoção de *Stopwords*:** Palavras extremamente comuns em uma língua (como "de", "o", "a", "que") são removidas, pois geralmente carregam pouco valor semântico e podem poluir a análise de relevância (Manning; Raghavan; Schuetze, 2009)
    - **Lematização:** Esta técnica, mais sofisticada que a simples stemização (stemming), reduz as palavras flexionadas à sua forma dicionarizada base, conhecida como lema. Por exemplo, os verbos "fui", "vou" e "iria" são todos mapeados para o lema "ir". Esse processo é crucial para agrupar palavras semanticamente equivalentes, garantindo que o sistema compreenda que elas se referem ao mesmo conceito (MANNING; RAGHAVAN; SCHÜTZE, 2009, p. 32).

Após a execução dessas etapas, o texto está pronto para ser convertido em um formato numérico, uma etapa essencial para que os algoritmos de *machine learning* possam extrair significado e realizar tarefas como a busca semântica.

2.5. Inteligência Artificial Generativa

A ascensão dos Grandes Modelos de Linguagem (*Large Language Models* - LLMs) representa uma evolução substancial para a Recuperação de Informação. Os LLMs são modelos de linguagem de grande escala treinados em um vasto conjunto de dados textuais e são capazes de compreender e gerar texto coerente e contextualmente relevante. Podem ser empregados para interpretar a intenção da consulta, gerar resumos ou até mesmo gerar respostas completas com base nas informações recuperadas (“*What is LLM*?”, *[S.d.]*). Essa capacidade de geração é diretamente alimentada pela proficiência desses modelos na interpretação de consultas.

2.5.1. Embedding

Os modelos de recuperação de informação tradicionais, como o TF-IDF que foi citado anteriormente, embora eficazes para a busca por termos, são baseados na contagem e correspondência de palavras dentro de um texto, não possuindo a capacidade de estabelecer relação entre significados. Então, não é possível que tenha uma relação entre palavras e siglas

como “Inteligência Artificial” e “IA”, ou entre palavras semanticamente próximas, como “Veículo” e “Carro”.

Para superar as limitações da busca por palavras-chave, a recuperação de informação moderna adotou a Recuperação Densa (*Dense Retrieval*), que utiliza *embeddings* — vetores densos gerados por modelos como o BERT — para representar consultas e documentos com base em seu significado. Esses vetores, criados por codificadores baseados em *Transformers*, capturam a semântica do texto de forma mais eficaz do que métodos esparsos como o TF-IDF (Li et al., [S.d.]). O princípio fundamental, conforme descrito por Turney e Pantel (2010), é que o significado de uma palavra pode ser inferido a partir do contexto em que ela aparece (BATISTA, 2024). O resultado é um espaço vetorial onde palavras com significados similares são mapeadas para vetores próximos. Essa proximidade pode ser calculada por meio de métricas como a similaridade de cossenos, permitindo que o sistema identifique relevância semântica mesmo na ausência de correspondência exata de palavras (BATISTA, 2024).

No contexto de sistemas de busca, os *embeddings* são a tecnologia que potencializa a busca semântica. Tanto as consultas dos usuários quanto os documentos do repositório são convertidos em vetores de *embedding*. O sistema, então, realiza a busca não mais por palavras-chave, mas pelo vetor da consulta, retornando os documentos cujos vetores estão mais próximos no espaço semântico.

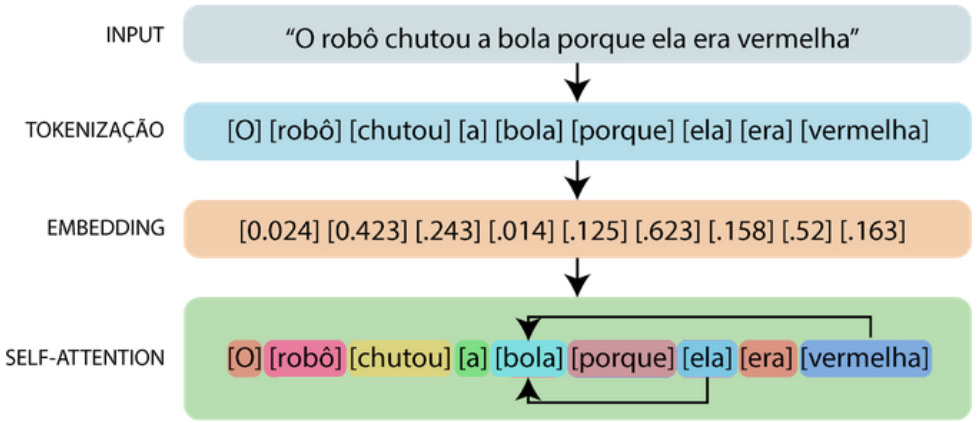
2.5.2. Transformers

Arquiteturas de *Transformers* como o BERT, por exemplo, revolucionaram a área ao permitir uma compreensão contextual profunda da intenção do usuário, indo muito além da análise de palavras-chave isoladas (Li et al., [S.d.]). Ela representa uma evolução dos modelos de redes neurais de codificador-decodificador (*encoder-decoder*) tradicionalmente usados para processar sequências de dados, como textos.

Os modelos de linguagem mais antigos processavam os dados de forma sequencial, palavra por palavra, o que os tornava lentos e suscetíveis à perda de contexto em textos longos, os *Transformers* introduziram uma inovação fundamental: o mecanismo de autoatenção (*self-attention*) (“What are Transformers?”, [S.d.]). A Figura 4 ilustra o processamento simultâneo de uma frase de forma simplificada. Para análise pelo *Transformer*, cada parte da frase é convertida em *tokens*, que são processados gerando *embeddings*

carregados de contexto sendo representados por valores. Na etapa de *self-attention*, são inferidas as relações e contextos das frases no texto.

Figura 4: Exemplo simplificado do fluxo de treinamento de um LLM para interpretação pelo mecanismo de *self-attention*. Nesse caso, o *Transformer* poderia concluir que "ela" estaria se tratando da "bola" nessa frase.



Fonte: Elaborado pelo autor.

Este mecanismo permite que o modelo analise todos os trechos de uma sequência de entrada simultaneamente, em vez de em ordem. Ao fazer isso, o modelo aprende a ponderar a importância de cada parte do texto em relação às outras, dando mais "atenção" aos trechos mais relevantes para prever a saída. ("What are Transformers?", *[S.d.]*). Essa capacidade de processamento paralelo não só torna os *Transformers* mais eficientes, permitindo seu treinamento em volumes de dados massivos, como também os torna mais eficazes em capturar relações de longo alcance e o contexto completo do texto, superando uma das principais limitações das arquiteturas anteriores.

2.6. Técnicas auxiliares para utilização de Inteligência Artificial Generativa

O poder interpretativo dos LLMs é, no entanto, acompanhado por um desafio inerente: a propensão à "alucinação", ou seja, a geração de informações que parecem corretas, mas são factualmente infundadas. Para evitar que isso ocorra, existem alguns métodos que são empregados, como *Retrieval Augmented Generation*, *Prompt Engineering* e *fine-tuning*.

2.6.1. Retrieval-Augmented Generation (RAG)

A RAG é uma técnica que funciona em um processo de duas etapas: primeiro, o sistema recupera um conjunto de documentos relevantes de uma base de conhecimento vetorial confiável (como um repositório institucional); em seguida, o LLM utiliza esses

documentos como contexto para gerar uma resposta final, que é, portanto, fundamentada nos dados recuperados (Li et al., *[S.d.]*). Essa técnica garante que as respostas não sejam apenas fluentes, mas também precisas e verificáveis.

A IA Generativa, quando estruturada através do RAG, não substitui a recuperação de informação, mas a potencializa. Ela cria um sistema onde a compreensão semântica, materializada pelos *embeddings*, serve para encontrar a informação correta, e o poder generativo do LLM serve para apresentá-la ao usuário da forma mais útil e direta possível. É a partir dessa arquitetura que soluções ainda mais avançadas, como o *self-querying*, podem ser construídas para refinar a etapa de recuperação.

2.6.2. Prompt engineering

A Engenharia de Prompts (*Prompt Engineering*) é a prática de projetar e refinar as entradas de texto (os *prompts*) fornecidas a um modelo de linguagem para obter respostas mais precisas, relevantes e alinhadas ao resultado desejado para guiar o modelo. Diferente de realizar todo treinamento ou modificar o modelo já existente, essa técnica foca na otimização da consulta, tratando o modelo como uma ferramenta a ser operada. De acordo com (Gadesha, 2025), a engenharia de *prompts* pode envolver a adição de instruções explícitas, contexto, exemplos (*few-shot learning*) ou a formatação da pergunta para guiar o raciocínio do modelo. A abordagem de *self-querying*, central neste trabalho, é uma aplicação de engenharia de *prompts*, na qual uma instrução bem elaborada é enviada ao LLM para que ele traduza a consulta do usuário em uma estrutura de busca que combina vetores semânticos e filtros de metadados (“LangChain: How to do ‘self-querying’ retrieval”, *[S.d.]*).

2.6.3. Fine-tuning

O Ajuste Fino (*Fine-tuning*) é um processo que modifica o próprio modelo de linguagem, normalmente usado para adaptar seus parâmetros internos a um domínio ou tarefa específica. Esta abordagem consiste em continuar o treinamento de um modelo pré-treinado (como o GPT ou BERT) utilizando um conjunto de dados menor e mais específico, que seja representativo do problema a ser resolvido. Conforme discutido na pesquisa sobre Recuperação de Informação Generativa, o ajuste fino é uma estratégia poderosa para especializar um modelo de propósito geral, fazendo com que ele aprenda a terminologia, o estilo e as nuances de uma área particular, como a jurídica, a médica ou, no caso deste trabalho, a acadêmica (Li et al., *[S.d.]*). Enquanto a engenharia de *prompts* busca extrair o

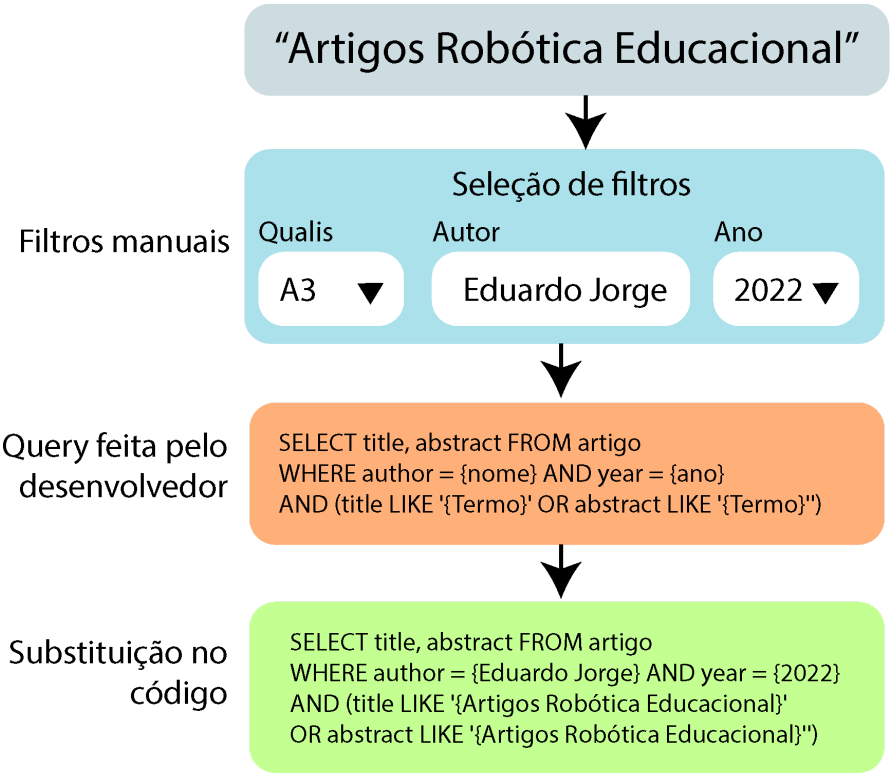
melhor de um modelo já existente, o ajuste fino altera os pesos da rede neural para criar uma versão especializada do modelo, o que geralmente exige um custo computacional maior e um esforço significativo na curadoria do conjunto de dados de treinamento.

2.7. Self-querying

Embora a abordagem RAG assegure a factualidade das respostas, ela não resolve nativamente um desafio importante: como aplicar filtros baseados em metadados específicos contidos em consultas em linguagem natural. Por exemplo, diante da pergunta "Quais artigos do autor Silva sobre IA foram publicados após 2022?", o sistema precisa interpretar não apenas o tema "IA", mas também identificar os critérios "autor" e "ano" como filtros relevantes. É nesse contexto que surge o conceito de *self-querying* (ou autoconsulta). Essa técnica propõe que o próprio modelo de linguagem (LLM) atue como planejador da consulta, sendo capaz de decompor uma pergunta em dois componentes principais: (i) uma *string* de busca para recuperação semântica e (ii) um conjunto de filtros estruturados baseados em metadados, previamente definidos e instruídos ao modelo. Essa abordagem, popularizada por *frameworks* como o *LangChain*, permite que a IA interprete, extraia e aplique automaticamente filtros diretamente da linguagem natural usada pelo usuário (BATISTA, 2024; "*LangChain: How to do 'self-querying' retrieval"*, *[S.d.]*)

Com base na abordagem do *self-query*, o programador não precisa mais construir manualmente as consultas no *backend*, como tradicionalmente era feito com instruções do tipo *SELECT* onde o programador define todos os filtros possíveis para uma pesquisa de termo (Figura 5). Assim, a responsabilidade pela formulação da consulta é transferida para a inteligência artificial.

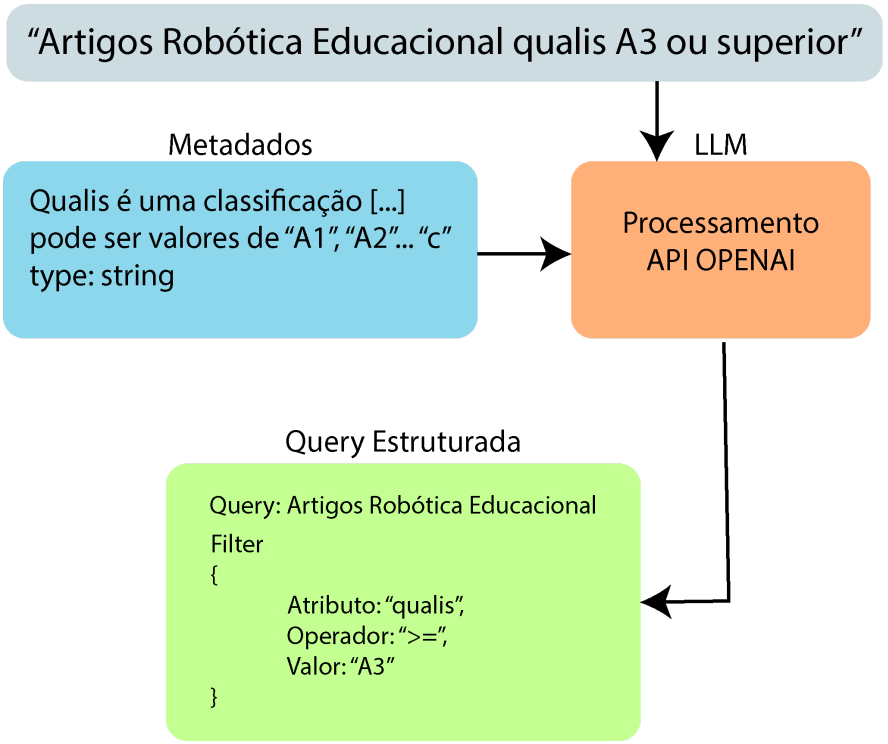
Figura 5: Fluxo de pesquisa pelo método tradicional com filtros manuais da interface e substituição dos valores recebidos em um código SQL.



Fonte: Elaborada pelo autor

O processo funciona da seguinte forma: o sistema fornece ao LLM uma descrição clara da estrutura dos metadados disponíveis, que é previamente descrita por um desenvolvedor (Figura 6). Com base nessa descrição e no *prompt* de instruções da biblioteca do *LangChain*, o modelo é capaz de interpretar a intenção na consulta e gerar de forma autônoma a separação dos elementos da frase, como uma *query* SQL ou outra forma de filtragem aplicável à base de dados *("LangChain: How to do 'self-querying' retrieval", [S.d.])*.

Figura 6: Fluxo de pesquisa pelo método de self-querying com a passagem da *query* e da definição dos metadados para obter a *query* estruturada em conteúdo e filtro.



