UNIVERSIDADE FEDERAL DO RIO GRANDE DO SUL
INSTITUTO DE INFORMÁTICA
DEPARTAMENTO DE INFORMÁTICA APLICADA
CURSO DE ESPECIALIZAÇÃO EM ENGENHARIA DE SOFTWARE PARA APLICAÇÕES DE CIÊNCIA DE DADOS

DOUGLAS LÁZARO S. P. SILVA

Hermeneutics 2.0 - Evolução Arquitetural de uma Ferramenta de auxílio de Revisão Sistemática da Literatura para Microsserviços

Monografia de Conclusão de Curso apresentada como requisito parcial para a obtenção do grau de Especialista em Engenharia de Software para Aplicações de Ciência de Dados

Orientador: Prof. Dr. Leandro Krug Wives

Porto Alegre
2026

CIP — CATALOGAÇÃO NA PUBLICAÇÃO

S. P. Silva, Douglas Lázaro

Hermeneutics 2.0 - Evolução Arquitetural de uma Ferramenta de auxílio de Revisão Sistemática da Literatura para Microsserviços / Douglas Lázaro S. P. Silva. – Porto Alegre: PPGC da UFRGS, 2026.

116 f.: il.

Monografia (especialização) – Universidade Federal do Rio Grande do Sul. Curso de Especialização em Engenharia de Software para Aplicações de Ciência de Dados, Porto Alegre, BR–RS, 2026. Orientador: Leandro Krug Wives.

1. Revisão sistemática da literatura. 2. Rotulação semântica. 3. Microsserviços. I. Krug Wives, Leandro. II. Título.

UNIVERSIDADE FEDERAL DO RIO GRANDE DO SUL
Reitora: Profª. Marcia Barbosa
Vice-Reitor: Prof. Pedro de Almeida Costa
Pró-Reitor de Pós-Graduação: Profª. Claudia Wasserman
Diretor do Instituto de Informática: Prof. Luciano Paschoal Gaspary
Coordenadora do Curso: Profª. Karin Becker
Bibliotecária-chefe do Instituto de Informática: Alexsander Borges Ribeiro

“If heaven is a void,

human beings will put their hope in science and technology ”

— EGBERT SCHUURMAN

AGRADECIMENTOS

Primeiramente, agradeço a Deus, por ter me sustentado, capacitado e concedido discernimento ao longo de toda esta jornada acadêmica e profissional. Reconheço que a possibilidade de desenvolver este trabalho, superar dificuldades e concluir esta etapa é fruto de Sua graça.

Agradeço de forma muito especial à minha namorada, hoje noiva e, em breve, esposa. Quando iniciei esta pós-graduação, há dois anos, vivíamos um momento marcante do nosso relacionamento, e desde então ela esteve presente em cada etapa, compartilhando desafios, expectativas e conquistas. No último ano, tive a alegria de pedi-la em noivado e, agora, nos preparamos para celebrar o casamento civil na semana seguinte à defesa desta monografia. Seu amor, paciência, compreensão e incentivo foram essenciais para que eu conseguisse atravessar períodos de intensa dedicação, madrugadas de estudo e finais de semana de implementação. Este trabalho também é resultado do seu apoio constante.

Agradeço à minha família e amigos, pelo carinho, incentivo e compreensão, mesmo nos momentos em que estive sobrecarregado pelos prazos do TCC e, simultaneamente, pelos desafios profissionais relacionados à migração e modernização de sistemas legados, que consumiram grande parte do meu tempo no último semestre.

Agradeço ao orientador, Prof. Dr. Leandro Krug Wives, pela disponibilidade, orientações e contribuições que auxiliaram na definição do escopo e no direcionamento deste trabalho.

Por fim, agradeço a todos que, direta ou indiretamente, contribuíram para minha formação acadêmica e profissional e para a realização deste projeto.

RESUMO

Revisões sistemáticas da literatura (RSL) são fundamentais para consolidar evidências e identificar lacunas de pesquisa em cenários com grande volume de publicações digitais. A ferramenta *Hermeneutics*, desenvolvida entre 2018 e 2020, aplicou mineração textual e rotulação semântica de resumos para auxiliar pesquisadores em processos de RSL. O protótipo original, concebido como aplicação monolítica em Django, apresentou limitações de escalabilidade, manutenibilidade e observabilidade: fila única de processamento, acoplamento entre módulos e baixa tolerância a falhas. Diante dessas limitações e da evolução das arquiteturas distribuídas, este trabalho apresenta *Hermeneutics 2.0*, uma versão reestruturada com base em microsserviços orientados a eventos. A decomposição do monólito foi realizada seguindo os princípios de *Domain-Driven Design* (DDD), resultando em serviços autônomos para Ingestão, Anotação, Persistência e Orquestração. A consistência eventual entre os serviços é garantida pelo padrão **Saga Coreografada**, implementado com Apache Kafka como *backbone* de *mensageria*. A camada de observabilidade, instrumentada com **OpenTelemetry**, representa uma das contribuições centrais deste trabalho. Diferentemente da versão anterior, a arquitetura v2 fornece transparência total através da correlação de Logs, Métricas e Traces distribuídos, permitindo rastrear transações desde a interface web até a persistência final. A avaliação da arquitetura foi realizada em ambiente local via Docker Compose, utilizando dados do *dataset* legado e *mocks* para simular latências de rede e o comportamento de modelos de linguagem. A integração com *Large Language Models* (LLMs) comerciais foi abstraída através de interfaces (Ports), aplicando o princípio YAGNI para postergar custos prematuros de APIs pagas. Os resultados demonstram a viabilidade da arquitetura proposta para suportar futuras integrações com modelos de IA em escala, estabelecendo uma fundação sólida para práticas de FinOps e MLOps.

Palavras-chave: Revisão sistemática da literatura. rotulação semântica. microsserviços.

Hermeneutics 2.0 – Architectural Evolution of a Systematic Literature Review Support Tool to Event-Driven Microservices

ABSTRACT

Systematic Literature Reviews (SLRs) are essential for consolidating evidence and identifying research gaps in domains with a large volume of digital publications. The *Hermeneutics* tool, developed between 2018 and 2020, applied text mining and semantic labeling of abstracts to assist researchers in SLR processes. The original prototype, designed as a monolithic Django application, exhibited limitations in scalability, maintainability, and observability: a single processing queue, tight coupling between modules, and low fault tolerance. In response to these limitations and the evolution of distributed architectures, this work presents *Hermeneutics 2.0*, a restructured version based on event-driven microservices. The monolith decomposition was performed following *Domain-Driven Design* (DDD) principles, resulting in autonomous services for Ingestion, Annotation, Persistence, and Orchestration. Eventual consistency between services is guaranteed by the **Choreographed Saga** pattern, implemented with Apache Kafka as the messaging backbone. The observability layer, instrumented with **OpenTelemetry**, represents one of the central contributions of this work. Unlike the previous version, the v2 architecture provides full transparency through the correlation of Logs, Metrics, and Distributed Traces, enabling end-to-end transaction tracking from the web interface to final persistence. The architecture evaluation was performed in a local environment using Docker Compose, with legacy dataset data and mocks to simulate network latencies and language model behavior. Integration with commercial LLMs was abstracted through interfaces (Ports), applying the YAGNI principle to defer premature costs from paid APIs. The results demonstrate the viability of the proposed architecture to support future AI model integrations at scale, establishing a solid foundation for FinOps and MLOps practices.

Keywords: systematic literature review, microservices, observability, event-driven architecture, OpenTelemetry.

LISTA DE ABREVIATURAS E SIGLAS

ACID	Atomicity, Consistency, Isolation, Durability
ADR	Architecture Decision Record
AGPL	Affero General Public License
AJAX	Asynchronous JavaScript and XML
API	Application Programming Interface
AWS	Amazon Web Services
BASE	Basically Available, Soft state, Eventual consistency
BERT	Bidirectional Encoder Representations from Transformers
BFF	Backend for Frontend
CD	Continuous Delivery
CI	Continuous Integration
CPU	Central Processing Unit
CQRS	Command Query Responsibility Segregation
CRUD	Create, Read, Update, Delete
DB	Database
DDD	Domain-Driven Design
DER	Diagrama Entidade-Relacionamento
DRY	Don’t Repeat Yourself
DX	Developer Experience
EPIC	Epic
ERP	Enterprise Resource Planning
ES	Engenharia de Software
ETL	Extract, Transform, Load
GCP	Google Cloud Platform

GPU	Graphics Processing Unit
HTML	HyperText Markup Language
HTTP	Hypertext Transfer Protocol
IA	Inteligência Artificial
ID	Identifier
IEEE	Institute of Electrical and Electronics Engineers
INPI	Instituto Nacional da Propriedade Industrial
JSON	JavaScript Object Notation
KISS	Keep It Simple, Stupid
LLM	Large Language Model
LTS	Long-Term Support
ML	Machine Learning
MTTK	Mean Time To Know
MTV	Model-Template-View
MVC	Model-View-Controller
NER	Named-entity recognition
NFR	Non-Functional Requirement
NLP	Natural Language Processing
NoSQL	Not Only SQL
OTLP	OpenTelemetry Protocol
PLN	Processamento de Linguagem Natural
PRISMA	Preferred Reporting Items for Systematic Reviews and Meta-Analyses
REST	Representational State Transfer
RSL	Revisão Sistemática da Literatura
SciERC	Scientific Entities Relations and Coreference
SLR	Systematic Literature Review

SOA	Service-Oriented Architecture
SpERT	Span-based Entity and Relation Transformer
SRL	Semantic Role Labeling
SUS	System Usability Scale
TCC	Trabalho de Conclusão de Curso
UI	User Interface
VM	Virtual Machine
YAGNI	You Ain't Gonna Need It

LISTA DE FIGURAS

Figura 4.1 Hermeneutics v1 — Fluxograma Principal 36
Figura 4.2 Hermeneutics v1 — Arquitetura Monolítica (AS-IS). 38

Figura 5.1 Hermeneutics v2 — Visão Geral da Arquitetura (TO-BE). 41
Figura 5.2 Hermeneutics v2 — Diagrama de Sequência - Saga Pattern 50

Figura 6.1 Estrutura do pipeline de CI para *Pull Requests* no GitHub Actions. 58
Figura 6.2 Dashboard de Infraestrutura: Uso de CPU e Memória dos 15+ contêineres. 62
Figura 6.3 Saúde dos Bancos de Dados: Conexões ativas e operações por segundo 62
Figura 6.4 Visão Geral OpenTelemetry: Latência, Taxa de Requisições e Taxa de Erros. 63
Figura 6.5 Detalhamento dos *Golden Signals*: Métricas agrupadas por endpoint HTTP. 63
Figura 6.6 Grafo de Dependências de Serviços (*Service Graph*) gerado via Jaeger. 64
Figura 6.7 Trace Timeline: Visualização Waterfall de uma transação distribuída completa. 65
Figura 6.8 Busca e visualização de Traces integrados no Grafana Tempo. 66
Figura 6.9 Logs centralizados no Loki com correlação automática de Trace ID. 67
Figura 6.10 Timeline de execução da Saga em modo Memória, evidenciando os delays simulados. 68
Figura 6.11 Logs no Loki durante execução em memória, mantendo rastreabilidade completa. 69
Figura 6.12 Dashboard FinOps projetando custos e tempos baseados na simulação com Jitter. 70
Figura 6.13 Tela inicial do Hermeneutics Web UI (interface inalterada v1→v2). 71
Figura 6.14 Barra de progresso assíncrona baseada no status da Saga. 72
Figura 6.15 Listagem de artigos processados e disponíveis para triagem. 72
Figura 6.16 Detalhe de artigo com anotações NER e SRL renderizadas. 73
Figura 6.17 Flamegraph de trace completo: visualização da profundidade de chamadas. 73
Figura 6.18 Estatísticas de trace por serviço: spans, duração e distribuição. 74
Figura 6.19 Dashboard FinOps com métricas reais de 500 artigos processados. 74
Figura 6.20 Logs no Loki evidenciando detecção de Cache Hit. 75
Figura 6.21 Trace curto no Jaeger: Cache Hit evita reprocessamento completo. 75

LISTA DE TABELAS

Tabela 2.1 Comparativo entre as abordagens de Kitchenham e PRISMA. 21

Tabela 3.1 Comparativo de ferramentas de RSL sob critérios arquiteturais. 32
Tabela 3.2 Estudos de caso acadêmicos sobre migração de monólitos para microsserviços 34

Tabela 5.1 ADRs Fundamentais da Arquitetura Hermeneutics 2.0 46
Tabela 5.2 Catálogo Completo de ADRs do Hermeneutics 2.0 49

Tabela 6.1 Principais EPICs do projeto Hermeneutics v2. 55
Tabela 6.2 Componentes cobertos por testes unitários no pipeline CI. 60
Tabela 6.3 Estratégia de Batching na Saga Coreografada. 71
Tabela 6.4 Comparativo Arquitetural: Hermeneutics v1 (AS-IS) vs. Hermeneutics v2 (TO-BE). 79

Tabela A.1 Índice de Decisões Arquiteturais (ADRs) do Projeto Hermeneutics v2.0 ... 90

Tabela B.1 Atributos Gerais do Produto Hermeneutics 2.0 103
Tabela B.2 Catálogo de PRDs Estratégicos do Hermeneutics 2.0 104
Tabela B.3 Requisitos Funcionais e Objetivos do Hermeneutics 2.0 105
Tabela B.4 Requisitos Não Funcionais e Arquitetura do Hermeneutics 2.0 106

Tabela C.1 Resumo dos Épicos de Engenharia do Hermeneutics v2.0 108

SUMÁRIO

1 INTRODUÇÃO 16
1.1 Definição do Problema 16
1.2 Objetivos 17
1.2.1 Objetivos Específicos 17
1.3 Contribuições e Avaliação 17
1.4 Organização do Texto 18
2 FUNDAMENTAÇÃO TEÓRICA 19
2.1 Revisão Sistemática da Literatura: Fundamentos e Evolução 19
2.1.1 Abordagens Metodológicas: de Kitchenham ao PRISMA 19
2.1.2 Comparativo e Sinergia entre Kitchenham e PRISMA 20
2.1.3 Implicações para a Evolução do Hermeneutics 2.0 21
2.2 Técnicas de Processamento de Linguagem Natural 21
2.3 Arquiteturas Orientadas a Domínio e Serviços 22
2.3.1 Domain-Driven Design 22
2.3.2 Arquitetura Orientada a Serviços 23
2.3.3 Arquitetura de Microsserviços 23
2.3.4 Arquitetura Hexagonal (*Ports and Adapters*) 23
2.3.5 Evolução Arquitetural 24
2.4 Padrões de Consistência em Sistemas Distribuídos 25
2.4.1 ACID *versus* BASE 25
2.4.2 Padrão Saga 25
2.4.3 Padrão *Database-per-Service* 26
2.5 Observabilidade e a Adoção do OpenTelemetry 27
2.5.1 Os Pilares da Observabilidade 27
2.5.2 OpenTelemetry: O Padrão Aberto para Observabilidade 28
2.5.2.1 Componentes Chave do OpenTelemetry: 29
2.5.2.2 Vantagens da Adoção do OpenTelemetry para a Observabilidade: 30
3 TRABALHOS RELACIONADOS 31
3.1 Ferramentas de Revisão Sistemática da Literatura 31
3.2 Migração de Sistemas Monolíticos para Microsserviços 32
3.2.1 Motivações para a Migração de Monólitos 32
3.2.2 Padrões e Práticas Recorrentes na Literatura 33
3.2.3 Estudos de Caso Documentados na Literatura 34
3.2.4 Lacunas e Contribuições Esperadas 35
4 HERMENEUTICS 1.0: CARACTERIZAÇÃO DO LEGADO 36
4.1 Visão Geral e Propósito 36
4.2 Arquitetura Monolítica e Tecnologias (AS-IS) 36
4.2.1 Stack Tecnológica 36
4.2.2 Estrutura do Monólito 37
4.3 Limitações Identificadas 38
4.4 Motivação para a Modernização 39
5 PROPOSTA ARQUITETURAL E MODERNIZAÇÃO 41
5.1 Nova Arquitetura (TO-BE) 41
5.1.1 Bounded Contexts e Decomposição em Serviços 42
5.1.2 Componentes e Fluxo de Dados 42
5.1.2.1 Ponto de Entrada e API Gateway 42
5.1.2.2 Serviços de Domínio (Microsserviços) 43
5.1.2.3 Estratégia de Dados: Database per Service 43

5.1.3 Evolução da Mensageria: De Celery para Kafka 44
5.1.4 Observabilidade e Telemetria 44
5.2 Decisões Arquiteturais (ADRs) 45
5.2.1 ADRs Estruturantes 45
5.2.1.1 ADR-001: Adoção da Arquitetura de Microsserviços (DDD) 45
5.2.1.2 ADR-002: Transição para Mensageria Assíncrona com Apache Kafka 46
5.2.1.3 ADR-003: Adoção do Padrão Database per Service 47
5.2.1.4 ADR-004: Implementação de API Gateway 47
5.2.1.5 ADR-005: Adoção de Observabilidade com OpenTelemetry 48
5.2.1.6 ADR-006: Padronização da Stack Tecnológica 48
5.2.2 Catálogo Completo de ADRs 48
5.3 Padrões Implementados 49
5.3.1 Padrão Saga Coreografada com Apache Kafka 49
5.3.1.1 Topologia de Tópicos Kafka 50
5.3.1.2 Coreografia da Saga 50
5.3.2 Ambiente de Avaliação: Docker Compose 51
5.3.2.1 Stack de Serviços 51
5.3.2.2 Feature Toggles para Modos de Operação 52
5.3.2.3 Delays com Jitter para Métricas FinOps 52
5.3.2.4 Limitações Explícitas do Ambiente 52
6 IMPLEMENTAÇÃO E ANÁLISE DE RESULTADOS 54
6.1 Metodologia de Desenvolvimento 54
6.1.1 Etapas do Projeto 54
6.1.2 Organização do Trabalho em EPICs e Stories 55
6.1.3 Aceleração do Desenvolvimento com *Prompt Engineering* 55
6.2 Ambiente de Desenvolvimento e DevOps 56
6.2.1 Orquestração com Docker Compose 56
6.2.2 Pipeline de Integração Contínua (CI) 57
6.2.3 Developer Experience (DX) e Pre-commit Hooks 58
6.2.4 Testes Unitários e Cobertura 59
6.3 Evidências de Observabilidade 60
6.3.1 Fundamentação e Infraestrutura como Código 60
6.3.2 Metodologia do Experimento 61
6.3.3 Monitoramento de Infraestrutura 62
6.3.4 Métricas de Aplicação (*Golden Signals*) 63
6.3.5 Rastreamento Distribuído (Distributed Tracing) 64
6.3.6 Correlação de Logs 65
6.4 Análise de Custos e Demonstração E2E 66
6.4.1 O Papel dos Mocks e Adapters na Estratégia de Testes 66
6.4.2 Metodologia da Simulação de Custos (FinOps) 67
6.4.3 Evidências do Experimento (Modo Memória) 68
6.4.3.1 Demonstração de Fluxo e Rastreabilidade 68
6.4.3.2 Correlação de *Logs* 68
6.4.3.3 Dashboard FinOps e Análise de Valor 69
6.4.4 Evidências do Experimento (Modo *Database*) 70
6.4.4.1 Cenário de Demonstração E2E 70
6.4.4.2 Interface do Usuário (Web UI) 71
6.4.4.3 Análise de Rastreabilidade em Escala 73
6.4.4.4 Dashboard FinOps (Escala Real) 74
6.4.4.5 Otimização: Cache Hit 75