Fonte: Elaborada pelo autor

Entre as principais vantagens dessa abordagem está a flexibilidade no desenvolvimento e na manutenção dos sistemas. Não é mais necessário modificar diretamente o código-fonte sempre que houver mudanças nos requisitos ou na lógica dos filtros, seja no *backend* para formulação de novas consultas ou no *frontend* com o replanejamento da interface. Caso um novo critério de busca seja introduzido, basta atualizar as instruções fornecidas ao modelo e adicionar os metadados novos para serem passados juntos aos documentos que estão sendo filtrados. Assim, reduz-se o tempo de desenvolvimento, minimizam-se erros humanos e a equipe técnica pode se concentrar em aspectos mais estratégicos da aplicação.

3. REVISÃO DA LITERATURA

A base teórica deste estudo foi construída por meio de uma Revisão Sistemática da Literatura (RSL). O desenvolvimento desta revisão tem como objetivo identificar, analisar, classificar e entender o estado da arte em relação aos métodos existentes para otimização e/ou refinamento de buscas com o auxílio de inteligência artificial usando processamento de linguagem natural. As etapas realizadas seguem o protocolo proposto por Kitchenham (2004), que consiste em elaborar perguntas de pesquisa, seleção das bases de dados de conhecimento e da definição da *string* de busca, estabelecer critérios de busca, avaliação qualitativa e extração dos dados.

Para auxiliar na revisão, foram elaboradas as questões de pesquisa presentes no Quadro 3. O intuito dessas questões é responder sobre o contexto que esse trabalho se propõe, que é o uso de Inteligência Artificial em buscas textuais além da identificação de tecnologias e métodos tradicionais e atuais que são usados nessa área.

Quadro 3: Perguntas de pesquisa da revisão sistemática.

Perguntas da revisão	Motivação
1) Como a IA está sendo utilizada para aprimorar a busca textual?	Para compreender o estado da arte, a pergunta central desta revisão sistemática está na utilização de IA no refinamento ou otimização de busca textual para obter melhores resultados e/ou melhor usabilidade.
2) Quais métodos existem para otimizar a recuperação de informações em bancos de dados e quais são as implicações de seu uso?	Como suporte à questão central, é fundamental compreender o que está sendo usado hoje na otimização de busca e as vantagens/desvantagens da aplicação dessas estratégias.
3) Quais tecnologias, ferramentas ou procedimentos são utilizados no sistema de busca atual?	Também é necessário entender quais ferramentas ou soluções existem nos sistemas de busca atuais, em quais etapas eles são aplicados e se são usados em ambientes reais.

Fonte: Elaborado pelo Autor

O protocolo da revisão foi executado seguindo etapas bem definidas. A busca pelos trabalhos foi realizada em importantes bases de dados científicas, como *IEEE Xplore, ACM Digital* e *SpringerLink*, reconhecidas pela alta qualidade de suas publicações. A *string* de busca, presente no Quadro 4, foi construída para abranger os conceitos centrais da pesquisa, combinando termos e adaptando a string de acordo com a restritividade de cada base. No caso

da *IEEE Xplore*, foram adicionados mais termos comumente relacionados à Recuperação de informação, como "Neural IR" OR "Neural Information Retrieval" e devido aos baixos resultados

Quadro 4: String de busca para as bases de conhecimento

ACM Digital e *Springer*	*("Natural Language Query" OR "Textual Input" OR "Text input" OR "User Query" OR "Text-based Search" OR "Text Search Interface")* *AND ("Search Optimization" OR "Search Engines" OR "Search System")* *AND ("Deep Learning" OR "Machine Learning" OR "Natural Language Query" OR "LLM" OR "Large Language Model" OR "self-query" OR "Fine-Tuning")* *AND ("Current methods" OR "existing approaches" OR "present techniques" OR "recent strategies" OR "Hybrid Approaches" OR "semantic search" OR "Text Embedding")* *AND ("Maintainability" OR "Retrieval Quality" OR "Search accuracy" OR "System Scalability" OR "Usability")*
IEEE *Xplore*	*("Search Optimization" OR "Search Engines" OR "Search System" OR "Information Retrieval Systems")* *AND ("Deep Learning" OR "Machine Learning" OR "Natural Language Query" OR "LLM" OR "Large Language Model" OR "self-query" OR "Fine-Tuning")* *AND ("Current methods" OR "Current approaches" OR "existing approaches" OR "present techniques" OR "recent Approaches" OR "Hybrid Approaches" OR "semantic search" OR "Text Embedding" OR "Neural IR" OR "Neural Information Retrieval" OR "Most Used")*

Fonte: Elaborado pelo Autor

Para garantir a relevância e a atualidade dos trabalhos, foram definidos critérios de seleção rigorosos após a obtenção dos artigos a partir das *strings* de busca. Nesse processo, os critérios são aplicados pela análise do título e resumo dos artigos. Os critérios de inclusão abrangeram artigos publicados em periódicos com revisão por pares, no período de 2020 a 2025, publicados em inglês. Ainda, como critério de aceitação buscou-se artigos que:

- Apresenta resultados empíricos, experimentos ou análise comparativa entre métodos de busca;
- Aborda o uso de IA, aprendizado de máquina ou LLMs aplicados à melhoria de sistemas de busca ou recuperação de informação;
- Descreve ou analisa métodos atuais/recentes de busca textual ou técnicas de recuperação de informação em bancos de dados;

- O artigo descreve, avalia ou propõe métodos de recuperação de informação textual em bases de dados, com foco em consultas feitas via linguagem natural;

Por outro lado, os critérios de exclusão descartaram artigos:

- O artigo não menciona ou utiliza IA, LLMs ou técnicas relacionadas à otimização de busca textual;
- O foco do artigo é a busca por imagem, voz, vídeo ou sensores, sem ou pouca relação com a entrada textual;
- Trabalhos não disponíveis na íntegra;

O Quadro 5 sintetiza o processo de filtragem iniciado com 48 artigos obtidos de três bases de dados, majoritariamente do *Spring Link*. Após a leitura de títulos e resumos e a aplicação dos critérios de exclusão, 20 artigos foram selecionados por sua aderência aos objetivos e questões de pesquisa. Destes, 6 foram excluídos devido à indisponibilidade de acesso ao texto completo, resultando em 14. A etapa final consistiu na análise de qualidade, utilizando perguntas sobre: a metodologia, o uso de LLM para otimização da busca, a clareza dos resultados (com métricas como acurácia, tempo, perda) e a discussão da aplicabilidade prática (manutenção, escalabilidade, usabilidade). Como resultado, 2 artigos sobre "*Knowledge Graphs*" foram removidos por não abordarem LLM para otimização de busca, não apresentarem métricas, não discutirem questões de aplicabilidade e focarem na construção de grafos de conhecimento em vez de busca textual.

Quadro 5: Número de artigos retornados em cada etapa da revisão sistemática.

Processo	Número de artigos
Total de artigos vindo da *string* de busca	48
Total de artigos após a leitura do título, *abstract* e palavras chave	20
Total de artigos que não foi possível obter o acesso	14
Remoção após leitura completa	12

Fonte: Elaborado pelo Autor

Em resposta à primeira questão de pesquisa, "Como a IA está sendo utilizada para aprimorar a busca textual?", a revisão identificou diversas abordagens. A IA é empregada no tratamento das consultas do usuário e na geração de respostas, através do Processamento de Linguagem Natural (PLN), da geração de respostas contextuais e da sumarização de dados.

Destaca-se também o uso de *embeddings*, com modelos como SBERT e BERT, para representar semanticamente textos e consultas, para a realização de busca vetorial.

Referente à segunda questão, "Quais métodos existem para otimizar a recuperação de informações em bancos de dados e quais são as implicações de seu uso?", os estudos analisados focam em métodos que aprimoram o processamento da consulta do usuário. Além das técnicas de IA já mencionadas, foi identificado o uso de *query expansion* (expansão de consulta). Este método busca adicionar termos semanticamente relacionados à pesquisa inicial, com o objetivo de obter resultados com relevância mais próxima da intenção do usuário e, potencialmente, maior diversidade. Observou-se também a exploração de métodos de ranqueamento alternativos, que visam evitar vieses de sistemas tradicionais (baseados apenas em frequência) e orientar o usuário, sugerindo contextos ou perguntas para aumentar a assertividade da busca.

Finalmente, em relação à terceira questão, "Quais tecnologias, ferramentas ou procedimentos são utilizados no sistema de busca atual?", a revisão apontou que métodos tradicionais e recentes, como TF-IDF e BM25, ainda são amplamente utilizados, baseando-se na frequência de termos nos documentos para calcular a relevância. Contudo, a principal tendência identificada são os esforços para melhorar a busca semântica através do uso de modelos de IA para consultas em linguagem natural.

Portanto, os trabalhos que compõem a base teórica desta monografia, como os estudos Li et al. (2025), e outros discutidos no capítulo de Trabalhos Correlatos, não são meramente exemplares, mas sim o resultado direto deste processo metodológico. Eles foram selecionados por representarem as contribuições mais significativas e alinhadas às questões de pesquisa, e é a partir da síntese e da identificação de lacunas nesses estudos que o presente Trabalho de Conclusão de Curso se justifica e se posiciona de forma original no campo de pesquisa.

É importante ressaltar que, complementarmente aos artigos selecionados através do protocolo formal de busca, a presente pesquisa teve como ponto de partida a análise aprofundada da monografia de Batista (2024). Este trabalho serve como precursor no contexto de busca semântica em dados acadêmicos brasileiros e da utilização de bibliotecas para o *self-query* que ocorreu em seu segundo ciclo de desenvolvimento. Além disso, foi a partir dos desafios práticos encontrados pelo autor, especialmente na aplicação da técnica de *self-querying*, que a principal lacuna de pesquisa foi identificada. Portanto, a problemática central deste TCC, focada na precisão dos filtros e, principalmente, na avaliação da

arquitetura de software proposta em relação à manutenibilidade e evolução para sistemas de busca, foi diretamente derivada dos caminhos para trabalhos futuros apontados por Batista (2024), alinhando este projeto a uma continuidade natural e necessária da pesquisa na área.

4. TRABALHOS CORRELATOS

A investigação sobre a otimização de sistemas de busca para repositórios de conhecimento é um campo ativo de pesquisa, com diversas abordagens sendo exploradas para aprimorar a recuperação de informação. Nesta seção, são analisados trabalhos que, embora compartilhem objetivos similares, empregam metodologias e tecnologias distintas. Isso permite contextualizar como o presente projeto contribui de forma específica para a área.

No que tange à tecnologia de base para a compreensão da linguagem, o trabalho de Chang et al. (2023) na construção de um robô interativo para a área da saúde oferece um paralelo tecnológico importante. Para que o sistema encontre a resposta correta em sua base de conhecimento, os autores utilizam um modelo *transformer,* baseado em atenção, similar ao BERT, para gerar *embeddings* de sentenças. Assim como este trabalho, Chang et al. (2023) emprega o uso de modelos LLM para capturar o significado semântico das consultas e dos documentos para realização de buscas semânticas. O autor diverge em relação ao objetivo desta pesquisa por construir um sistema para gerar perguntas e respostas, enquanto esta monografia foca na recuperação de informação, que são os artigos de uma base de dados, a partir de uma *query* de usuário.

Outra abordagem relevante para a organização do conhecimento acadêmico é apresentada por Subramanian et al. (2025) no desenvolvimento do KMSBOT, um sistema de busca para uma instituição de ensino. A principal semelhança com este projeto reside no objetivo de criar uma interface de busca centralizada e semântica para um repositório de conhecimento institucional. Para isso, os autores também empregam técnicas de PLN para interpretar as consultas dos usuários, da mesma forma como será empregado na solução desta monografia. No entanto, a arquitetura diverge na tecnologia empregada para a realização das buscas: enquanto a solução do KMSBOT é baseada na construção de um Grafo de Conhecimento para modelar previamente as relações entre as informações, a abordagem deste trabalho utiliza a capacidade dos LLMs de interpretar e estruturar essas relações por meio de *embeddings* gerados pelos modelos, sem a necessidade de uma modelagem de dados tão rígida e com manutenção intensiva.

O trabalho mais próximo desta pesquisa, servindo como um precursor direto, é a monografia de Batista (2024). O autor desenvolveu um mecanismo de busca semântica para a Plataforma Lattes, demonstrando a eficácia do uso de *embeddings* para a recuperação de informação no domínio acadêmico brasileiro. A investigação de Batista (2024) explorou em

um segundo ciclo de desenvolvimento, a aplicação da técnica de *self-querying* usando uma biblioteca do Langchain. Nesta fase exploratória, o autor encontrou desafios relacionados à precisão e à confiabilidade dos filtros gerados pelo LLM, um problema que ele próprio reconheceu e apontou como uma lacuna relevante para trabalhos futuros. De forma similar ao realizado por Batista (2024), esse trabalho usará a LLM para gerar os *embeddings* para a busca semântica usando as funções do PGVector do PostgreSQL. O que difere, principalmente, em relação ao ciclo 2 de desenvolvimento dele é que no artefato computacional criado nesta monografia não será usada a busca semântica com filtragem de forma nativa da biblioteca do Langchain.

A proposta deste TCC, portanto, contribui em duas frentes complementares. Primeiramente, busca avançar na solução do problema técnico da precisão dos filtros em *self-querying* identificados por Batista (2024). Em segundo lugar, e como diferencial principal, analisar os benefícios arquiteturais dessa abordagem em termos de manutenibilidade e evolução do código. O objetivo é demonstrar como uma arquitetura baseada em *self-querying* pode desacoplar a lógica de filtragem do código da aplicação, resultando em um sistema mais flexível, cuja manutenção e evolução — como a adição de novos filtros — possam ser realizadas com mínimo ou nenhum retrabalho de codificação, impactando positivamente o ciclo de vida do software.

Quadro 6: Resumo dos objetivos, tecnologias e principais diferenças entre os trabalhos correlatos e a monografia.

Trabalho de Referência	Objetivo Principal	Tecnologia/Técnica Principal	Principal Diferencial e Relação com este TCC
(Chang et al., 2023)	Pergunta e Resposta na área da saúde.	• Embeddings de sentenças com modelos baseados em atenção (BERT).	Semelhança tecnológica: Uso de embeddings de *Transformers* para compreensão semântica. Diferença de escopo: Foco em Q&A (uma resposta) vs. RI (lista de documentos).
(Subramanian et al., 2025)	Gestão e busca em conhecimento de uma instituição de ensino.	• Grafo de Conhecimento (Knowledge Graph). • Processamento de Linguagem Natural (PLN)	Semelhança de objetivo: Centralizar e facilitar o acesso ao conhecimento acadêmico. Diferença de arquitetura: Abordagem rígida baseada em grafo vs. abordagem flexível com LLMs.

(BATISTA, 2024)	Busca semântica de alta performance e exploração inicial de *self-querying*.	• *Word Embeddings* para busca semântica. • Teste exploratório de *self-querying*.	Avanço sobre a lacuna: Foco em resolver o problema de precisão identificado por Batista (2024). Contribuição original: Análise sob a perspectiva de Engenharia de *Software*, avaliando a manutenibilidade e evolução da arquitetura.

Fonte: Elaborado pelo Autor

5. METODOLOGIA

Esta pesquisa pode ser caracterizada como de natureza aplicada, pois visa a resolução de um problema prático relacionado à recuperação de informação em sistemas de busca acadêmicos. Ela possui também uma abordagem empírica uma vez que envolve a construção e avaliação de um protótipo de software funcional. Sendo assim, a metodologia adotada para a pesquisa é o *Design Science Research* (DSR). O enfoque do DSR é estruturar um percurso metodológico para a concepção e construção de artefatos que solucionem problemas práticos (Lacerda et al., 2013).

Como etapa fundamental e predecessor ao ciclo de design, foi realizada uma Revisão Sistemática da Literatura (RSL). Este processo metodológico foi conduzido com o objetivo de identificar, avaliar e sintetizar as evidências científicas mais relevantes e atuais relacionadas ao problema de pesquisa. A RSL garantiu que a fundamentação teórica deste trabalho fosse construída sobre uma base sólida, permitindo a identificação dos principais conceitos, das tecnologias e, crucialmente, das lacunas existentes na literatura. Os trabalhos correlatos e as teorias apresentadas nos capítulos anteriores são, portanto, o resultado direto desta revisão, que orientou tanto a definição do problema quanto o delineamento da solução proposta.