6.5 Análise Comparativa: Do Monólito à Arquitetura Distribuída 76
6.5.1 Persistência e Estratégia de Dados 76
6.5.2 Acoplamento, Coesão e Manutenibilidade 77
6.5.3 Estratégia de Deploy e Escalabilidade 77
6.5.4 Consistência de Dados e Mensageria 78
6.5.5 Observabilidade e Suporte a FinOps 78
6.5.6 Síntese Comparativa 78
7 CONCLUSÃO 80
7.1 Síntese das Contribuições 80
7.1.1 Observabilidade como Ativo de Dados 80
7.1.2 Fundação para FinOps 81
7.1.3 Extensibilidade para *Machine Learning* 81
7.2 Limitações do Trabalho 82
7.2.1 Uso de *Mocks* para Modelos de Linguagem 82
7.2.1.1 Complexidade do Modelo SpERT 82
7.2.2 Ausência de Deploy em Cloud Real 83
7.2.3 APIs de Bibliotecas Digitais Simuladas 83
7.2.4 Ausência de Avaliação Experimental com Usuários 83
7.3 Trabalhos Futuros 84
7.3.1 Infraestrutura e *Deploy* 84
7.3.2 Integração com Modelos de Linguagem 84
7.3.3 Integração com Bibliotecas Digitais 85
7.3.4 Evolução de FinOps 85
7.3.5 Qualidade e Testes 85
7.3.6 Evolução da Interface 85
7.3.7 Evolução Metodológica 85
7.3.8 Perspectivas de Continuidade 86
REFERÊNCIAS 87
APÊNDICE A — REGISTRO DE DECISÕES ARQUITETURAIS (ADRS) 90
A.1 Decisões de Arquitetura Core 92
A.1.1 ADR-001: Adoção da Arquitetura de Microsserviços (DDD) 92
A.1.2 ADR-002: Mensageria Assíncrona com Apache Kafka e Saga Pattern 93
A.1.3 ADR-003: Adoção do Padrão Database per Service 93
A.1.4 ADR-004: Implementação de API Gateway 94
A.1.5 ADR-005: Adoção de Observabilidade com OpenTelemetry 94
A.2 Decisões de Stack e Infraestrutura 95
A.2.1 ADR-006: Padronização da Stack Tecnológica 95
A.2.2 ADR-007: Contêiner Inicializador para Tópicos Kafka 96
A.2.3 ADR-008: Estratégia de Build Unificado (Dockerfile Template) 96
A.2.4 ADR-009: Shared Kernel para Infraestrutura Kafka 96
A.2.5 ADR-010: Object Storage Local com MinIO 97
A.3 Decisões de Observabilidade 97
A.3.1 ADR-011: Métricas de Longo Prazo com Thanos 97
A.3.2 ADR-012: Rastreamento Distribuído com Tempo 97
A.3.3 ADR-013: Agregação de Logs com Loki 97
A.3.4 ADR-014: Monitoramento de Golden Signals 98
A.3.5 ADR-015: Estratégia de Instrumentação OpenTelemetry 98
A.3.6 ADR-016: Abordagem Híbrida OpenTelemetry 98
A.4 Decisões de Padronização 98
A.4.1 ADR-017: Upgrade do Django para 4.2 LTS 99
A.4.2 ADR-018: Uso do Padrão Facade no Maître (Mock E2E) 99

A.4.3 ADR-019: Docker Compose Naming Convention 99
A.4.4 ADR-020: Makefile Naming Convention 99
A.4.5 ADR-021: API Gateway com Subdomínios Localhost 99
A.4.6 ADR-022: Shared Kernel para FastAPI (App Factory) 100
A.4.7 ADR-023: Padronização SCM (.editorconfig e .gitattributes) 100
A.5 Decisões de Persistência 100
A.5.1 ADR-024: Estratégia de Banco de Dados (Single Container) 100
A.5.2 ADR-025: Shared Kernel para Conectividade PostgreSQL 100
A.5.3 ADR-026: Shared Kernel para Cache Redis 101
A.5.4 ADR-027: ConnectionMixin Pattern para Repositórios 101
A.6 Decisões de Feature Toggles e FinOps 101
A.6.1 ADR-028: Feature Toggles para Modos de Operação 101
A.6.2 ADR-029: BFF Stateless com Redis Sessions 101
A.6.3 ADR-030: Kafka Batch Publishing para Mensagens Grandes 102
A.6.4 ADR-031: ETL Strategy para Migração de Legados 102
A.6.5 ADR-032: Delays Centralizados com Jitter (FinOps) 102
APÊNDICE B — DOCUMENTO DE REQUISITOS DO PRODUTO (PRD) 103
B.1 Atributos do Produto 103
B.2 Catálogo de PRDs Estratégicos 103
B.3 Requisitos Funcionais e Objetivos 105
B.4 Requisitos Não Funcionais e Arquitetura 105
B.5 Gestão de Documentação (Metodologia) 106
APÊNDICE C — ÉPICOS DE ENGENHARIA E USER STORIES 108
C.1 Visão Geral dos Épicos 108
C.2 EPIC-001: Reorganização e Estruturação de Engenharia de IA 108
C.2.1 User Stories Associadas 109
C.3 EPIC-002: Infraestrutura de Microsserviços e Refatoração Core 109
C.3.1 User Stories Associadas 110
C.4 EPIC-003: Integração Kafka e Padrão Saga 111
C.4.1 User Stories Associadas 111
C.5 EPIC-004: Observabilidade Full-Stack e FinOps 112
C.5.1 User Stories Associadas 112
C.6 EPIC-005: Padronização da Arquitetura 113
C.6.1 User Stories Associadas 113
C.7 EPIC-006: Persistência, Conectividade e Migração de Dados 114
C.7.1 User Stories Associadas 114
C.8 User Stories Técnicas (Quality Gates) 115
C.8.1 US-DEV-001: Configuração de Pre-Commit Hooks 115
C.8.2 US-DEV-002: Implementação de CI Pipeline (TDD) 116

1 INTRODUÇÃO

As Revisões Sistemáticas da Literatura (RSL) são essenciais para a síntese de evidências científicas (KITCHENHAM, 2004). No entanto, a longevidade e a evolução das ferramentas de apoio a esse processo enfrentam barreiras técnicas significativas. O *Hermeneutics*, originalmente desenvolvido como um monólito em Django (SILVA, 2021), exemplifica o desafio de manter softwares acadêmicos: o acoplamento excessivo e a falta de observabilidade dificultam a integração de novas tecnologias, como Modelos de Linguagem de Grande Escala (LLMs). Este trabalho aborda a modernização arquitetural necessária para superar tais limitações.

1.1 Definição do Problema

A ferramenta original, embora funcional como protótipo, representa uma típica **Aplicação Intensiva em Dados** (*Data-Intensive Application*), que utiliza modelos de Processamento de Linguagem Natural (PLN), como o SpERT (*Span-based Entity and Relation Transformer*) e o BERT (*Bidirectional Encoder Representations from Transformers*), para processamento semântico. Contudo, acumulou uma significativa dívida técnica arquitetural ao acoplar rigidamente a lógica de apresentação (Web), a lógica de negócios (RSL) e o processamento intensivo de dados (NLP) em um único processo monolítico. O desafio de engenharia neste contexto não reside no treinamento de modelos, mas sim na **orquestração do ciclo de vida, inferência e persistência** dos dados gerados por esses modelos.

Esse modelo impõe barreiras críticas à evolução do software: (1) **Escalabilidade Ineficiente**, onde não é possível escalar apenas o módulo de anotação semântica sem replicar todo o *stack web*; (2) **Fragilidade Sistêmica**, onde uma falha de memória durante o processamento de um lote de artigos derruba a interface do usuário; e (3) **Bloqueio Tecnológico**, dificultando a experimentação com novas bibliotecas de IA que exigem ambientes ou versões de linguagem conflitantes com o legado.

O problema de pesquisa, portanto, é redefinido não como a criação de novos modelos de IA, mas como: *Como arquitetar uma plataforma científica evolutiva que mitigue a rigidez do legado através de microsserviços, garantindo observabilidade e resiliência para suportar a próxima geração de ferramentas de RSL?*

1.2 Objetivos

O objetivo geral é projetar e implementar a evolução arquitetural da ferramenta *Hermeneutics* (v2), migrando de um monólito legado para uma arquitetura distribuída baseada em microsserviços, orientada a eventos e observabilidade.

1.2.1 Objetivos Específicos

- Decompor o monólito original, aplicando os princípios de *Domain-Driven Design* (DDD) para definição de contextos delimitados e Arquitetura Hexagonal para isolamento de domínio;
- Garantir a consistência eventual e a integridade de fluxos de longa duração através da implementação do padrão **Saga Coreografada** sobre uma infraestrutura de *mensageria* baseada em Apache Kafka;
- Implementar uma camada de observabilidade abrangente com o padrão **OpenTelemetry**, correlacionando Logs, Métricas e Traces para viabilizar análises de *FinOps* e monitoramento de custos em granularidade de transação;
- Projetar e validar uma estratégia de persistência poliglota (*Database-per-Service*), isolando os dados de cada microsserviço para eliminar o acoplamento em nível de banco de dados;
- Validar a eficácia da arquitetura proposta em cenários de carga simulada, utilizando um ambiente de orquestração local (Docker Compose) para coletar métricas de resiliência e latência distribuída.

1.3 Contribuições e Avaliação

Diferente de abordagens focadas puramente em funcionalidades, este trabalho contribui com uma arquitetura de referência para a modernização de software científico. A avaliação é realizada através da análise de telemetria (traces distribuídos e métricas de recursos) em cenários de carga simulada, demonstrando a viabilidade técnica e a observabilidade da solução proposta. Adicionalmente, introduz-se uma camada de FinOps para

monitoramento de custos computacionais, essencial para a sustentabilidade de projetos acadêmicos em nuvem (FinOps Foundation Technical Advisory Council, 2025).

1.4 Organização do Texto

A monografia está organizada da seguinte forma: o Capítulo 2 apresenta a fundamentação teórica sobre RSL, microsserviços e observabilidade. O Capítulo 3 descreve os trabalhos relacionados e o estado da arte. O Capítulo 4 caracteriza a ferramenta Hermeneutics 1.0 e diagnostica as limitações que motivaram a modernização. O Capítulo 5 detalha a proposta arquitetural do Hermeneutics 2.0. O Capítulo 6 discute a implementação e os resultados da avaliação técnica e de FinOps. Por fim, o Capítulo 7 apresenta as considerações finais e trabalhos futuros.

2 FUNDAMENTAÇÃO TEÓRICA

Neste capítulo, são apresentados os fundamentos teóricos que sustentam as decisões arquiteturais e de engenharia deste trabalho. A primeira seção aborda a Revisão Sistemática da Literatura (RSL), domínio de aplicação da ferramenta. As seções subsequentes fundamentam as escolhas técnicas: Arquitetura de Microsserviços, *Domain-Driven Design* (DDD), padrões de consistência eventual (Saga) e Observabilidade distribuída com OpenTelemetry.

2.1 Revisão Sistemática da Literatura: Fundamentos e Evolução

A Revisão Sistemática da Literatura (RSL) é um método de investigação científica secundária cujo objetivo é identificar, avaliar e interpretar de forma criteriosa toda a evidência disponível e relevante sobre uma questão de pesquisa específica, um tópico ou um fenômeno de interesse. Amplamente consolidada em áreas como medicina, biologia e ciências da saúde, a RSL se distingue de revisões tradicionais por empregar uma metodologia explícita, rigorosa e auditável, garantindo que o processo seja transparente e replicável por outros pesquisadores (KITCHENHAM, 2004).

Na Engenharia de Software (ES) e, por extensão, na Ciência de Dados, a maioria das pesquisas inicia-se com alguma forma de revisão da literatura. Contudo, para que essa revisão possua valor científico, ela precisa ser completa, imparcial e sistemática (KITCHENHAM; CHARTERS, 2007). A RSL cumpre esse papel ao fornecer um panorama robusto do estado da arte, permitindo identificar lacunas de pesquisa, sintetizar achados de estudos primários e resolver conflitos entre resultados de diferentes trabalhos.

2.1.1 Abordagens Metodológicas: de Kitchenham ao PRISMA

Para guiar a execução de uma RSL, a comunidade de Engenharia de Software tem se apoiado fortemente nas diretrizes propostas por Barbara Kitchenham e seus colaboradores. O trabalho *"Procedures for performing systematic reviews"* (KITCHENHAM, 2004) e sua posterior especialização para a área, *"Guidelines for performing systematic literature reviews in software engineering"* (KITCHENHAM; CHARTERS, 2007), estabeleceram um processo claro e bem definido, dividido em três fases principais:

1. **Planejamento da Revisão:** Definição da questão de pesquisa, elaboração do protocolo da revisão, incluindo a estratégia de busca, os critérios de inclusão e exclusão de estudos e os métodos para extração e síntese de dados.

2. **Condução da Revisão:** Execução da busca nas fontes de dados, seleção dos estudos primários com base nos critérios predefinidos, avaliação da qualidade dos estudos e extração dos dados relevantes.

3. **Relato da Revisão:** Síntese e análise dos dados extraídos e a documentação completa dos resultados em um relatório final.

Este modelo processual, proposto por Kitchenham, foi a base que norteou o desenvolvimento da versão inicial da ferramenta Hermeneutics, focando em apoiar as atividades de busca, seleção e extração de dados.

Contudo, enquanto a abordagem de Kitchenham se concentra no *processo de condução*, o estado da arte na comunicação científica evoluiu para um padrão de *relato* que garante transparência e completude: o **PRISMA (*Preferred Reporting Items for Systematic Reviews and Meta-Analyses*)** (PAGE et al., 2021). O PRISMA não é uma metodologia de execução, mas sim um *framework* de relato que especifica os itens essenciais que devem ser reportados em uma RSL. Sua versão mais recente, o PRISMA 2020, consiste em um *checklist* de 27 itens e um fluxograma detalhado que documenta o fluxo de seleção dos estudos.

A crescente adoção do PRISMA e suas extensões, como o **PRISMA-S** (RETHLEFSEN et al., 2021) para o relato detalhado das estratégias de busca, reflete uma demanda da comunidade científica por maior rigor e reprodutibilidade. O *framework* PRISMA também tem sido explorado em contextos de automação com Inteligência Artificial (IA) e Processamento de Linguagem Natural (PLN) para acelerar etapas como a triagem de artigos e a extração de dados (MARSHALL; WALLACE, 2019).

2.1.2 Comparativo e Sinergia entre Kitchenham e PRISMA

As abordagens de Kitchenham e PRISMA não são mutuamente exclusivas; pelo contrário, são complementares. Kitchenham oferece o "como fazer", um guia processual para o pesquisador de Engenharia de Software. O PRISMA oferece o "como relatar", garantindo que o resultado final seja compreensível, transparente e auditável para a comunidade científica global. A Tabela 2.1 ilustra essa complementaridade.

Tabela 2.1: Comparativo entre as abordagens de Kitchenham e PRISMA.

Critério	**Diretrizes de Kitchenham**	**Framework PRISMA**
Foco Principal	Processo de execução da RSL.	Padrão de relato dos resultados.
Objetivo	Guiar o pesquisador nas etapas.	Garantir transparência e completude.
Artefato Principal	Protocolo de revisão.	Checklist de 27 itens e fluxograma.
Natureza	Metodologia prescritiva.	Framework de relato (checklist).
Aplicação em ES	Amplamente adotada e validada.	Padrão emergente e recomendado.

A combinação das duas abordagens representa o padrão-ouro para a condução de RSLs em Engenharia de Software. Utilizar o processo de Kitchenham para planejar e conduzir a revisão e, em seguida, usar o *checklist* do PRISMA 2020 para estruturar o relatório final, maximiza tanto o rigor metodológico quanto a qualidade da comunicação científica.

2.1.3 Implicações para a Evolução do Hermeneutics 2.0

A compreensão das metodologias de RSL — em especial os processos de Kitchenham e os padrões de relato do PRISMA — fundamenta as decisões arquiteturais da ferramenta Hermeneutics 2.0. Embora o escopo deste trabalho não abranja a implementação completa de funcionalidades específicas do PRISMA, a arquitetura proposta foi projetada para viabilizar futuras extensões nessa direção.

Essa base arquitetural, validada através de cenários de carga simulada em ambiente local (Docker Compose), prepara a plataforma para futuras evoluções funcionais, incluindo a eventual incorporação de suporte nativo ao *framework* PRISMA.

2.2 Técnicas de Processamento de Linguagem Natural

O Processamento de Linguagem Natural (PLN) constitui a base tecnológica para a extração automatizada de informações em revisões sistemáticas. Técnicas como Reconhecimento de Entidades Nomeadas (NER) e Extração de Relações têm sido aplicadas para identificar conceitos-chave em artigos científicos (NADEAU; SEKINE, 2007; LUAN et al., 2018). A mineração de textos, conforme definida por Sebastiani (2001), consiste no processo de extrair conhecimento não trivial a partir de coleções de documentos não estruturados, englobando tarefas como Classificação, Agrupamento (*Clustering*) e Sumarização.

A evolução recente dos *Large Language Models* (LLMs) ampliou significativamente as possibilidades de automação em PLN (BROWN et al., 2020). Diferentemente dos sistemas anteriores, que exigiam grandes volumes de dados rotulados para treinamento supervisionado, os LLMs permitem abordagens *zero-shot* e *few-shot*, onde o modelo executa tarefas com base apenas em instruções em linguagem natural (WEI et al., 2022). Essa capacidade reduz drasticamente o esforço de engenharia de dados e democratiza o acesso a técnicas avançadas de análise textual.

Contudo, a integração efetiva de modelos de PLN em ferramentas de produção requer uma infraestrutura robusta, observável e resiliente. Enquanto os avanços em LLMs oferecem novas capacidades de processamento, a camada arquitetural que os suporta — microsserviços, *mensageria* assíncrona e telemetria distribuída — torna-se igualmente crítica para garantir a escalabilidade e a manutenibilidade do sistema. É nesse contexto que as decisões arquiteturais deste trabalho se fundamentam, priorizando a construção de uma plataforma preparada para incorporar modelos de IA em escala.

2.3 Arquiteturas Orientadas a Domínio e Serviços

O crescimento da complexidade de sistemas e a necessidade de escalar funcionalidades de forma autônoma e resiliente levaram ao surgimento de abordagens arquiteturais que priorizam modularidade, separação de responsabilidades e independência entre componentes. Nesse contexto, destacam-se o *Domain-Driven Design* (DDD), a Arquitetura Orientada a Serviços (SOA) e os microsserviços, que, embora compartilhem fundamentos conceituais, distinguem-se em práticas e escopo de aplicação.

2.3.1 Domain-Driven Design

O *Domain-Driven Design* (DDD), segundo (OZKAYA, 2023), é uma abordagem de modelagem centrada no domínio do problema, na qual a arquitetura do sistema é organizada a partir de unidades conceituais chamadas *bounded contexts*. Cada *bounded context* representa uma fronteira explícita onde um modelo de domínio é definido, isolado e mantido com consistência. Essa segmentação favorece o baixo acoplamento e a alta coesão, elementos essenciais para a manutenção de sistemas evolutivos. DDD também enfatiza a colaboração entre especialistas do domínio e desenvolvedores, promovendo

um vocabulário onipresente (*ubiquitous language*) que permeia tanto o código quanto a comunicação da equipe.

2.3.2 Arquitetura Orientada a Serviços

A Arquitetura Orientada a Serviços (SOA[1]), por sua vez, surgiu como resposta à necessidade de integrar sistemas heterogêneos em grandes organizações. Conforme descrito por (NEWMAN, 2021), SOA promove a composição de aplicações a partir de serviços reutilizáveis, os quais se comunicam geralmente via mensagens padronizadas. Embora compartilhe com a arquitetura de microsserviços o princípio da separação de responsabilidades, SOA tende a operar em ambientes mais formalizados, frequentemente com ferramentas de *middleware* robustas (p.ex., *Enterprise Service Buses* - ESBs), e com foco em interoperabilidade entre domínios organizacionais diversos.

2.3.3 Arquitetura de Microsserviços

A arquitetura de microsserviços, como explorada por Newman (2021), representa uma evolução prática dos conceitos de modularização de software. Ela propõe que uma aplicação seja decomposta em pequenos serviços independentes, cada um responsável por uma funcionalidade específica do sistema e implementado de forma autônoma, podendo inclusive ser escrito em linguagens distintas e implantado separadamente. Cada serviço possui seu próprio ciclo de vida, base de dados (quando necessário) e expõe Interfaces de Programação de Aplicações (APIs) bem definidas, geralmente seguindo o estilo REST (*Representational State Transfer*). Os microsserviços alinham-se naturalmente a práticas modernas de DevOps, automação de deploy e escalabilidade horizontal via contêineres, como Docker e orquestradores como Kubernetes.

2.3.4 Arquitetura Hexagonal (*Ports and Adapters*)

A Arquitetura Hexagonal, ou *Ports and Adapters*, introduz a ideia de que a aplicação se comunica com o mundo exterior através de "portas" (interfaces) e "adaptadores" (implementações). O núcleo da aplicação, que contém a lógica de negócio, não sabe nada

[1] do inglês, *Service Oriented Architecture.*

sobre os adaptadores que o invocam ou que ele invoca (MARTIN, 2020).

As portas definem os contratos que a aplicação oferece (para *driving adapters* como interfaces de usuário) e os contratos de que a aplicação precisa (para *driven adapters* como bancos de dados). Os adaptadores atuam como tradutores, convertendo as requisições externas para o formato compreendido pelas portas do núcleo e vice-versa. Isso promove o **desacoplamento** entre o domínio e a infraestrutura, facilitando a substituição de tecnologias externas sem impactar a lógica de negócio.

As vantagens dessa abordagem incluem:

- **Testabilidade:** O domínio pode ser testado isoladamente com adaptadores simulados (*mocks*).
- **Flexibilidade tecnológica:** Mudanças na infraestrutura não afetam a lógica de negócio.
- **Clareza das dependências:** O fluxo de dependências sempre aponta para dentro (em direção ao domínio).