O desenvolvimento da pesquisa seguirá as seguintes etapas principais do ciclo da DSR, que foram adaptadas de autores como Manson (2006) e Vaishnavi & Kuechler (2009):

- Conscientização do Problema: Esta fase consiste no levantamento e na formalização do problema de pesquisa. Uma revisão sistemática da literatura é realizada para identificar as limitações dos métodos atuais de recuperação de informação e das tecnologias envolvidas no processo de *self-querying*.
- Sugestão: Com base na compreensão do problema, esta etapa envolve a concepção de uma ou mais propostas de solução. Nesta etapa, projeta-se a arquitetura do artefato, que utiliza a Generativa e técnicas de *self-querying* para a criação de filtros dinâmicos. As premissas e os requisitos para a construção do artefato são explicitados e justificados além das definições das camadas de *frontend*, relacionado ao aspecto visual da aplicação e comunicação com a API e o *backend*, que terá a parte de tratamento da base de dados e os controles relacionados com o processamento da consulta do usuário.
- Desenvolvimento: Esta etapa corresponde à construção do artefato em si, ou seja, a implementação do protótipo funcional. O desenvolvimento é guiado pela arquitetura

definida na fase anterior, resultando em uma instanciação capaz de ser testada e avaliada.

- Avaliação: Na avaliação é verificada a utilidade, a qualidade e a eficácia do artefato desenvolvido em relação aos objetivos propostos, com ênfase no impacto da arquitetura *self-querying* na manutenibilidade e evolução do sistema. Para a avaliação dos resultados do artefato, será feita uma comparação entre um sistema de *baseline* do Sistema de Mapeamento de Competências Científicas (SIMCC), fornecido pelo orientador, e o sistema desenvolvido. Inicialmente, será realizada a contabilização de métricas gerais quantitativas como complexidade ciclomática e linhas de código, e qualitativas como acoplamento e coesão das classes utilizadas. Em seguida, a solução desenvolvida terá sua flexibilidade avaliada através da incorporação de novos filtros com o objetivo de analisar o deles no sistema de *baseline* e no sistema elaborado.
- Comunicação: Por fim, a etapa de Comunicação será apresentado os resultados e o conhecimento gerado ao longo do processo. Serão sintetizadas as principais aprendizagens, a contribuição do trabalho para a "classe de problemas" em questão e as implicações para futuras pesquisas, comunicando os resultados tanto para a comunidade acadêmica quanto para profissionais da área.

As próximas subseções descrevem com mais detalhes sobre o ambiente experimental e os critérios de avaliação adotados. Primeiramente, é apresentado sobre o sistema de *baseline* que o artefato desenvolvido será comparado, o Sistema de Mapeamento de Competências Científicas (SIMCC), detalhando suas características arquiteturais. Posteriormente, são apresentadas as fases de validação para mensurar o possível ganho de manutenibilidade e evolução pela nova abordagem em comparação ao sistema de referência.

5.1. Sistema do SIMCC

Esta monografia utiliza o Sistema de Mapeamento de Competências Científicas (SIMCC) como *baseline* do sistema tradicional. O SIMCC é uma ferramenta desenvolvida pelos pesquisadores da Universidade Estadual de Santa Cruz (UESC), Universidade Federal de Minas Gerais, Universidade Federal do Recôncavo da Bahia e Universidade Estadual da Bahia para facilitar a identificação de docentes das Universidades que tenham histórico de familiaridade com temas de interesse de colaboradores, sua base de dados é composta por currículos dos docentes da UESC e foi feita a partir dos dados do Currículo Lattes (“Sistema de Mapeamento de Competências - SIMC | Nit Uesc”, *[S.d.]*). O sistema possui diferentes

tipos de busca, como busca por pesquisadores, patentes, trabalhos acadêmicos, visualização das produções acadêmicas por tema e entre outros.

A versão cedida do código, que foi disponibilizada pelo orientador e está pública no GitHub, é um fragmento relacionado apenas à parte de busca lexical e a filtragem. Esta *baseline* implementa a lógica de filtragem de forma tradicional, ou seja, fortemente acoplada ao código-fonte da aplicação que também é feita usando Python e FastAPI. Sendo assim, servindo como um ponto de comparação ideal para a arquitetura proposta que visa o desacoplamento dessa lógica. Vale ressaltar que a validação realizada se limita na análise apenas do *backend* para um escopo mais reduzido e mais detalhado do trabalho.

5.2. Validação

A avaliação comparativa será conduzida em duas fases principais que serão detalhadas nesta seção. Ela tem como objetivo medir a complexidade estática e o esforço de evolução de ambos os sistemas.

5.2.1. Fase 1: Análise estática

Inicialmente, será realizada uma visão geral de ambos os sistemas (*baseline* e proposto) através da contabilização e análise de métricas de qualidade de software nos módulos responsáveis pela filtragem. O objetivo é quantificar a complexidade estrutural e o potencial de manutenibilidade. As métricas selecionadas para a avaliação, fundamentadas em Pressman e Maxim (2021), incluem:

- **Métricas Quantitativas:**
    - Linhas de Código: medida do tamanho do código responsável pela lógica de filtragem. Essa medida de volume direta auxilia a dar uma visão geral do código apesar de penalizar programadores que comentam o código ou deixam espaçamentos.
    - Complexidade Ciclomática (CC): medida da complexidade estrutural e do número de caminhos de teste independentes no código .
- **Métricas Qualitativo-Conceituais:**
    - Acoplamento (*Coupling*): nível de interdependência entre o módulo de filtragem e os demais componentes do sistema.
    - Coesão (*Cohesion*): nível em que as responsabilidades dentro do módulo de filtragem estão relacionadas e focadas.

5.2.2. Fase 2: Análise de evolução

Posteriormente, a flexibilidade e o esforço de evolução das arquiteturas serão avaliados através da simulação de cenários de mudança realistas. O objetivo é analisar o impacto da adição de novos filtros em ambos os sistemas, medindo métricas como a quantidade de arquivos ou módulos alterados, e linhas de código alteradas. Para isso, serão consideradas apenas as modificações dentro do sistema do *backend*, sem levar em conta possíveis processos de carregamento e transformação dos dados necessários para a captura e carregamento do novo metadado.

Os cenários de teste de evolução planejados são:

- Cenário A (Adição de Filtro Simples): Implementação do filtro "Tipo de publicação" que pode ser filtrado apenas 1 ou mais tipos de publicação na mesma busca. Este cenário visa testar a adição de um novo campo de metadados que será usado apenas para filtragem (recebido pelo endpoint), mas não vai alterar a estrutura de retorno da API pois não há mudança no componente que mostra os artigos.
- Cenário B (Adição de Filtro e Retorno): Implementação do filtro "Número de citações". Este cenário é mais complexo, pois exige que o novo campo seja usado tanto para a lógica de filtragem quanto para ser retornado na resposta da API .

Espera-se que, por meio desses cenários, o sistema baseline exija mais alterações diretas no código-fonte para ambos os cenários (aumentando sua CC), enquanto o protótipo proposto deverá acomodar as mudanças com impacto mínimo ou nulo no código da aplicação, validando a hipótese de maior manutenibilidade e facilidade de evolução.

6. DESCRIÇÃO DO PROJETO

Esta seção apresenta o relatório técnico do desenvolvimento da solução computacional, detalhando a arquitetura do sistema, as tecnologias empregadas, os processos de tratamento de dados e o fluxo de execução da funcionalidade central de busca. O objetivo é descrever as etapas e decisões técnicas que permitiram a construção do protótipo funcional para o problema de pesquisa investigado.

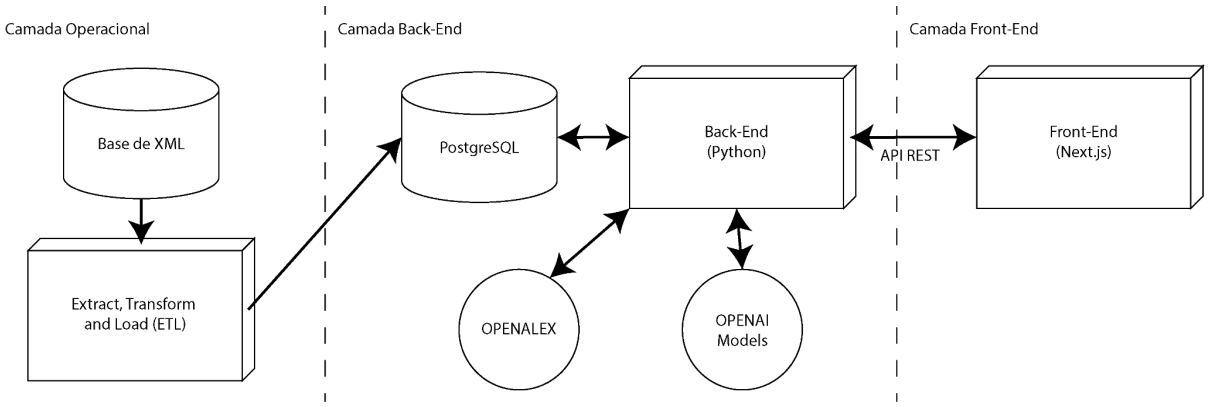
Para a validação e demonstração da abordagem proposta, será utilizada como base de dados a produção científica do Núcleo de Pesquisa Aplicada em Inteligência Artificial (NPAI). Este acervo é composto por cerca de 274 artigos científicos, abrangendo a produção de nove pesquisadores e professores vinculados ao grupo. A diversidade e a especificidade deste conjunto de dados oferecem um cenário realista e relevante para testar a eficácia da recuperação de informação semântica e a filtragem dinâmica de metadados.

6.1. Arquitetura geral do sistema

A solução foi desenvolvida seguindo o padrão de arquitetura cliente-servidor, com uma separação clara entre a camada de apresentação (*frontend*) e uma aplicação de *backend*, que se comunicam através de uma API Rest (Figura 7). Essa abordagem permite o desenvolvimento independente das camadas, ao mesmo tempo que centraliza a lógica de negócio no servidor. A figura 4 mostra uma visão geral das camadas planejadas para o sistema:

- Camada Operacional: Ela possui uma base XML que será tratada por um orquestrador de dados, o intuito é formatar esses dados para serem consultados em um banco de dados exclusivo e otimizado para consultas, não impactando no banco operacional do grupo de pesquisa.
- Camada *Backend:* nessa camada os dados processados vão ser armazenados em um banco PostgreSQL para poder ser utilizado e processado pela API *backend* (Python) desenvolvida. Por se tratar de uma lógica sequencial, optou-se por reduzir a complexidade no desenvolvimento de um *backend* seguindo uma arquitetura em camadas com seus dados centralizados no PostgreSQL.
- Camada *Frontend:* realizando a comunicação com a API do *backend* para transmitir as consultas do usuário e mostrar o resultado delas.

Figura 7: Diagrama da arquitetura geral da solução *self-querying.* A camada Operacional que contém o banco de operação do grupo NPAI e a orquestração dos dados. A camada Backend que possui o banco PostgreSQL otimizado para consultas, o *backend* em python para orquestrar a busca híbrida e o uso das APIs externas. A camada *Frontend* responsável por realizar requisições e mostrar os resultados ao usuário.



Fonte: Elaborada pelo autor

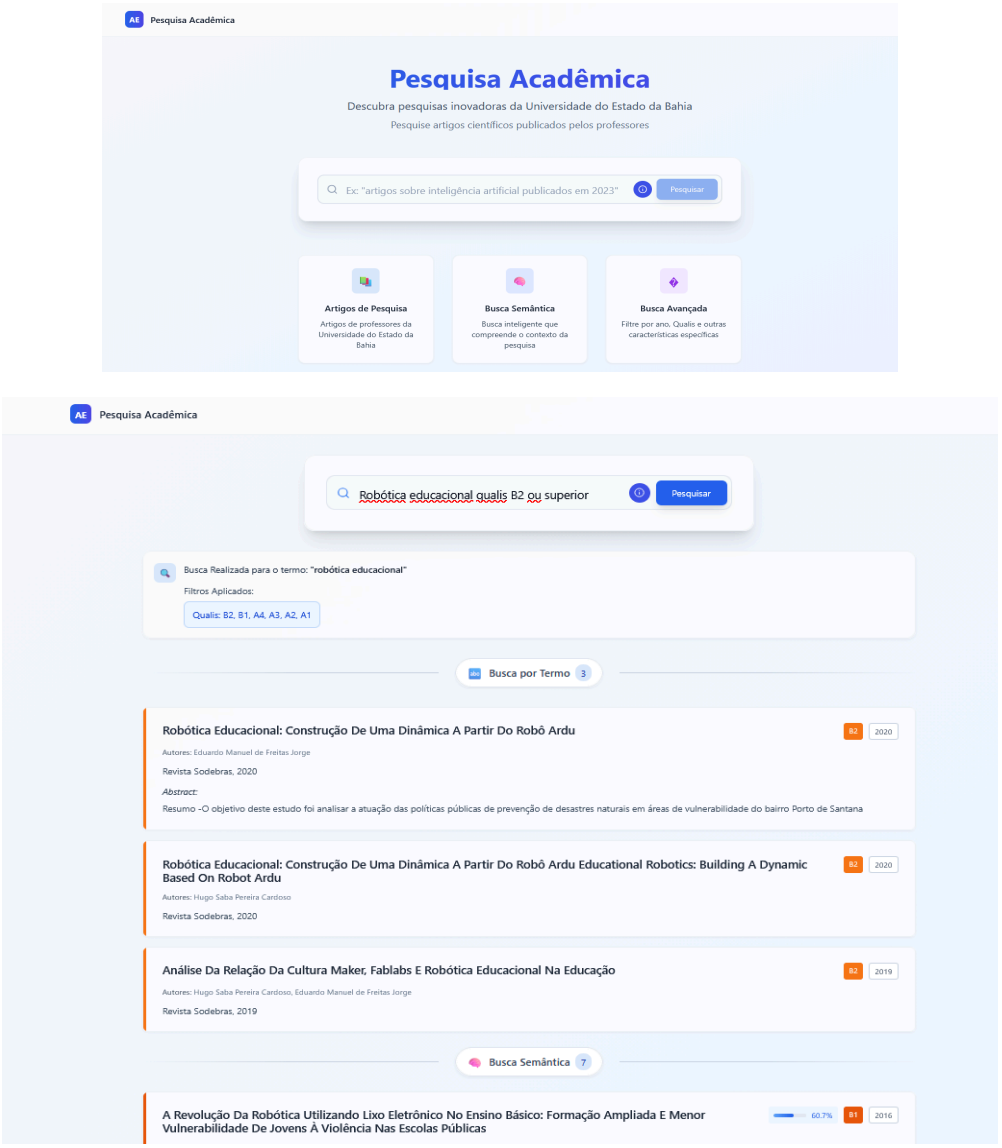
Internamente, o *backend* foi organizado seguindo a arquitetura MVC (*Model-View-Controller*), que é dividida em camadas para promover a separação de responsabilidades e a alta coesão, conforme refletido na estrutura de diretórios do projeto:

- Camada de Controle (*Controller*): Implementada com FastAPI, é responsável por gerenciar os *endpoints* da API, receber as requisições HTTP e orquestrar as respostas.
- Camada de Serviço (*Service*): Classes de serviço que isolam e abstraem as lógicas de negócio específicas, como a comunicação com a API da OpenAI, a lógica do *self-querying* com LangChain e a busca textual. Essa camada serve para desacoplar a lógica de negócio dos detalhes da API.
- Camada de Acesso a Dados (DAO - *Data Access Object*): Responsável pela comunicação direta com o banco de dados, abstraindo as consultas SQL.
- Camada de modelos de dados (*Model*) - Responsável por representar as entidades do banco de dados que são retornadas nas consultas. Possui *Data Transfer Object* (DTO) que são classes para comunicação de dados sem lógica de negócio entre diferentes camadas do sistema, como no caso de respostas que envolvem múltiplas tabelas do banco, para serem formatadas para o *frontend.* Por fim, temos os *Mappers* que vão fazer a conversão de um dicionário retornado do PostgreSQL para objetos do sistema, como um DTO.

O *backend* foi estruturado utilizando Python e o framework FastAPI. A escolha do Python foi motivada pelo seu vasto ecossistema de bibliotecas para Inteligência Artificial, especialmente LangChain, que é central para a implementação do mecanismo de *self-querying*. O FastAPI foi um *framework* selecionado pela sua alta performance, fácil testagem dos *endpoints* via *localhost* e pela sua documentação simplificada de *endpoints*.