2.3.5 Evolução Arquitetural

A modernização da ferramenta Hermeneutics será orientada por uma transição arquitetural progressiva, seguindo os princípios de decomposição de sistemas monolíticos descritos por (NEWMAN, 2020). Em vez de uma reescrita completa, adotar-se-á uma estratégia incremental baseada na extração gradual de funcionalidades para serviços independentes, utilizando padrões como isolamento de componentes, encapsulamento de chamadas internas e construção de fachadas de integração. Essa abordagem visa mitigar riscos técnicos e operacionais comuns em reestruturações radicais, ao mesmo tempo que permite a adoção controlada de tecnologias modernas, automação de implantação e mecanismos de observabilidade distribuída. Complementarmente, os conceitos de *Domain-Driven Design* (DDD) serão aplicados para delimitar os domínios funcionais da aplicação em *bounded contexts* bem definidos, servindo como base para o desenho dos microsserviços e favorecendo baixa dependência entre módulos. Cada componente extraído será reimplementado como um serviço conteinerizado e autônomo, com APIs e persistência segregadas, alinhando-se à proposta de modularidade, escalabilidade e resiliência. Essa nova organização não só facilita a manutenção e evolução da ferramenta, como também

a prepara para futuras extensões funcionais, como a incorporação de modelos preditivos, algoritmos de ranqueamento e mecanismos semânticos mais sofisticados.

2.4 Padrões de Consistência em Sistemas Distribuídos

A transição de uma arquitetura monolítica para microsserviços exige um mecanismo robusto para garantir a consistência dos dados. O desafio surge porque as garantias transacionais tradicionais, como *Two-Phase Commit* (2PC), são inaplicáveis a arquiteturas reativas e distribuídas devido ao seu caráter de bloqueio, que impacta a resiliência (ŠTEFANKO et al., 2019). Além disso, como cada microsserviço tipicamente gerencia seu próprio banco de dados, as garantias ACID tradicionais não se aplicam a múltiplos armazenamentos independentes.

2.4.1 ACID *versus* BASE

Enquanto arquiteturas monolíticas seguem os princípios **ACID** — *Atomicidade* (tudo ou nada), *Consistência* (transição entre estados válidos), *Isolamento* (transações não interferem umas nas outras) e *Durabilidade* (dados persistem após a conclusão) —, sistemas distribuídos frequentemente adotam uma abordagem mais flexível.

Para alcançar maior disponibilidade em ambientes distribuídos, o modelo de transação **BASE** relaxa o requisito de Isolamento:

- ***Basically Available*** (Basicamente Disponível): O sistema prioriza a disponibilidade em detrimento da consistência imediata.
- ***Soft State*** (Estado Suave): O estado do sistema pode mudar ao longo do tempo, mesmo sem entrada explícita.
- ***Eventual Consistency*** (Consistência Eventual): O sistema garante que, se nenhuma nova atualização for feita, os dados eventualmente atingirão um estado consistente.

2.4.2 Padrão Saga

O **Padrão Saga** gerencia transações distribuídas, dividindo-as em uma sequência de **transações locais**. Em cada etapa, um serviço executa sua operação, atualiza atomi-

camente seu próprio banco de dados e inicia o próximo passo da sequência através da publicação de um evento (ŠTEFANKO et al., 2019).

Caso uma transação local falhe, a saga executa uma série de **transações compensatórias**, que são operações responsáveis por desfazer semanticamente o trabalho realizado pelas etapas anteriores, revertendo o sistema a um estado consistente.

O padrão pode ser implementado por meio de duas abordagens principais (RICHARDSON, 2025):

- **Saga Orquestrada:** Um coordenador central (*orquestrador*) controla a sequência de passos e decide quando acionar compensações em caso de falha.
- **Saga Coreografada:** Não há coordenador central. Cada serviço reage a eventos publicados por outros serviços, formando uma coreografia descentralizada.

A Saga Coreografada reduz o acoplamento entre serviços ao eliminar o ponto único de coordenação, embora exija maior disciplina na definição de eventos de compensação.

2.4.3 Padrão *Database-per-Service*

Complementando o padrão Saga, a estratégia *Database-per-Service* determina que cada microsserviço seja responsável exclusivamente por seu próprio esquema de banco de dados. Essa abordagem elimina dependências ocultas e permite que cada serviço evolua seu modelo de dados de forma independente (Amazon Web Services, 202X).

As principais vantagens incluem:

- **Autonomia de dados:** Cada serviço pode escolher a tecnologia de banco de dados mais adequada às suas necessidades.
- **Isolamento de falhas:** Problemas de performance ou disponibilidade em um banco não afetam os demais serviços.
- ***Deploy* independente:** Alterações no esquema de um serviço não exigem coordenação com outros serviços.

O isolamento dos dados, contudo, exige a garantia de **consistência eventual** através do Padrão Saga, uma vez que transações distribuídas tradicionais não são aplicáveis.

2.5 Observabilidade e a Adoção do OpenTelemetry

A **observabilidade** é a capacidade de compreender o estado interno de um sistema a partir da análise de suas saídas externas, permitindo questionar o sistema sem conhecer seus detalhes internos. Em sistemas de software, isso significa entender o estado interno a partir de dados de telemetria, como ***traces*, métricas e logs**. Ela é fundamental para solucionar problemas novos e inesperados ("desconhecidos") e para responder à pergunta "Por que isso está acontecendo?". Um sistema é considerado devidamente instrumentado para observabilidade quando os desenvolvedores não precisam adicionar mais instrumentação para solucionar um problema, pois já possuem todas as informações necessárias.(OpenTelemetry Community, 2024)

Em ambientes modernos, especialmente com **arquiteturas de microsserviços** e sistemas distribuídos, a observabilidade é crucial para garantir a confiabilidade e o desempenho. Ela permite a avaliação em tempo real do comportamento dos subsistemas, fornecendo o **contexto e a correlação de sinais** necessários para depurar sistemas complexos o mais rápido possível. O objetivo é reduzir o **Mean Time to Detect (MTTD)**, **Mean Time to Know (MTTK)** e **Mean Time to Verify (MTTV)**.

2.5.1 Os Pilares da Observabilidade

A telemetria, que é a base da observabilidade, é composta por três sinais principais, frequentemente chamados de "pilares":

- **Logs:** Mensagens com registro de tempo emitidas por serviços ou componentes. Podem ser estruturadas (recomendado) ou não estruturadas. Embora logs tradicionais possam carecer de contexto, eles se tornam muito mais úteis quando associados a um **span** ou correlacionados com um **trace**. São frequentemente usados para determinar a causa raiz de um problema.[2]

- **Métricas:** Agregações de dados numéricos sobre a infraestrutura ou aplicação ao longo de um período de tempo. Exemplos incluem a taxa de erro do sistema, a utilização da CPU e a taxa de requisições. As métricas fornecem sinais estáveis e completos, ideais para monitoramento de **indicadores-chave de desempenho (KPIs)** e

[2]OTel Logs <https://opentelemetry.io/docs/concepts/signals/logs/>

para acionar alertas.[3]

- **Traces (Rastreamento Distribuído):** Registram os caminhos que as requisições percorrem em sistemas distribuídos complexos, como aplicações de microsserviços. Um trace é composto por um ou mais **spans**.[4]

  - **Spans:** Representam uma unidade de trabalho ou operação específica dentro de um trace. Contêm nome, dados de tempo (início e fim), mensagens de log estruturadas (eventos) e outros metadados (**atributos**) que descrevem a operação. Spans podem ser aninhados, com spans filhos representando sub-operações, permitindo uma captura mais precisa do trabalho realizado.[5]

  - **Context Propagation:** É o conceito central que permite o rastreamento distribuído, correlacionando spans entre si e montando-os em um trace completo, independentemente de onde são gerados.[6]

2.5.2 OpenTelemetry: O Padrão Aberto para Observabilidade

O **OpenTelemetry (OTel)** é uma estrutura e kit de ferramentas de **código aberto, agnóstico a fornecedores e ferramentas**, projetado para facilitar a geração, exportação e coleta de dados de telemetria, como *traces*, métricas e logs. Seu principal objetivo é permitir a **instrumentação fácil e ubíqua** de aplicações e sistemas, independentemente da linguagem de programação, infraestrutura ou ambientes de execução (OpenTelemetry Community, 2025).

O OpenTelemetry surgiu da fusão de dois projetos anteriores da *Cloud Native Computing Foundation* (CNCF), **OpenTracing e OpenCensus**, que visavam resolver a falta de um padrão para instrumentação e envio de dados de telemetria. Ao combinar as forças de ambos, o OTel se tornou o padrão líder da indústria para rastreamento distribuído e, mais amplamente, para coleta e transmissão de dados de observabilidade. A missão do OpenTelemetry é "habilitar uma observabilidade eficaz, tornando a telemetria de alta qualidade e portátil ubíqua".

[3] OTel Métricas <https://opentelemetry.io/docs/concepts/signals/metrics/>
[4] OTel Traces <https://opentelemetry.io/docs/concepts/signals/traces/>
[5] OTel Spans <https://opentelemetry.io/docs/concepts/spans/>
[6] Otel *Context Propagation* <https://opentelemetry.io/docs/concepts/context-propagation/>

2.5.2.1 Componentes Chave do OpenTelemetry:

O OpenTelemetry é composto por diversos componentes que trabalham em conjunto para entregar uma solução de observabilidade robusta e padronizada.

- **Especificação (*Specification*):** Descreve os requisitos e expectativas para todas as implementações em diferentes linguagens. Define APIs, SDKs e formatos de dados (como OTLP e convenções semânticas). A versão 1.0.0 da especificação estabilizou o rastreamento, contexto e *baggage*.
- **APIs (*Application Programming Interfaces*):** Interfaces públicas que definem como gerar dados de telemetria.
- **SDKs (*Software Development Kits*):** Implementações específicas de linguagem das APIs, que fornecem funcionalidades para configuração, processamento de dados e exportação.
- **Bibliotecas de Instrumentação (*Instrumentation Libraries*):** Componentes que geram dados de telemetria automaticamente a partir de bibliotecas e *frameworks* populares, sem a necessidade de modificar o código-fonte da aplicação.
- **Conectores (*Exporters*):** Responsáveis por enviar os dados de telemetria para um consumidor, que pode ser a saída padrão (para depuração), o **OpenTelemetry Collector** ou qualquer *backend* (*open source* ou comercial). Os exportadores OTLP (OpenTelemetry Protocol) são projetados para o modelo de dados do OpenTelemetry, garantindo a exportação sem perda de informações.
- **OpenTelemetry Collector:** Um *proxy* agnóstico a fornecedores que pode receber, processar e exportar dados de telemetria. Ele suporta múltiplos formatos (OTLP, Jaeger, Prometheus) e pode enviar dados para um ou mais *backends*, além de processar e filtrar dados antes da exportação. É essencial para integrar sistemas legados e aplicar transformações globais.
- **Propagadores (*Cross-Service Propagators*):** Mecanismos que movem dados (como contexto de trace e *baggage*) entre serviços e processos. São fundamentais para construir informações causais em sistemas distribuídos.
- **Baggage:** Informação contextual no formato chave-valor que é propagada entre os sinais de telemetria. O *Baggage* permite que dados como IDs de usuário ou de

conta sejam passados através de serviços e processos, tornando-os disponíveis para serem adicionados a traces, métricas ou logs mais adiante na cadeia de requisições, o que facilita análises mais profundas. É um armazenamento chave-valor separado dos atributos, e precisa ser explicitamente lido e adicionado aos atributos dos spans, métricas ou logs quando desejado.[7]

2.5.2.2 Vantagens da Adoção do OpenTelemetry para a Observabilidade:

A adoção do OpenTelemetry oferece várias vantagens significativas para a observabilidade:

- **Padronização e Portabilidade:** Elimina o "*vendor lock-in*", permitindo que as organizações gerem dados de telemetria padronizados que podem ser enviados para qualquer *backend* de observabilidade.
- **Contexto e Correlação Abrangentes:** O OpenTelemetry é projetado para permitir que logs, traces e métricas sejam correlacionados de maneiras significativas, oferecendo uma visão holística do sistema.
- **Redução do Tempo de Resolução de Incidentes:** Ao fornecer contexto de depuração padronizado e automático, o OTel reduz drasticamente o MTTK, capacitando engenheiros a identificar a causa raiz dos incidentes mais rapidamente.
- **Melhoria na Qualidade dos Dados:** As **convenções semânticas** definem um esquema de nomenclatura padrão para tipos de dados de telemetria comuns, facilitando a correlação entre sinais e serviços e garantindo que a telemetria seja uniformemente compreendida.
- **Otimização de Custos e Desempenho:** Técnicas como amostragem de *traces* (*sampling*) e processamento de dados no Collector ajudam a reduzir o volume de telemetria, minimizando os custos de transferência, armazenamento e processamento sem comprometer a observabilidade.

Ao adotar o OpenTelemetry, as equipes podem instrumentar suas aplicações uma única vez e exportar dados para diversas ferramentas, garantindo uma base sólida e flexível para a observabilidade de sistemas distribuídos.

[7] Baggage <https://opentelemetry.io/docs/concepts/baggage/>

3 TRABALHOS RELACIONADOS

Neste capítulo, são apresentados os trabalhos relacionados que embasam e posicionam a contribuição deste estudo. A análise é estruturada em dois eixos: (1) ferramentas existentes para apoio à Revisão Sistemática da Literatura, avaliadas sob critérios arquiteturais; e (2) estudos sobre migração de sistemas monolíticos para microsserviços, que fundamentam a abordagem de modernização adotada. A caracterização detalhada do Hermeneutics v1, incluindo sua arquitetura, limitações e motivação para a modernização, é apresentada no Capítulo 4.

3.1 Ferramentas de Revisão Sistemática da Literatura

A análise de ferramentas existentes para apoio à RSL é fundamental para posicionar a contribuição deste trabalho. Diferentemente de abordagens que priorizam funcionalidades de processamento de linguagem natural, esta análise foca em **critérios arquiteturais**, essenciais para a evolução e sustentabilidade de ferramentas científicas:

- **Plataforma e Arquitetura:** Desktop (monolítico) versus Web/Cloud (potencialmente distribuído);
- **Licenciamento:** *Open Source* versus Proprietário — impacta a capacidade de evolução pela comunidade;
- **Extensibilidade:** Disponibilidade de APIs ou mecanismos de integração com outros sistemas;
- **Escalabilidade:** Capacidade de processar grandes volumes de artigos simultaneamente.

A Tabela 3.1 apresenta uma síntese comparativa das principais ferramentas identificadas na literatura, categorizadas segundo esses critérios.

A análise revela que a maioria das ferramentas gratuitas e de código aberto apresenta arquitetura monolítica, o que limita sua capacidade de evolução e integração com tecnologias modernas. Ferramentas comerciais como EPPI-Reviewer e Rayyan oferecem maior escalabilidade, porém, o modelo proprietário restringe a extensibilidade pela comunidade acadêmica.

Tabela 3.1: Comparativo de ferramentas de RSL sob critérios arquiteturais.

Ferramenta	Plataforma	Licença	API/Extensível	Escalabilidade	Arquitetura
StArt	Desktop	Gratuita	Não	Limitada	Monolítica
Parsifal	Web	Gratuita	Não	Moderada	Monolítica
Rayyan (OUZZANI et al., 2016)	Web/Mobile	Freemium	Parcial	Alta	Cloud
EPPI-Reviewer	Web	Comercial	Sim	Alta	Cloud
Hermeneutics v1 (SILVA, 2021)	Web	Open Source	Não	Limitada	Monolítica
Hermeneutics v2	Web	Open Source	Sim (Kafka/REST)	Projetada	Microsserviços

Nesse contexto, o Hermeneutics v2 se diferencia ao combinar **código aberto**[1], com uma **arquitetura de microsserviços**, viabilizando tanto a escalabilidade horizontal quanto a extensibilidade por meio de APIs padronizadas e *mensageria* assíncrona.

3.2 Migração de Sistemas Monolíticos para Microsserviços

A crescente adoção da arquitetura de microsserviços em ambientes de software modernos motivou uma série de estudos acadêmicos e industriais voltados para a transição de sistemas legados com arquitetura monolítica. Esta seção explora os principais motivos para essa migração, apresenta estudos de caso documentados na literatura, discute padrões recorrentes aplicados nesse processo e posiciona o presente trabalho diante das lacunas observadas.

3.2.1 Motivações para a Migração de Monólitos

A literatura especializada aponta diversas razões técnicas e organizacionais que impulsionam a migração de sistemas monolíticos para arquiteturas baseadas em microsserviços. Entre as mais recorrentes, destacam-se a necessidade de escalabilidade granular, redução do acoplamento entre módulos, maior autonomia de equipes, e facilitação de *deploys* independentes e contínuos (NEWMAN, 2020). A arquitetura monolítica, embora eficaz em fases iniciais de sistemas, tende a dificultar a evolução contínua à medida

[1] A licença GNU AGPLv3 (Affero General Public License) foi selecionada estrategicamente para o Hermeneutics 2.0. Ao contrário de licenças permissivas (como MIT ou Apache), a AGPL assegura que quaisquer modificações disponibilizadas como serviço em rede (SaaS) devem ter seu código-fonte aberto à comunidade. Essa cláusula protege o ativo intelectual da universidade contra a exploração comercial proprietária por terceiros ("*vendor lock-in*") sem a devida contribuição de volta ao ecossistema de pesquisa aberta

que o código cresce e o acoplamento entre funcionalidades se intensifica (NEWMAN, 2021). Nesse contexto, a abordagem evolutiva defendida por Newman (2020) propõe uma transição gradual e incremental, mitigando os riscos associados a reescritas completas. Complementarmente, Ozkaya (2023) ressalta a importância de se alinhar a modularidade técnica com limites de domínio bem definidos, utilizando princípios de *Domain-Driven Design* (DDD) para garantir que a nova estrutura reflita as regras e responsabilidades de negócio com maior clareza e manutenibilidade (OZKAYA, 2023).

3.2.2 Padrões e Práticas Recorrentes na Literatura

A literatura especializada descreve diversos padrões arquiteturais comumente aplicados durante a migração de sistemas monolíticos para microsserviços, visando modularidade, escalabilidade e manutenção evolutiva.

O padrão *Strangler Fig*[2] possibilita uma transição incremental a partir da aplicação existente, por meio da extração progressiva de funcionalidades para novos serviços, permitindo que o monólito seja gradualmente descontinuado sem interromper a operação global.

Já o padrão *API Gateway*[3] atua como um ponto de entrada unificado para clientes externos, gerenciando roteamento de chamadas, agregações de respostas, autenticação e segurança, além de reduzir a complexidade no cliente final.

O padrão de *Banco de Dados por Serviço*[4] promove isolamento de persistência ao atribuir a cada microsserviço seu próprio esquema ou instância de armazenamento, eliminando dependência centralizada e facilitando escalabilidade e governança de dados.

O *Event Sourcing*[5] enfatiza o armazenamento de eventos de estado como fonte primária de dados, em vez de registros mutáveis, possibilitando audibilidade, reconstrução de estado e consistência eventual em conjunto com *CQRS* (*Command Query Responsibility Segregation*).

Finalmente, o *CQRS*[6] separa explicitamente os modelos de leitura e escrita, permitindo otimização independente do desempenho e escalabilidade para cada fluxo, frente

[2] *Strangler Fig Application* <https://martinfowler.com/bliki/StranglerFigApplication.html>

[3] AWS API Gateway <https://docs.aws.amazon.com/prescriptive-guidance/latest/modernization-integrating-microservices/api-gateway-pattern.html>

[4] *Pattern database-per-service* <https://microservices.io/patterns/data/database-per-service.html>

[5] AWS Event Sourcing <https://docs.aws.amazon.com/prescriptive-guidance/latest/modernization-data-persistence/service-per-team.html>

[6] Microsoft CQRS <https://learn.microsoft.com/en-us/azure/architecture/patterns/cqrs>

às operações tradicionalmente mescladas em um único CRUD.

Esses padrões surgem com frequência na literatura como parte essencial dos processos de migração bem-sucedida e embasam a proposta arquitetural apresentada neste trabalho.

3.2.3 Estudos de Caso Documentados na Literatura

Diversos trabalhos relatam experiências práticas de migração, documentando os desafios enfrentados, os padrões utilizados e os benefícios observados após a adoção de microsserviços.

Tabela 3.2: Estudos de caso acadêmicos sobre migração de monólitos para microsserviços.

ID	Citação	Tipo	Domínio	Padrões	Tecnologias
1	(JUNIOR, 2022)	Monografia (Esp.)	ERP Web	API Gateway, DB por serviço	Spring Boot, Docker
2	(STRADOLINI, 2022)	TCC (Grad.)	e-Commerce	Strangler Fig, Event Sourcing	Spring Boot, RabbitMQ
3	(LIMA, 2025)	TCC (Grad.)	Sistema Acadêmico	CQRS, DB por serviço	Node.js, MongoDB

O trabalho de Junior (2022) apresenta um relato de experiência sobre a extração de módulos de um sistema ERP monolítico, com foco em desacoplamento arquitetural e implantação de microsserviços. A aplicação de padrões como API Gateway e banco de dados por serviço possibilitou uma transição mais controlada, permitindo *deploys* independentes e maior autonomia das equipes envolvidas.