O *frontend*, responsável pela interface do usuário, foi desenvolvido com Next.js (Figura 6), um framework React que oferece agilidade no desenvolvimento e otimização de performance, dada a familiaridade com a tecnologia e de ferramentas de construção de componentes.

Figura 8: Frontend em Next.js. A primeira imagem mostra a interface geral. A segunda imagem mostra o resultado de uma busca com campos de conteúdo identificado e filtros aplicados.



Fonte: Elaborada pelo autor

6.2. Processo de Extract, *Transform and Load* (ETL)

A construção da base de conhecimento do sistema iniciou-se com a coleta de dados brutos, fornecidos pelo pesquisador do grupo de pesquisa NPAI em formato de arquivos XML extraídos da Plataforma Lattes. Para orquestrar a extração, transformação e carga desses dados (processo de ETL), foi utilizada a ferramenta Apache Hop.

O fluxo de trabalho no Apache Hop foi configurado para:

1. Extrair: Ler os arquivos XML e extrair os dados bibliográficos relevantes, como títulos de artigos, autores, periódicos e ano de publicação.
2. Transformar: Realizar a limpeza e a normalização dos dados, tratando inconsistências e padronizando formatos para garantir a integridade da base.
3. Carregar: Inserir os dados estruturados nas tabelas do banco de dados PostgreSQL.

Paralelamente, para enriquecer a base de dados com informações que serão tratadas nas buscas semânticas, foi implementado um serviço que consome a API da OpenAlex. Este serviço busca pelos artigos presentes na base e faz uma requisição à essa API externa para obter os dados dos resumos e atualizar o banco. Essa etapa é fundamental para gerar *embeddings* mais enriquecidos semanticamente para cada artigo e não se restringir ao título do artigo.

6.2.1. Processo de Inicialização e Indexação Semântica

Além do ETL inicial para carregamento da base de dados, a aplicação executa também um processo de inicialização para eventuais atualizações da base, gerenciado pelo mecanismo de *lifespan* do FastAPI. Este processo garante que o sistema esteja pronto para realizar as buscas híbridas e inclui as seguintes etapas:

- Sincronização de Dados: São executadas rotinas de sincronização para dados voláteis, como os resumos de artigos que foram adicionados.
- Geração de Embeddings: O serviço *EmbeddingService* é invocado para verificar se existem artigos na base de dados com a coluna de *embeddings* vazias, e gerando se necessário. Esta etapa utiliza o modelo *text-embedding-3-small* da OpenAI para converter o conteúdo textual dos artigos em vetores semânticos.

- Indexação Semântica: O serviço *SemanticSearchService* é chamado para indexar os *embeddings* gerados, preparando a estrutura de dados para a busca por similaridade de cosseno.

6.2.2. Centralização da Lógica de Dados com *view*

Para apoiar a arquitetura de desacoplamento, foi implementada uma camada de abstração diretamente no banco de dados PostgreSQL. O objetivo estratégico desta decisão é centralizar o ponto de manutenção para a lógica de filtragem, permitindo que o sistema evolua com o mínimo de atrito no código-fonte da aplicação.

No banco PostgreSQL, foi criada uma *view* denominada *vw_artigos_completos* como pode ser vista da Figura 9. Esta *view* atua como a única fonte de dados para o sistema de busca semântica e por termos, normalizando e unificando, todos os campos relevantes para a consulta e que são usados pela filtragem do *self-query*, originários de múltiplas tabelas (como artigo, periódico e pesquisador).

Figura 9: Script da view criada no PostgreSQL que faz o papel de uma camada de abstração para o backend.

```
CREATE OR REPLACE VIEW vw_artigos_completos AS
SELECT
    a.id_artigo as id,
    a.nome as title,
    per.nome as journal,
    a.ano as year,
    a.resumo as abstract,
    a.doi,
    per.qualis,
    p.id_pesquisador as author_id,
    p.nome as authors,
    a.embedding
FROM artigo a
JOIN periodico per ON a.id_periodico = per.id_periodico
JOIN pesquisador p ON a.id_pesquisador = p.id_pesquisador;
```

Fonte: Elaborado pelo autor

A adoção desta *view* é uma decisão de projeto estratégica para a manutenibilidade do sistema. Ela garante que a lógica de junção dos dados resida em um único local (o banco de dados). Dessa forma, o sistema atinge o desacoplamento desejado: quando um novo metadado

filtrável precisa ser adicionado (por exemplo, adicionar um campo de uma nova tabela), a manutenção é centralizada em apenas dois artefatos, sem alteração no código-fonte das camadas de serviço ou controle:

1. A *view vw_artigos_completos* é atualizada para incluir o novo campo.
2. O arquivo *metadata_config.json* é atualizado para informar ao LLM sobre a existência e o tipo do novo filtro.

O *ArtigoDAO*, portanto, realiza todas as suas consultas (tanto lexicais quanto semânticas) exclusivamente sobre esta *view*, simplificando drasticamente o código da aplicação. Ainda, vale ressaltar que existem algumas manutenções que podem impactar em alterações no *backend* da aplicação. Como no caso de uma necessidade de repassar mais informações ou modificar como as informações são passadas para o *frontend*.

6.3. Fluxo da consulta híbrida com *self-querying*

A solução foi feita, inicialmente, usando a solução do Langchain pela biblioteca self-querying, assim como descrito na monografia de João Batista (2024) que realizou um protótipo de busca semântica com filtragem automática. Porém, foi observado que os resultados eram imprecisos e limitados à uma busca semântica. Para contornar esses problemas, a solução foi desenvolvida usando o componente *query_constructor* do Langchain.

O *query_constructor* realiza apenas a separação da consulta do usuário retornando um *json* com os filtros identificados de forma estruturada. Dessa forma, o *backend* tem total controle sobre a aplicação dos filtros diretamente no banco de dados PostgreSQL, evitando inconsistências e possibilitando uma maior personalização. Como na realização de uma busca híbrida, sendo ela por termos e semântica, ou PLN como a remoção de acentos e normalização dos textos para a personalização da busca.

O núcleo da solução reside no *endpoint* de busca híbrida (*/self_query/busca_hibrida*). A implementação, detalhada no *SelfQueryService* que atua como uma fachada, coordena o processamento da consulta do usuário em linguagem natural seguindo um fluxo necessário para garantir a aplicação correta dos filtros e a combinação dos resultados.

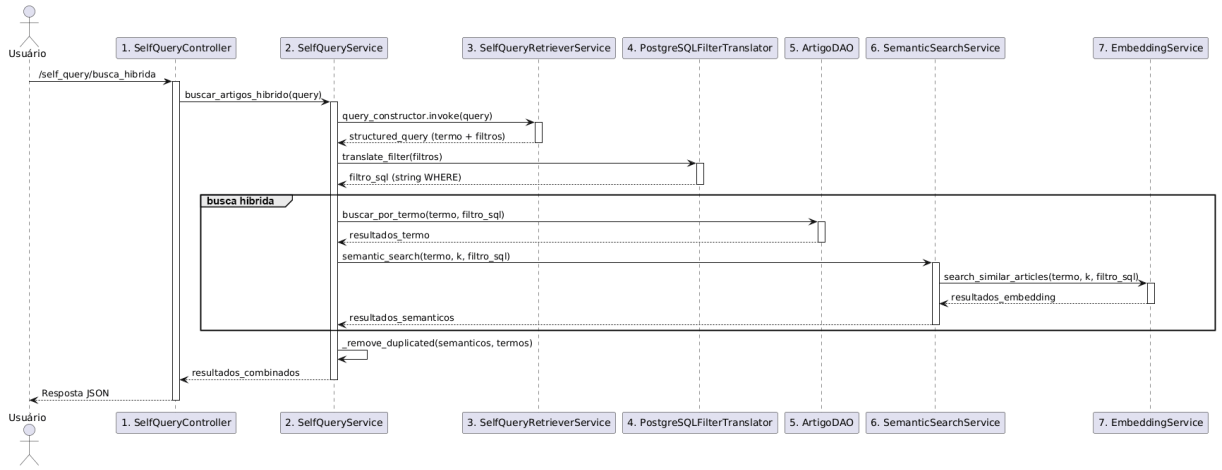
O fluxo de execução segue as seguintes etapas que são ilustradas na Figura 10:

1. **Recebimento da Consulta:** O *SelfQueryController* recebe a *query* do usuário que pode conter título de artigo e metadados descritos em linguagem natural (ex: "artigos sobre *machine learning* publicados após 2020 com qualis A1").
2. **Interpretação com *self-querying*:** A classe *SelfQueryService* atua como um orquestrador centralizando o fluxo da consulta. A requisição do usuário é encaminhada para o *SelfQueryRetrieverService*. Este serviço utiliza o *query_constructor* da biblioteca **LangChain,** que por sua vez invoca um LLM da OpenAI, especificamente o *gpt-5-mini-2025-08-07* (pelo baixo custo e precisão dos resultados em comparação com modelos mais leves ou locais). O LLM, instruído pela instrução do *query_constructor* e da configuração definida no arquivo *metadata_config.json*, interpreta a consulta e a divide em dois componentes:
    - ***Content_query*:** A parte da *query* destinada à busca (ex: "artigos sobre *machine learning*").
    - ***Filter*:** Metadados extraídos da consulta traduzidos em um filtro estruturado como se fosse um código intermediário (ex: *Comparator=<Comparator.IN: 'IN'> attribute = 'qualis' value=IN[A1]*).
3. **Tradução de Filtros para SQL:** O filtro estruturado gerado pelo LangChain é processado por um tradutor customizado, o *PostgreSQLFilterTranslator*, que pode ser substituído para se adequar a qualquer outro banco de dados. Este componente converte a estrutura de filtro do LangChain em uma cláusula *WHERE* válida para o banco de dados PostgreSQL (ex: *AND year* > 2020 *AND qualis* = 'A1').
4. **Execução da Busca Híbrida (Filtrada):** A busca é realizada em duas frentes paralelas, e o mais importante: a cláusula *WHERE* gerada na etapa anterior é aplicada em ambas as buscas, garantindo que apenas documentos que já satisfazem os metadados sejam considerados:
    - **Busca Léxica (Termos):** O *ArtigoDAO* executa uma busca textual tradicional no PostgreSQL pela *content_query*, já incluindo a cláusula *WHERE* dos filtros.
    - **Busca Semântica:** O *SemanticSearchService* realiza a busca por similaridade de vetores pela *content_query* identificada. Nesse processo, o texto é convertido para um *embedding* usando o modelo *text-embedding-3-small* de

forma igual ao que foi feito com os artigos que já foram processados. Por fim, é aplicado o filtro SQL e encaminhado ao PostgreSQL.

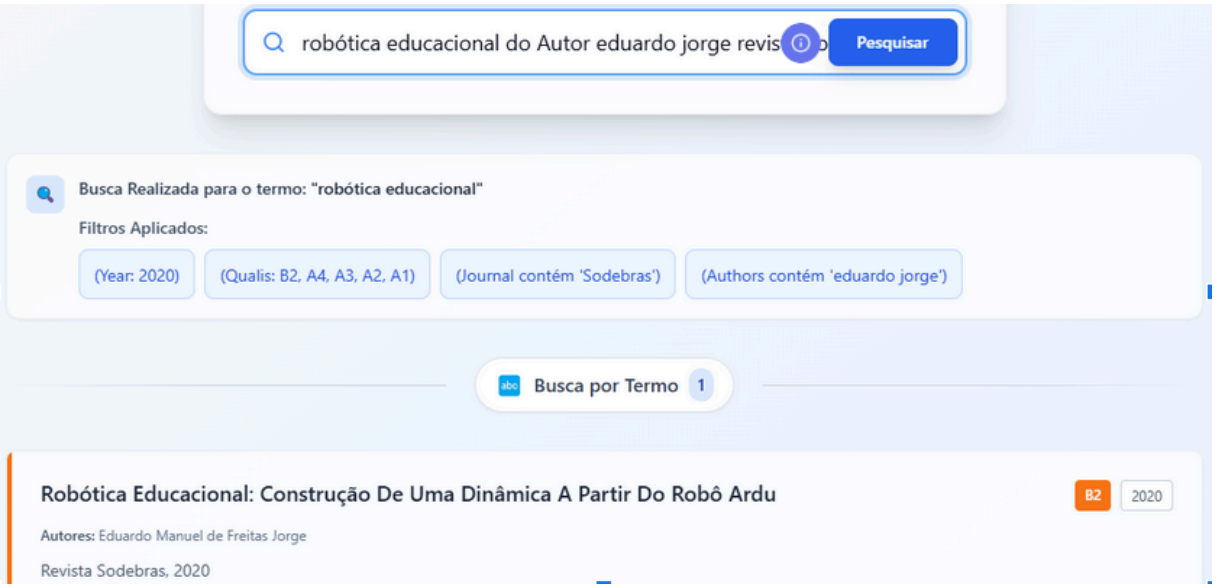
5. **Desduplicação e Retorno:** Os resultados das duas buscas (lexical e semântica) são combinados. Um método de desduplicação (*_remove_duplicated*) é executado para garantir que um mesmo artigo não apareça em ambas as listas, priorizando sua aparição na lista de termos (correspondência exata). Os resultados consolidados são retornados ao *frontend*.
6. **Exibição:** O resultado, presente na Figura 11, mostra ao usuário os filtros identificados na consulta realizada e o conteúdo da pesquisa. Na listagem dos artigos existe a separação visual de ambas as buscas realizadas (Semântica e lexical).

Figura 10: Diagrama de sequência mostrando os métodos e classes envolvidas no fluxo de busca híbrida com *self-querying*.



Fonte: Elaborada pelo Autor

Figura 11: Exemplo de uma consulta realizada no sistema que inclui múltiplos filtros, como autor, nome de revista, qualis e ano de publicação. Abaixo é exibido os filtros aplicados e o conteúdo pesquisado.



Fonte: Elaborada pelo Autor

Este mecanismo, ao traduzir a linguagem natural em filtros SQL antes da execução da busca, otimiza a performance ao deixar essa busca ser realizada no PostgreSQL ao invés de ser feita pela biblioteca self-querying do LangChain. Ainda, ela válida a arquitetura de desacoplamento, pois a lógica de filtragem reside nas instruções do LLM, guiadas pelo *metadata_config.json*, e não sendo realizada diretamente pelo código-fonte de uma ou mais classes.

6.4. Módulo de Filtragem

A filtragem do sistema ocorre a partir de instruções que são passadas para o LLM, que inclui um arquivo de configuração em *json* chamado: *metadata_config*. Na Figura 12, temos um fragmento do arquivo que inclui os seguintes campos:

- *Name* - O nome do metadado que também precisa ser igual ao nome da coluna presente no retorno da consulta do banco de dados.
- *Description* - A descrição do que é para o LLM separar na consulta do usuário. Note que ela possui também detalhes de operadores que ela deve usar e alguns exemplos para melhorar a precisão.
- *Type* - tipo do dado que será passado, em caso de valores numéricos ou *strings*.
- *Filter_description* - Descrição do que é o filtro, esse campo será passado apenas para o *frontend* com o objetivo de instruir o usuário sobre os filtros existentes. Essa lista de

filtros é mostrada em formato de *tooltip* para o usuário ao lado da barra de busca (Figura 12).

Figura 12: Fragmento do arquivo json contendo descrições dos filtros e a visualização deles no *frontend.*

```
{
"metadata_fields": [
    {
      "name": "year",
      "description": "Ano de publicação do artigo no
formato numérico (ex.: 2023). Use para filtrar artigos
por ano específico ou por intervalo de anos. Make sure
that you filter year only using = (eq), > (gt), <
(lt), >= (gte), <= (lte) operators.",
      "type": "integer",
      "filter_description": "Ano de publicação do
artigo, pode ser usado para filtrar artigos por ano
específico ou por intervalo de anos"
    }]
}
```

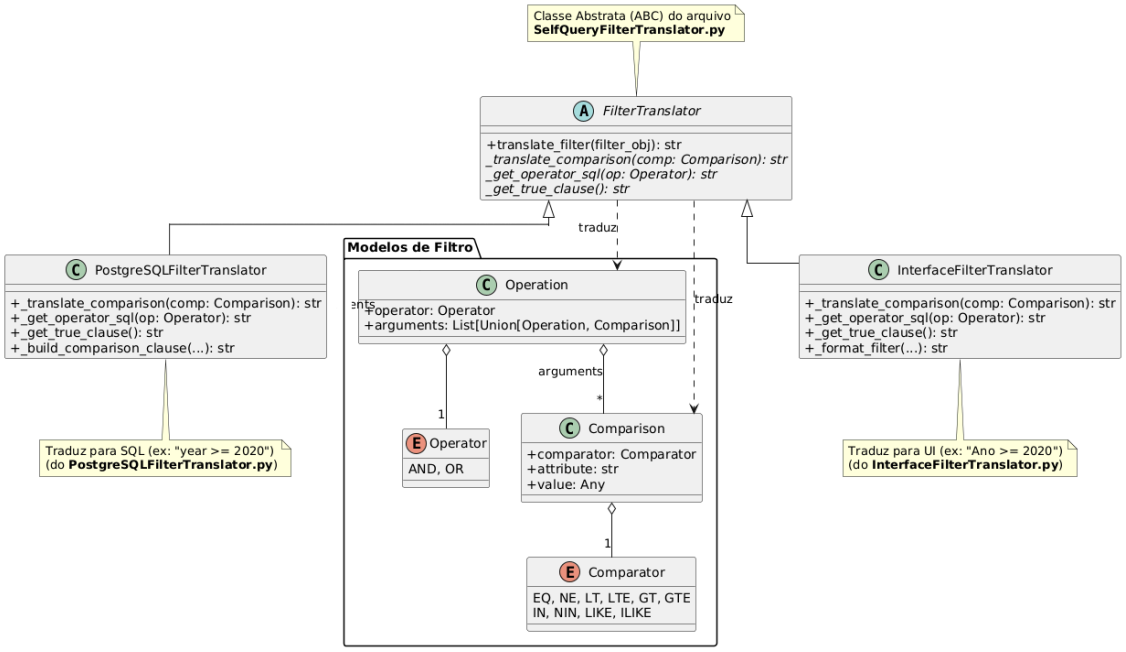
Fonte: Elaborada pelo Autor

Depois de encaminhar o *metadata_config* para a LLM através da biblioteca *query_constructor* do Langchain, ela retorna um json que traz o conteúdo identificado da consulta do usuário (*content_query* e *filter*). Porém, o filtro gerado ainda está escrito em um código intermediário que é tratado internamente por outras classes que fazem a adaptação do retorno para a linguagem do banco de dados alvo, que no caso é o PostgreSQL. Dessa forma, foi elaborada uma estrutura seguindo o padrão de projeto *Strategy*.

No diagrama de classe, ilustrado na Figura 13, existe uma classe abstrata que define métodos obrigatórios de conversão dos operadores e comparadores que vem da *query* estruturada, descritos no *enum* "*operator*" e "*comparator*", respectivamente. Dessa forma, o filtro que vem da LLM passa pelo *PostgreSQLFilterTranslator* que vai converter a *query*

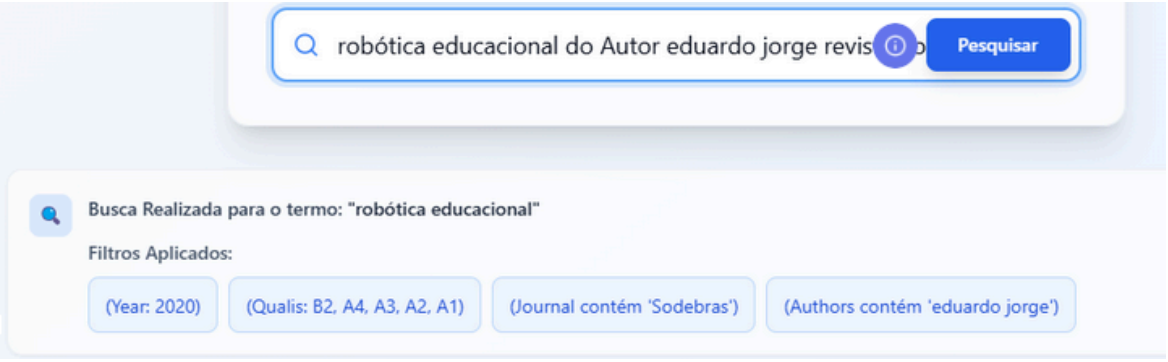
intermediária para a cláusula *WHERE* do banco de dados. Em seguida, o mesmo filtro da LLM passa pelo *InterfaceTranslator* para mostrar os filtros identificados de forma visual na interface do *frontend*, que pode ser visto na Figura 14. A principal vantagem dessa abordagem está na separação que o LLM faz ao extrair os dados do arquivo *metadata_config.json*. Isso elimina a necessidade de mencionar o nome da coluna diretamente no código-fonte.

Figura 13: Diagrama de classe do módulo de filtragem que realiza a conversão do filtro gerado do LLM para a cláusula *WHERE* e filtro de interface.



Fonte: Elaborada pelo autor.

Figura 14: Filtro gerado pela classe InterfaceFilterTranslator que aparece abaixo da barra de busca.



Fonte: Elaborada pelo Autor

7. RESULTADOS

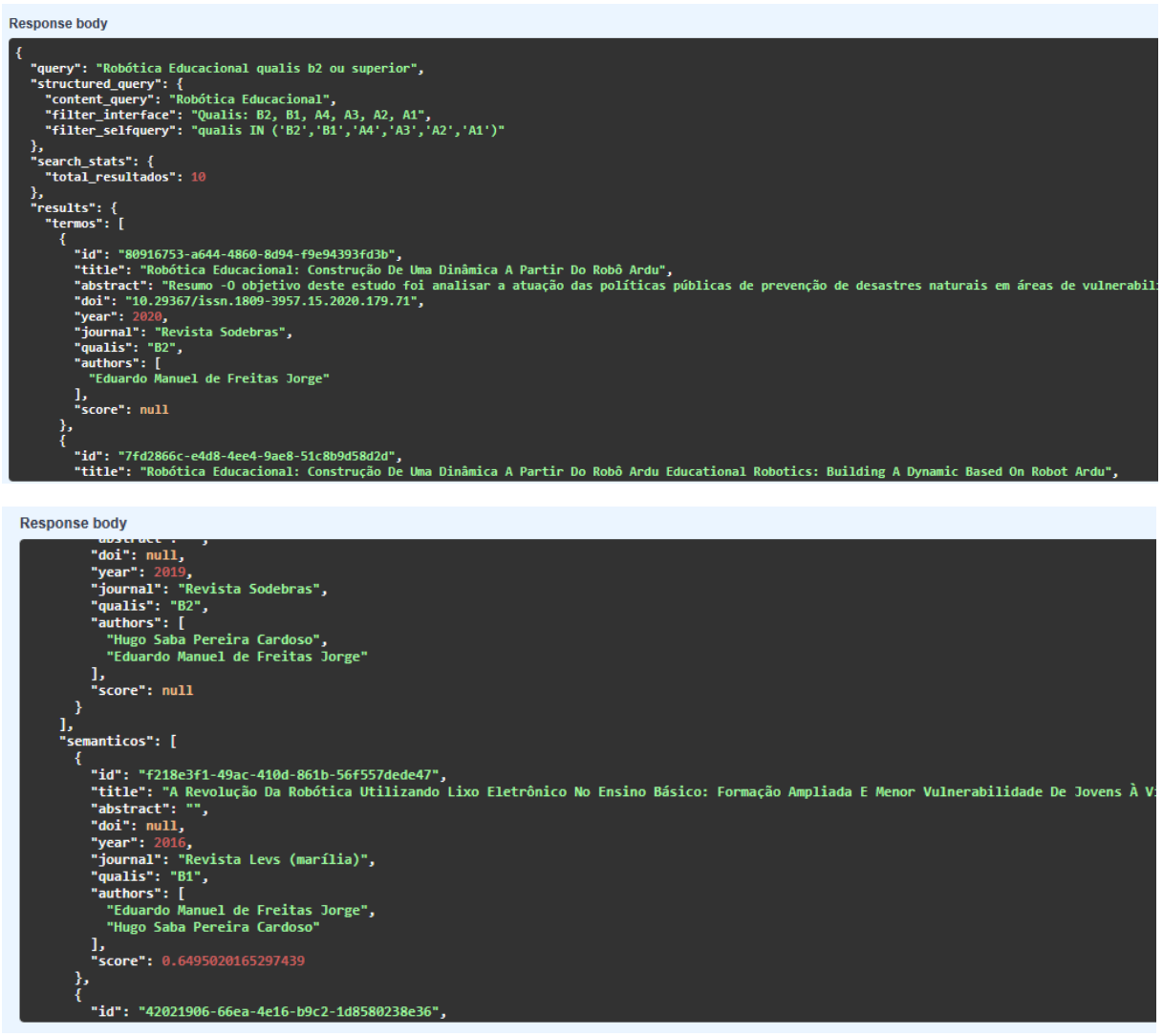
Nesta seção, são apresentados os resultados obtidos com o artefato computacional, demonstrando a aplicação da arquitetura de busca híbrida com *self-querying*. De início, será demonstrado alguns exemplos de consultas simples e compostas para mostrar a capacidade do sistema de interpretar consultas em linguagem natural. Em seguida, serão feitas as análises qualitativas e quantitativas de ambos os sistemas, o *baseline* e o que usa *self-querying* para visualizar de modo geral os aspectos de manutenibilidade e evolução do sistema. Por fim, será feito os dois testes de implementação para análise comparativa dos dois sistemas.

O sistema é capaz de realizar a geração de filtros simples e compostos a partir de consultas em linguagem natural. A Figura 15 exibe a resposta da API para a consulta: "Robótica Educacional qualis B2 ou superior". Por meio da análise da resposta da API, é notável que a abordagem de self-querying apresentou resultados promissores, demonstrando a capacidade do sistema em interpretar a consulta complexa e segmentá-la adequadamente em dois componentes essenciais:

- Alvo da Pesquisa (Consulta de Conteúdo): O termo "Robótica Educacional" foi identificado como o núcleo semântico da busca. Este valor é utilizado para realizar tanto a busca textual tradicional (por meio de *ILIKE* no banco de dados) quanto a busca por similaridade de vetores (*embeddings*) usando o *PGVector* do PostgreSQL.
- Filtro Estruturado: A filtragem dinâmica foi capaz de separar os filtros e traduzir usando as classes *InterfaceFilterTranslator* e do *PostgreSQLFilterTraslator* que realizam a formatação do *filter_interface* e do *filter_selfquery*, respectivamente.

O resultado final exibe uma separação em busca por termos e a busca semântica, tendo uma quantidade somada de 10 artigos retornados.

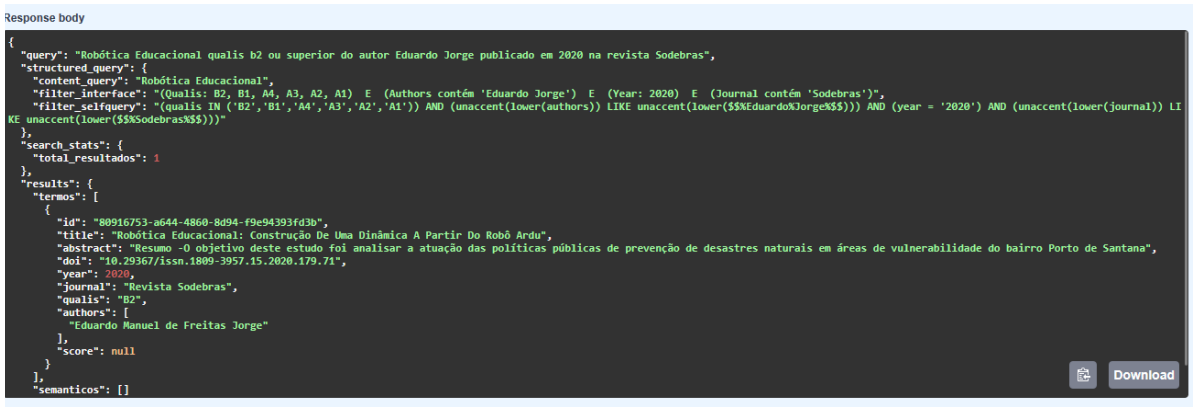
Figura 15 - Resposta da API para uma *query* simples: “Robótica Educacional qualis b2 ou superior”. Apresentando resultados da busca híbrida



Fonte: Elaborada pelo autor

Já na Figura 16, temos uma *query* com múltiplos filtros dependentes: “Robótica Educacional qualis b2 ou superior do autor Eduardo Jorge publicado em 2020 na revista Sodebras”. Como resultado, temos apenas 1 artigo do sistema que combinava com todos os filtros e que é mostrado em resultados na busca por termos. Nesse caso, a busca semântica não retorna nada, mesmo que os artigos cumpram com o requisito do filtro. O artigo ainda pode não ter um *score* suficiente para ser mostrado, já que nesse sistema foi estabelecido de forma empírica um valor mínimo de 0.4 de similaridade de cosseno.

Figura 16 - Resposta da API para uma *query* mais complexa: “Robótica Educacional qualis b2 ou superior do autor Eduardo Jorge publicado em 2020 na revista Sodebras”.



Fonte: Elaborada pelo autor

7.1. Análise dos sistemas

Para analisar os aspectos de manutenibilidade e evolução do código em ambos os sistemas será realizada a contabilização das métricas apenas nos métodos e das classes envolvidas no processo de filtragem dos dados, pois a natureza de ambos os sistemas é diferente em relação ao tamanho. No sistema do SIMCC o arquivo contém buscas de múltiplos *endpoints* para diferentes finalidades para busca de pesquisadores, artigos, patentes etc resultando em um arquivo de 1171 linhas, enquanto o sistema *self-query* contém apenas os relacionados a busca por termos e semântica com apenas 151 linhas. Ainda, o *endpoint* do sistema tradicional trata apenas da busca por termos, enquanto o sistema de *self-query* realiza uma busca híbrida pela combinação dos resultados.

7.1.1. Análise dos sistema SIMCC

O fluxo da busca por termos e filtragem no sistema do SIMCC realiza no método *lists_bibliographic_production_article_researcher_db* a orquestração da busca, os detalhes da implementação estão no Apêndice A. Esse arquivo, levando em consideração apenas o método que orquestra a busca, realiza desde a formatação de alguns filtros quanto a paginação, formulação do código SQL e a chamada para o banco de dados, levando à um alto acoplamento e baixa coesão.

Como pode ser visualizado no Quadro 7 e na Figura 17, que foram elaborados a partir do código-fonte presente no no Apêndice A, o método que orquestra a busca do sistema tradicional: *lists_bibliographic_production_article_researcher_db* possui uma complexidade ciclomática 6, e os métodos relacionados à montagem dos filtros possui uma complexidade 8

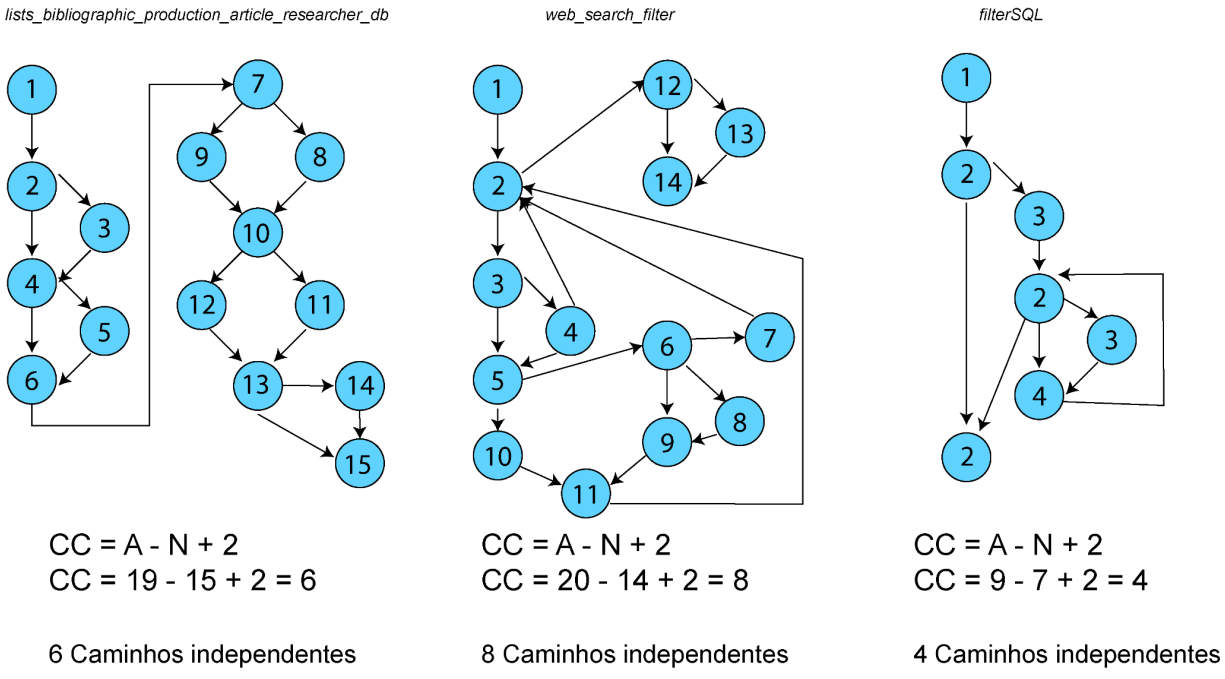
e 4. Esse valor representa a quantidade de caminhos independentes existentes nos métodos e que precisam ser testados para garantir a qualidade do *software*. Vale ressaltar que esses métodos são os mais impactados em caso de manutenções para a alteração de filtros adicionados ou para modificar o tratamento deles.