Stradolini (2022) explora um caso prático de migração de um módulo de pagamentos de um sistema e-*commerce*, aplicando conceitos de DDD, SOA e eventos para alcançar resiliência e flexibilidade. O padrão Strangler Fig foi utilizado para permitir a substituição gradual de componentes críticos, enquanto o *Event Sourcing* foi adotado como estratégia de consistência e rastreabilidade.

Lima (2025) conduz um estudo comparativo entre as abordagens *Monolith First* e *Microservices First* no desenvolvimento de aplicações acadêmicas. O trabalho utiliza métricas de tempo, esforço e complexidade técnica para avaliar as duas estratégias, demonstrando cenários nos quais a abordagem incremental se mostrou mais eficaz para equipes reduzidas.

3.2.4 Lacunas e Contribuições Esperadas

A análise dos trabalhos relacionados revela uma lacuna significativa na interseção entre ferramentas de RSL e práticas modernas de engenharia de software. **Embora existam ferramentas com capacidades de IA** (como Rayyan, que oferece sugestões de inclusão/exclusão), **elas operam como caixas-pretas proprietárias**, cujo código-fonte e arquitetura não estão acessíveis à comunidade acadêmica. **Faltava uma arquitetura aberta e distribuída**, preparada para a incorporação evolutiva de modelos de linguagem em larga escala.

Adicionalmente, apesar da vasta literatura sobre migração de sistemas legados para microsserviços, são escassos os estudos que aplicam essas práticas especificamente em ferramentas voltadas ao apoio de revisões sistemáticas da literatura. A maioria dos casos documentados concentra-se em domínios como e-*commerce*, gestão educacional ou serviços web corporativos.

O presente trabalho busca preencher essa dupla lacuna ao: (1) disponibilizar uma ferramenta de RSL **open source** sob licença AGPL, garantindo que evoluções permaneçam abertas à comunidade; e (2) aplicar princípios modernos de engenharia de software – incluindo *Domain-Driven Design*, Arquitetura Hexagonal, Padrão Saga Coreografada e Observabilidade com OpenTelemetry – à evolução da ferramenta Hermeneutics, cujo foco está na análise semântica de artigos científicos.

Ao desacoplar os componentes da ferramenta original e torná-los implantáveis independentemente, espera-se não apenas melhorar sua escalabilidade e manutenção, mas também preparar uma **base arquitetural sólida para a futura incorporação de modelos de linguagem de grande escala (LLMs)**, de forma evolutiva e com baixo risco de regressão. Essa preparação inclui a instrumentação completa da plataforma com telemetria distribuída, viabilizando análises de *FinOps* e monitoramento de custos computacionais em granularidade de transação.

4 HERMENEUTICS 1.0: CARACTERIZAÇÃO DO LEGADO

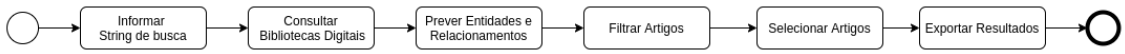
Este capítulo caracteriza a versão original da ferramenta, detalhando sua arquitetura, tecnologias, funcionalidades e as limitações que motivaram a modernização proposta neste trabalho.

4.1 Visão Geral e Propósito

A primeira versão da ferramenta, desenvolvida como Trabalho de Conclusão de Curso na graduação (SILVA, 2021), teve como propósito demonstrar a viabilidade de uma solução *open source* para apoio à RSL, integrando busca automatizada em bases bibliográficas com processamento semântico de resumos.

Entre as funcionalidades implementadas na versão original, destacam-se: busca automatizada em bases bibliográficas (arXiv, IEEE, Scopus, ScienceDirect, Springer); visualização de resultados com rotulação semântica de entidades; algoritmo de ranqueamento baseado em termos de busca e contexto semântico; e exportação BibTeX de resultados selecionados.

Figura 4.1: Hermeneutics v1 — Fluxograma Principal



Fonte: (SILVA, 2021).

4.2 Arquitetura Monolítica e Tecnologias (AS-IS)

4.2.1 Stack Tecnológica

O *frontend* foi desenvolvido com uso de *Bootstrap 5*[1], garantindo compatibilidade entre navegadores e *layout* responsivo. O *backend* utilizou *Python* com *Django*[2], integrando bibliotecas como *spaCy*[3] para mineração textual e anotação semântica. Para

[1] Bootstrap Framework 5.0.1 <https://getbootstrap.com/>.
[2] Django Framework <https://www.djangoproject.com/>
[3] SpaCy <https://spacy.io/>

mensageria assíncrona, foram adotados *Celery*[4] e *RabbitMQ*[5], enquanto a persistência de dados evoluiu de *SQLite*[6] para *PostgreSQL*[7] na versão em produção.

4.2.2 Estrutura do Monólito

A versão inicial foi desenvolvida como um protótipo funcional com arquitetura monolítica baseada no padrão Model-Template-View (MTV)[8] do *framework Django*, adotando boas práticas de Engenharia de Software. Foram elaborados diagramas UML de classes e de relacionamento com o banco de dados para orientar a implementação da lógica e do modelo de dados.

A arquitetura concentrava todas as responsabilidades em um único processo Django, incluindo:

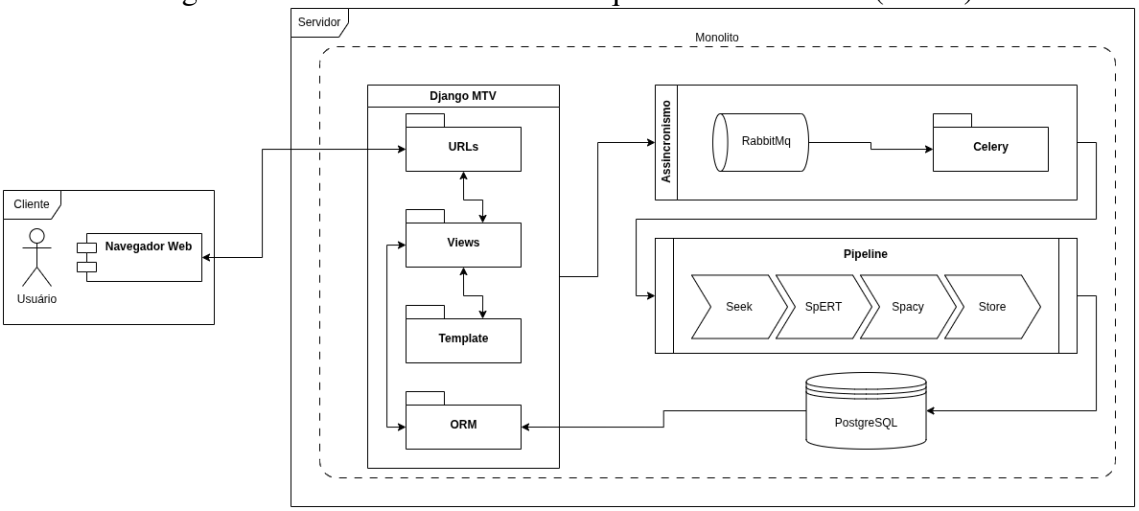
- **Interface Web:** Renderização de *templates* HTML para interação com o pesquisador.
- **Lógica de Negócio:** Regras de busca, seleção e ranqueamento de artigos.
- **Integração com Bibliotecas Digitais:** Comunicação direta com APIs externas (arXiv, IEEE, Scopus).
- **Processamento de Linguagem Natural:** Extração de entidades nomeadas e relacionamentos via SpERT/SciERC com renderização via SpaCy.
- **Persistência:** Acesso direto ao banco de dados PostgreSQL compartilhado.
- **Processamento Assíncrono:** Filas de tarefas gerenciadas por Celery e RabbitMQ.

Para garantir responsividade durante os testes de usabilidade, foi implementado um fluxo assíncrono usando AJAX para atualização dinâmica da interface, além do uso do Celery em conjunto com o RabbitMQ para orquestrar tarefas em segundo plano, como o processo de anotação semântica dos artigos. O banco de dados utilizado inicialmente foi o SQLite, posteriormente migrado para o PostgreSQL na versão de testes em servidor remoto.

[4]Celery 5.1.2 <https://docs.celeryproject.org/en/stable/>.
[5]RabbitMQ 3.6.10 <https://www.rabbitmq.com/>.
[6]SQLite <https://www.sqlite.org/>
[7]PostgreSQL <https://www.postgresql.org/>
[8]Django FAQs MTV x MVC <https://docs.djangoproject.com/en/2.2/faq/general/>.

Apesar de ter sido concebida com um esboço para containerização via Docker, as limitações de tempo e escopo da graduação direcionaram a implantação para ambientes tradicionais (*localhost* e VM), sem orquestração formal ou isolamento completo dos serviços.

Figura 4.2: Hermeneutics v1 — Arquitetura Monolítica (AS-IS).



Fonte: (SILVA, 2021).

4.3 Limitações Identificadas

A análise crítica da arquitetura original, documentada no *Architecture Decision Record* ADR-001, identificou as seguintes limitações:

(i) **Alto Acoplamento entre Módulos:** Os módulos de busca (*crawlers*), processamento semântico (NLP) e interface web compartilhavam o mesmo processo e base de código. Alterações em um módulo frequentemente impactavam outros, dificultando a manutenção e aumentando o risco de regressões.

(ii) **Escalabilidade Limitada:** O módulo de Processamento de Linguagem Natural (NLP), computacionalmente intensivo, competia por recursos com a interface web. Não era possível escalar apenas o componente de anotação semântica sem escalar toda a aplicação, resultando em uso ineficiente de recursos computacionais.

(iii) **Fila Única de Processamento:** O uso de Celery com RabbitMQ resultava em uma fila única de processamento, onde todas as tarefas assíncronas (busca, anotação, persistência) competiam pelos mesmos *workers*. Em picos de carga, isso gerava gargalos e potencial perda de mensagens, conforme documentado no ADR-002.

(iv) **Banco de Dados Compartilhado:** Todos os módulos acessavam diretamente o mesmo esquema PostgreSQL, criando dependências ocultas entre componentes. A evolução do esquema de dados de um módulo exigia coordenação com todos os outros, dificultando a independência de desenvolvimento (ADR-003).

(v) **Ausência de Observabilidade:** A depuração de problemas era limitada a logs locais e inspeção manual do banco de dados. Não havia instrumentação para rastreamento de requisições através dos diferentes módulos, tornando a identificação de gargalos e falhas um processo moroso e impreciso (ADR-005).