Quadro 7: Contabilização do número de linhas de código e complexidade ciclomática por método do fluxo de filtragem na busca

Arquivo/Classe	Método	Linhas de Código	Complexidade Ciclomática
termFlowSQL	*lists_bibliographic_production_article_researcher_db*	173	6
util	*web_search_filter*	30	8
	filterSQL	22	4

Fonte: Elaborado pelo autor

Figura 17: Visualização grafo e cálculo da Complexidade Ciclomática dos métodos lists_bibliographic_production_article_researcher_db, *websearch* e *filterSQL*.



Fonte: Elaborado pelo autor

A seguir, em relação aos aspectos de acoplamento e coesão dos métodos mencionados no Quadro 7, visto que esse sistema não está orientado a objeto, e com base nos códigos do Apêndice A, podemos fazer as seguintes análises:

- *lists_bibliographic_production_article_researcher_db* possui muitas diferentes responsabilidades como: montar os filtros manualmente como o qualis, requisitar os filtros de outros métodos (como o *web_search_filter* e o *filterSQL)*, adicionar paginação, montar as cláusulas SQL definindo colunas retornadas fazendo *JOINS,* solicitar a execução para a classe *sgbdSQL* e, por fim, o tratamento dos dados retornados do banco antes de devolver ao *endpoint*. Sendo assim, possui baixa coesão e um alto acoplamento em relação aos arquivos *util*, *sgbdSQL* e ao retorno esperado pelo *frontend*.
- *web_search_filter* e *filterSQL* possuem alta coesão por serem métodos responsáveis unicamente por montar os filtros booleanos e por correspondência exata, respectivamente. Eles possuem baixo acoplamento por não dependerem de nenhuma outra classe ou método do sistema.

7.1.2. Análise dos sistema *self-querying*

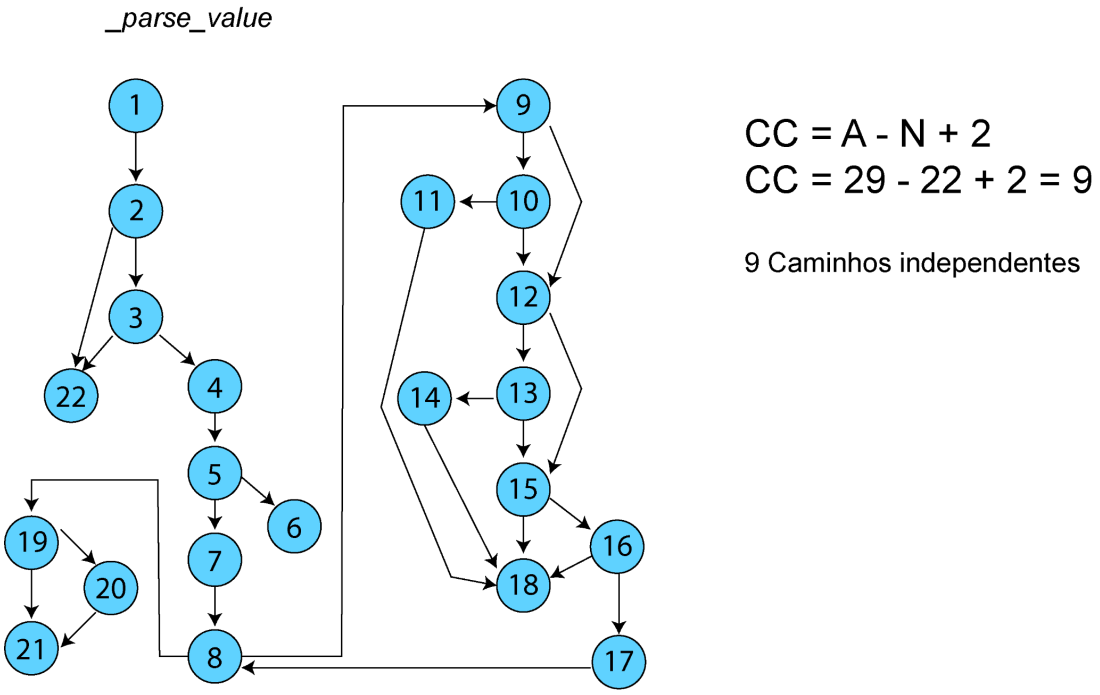
O fluxo da busca híbrido do sistema de *self-querying* realiza a orquestração da busca no método *buscar_artigos_hibrido,* os detalhes da implementação podem ser visualizados no Apêndice B. De modo geral, ele deixa para outras classes abstrairem a lógica da construção de filtros, a montagem do SQL e a busca no banco de dados. Dessa forma, existe um maior volume de classes/arquivos sendo usados nesse fluxo, como pode ser visto no Quadro 8. Vale pontuar que nos métodos existentes o *_parse_value* se destaca com 9 de complexidade ciclomática sendo um ponto de atenção para o sistema desenvolvido. Esse método, que sua complexidade e fluxo podem ser visualizados na Figura 18, é impactado por mudanças na formatação recebida pelo json do LLM. Sendo assim, em caso de manutenção por eventual atualização da biblioteca do Langchain, esse método pode ser mais complexo de realizar mudanças.

Quadro 8: Contabilização do número de linhas de código e complexidade ciclomática por método do fluxo de filtragem na busca para o sistema de *self-query.*

Arquivo/Classe	Método	Linhas de código	Complexidade Ciclomática
selfquery_service	buscar_artigos_hibrido	62	3
SelfQueryFilterTranslator	*get_where_clause*	10	3
FilterParser	*parse_filter_string*	11	2
	_parse_operation_string	15	1
	_parse_arguments_list	25	1
	_parse_single_argument	10	1
	_parse_nested_operation	7	1
	_parse_comparison_string	31	1
	_parse_value	32	9
PostgreSQLFilterTranslator	*_get_operator_sql*	4	1
	_translate_comparison	3	1
	_build_comparison_clause	16	1
	_build_in_clause	4	1
	_build_not_in_clause	4	1
	_build_like_clause	1	1
	_build_ilike_clause	1	1
	_quote_value	17	2

Fonte: Elaborado pelo autor

Figura 18: Visualização grafo e cálculo da Complexidade Ciclomática do método *_parse_value*.



Fonte: Elaborado pelo autor

Agora, em relação aos aspectos de acoplamento e coesão das classes podemos ter as seguintes análises ao analisar o código do Apêndice B:

- *SelfQueryService* está fazendo o papel de uma fachada para o *controller*, logo sua coesão acaba sendo mediada por precisar orquestrar diferentes sistemas de busca, como requisitar separação da consulta do usuário ou unir o retorno das buscas para o frontend. Ela depende de classes como *SemanticSearchService* e *ArtigoDAO* para busca híbrida e do *PostgreSQLFilterTranslator* e *InterfaceFilterTranslator* para formatação do SQL e do retorno desses filtros para o *frontend*, respectivamente.

- *PostgreSQLFilterTranslator* faz exclusivamente a implementação da classe abstrata para converter o filtro para a cláusula *WHERE*, tendo assim uma alta coesão. Ele depende apenas da classe abstrata e de componentes como *enumerators* para o processamento dos operadores e comparadores.

- *FilterParser* mesmo que possua a única responsabilidade de fazer a conversão direta vinda do retorno da LLM, a coesão pode ser dita como frágil por lidar com diferentes formas de expressões regulares para formatar o filtro que vai ser passado para o *PostgreSQLFilterTranslator*. A classe então tem forte acoplamento em relação à

implementação do *LangChain*, como era o esperado, pelo sistema desacoplar a dependência dos filtros no código e extrair para o uso usando apenas a LLM.

- *Get_where_clause* é citado no Quadro 8, mas é um método que não está dentro de um objeto. Ele depende do *PostgreSQLFilterTranslator* por ser o tradutor padrão, em caso de não declarar nenhum, e do *FilterParser* pois o método faz uma simplificação da chamada para o *SelfQueryService*. Sua coesão é média por fazer quase que um papel de fachada para outra classe. Ainda, possui um maior acoplamento em relação a essas classes mencionadas anteriormente.

7.2. Testes de implementação em ambos os sistemas

Nesta seção serão realizados os testes de implementação de ambos os cenários para os dois sistemas com o objetivo de verificar o impacto da manutenibilidade e evolução de ambos os código-fonte. Vale pontuar que essas modificações em ambos os códigos foi realizada pelo próprio pesquisador. Nesses cenários, vai ser desconsiderado o processo de ETL necessário para ambos os sistemas para adição de novos filtros. Dessa forma, vai ser contabilizado apenas as alterações em código, arquivos de configuração e códigos SQL relacionados ao *SELECT*.

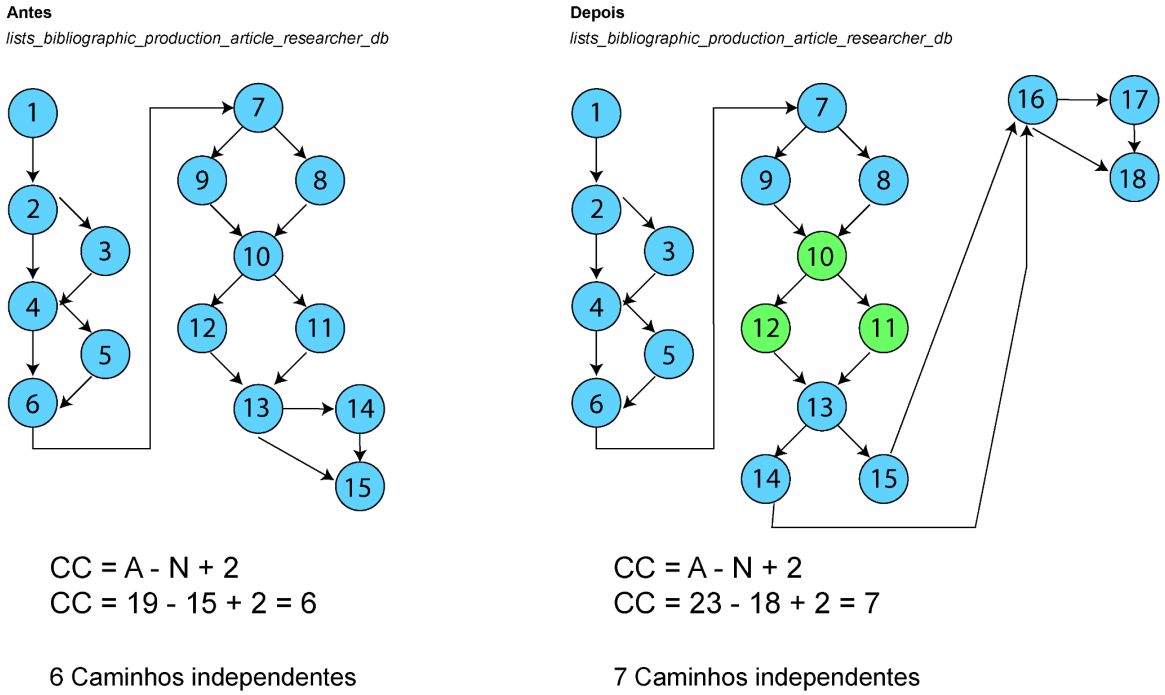
7.2.1. Cenário A - Filtro “tipo de publicação”

Primeiro, será feita a implementação do Cenário A, que é a adição do filtro “Tipo de publicação”. Esse filtro pode vir sozinho como “Apenas Artigos de conferência” ou pode vir em formato de lista para filtrar múltiplos tipos. Ainda, ele não será retornado ao *frontend*, sendo apenas um recurso de filtragem interno.

Para o sistema do SIMCC, nota-se no Quadro 9 que à medida que novos filtros são adicionados, como o caso do tipo de publicação, ao fluxo de busca isso ocasiona alterações em duas partes centrais da busca, o *endpoint* da aplicação e no método que orquestra a busca. No método *bibliographic_production_researcher,* que é o *endpoint* do sistema, ele precisa fazer a captura dos novos filtros que são passados pelo *frontend*. Já no método *lists_bibliographic_production_article_researcher_db*, por ele centralizar múltiplas responsabilidades, é preciso fazer a adição de um condicional para detecção desse novo filtro. Dessa forma, sempre que um filtro é adicionado, a complexidade ciclomática é aumentada como pode ser visualizado na Figura 19. Ainda, o método possui duplicidade na declaração do sql, precisando ser adicionado o retorno “*tipo_pub*”e o filtro “*filter_tipo_pub*” em ambos

os SQL. Por fim, é necessário realizar uma possível criação de um novo caso de teste para cobrir esse novo caminho e também fazer a implantação novamente do sistema.

Figura 19: Visualização grafo e cálculo da Complexidade Ciclomática do método *list_bibliographic_production_article_researcher_db* atual e com a adição do tipo de publicação.



Fonte: Elaborada pelo autor.

Quadro 9: Visualização das linhas de código adicionadas e a mudança na complexidade ciclomática após a adição do tipo de publicação para o Sistema SIMCC.

Método/Arquivo	Linhas modificadas	Complexidade Ciclomática	Alteração
Método: *lists_bibliographic_production_article_researcher_db*	10 novas linhas	6 → 7	Adicionado campo na declaração do método: *"tipo_pub: str = None"* Verificação da existência do filtro *"if tipo_pub:* *filter_tipo_pub = util.filterSQL(tipo_pub, ";", "or", "b.type")* *else:* *filter_tipo_pub = str()"* Adição do retorno e do filtro na consulta de ambos os códigos sql do método *"tipo_pub"* *"{filter_tipo_pub}"*
Método: *bibliographic_production_researcher*	2 novas linhas	1 → 1	Captura de novo no *endpoint* Adição do novo parâmetro alterado na declaração do método *lists_bibliographic_production_article_researcher_db.*

			“tipo_pub = request.args.get("tipo_pub") *tipo_pub=tipo_pub”*

Fonte: Elaborado pelo autor.

No sistema do *self-query*, pode-se observar pelo Quadro 10 que, para a adição de filtros que não retornam para o *frontend*, o esforço se concentra apenas em modificar o *json* que instrui o LLM. O *endpoint* da aplicação permanece inalterado pois o filtro é processado pela LLM e formatado pelas classes auxiliares *PostgreSQLFilterTranslator* que também não são modificadas. Outro local de mudança está na alteração da *view* que é configurada no PostgreSQL, sendo assim, não há a necessidade de implantação do código novo.

Quadro 10: Visualização das linhas de código adicionadas e a mudança na complexidade ciclomática após a adição do tipo de publicação para o Sistema *self-query*.

Arquivo / SQL / Métodos	Linhas modificadas	Complexidade Ciclomática	Alteração
Arquivo de configuração: *metadata_config.json*	6 novas linhas	NA	Adicionado novo item na lista do json de filtros: *{* *"name": "tipo_publicacao",* *"description": "Tipo de publicação do artigo. Valores possíveis: ['Artigo de periódico', 'Capítulo de livro', 'Dissertação', 'Tese']. Use only IN for this metadata field. Few shots learning, Example 1: Query: 'Artigos sobre computação que são capítulos de livros', the content_query is 'computação' and the filter must be 'Comparison(comparator=<Comparator.IN: 'in'> attribute='tipo_publicacao' value=['Capítulo de livro'])'.",* *"type": "string",* *"filter_description": ""* *},*
SQL view PostgreSQL	1 nova linha	NA	Adição da coluna retornada do banco *a.tipo_publicacao,*

Fonte: Elaborado pelo autor.

7.2.2. Cenário B - Filtro “quantidade de citações”

Em segundo, será feita a implementação do Cenário B, que é a adição do filtro “número de citações”. Esse filtro terá impacto no *frontend* da aplicação, sendo assim uma implementação possivelmente mais complexa.

No sistema do SIMCC, é perceptível no Quadro 11 que os impactos são iguais aos filtros mais simplificados do Cenário A. A única redução que houve foi em relação ao número de linhas que está menor. Isso ocorre porque o sistema do SIMCC já realiza o retorno de

múltiplas informações que são implementadas no *frontend* que está em produção, caso contrário seriam 11 linhas adicionadas ao invés de 8. Assim como o primeiro cenário, a adição de mais filtros acaba aumentando a complexidade ciclomática de forma igual ao que foi mostrado na Figura 19, que a parte verde reflete a existência dos condicionais adicionados para o novo filtro.

Por meio desses dois cenários é perceptível que a adição de novos filtros no sistema tradicional como o SIMCC escala cada vez mais a complexidade do código e exige sempre a implantação novamente do sistema.

Quadro 11: Visualização das linhas de código adicionadas e a mudança na complexidade ciclomática após a adição do filtro de citações para o Sistema SIMC.

Método/Arquivo	Linhas modificadas	Complexidade Ciclomática	Alteração
Método: *lists_bibliographic_production_article_researcher_db*	8 novas linhas	6 → 7	Adicionado campo na declaração do método: "min_citacoes: int = None," Verificação da existência do filtro *""if min_citacoes:* *filter_citacoes= f"""AND opa.citations_count >= {min_citacoes}"""* *else:* *filter_citacoes= str()""* Adição do filtro na consulta de ambos os códigos sql do método *"{filter_citacoes}"*
Método: *bibliographic_production_researcher*	2 novas linhas	1 → 1	Captura de novo no *endpoint* Adição do novo parâmetro alterado na declaração do método *lists_bibliographic_production_article_researcher_db.* *"min_citacoes = request.args.get("min_citacoes", 0, int)* *min_citacoes=min_citacoes"*

Fonte: Elaborado pelo autor.

Com a implementação do cenário B, o sistema do *self-query* acaba precisando também realizar mudanças no código fonte, como pode ser observado pelo Quadro 12. À medida que novos filtros são retornados para a interface, existe a necessidade de mapear o retorno do banco para passar para o *frontend*. Nesse caso, as classes *ArtigoBuscaDTO* e *ArtigoDTOMapper* precisam ser atualizadas pois impacta diretamente nesse processo de mapear o resultado do banco. A complexidade de ambos permanece inalterada, pois não existe adição de lógica condicional no mapeamento, realizando apenas a conversão de um dicionário para uma entidade da classe *ArtigoBuscaDTO*.

Da mesma forma que o cenário A, para a adição desses novos filtros, é preciso fazer a adição do novo metadado no arquivo de configuração e a adição dele no retorno da *view*.

Quadro 12: Visualização das linhas de código adicionadas e a mudança na complexidade ciclomática após a adição do filtro de citações para o Sistema *self-query*.

Arquivo / SQL / Métodos	Linhas modificadas	Complexidade Ciclomática	Alteração
Arquivo de configuração: *metadata_config.json*	6 novas linhas	NA	Adicionado novo item na lista do json de filtros: *{* *"name": "citacoes",* *"description": "Número de citações que o artigo recebeu, expresso como um valor numérico inteiro. Use para filtrar artigos com base na quantidade de citações, aplicando operadores como = >= (gte), <= (lte). Few shots learning, Example 1: Query: 'Artigos sobre computação com mais de 100 citações', the content_query is 'computação' and the filter must be 'Comparison(comparator=<Comparator.GTE: 'gte'> attribute='citacoes' value=100)'. ",* *"type": "integer",* *"filter_description": "Número de citações do artigo, pode ser usado para filtrar artigos com base na quantidade de citações"* *},*
ArtigoBuscaDTO	3 novas linhas	1 → 1	Adição do atributo novo *"citacoes: Optional[int] = None"* Adição do atributo novo no construtor *"citacoes: Optional[int] = None, citacoes=citacoes"*
ArtigoDTOMapper, método to_artigo_busca_dto_from_sql_rows	2 novas linhas	2 → 2	Captura do atributo a partir da linha retornada do banco *"citacoes = linha.get('citacoes', None)"* Passagem do parâmetro novo adicionado no *ArtigoBuscaDTO* *"citacoes=citacoes,"*
SQL view PostgreSQL	1 nova linha	NA	Adição da coluna retornada do banco *a.citacoes,*

Fonte: Elaborado pelo autor.

8. CONSIDERAÇÕES FINAIS

Este trabalho buscou investigar de que forma a implementação de uma arquitetura de *self-querying* com LLM impactaria nos aspectos de manutenibilidade e evolução de sistemas de busca acadêmicos ao desacoplar a lógica de filtragem do código fonte. A motivação central se originou da lacuna identificada no trabalho de Batista (2024), que apontou as limitações encontradas na integração dos sistemas de busca híbrida com a filtragem dinâmica pela implementação direta da biblioteca *Self Query Retriever* do Langchain. Assim, o artefato computacional desenvolvido buscou resolver essa problemática utilizando um LLM não apenas para a busca semântica, mas para realizar a interpretação e separação da consulta do usuário nos componentes de conteúdo de pesquisa e em filtros de metadados.

A adoção da metodologia DSR se demonstrou adequada na condução dessa pesquisa. Ela serviu como um guia para as etapas de conscientização do problema, identificação de tecnologias que foram mais tarde empregadas na solução, na definição de como seria o artefato computacional e o modo de validação dele. Dessa forma, foi possível realizar uma formalização dos conhecimentos gerados ao longo desse processo de elaboração e avaliação da pesquisa, ainda, identificando também limitações da solução encontrada.

As implicações da pesquisa sugerem uma possível mudança no cenário de desenvolvimento dos sistemas de busca acadêmicos. Os sistemas tradicionais, como o SIMCC que foi analisado, necessitam de intervenções manuais no código fonte trazendo maior complexidade ao código nos cenários de evolução. A arquitetura *self-query* proposta contribui para a mudança nessa realidade ao promover o desacoplamento da lógica de filtragem. Ao transferir essa lógica, o sistema passa a ser mais adaptável permitindo que o sistema sofra manutenções minimizando as intervenções no código fonte, evitando a necessidade de realizar novas implantações do mesmo sistema.

Por fim, o processo de avaliação do artefato também revelou alguns pontos de atenção que devem ser considerados na utilização dessa arquitetura de *self-query*. O acoplamento do filtro deixou de ser em relação ao código-fonte da aplicação e foi deslocado para um acoplamento com a biblioteca externa do Langchain e o formato de saída do LLM que é de um serviço de nuvem, perdendo a independência do sistema em relação a tecnologias externas.

8.1. Limitações do trabalho

Apesar dos resultados positivos apresentados em relação à manutenibilidade, o desenvolvimento e validação do artefato pela metodologia DSR permitiu identificar as seguintes limitações:

- Viés de validação: A segunda fase da avaliação, relacionada a simulação de manutenção do código de ambos os sistemas, foi realizada pelo próprio pesquisador. Dessa forma, embora os cenários tenham sido criados de forma objetiva, a familiaridade com o código desenvolvido pode influenciar a percepção de esforço na implementação das mudanças.
- Dependência de bibliotecas: Mesmo que sendo algo esperado, a solução proposta deixa de ser acoplada ao código fonte e começa a ter um acoplamento considerável em relação a biblioteca do Langchain e ao formato de saída do LLM. As alterações nesses componentes externos podem impactar no módulo de filtragem, mais especificamente o *FilterParser*, necessitando de atualizações do código fonte que trabalha com expressões regulares.
- Latência e dependência de nuvem: Devido a necessidade de fazer uma requisição a uma API na nuvem sempre que existe uma consulta, isso traz para o sistema uma dependência dessa API, uma latência pelo processamento nesse fluxo da busca e um custo atrelado ao modelo usado. Nesse fluxo, são necessárias 2 chamadas para o OpenAI, a primeira, para realizar a identificação dos componentes da busca e a segunda para a geração do *embedding* da consulta, sendo assim, influenciada diretamente pela velocidade do modelo.

8.2. Trabalhos Futuros

A partir das conclusões e limitações identificadas, sugere-se a seguinte continuidade para esta pesquisa:

- Testes de usabilidade e experiencia de usuário: o protótipo validou a viabilidade técnica e o impacto da manutenibilidade da arquitetura que foi proposta. Uma etapa essencial seria a realização de uma análise de como o usuário interage com esse método diferente de interação com sistemas de busca. Pois surgem questões de

usabilidade como: O usuário consegue formular consultas complexas? A ausência de botões e *dropdowns* de filtro prejudica ou facilita a descoberta e filtragem de artigos?

- Robustecimento de testes e refatoração do *FilterParser*: a análise de manutenibilidade apontou a classe *FilterParser* como um ponto de acoplamento em relação ao modelo de LLM e a biblioteca LangChain. O refatoramento e a realização de testes com as respostas do LLM garantiria maior confiabilidade ao sistema.
- Criação de uma interface administrador para mudar os filtros sem necessidade de alterar arquivos do *backend*: essa abordagem necessitaria de um outra forma de selecionar as colunas que são passadas para o *frontend* para evitar ter que implantar novamente o sistema. A implementação atual faz o uso de *views* que pode ser um ponto de partida para alterar o SQL por meio do painel administrador. Ainda, a implementação atual faz com que a pessoa que altera o *metadata_config* precise conhecer como utilizar as cláusulas *WHERE* para maior precisão, como o *IN, LIKE, EQUAL* etc
- Análise do impacto da arquitetura proposta de forma mais abrangente: como identificado nas limitações do trabalho, é necessário uma validação mais ampla para comprovação dos benefícios dessa arquitetura, possivelmente trabalhando com diferentes desenvolvedores e analisando impactos tanto no *backend* quanto o *frontend* de um sistema de busca acadêmico com arquitetura *self-query*.
- Dependência da nuvem para realização das buscas: como foi relatado nas considerações finais, a arquitetura depende da utilização de serviços em nuvem pois não é possível obter resultados satisfatórios na utilização de modelos locais ou menores na nuvem. Assim, um possível aprimoramento do sistema seria a utilização de modelos locais para interpretação da *query* para evitar dependências externas e possivelmente reduzir a latência da fila dos modelos menores da OPENAI.

REFERÊNCIAS

BARNARD, Joel. **What is Embedding? | IBM**. Disponível em: <https://www.ibm.com/think/topics/embedding>. Acesso em: 22 jun. 2025.

BATISTA, JOÃO. **MECANISMO DE BUSCA SEMÂNTICA BASEADO EM WORD EMBEDDINGS EM DADOS DO CURRÍCULO LATTES, PROGRAMAS DE PÓS GRADUAÇÃO E GRUPOS DE PESQUISA**. SALVADOR: UNIVERSIDADE DO ESTADO DA BAHIA, 2024.

CHANG, Yu-Hsuan *et al.* Interactive Healthcare Robot Using Attention-Based Question-Answer Retrieval and Medical Entity Extraction Models. **IEEE Journal of Biomedical and Health Informatics**, v. 27, n. 12, p. 6039–6050, dez. 2023.

GADESHA, Vrunda. **What Is Prompt Engineering? | IBM**. Disponível em: <https://www.ibm.com/think/topics/prompt-engineering>. Acesso em: 22 jun. 2025.

KHAN, Tajmir; RASHID, Umer; KHAN, Abdur Rehman. End-to-end pseudo relevance feedback based vertical web search queries recommendation. **Multimedia Tools and Applications**, v. 83, n. 31, p. 75995–76033, 21 fev. 2024.

KITCHENHAM, Barbara. Procedures for Performing Systematic Reviews. *[S.d.]*.

LACERDA, Daniel Pacheco *et al.* Design Science Research: método de pesquisa para a engenharia de produção. **Gestão & Produção**, v. 20, p. 741–761, 2013.

LangChain: How to do "self-querying" retrieval. Disponível em: <https://python.langchain.com/docs/how_to/self_query/>. Acesso em: 22 jun. 2025.

LI, Xiaoxi *et al.* From Matching to Generation: A Survey on Generative Information Retrieval. **ACM Transactions on Information Systems**, v. 0, n. ja, *[S.d.]*.

MANNING, Christopher; RAGHAVAN, Prabhakar; SCHUETZE, Hinrich. Introduction to Information Retrieval. 2009.

PRESSMAN, Roger; MAXIM, Bruce. **Engenharia de software: uma abordagem profissional**. 8. ed. *[S.l.]*: AMGH, 2021.

ROZSA, Vitor *et al.* Aplicação de Tecnologias da Web Semântica em Motores de Busca na Internet. **Investigación bibliotecológica**, v. 33, n. 78, p. 165–191, mar. 2019.

Sistema de Mapeamento de Competências - SIMC | Nit Uesc. Disponível em: <https://nit.uesc.br/sistema-de-mapeamento-de-competencias-simc>. Acesso em: 10 nov. 2025.

SUBRAMANIAN, D. Venkata *et al.* KMSBOT: enhancing educational institutions with an AI-powered semantic search engine and graph database. **Soft Computing**, v. 29, n. 1, p. 1–15, jan. 2025.

What are Transformers? - Transformers in Artificial Intelligence Explained - AWS.

Disponível em: <https://aws.amazon.com/what-is/transformers-in-artificial-intelligence/>. Acesso em: 25 jul. 2025.

What is LLM? - Large Language Models Explained - AWS. Disponível em: <https://aws.amazon.com/what-is/large-language-model/>. Acesso em: 22 jun. 2025.

XIE, Yutao *et al.* Survey of Code Search Based on Deep Learning. **ACM Transactions on Software Engineering and Methodology**, v. 33, n. 2, p. 1–42, 23 dez. 2023.

APÊNDICE A - MÉTODOS RELACIONADOS À FILTRAGEM DO SISTEMA DO SIMCC

Este apêndice apresenta os principais trechos do código-fonte da lógica de filtragem do Sistema de Mapeamento de Competências Científicas (SIMCC) que foi adotado como *baseline* comparativo desta pesquisa. A documentação destes métodos tem por objetivo fornecer evidências materiais sobre a arquitetura utilizada na solução do SIMC, permitindo a verificação da estrutura de acoplamento rígido descrita na fundamentação e validação. Esse código foi usado para as fases 1 e 2 da validação que estão descritas no capítulo de resultados.

```
Arquivo termFlowSQL.py

def lists_bibliographic_production_article_researcher_db(
    page: int,
    term: str = None,
    researcher_id: str = None,
    year: int = None,
    type: str = None,
    qualis: str = None,
):
    filter = str()
    if term:
        filter = f"AND {util.web_search_filter(term, 'title')}"

    filter_qualis = str()
    if qualis:
        filter_qualis = util.filterSQL(qualis, ";", "or", "qualis")

    if researcher_id:
        filter_id = f"""AND r.id = '{researcher_id}'"""
    else:
        filter_id = str()

    if page:
        page = page - 1
        pagination = f"""
        OFFSET {30 * page}
        LIMIT 30
        """
    else:
        pagination = str()

    if type == "ARTICLE":
        script_sql = f"""
            SELECT DISTINCT
                b.id AS id,
                title,
                b.year AS year,
                type,
                doi,
                ba.qualis,
                periodical_magazine_name AS magazine,
                r.name AS researcher,
                r.lattes_10_id AS lattes_10_id,
                r.lattes_id AS lattes_id,
```

```
                jcr AS jif,
                jcr_link,
                r.id as researcher_id,
                opa.abstract,
                opa.article_institution,
                opa.authors,
                opa.authors_institution,
                opa.citations_count,
                ba.issn,
                opa.keywords,
                opa.landing_page_url,
                opa.language,
                opa.pdf,
                b.has_image,
                b.relevance
            FROM
                bibliographic_production b
                LEFT JOIN bibliographic_production_article ba ON b.id =
ba.bibliographic_production_id
                LEFT JOIN researcher r ON r.id = b.researcher_id
                LEFT JOIN institution i ON r.institution_id = i.id
                LEFT JOIN openalex_article opa ON opa.article_id = b.id
            WHERE
                year_ >= {year}
                {filter}
                {filter_qualis}
                {filter_id}
                AND b.type = 'ARTICLE'
            ORDER BY
                year DESC
            {pagination}
            """

    if type == "ABSTRACT":
        script_sql = f"""
        SELECT DISTINCT
            b.id AS id,
            title,
            year,
            type,
            doi,
            ba.qualis,
            periodical_magazine_name AS magazine,
            r.name AS researcher,
            r.lattes_10_id AS lattes_10_id,
            r.lattes_id AS lattes_id,
            jcr AS jif,
            jcr_link,
            r.id,
            opa.abstract,
            opa.article_institution,
            opa.authors,
            opa.authors_institution,
            opa.citations_count,
            ba.issn,
            opa.keywords,
            opa.landing_page_url,
            opa.language,
            opa.pdf,
            b.has_image,
            b.relevance
        FROM
            bibliographic_production b
```

```
            LEFT JOIN bibliographic_production_article ba ON b.id = ba.bibliographic_production_id
            LEFT JOIN researcher r ON r.id = b.researcher_id
            LEFT JOIN institution i ON r.institution_id = i.id
            LEFT JOIN openalex_article opa ON opa.article_id = b.id
        WHERE
            year_ >= {year}
            {filter}
            {filter_qualis}
            {filter_id}
        ORDER BY
            year DESC
        {pagination}
        """
    print(script_sql)
    reg = sgbdSQL.consultar_db(script_sql)
    data_frame = pd.DataFrame(
        reg,
        columns=[
            "id",
            "title",
            "year",
            "type",
            "doi",
            "qualis",
            "magazine",
            "researcher",
            "lattes_10_id",
            "lattes_id",
            "jif",
            "jcr_link",
            "researcher_id",
            "abstract",
            "article_institution",
            "authors",
            "authors_institution",
            "citations_count",
            "issn",
            "keywords",
            "landing_page_url",
            "language",
            "pdf",
            "has_image",
            "relevance",
        ],
    )

    return data_frame.fillna("").to_dict(orient="records")
```

```
Arquivo util.py

def filterSQL(text, split, booleanOperator, attribute):
    filter = " "
    if text != "":
        t = []
        t = text.split(split)
        filter = ""
        i = 0
        for word in t:
            filter = (
                " unaccent(LOWER("
                + attribute
                + "))='"
```

```
                + unidecode.unidecode(word.lower())
                + "' "
                + booleanOperator
                + ""
                + filter
            )
            i = i + 1
        x = len(filter)
        filter = filter[0 : x - 3]
        filter = " AND (" + filter + ")"
    return filter

def web_search_filter(string_of_terms, column):
    skip = 0
    term = str()
    filter_terms = str()
    sintax_simbols = [";", ".", "|", "("]
    grammatic = {";": " AND \n\n", ".": " AND NOT \n\n", "|": " OR \n\n"}

    for position, char in enumerate(string_of_terms):
        if skip:
            skip -= 1
            continue

        if char in sintax_simbols:
            if char == "(":
                start = position + 1
                end = string_of_terms.find(")", start)
                part = string_of_terms[start:end]
                skip = len(part) + 1
                filter_terms += web_search_filter(part, column)
                continue
            elif term:
                term = unidecode.unidecode(term.lower())
                filter_terms +=
rf"""ts_rank(to_tsvector(translate(unaccent(LOWER({column})),'-\.:;''',' ')),
websearch_to_tsquery('"{term}"')) > 0.04"""
            filter_terms += grammatic[char]
            term = str()
        else:
            term += char
    if term:
        term = unidecode.unidecode(term.lower())
        filter_terms += rf"""ts_rank(to_tsvector(translate(unaccent(LOWER({column})),'-\.:;''','
')), websearch_to_tsquery('"{term}"')) > 0.04"""
    return f"""({filter_terms})"""
```

APÊNDICE B - MÉTODOS RELACIONADOS À FILTRAGEM DO SISTEMA DO *SELF QUERY*

Neste apêndice estão detalhados alguns fragmentos do artefato de software, incluíndo métodos e classes desenvolvidos na implementação da arquitetura proposta baseada em *self-query*. A demonstração destes trechos de código visa apresentar os métodos e classes usados no fluxo de filtragem dos metadados e que são avaliados no capítulo de resultados.

```
class SelfQueryService:
    def buscar_artigos_hibrido(
            self, query: str, max_results: int = 20
        ) -> Dict[str, Any]:
        """
        Realiza busca híbrida combinando busca por termos, semântica e filtros automáticos.

        Args:
            query: Consulta em linguagem natural
            max_results: Número máximo de resultados

        Returns:
            Resultados da busca híbrida com duas listas separadas
        """
        try:
            # Passo 1: Separar query de conteúdo e filtros
            structured_query = self.self_query.query_constructor.invoke(
                {"query": query}
            )
            content_query = (
                structured_query.query
                if hasattr(structured_query, "query") and structured_query.query
                else query
            )
            filters = (
                structured_query.filter if hasattr(structured_query, "filter") else None
            )

            logger.info(
                f"Query separada - Conteúdo: '{content_query}', Filtros: {filters}"
            )

            # Passo  2: Traduzir filtros para SQL
            filter_query = ""
            interface_filter = ""
            if filters:
                filter_query = get_where_clause(str(filters), self.sql_query_translator)
                interface_filter = get_where_clause(str(filters), self.interface_translator)

            # Passo 3: Realizar buscas por termos e semântica com filtros traduzidos
            resultados_termos = self.dao.buscar_por_termo(content_query, filter_query)
            resultados_semanticos = self.semantic.semantic_search(
                content_query, k=max_results, filter=filter_query
            )

            # Passo 4: Remover artigos duplicados que possuem no termo e no semantico.
            resultados_semanticos = self._remove_duplicated(
                resultados_semanticos, resultados_termos
            )

            return {
                "query": query,
```

```
                "structured_query": {
                    "content_query": content_query,
                    "filter_interface": interface_filter if filters else None,
                    "filter_selfquery": filter_query if filters else None,
                },
                "search_stats": {
                    "total_resultados":F len(resultados_termos)
                    + len(resultados_semanticos)
                },
                "results": {
                    "termos": resultados_termos,
                    "semanticos": resultados_semanticos,
                },
            }

        except Exception as e:
            logger.error(f"Erro na busca híbrida: {e}")
            raise RuntimeError(f"Erro ao processar busca híbrida: {str(e)}")
```

```
class ArtigoBuscaDTO(BaseModel):
    model_config = ConfigDict(from_attributes=True)

    id: str
    title: Optional[str]
    abstract: Optional[str]
    doi: Optional[str]
    year: Optional[int]
    journal: Optional[str]
    qualis: Optional[str]
    authors: Optional[List[str]] = None
    citacoes: Optional[int] = None
    score: Optional[float] = None

    def __init__(self, id: str, title: Optional[str] = None, abstract: Optional[str] = None,
                 doi: Optional[str] = None, year: Optional[int] = None,
                 journal: Optional[str] = None, qualis: Optional[str] = None,
                 authors: Optional[List[str]] = None, citacoes: Optional[int] = None, score:
Optional[float] = None,
                 embedding: Optional[str] = None):
        super().__init__(id=id, title=title, abstract=abstract, doi=doi,
                         year=year, journal=journal, qualis=qualis, authors=authors,
citacoes=citacoes, score=score)
```

```
class ArtigoDTOMapper:
    @staticmethod
    def to_artigo_busca_dto_from_sql_rows(linhas: List[Dict]) -> List[ArtigoBuscaDTO]:
        """
        Agrupa linhas SQL que representam possivelmente o mesmo artigo (mesmo título, journal, year,
doi)
        e converte para uma lista de ArtigoBuscaDTO, agregando autores únicos.
        """
        if not linhas:
            return []

        artigos_map: Dict[str, ArtigoBuscaDTO] = {}
        for linha in linhas:
            id_artigo = linha.get('id')
            title = linha.get('title')
            journal = linha.get('journal')
            year = linha.get('year')
```

```
            abstract = linha.get('abstract')
            doi = linha.get('doi')
            qualis = linha.get('qualis')
            authors = linha.get('authors')
            citacoes = linha.get('citacoes', None)
            score = linha.get('similarity_score', None)

            key = ArtigoDTOMapper._make_key(title, journal, year, doi)

            if key not in artigos_map:
                artigos_map[key] = ArtigoBuscaDTO(
                    id=str(id_artigo),
                    title=title,
                    journal=journal,
                    year=year,
                    abstract=abstract or "",
                    doi=doi,
                    qualis=qualis,
                    authors=[],
                    citacoes=citacoes,
                    score=score
                )

            ArtigoDTOMapper._append_authors(artigos_map[key], authors)

        return list(artigos_map.values())
```

```
class FilterParser:
    def parse_filter_string(self, filter_string: str) -> Union[Operation, Comparison, None]:
        if not filter_string or filter_string.strip() == "":
            return None

        filter_string = filter_string.strip()

        if filter_string.startswith("operator="):
            return self._parse_operation_string(filter_string)
        elif filter_string.startswith(("Comparison(", "comparator=")):
            return self._parse_comparison_string(filter_string)
        else:
            raise ValueError(f"Unknown filter format: {filter_string}")

    def _parse_operation_string(self, op_string: str) -> Operation:
        operator_match = re.search(r"operator=<Operator\.(\w+):", op_string)
        if not operator_match:
            raise ValueError(f"Could not extract operator from: {op_string}")

        operator_name = operator_match.group(1)
        operator = Operator.AND if operator_name == "AND" else Operator.OR

        args_match = re.search(r"arguments=\[(.*)\]", op_string, re.DOTALL)
        if not args_match:
            raise ValueError(f"Could not extract arguments from: {op_string}")

        args_string = args_match.group(1)
        arguments = self._parse_arguments_list(args_string)

        return Operation(operator=operator, arguments=arguments)

    def _parse_arguments_list(self, args_string: str) -> List[Union[Operation, Comparison]]:
        arguments = []
        current_arg = ""
        paren_count = 0
```

```
        bracket_count = 0

        for char in args_string:
            if char == '(':
                paren_count += 1
            elif char == ')':
                paren_count -= 1
            elif char == '[':
                bracket_count += 1
            elif char == ']':
                bracket_count -= 1
            elif char == ',' and paren_count == 0 and bracket_count == 0:
                if current_arg.strip():
                    arguments.append(self._parse_single_argument(current_arg.strip()))
                current_arg = ""
                continue

            current_arg += char

        if current_arg.strip():
            arguments.append(self._parse_single_argument(current_arg.strip()))

        return arguments

    def _parse_single_argument(self, arg_string: str) -> Union[Operation, Comparison]:
        arg_string = arg_string.strip()

        if arg_string.startswith("operator="):
            return self._parse_operation_string(arg_string)
        elif arg_string.startswith("Comparison("):
            return self._parse_comparison_string(arg_string)
        elif arg_string.startswith("Operation("):
            return self._parse_nested_operation(arg_string)
        else:
            raise ValueError(f"Unknown argument format: {arg_string}")

    def _parse_nested_operation(self, op_string: str) -> Operation:
        content = op_string[10:]
        last_paren = content.rfind(')')
        if last_paren == -1:
            raise ValueError(f"Could not find closing parenthesis in: {op_string}")

        content = content[:last_paren]
        return self._parse_operation_string(content)

    def _parse_comparison_string(self, comp_string: str) -> Comparison:
        comparator_match = re.search(r"comparator=<Comparator\.(\w+):", comp_string)
        if not comparator_match:
            raise ValueError(f"Could not extract comparator from: {comp_string}")

        comparator_name = comparator_match.group(1)

        comparator_map = {
            'EQ': Comparator.EQ, 'NE': Comparator.NE, 'LT': Comparator.LT, 'LTE': Comparator.LTE,
            'GT': Comparator.GT, 'GTE': Comparator.GTE, 'IN': Comparator.IN, 'NIN': Comparator.NIN,
            'LIKE': Comparator.LIKE, 'ILIKE': Comparator.ILIKE
        }

        if comparator_name not in comparator_map:
            raise ValueError(f"Unknown comparator: {comparator_name}")

        comparator = comparator_map[comparator_name]
```

```
        attr_match = re.search(r"attribute='([^']+)'", comp_string)
        if not attr_match:
            raise ValueError(f"Could not extract attribute from: {comp_string}")

        attribute = attr_match.group(1)

        value_match = re.search(r"value=(.+?)(?:\)|$)", comp_string)
        if not value_match:
            raise ValueError(f"Could not extract value from: {comp_string}")

        value_string = value_match.group(1)
        value = self._parse_value(value_string)

        return Comparison(comparator=comparator, attribute=attribute, value=value)

    def _parse_value(self, value_string: str) -> Any:
        value_string = value_string.strip()

        if value_string.startswith('[') and value_string.endswith(']'):
            list_content = value_string[1:-1]
            if not list_content.strip():
                return []

            items = []
            current_item = ""
            in_quotes = False
            quote_char = None

            for char in list_content:
                if char in ['"', "'"] and not in_quotes:
                    in_quotes = True
                    quote_char = char
                elif char == quote_char and in_quotes:
                    in_quotes = False
                    quote_char = None
                elif char == ',' and not in_quotes:
                    items.append(self._clean_value(current_item.strip()))
                    current_item = ""
                    continue

                current_item += char

            if current_item.strip():
                items.append(self._clean_value(current_item.strip()))

            return items

        return self._clean_value(value_string)

    def _clean_value(self, value: str) -> Any:
        value = value.strip()

        if (value.startswith('"') and value.endswith('"')) or \
           (value.startswith("'") and value.endswith("')):
            return value[1:-1]

        try:
            return float(value) if '.' in value else int(value)
        except ValueError:
            pass

        if value.lower() == 'true':
            return True
```

```
        elif value.lower() == 'false':
            return False
        elif value.lower() in ('null', 'none'):
            return None

        return value
```

```
class PostgreSQLFilterTranslator(FilterTranslator):
    def _get_true_clause(self) -> str:
        return "TRUE"

    def _get_operator_sql(self, operator: Operator) -> str:
        operator_map = {Operator.AND: "AND", Operator.OR: "OR"}
        if operator not in operator_map:
            raise ValueError(f"Unsupported operator: {operator}")
        return operator_map[operator]

    def _translate_comparison(self, comparison: Comparison) -> str:
        column_name = self._get_mapped_column_name(comparison.attribute)
        self._validate_column_name(column_name)
        return self._build_comparison_clause(column_name, comparison.comparator, comparison.value)

    def _build_comparison_clause(self, column_name: str, comparator: Comparator, value: Any) -> str:
        comparison_builders = {
            Comparator.EQ: lambda c, v: f"{c} = {self._quote_value(v)}",
            Comparator.NE: lambda c, v: f"{c} != {self._quote_value(v)}",
            Comparator.LT: lambda c, v: f"{c} < {v}",
            Comparator.LTE: lambda c, v: f"{c} <= {v}",
            Comparator.GT: lambda c, v: f"{c} > {v}",
            Comparator.GTE: lambda c, v: f"{c} >= {v}",
            Comparator.IN: self._build_in_clause,
            Comparator.NIN: self._build_not_in_clause,
            Comparator.LIKE: self._build_like_clause,
            Comparator.ILIKE: self._build_ilike_clause
        }

        builder = comparison_builders.get(comparator)
        if not builder:
            raise ValueError(f"Unsupported comparator: {comparator}")

        return builder(column_name, value)

    def _build_in_clause(self, column_name: str, value: Any) -> str:
        if not isinstance(value, list):
            raise ValueError("IN operator requires a list of values")
        values_str = ",".join([self._quote_value(v) for v in value])
        return f"{column_name} IN ({values_str})"

    def _build_not_in_clause(self, column_name: str, value: Any) -> str:
        if not isinstance(value, list):
            raise ValueError("NIN operator requires a list of values")
        values_str = ",".join([self._quote_value(v) for v in value])
        return f"{column_name} NOT IN ({values_str})"

    def _build_like_clause(self, column_name: str, value: Any) -> str:
        return f"unaccent(lower({column_name})) LIKE
unaccent(lower({self._quote_value(f'%{value}%')}))"

    def _build_ilike_clause(self, column_name: str, value: Any) -> str:
        return f"unaccent(lower({column_name})) ILIKE
```

```
unaccent(lower({self._quote_value(f'%{value}%')}))"

    def _quote_value(self, value: Any) -> str:
        if isinstance(value, str):
            if '%' in value:
                if ' ' in value:
                    value = value.replace(' ', '%')
                return f"$${value}$$"
            else:
                escaped_value = value.replace("'", "''")
                return f"'{escaped_value}'"
        elif isinstance(value, (int, float)):
            return str(value)
        elif isinstance(value, bool):
            return "TRUE" if value else "FALSE"
        elif value is None:
            return "NULL"
        else:
            escaped_value = json.dumps(value).replace("'", "''")
            return f"'{escaped_value}'"
```