4.4 Motivação para a Modernização

As limitações identificadas motivaram a reestruturação da ferramenta para uma arquitetura de microsserviços. Sob a ótica de **MLOps**[9], a arquitetura monolítica impedia a escalabilidade horizontal específica do módulo de inferência — o serviço de anotação semântica que consome GPU/CPU intensivamente — prejudicando toda a aplicação durante picos de processamento. A separação em microsserviços permite escalar o "cérebro" (serviço de IA/Anotação) independentemente do "corpo" (serviços de CRUD e Interface).

Os objetivos específicos da modernização incluem:

1. **Escalabilidade Granular:** Permitir que o módulo de anotação semântica seja escalado independentemente da interface web, atendendo às demandas de inferência em larga escala.

2. **Resiliência:** Garantir que falhas em um serviço não comprometam toda a aplicação.

3. **Manutenibilidade:** Facilitar a evolução independente de cada módulo com ciclos de *deploy* autônomos.

4. **Observabilidade:** Implementar instrumentação completa para rastreamento distribuído e monitoramento de métricas.

A estratégia de modernização segue os princípios de migração incremental descritos por (NEWMAN, 2020), decomposição por *Bounded Contexts* do DDD (EVANS,

[9]MLOps é definido como "a extensão da metodologia DevOps para incluir ativos de Machine Learning e Ciência de Dados como cidadãos de primeira classe dentro da ecologia DevOps". Disponível em: <https://aws.amazon.com/pt/what-is/mlops/>

2016) e aplicação de padrões arquiteturais consolidados como *Clean Architecture* (MARTIN, 2020).

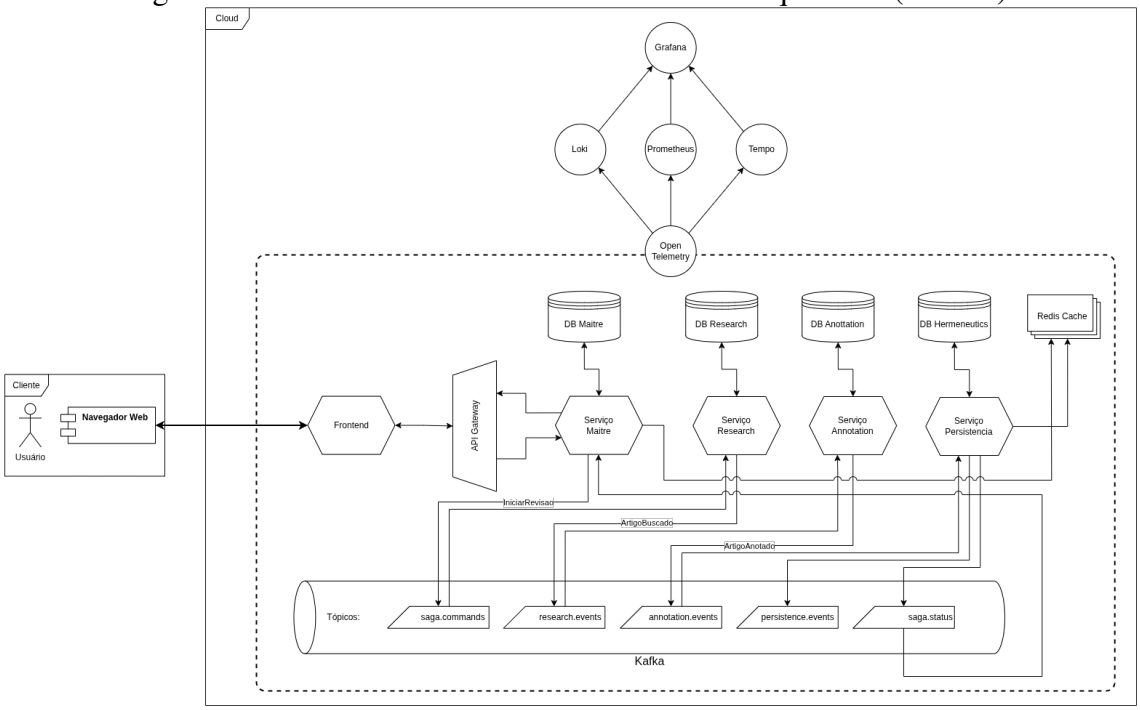
5 PROPOSTA ARQUITETURAL E MODERNIZAÇÃO

Este capítulo apresenta a arquitetura proposta para o Hermeneutics 2.0, considerando as limitações de acoplamento e escalabilidade detalhadas no Capítulo 4. A Seção 5.1 detalha a nova arquitetura (TO-BE). A Seção 5.2 documenta as decisões arquiteturais fundamentais que guiaram essa transição. Por fim, a Seção 5.3 descreve os padrões implementados e o ambiente de avaliação.

5.1 Nova Arquitetura (TO-BE)

A arquitetura TO-BE, ilustrada na Figura 5.1, representa a concretização dos princípios de *Domain-Driven Design* (DDD) e microsserviços discutidos na fundamentação teórica. Diferentemente da versão monolítica (AS-IS), onde o acoplamento temporal e de dados limitava a escalabilidade, a nova arquitetura distribui as responsabilidades em serviços autônomos, comunicando-se de forma reativa através de um barramento de eventos robusto.

Figura 5.1: Hermeneutics v2 — Visão Geral da Arquitetura (TO-BE).



Fonte: Autor.

A estrutura apresentada materializa a aplicação dos estilos arquiteturais *Clean Architecture* e Hexagonal (*Ports and Adapters*), garantindo que o núcleo de negócio de cada serviço permaneça isolado da infraestrutura externa.

5.1.1 Bounded Contexts e Decomposição em Serviços

A decomposição do sistema segue os *Bounded Contexts* identificados na etapa de modelagem estratégica com DDD. Cada contexto representa um futuro microsserviço ou grupo coeso de microsserviços, com responsabilidades bem definidas:

- **Contexto de Ingestão de Artigos (Research):** Responsável pela busca automatizada em bibliotecas digitais (arXiv, IEEE Xplore, Scopus, ScienceDirect, Springer) e coleta inicial dos metadados.
- **Contexto de Anotação Semântica (Annotation):** Encarregado do processamento de resumos para extração e classificação de Entidades Nomeadas e Relacionamentos utilizando técnicas de PLN.
- **Contexto de Persistência de Dados (Persistence):** Gerencia o armazenamento definitivo de todos os dados do sistema, incluindo artigos processados, entidades e relacionamentos extraídos.
- **Contexto de Orquestração de Processos (Maître):** Coordena os fluxos de trabalho assíncronos através do padrão Saga, garantindo consistência eventual entre os serviços.
- **Contexto de Interface do Usuário (Web UI/BFF):** Responsável pela apresentação dos dados e interação com o pesquisador, atuando como *Backend for Frontend*.

5.1.2 Componentes e Fluxo de Dados

5.1.2.1 Ponto de Entrada e API Gateway

O acesso do cliente (navegador web) não ocorre mais diretamente nos serviços de negócio. Foi introduzido um **API Gateway** baseado em Nginx que atua como a porta de entrada única do sistema. Este componente abstrai a complexidade da rede de microsserviços, gerenciando o roteamento através de subdomínios (conforme ADR-021):

- `web.localhost` → Web UI (Django 4.2 LTS)
- `maitre.localhost` → Serviço Maître (FastAPI)
- `research.localhost` → Serviço Research (FastAPI)

- `annotation.localhost` → Serviço Annotation (FastAPI)
- `grafana.localhost` → Dashboard de Observabilidade

5.1.2.2 Serviços de Domínio (Microsserviços)

Os hexágonos centrais na Figura 5.1 representam os serviços autônomos, cada um encapsulando um domínio específico e possuindo seu próprio ciclo de vida:

- **Serviço Maître (Orquestração):** Atua como o coordenador da Saga e *aggregator* de dados (padrão Facade, conforme ADR-018). Gerencia o estado da transação distribuída, emitindo comandos para o tópico `saga.commands` e escutando o status final em `saga.status`. Implementado com FastAPI e Python 3.12 LTS (ADR-006).
- **Serviço Research (Ingestão):** Representa a camada de adaptação (*Anti-Corruption Layer*) para as bibliotecas digitais externas. Consome comandos de busca e publica os metadados brutos no tópico `research.events`, isolando o restante do sistema das instabilidades das APIs externas.
- **Serviço Annotation (Processamento Semântico):** É o núcleo de inteligência da ferramenta. Ao contrário da versão anterior, que competia por recursos na mesma fila do Celery, este serviço agora escala independentemente, consumindo eventos de `annotation.events` conforme sua capacidade de processamento (*Backpressure*).
- **Serviço Persistence (Escrita Definitiva):** Centraliza a responsabilidade de gravação definitiva. Escuta os resultados anotados e os persiste, emitindo o evento final de `persistence.events` que sinaliza ao Maître a conclusão do fluxo.

5.1.2.3 Estratégia de Dados: Database per Service

Abandonando o banco de dados compartilhado do monólito, a arquitetura adota o padrão *Database per Service* (ADR-003). Como visualizado na arquitetura:

- **DB Maître, DB Research, DB Annotation, DB Persistence:** Cada serviço possui seu próprio esquema ou instância PostgreSQL isolada, garantindo que nenhum serviço possa acessar diretamente as tabelas de outro, forçando a comunicação via APIs ou eventos.

- **Redis Cache:** Introduzido para otimizar leituras frequentes, gerenciar estados transitórios e armazenar sessões da Web UI (ADR-029), reduzindo a latência nas consultas do *Frontend*.

5.1.3 Evolução da Mensageria: De Celery para Kafka

Uma das mudanças mais críticas ilustradas no diagrama é a substituição do mecanismo de filas baseado em Celery/RabbitMQ pelo **Apache Kafka** (ADR-002).

No modelo AS-IS, o Celery gerenciava tarefas em uma fila de trabalho volátil, adequada para processamento em *background* simples, mas limitada para orquestração de eventos complexos e persistência de logs de eventos. Na arquitetura TO-BE, o Kafka assume o papel de *Event Bus* durável e de alta vazão.

A Figura 5.1 detalha a topologia dos tópicos que implementam a coreografia da Saga:

- `saga.commands`: Canal para comandos de iniciação de processos.
- `research.events` e `annotation.events`: Canais de fatos de domínio, permitindo que múltiplos consumidores reajam aos dados sem alterar o produtor.
- `saga.status`: Canal de fechamento do ciclo de *feedback* para o orquestrador.

Essa mudança permite não apenas a resiliência (*replay* de eventos em caso de falha), mas habilita uma arquitetura verdadeiramente orientada a eventos (*Event-Driven Architecture*), alinhada aos princípios de desacoplamento do DDD.

5.1.4 Observabilidade e Telemetria

A camada superior do diagrama destaca a infraestrutura de observabilidade, essencial para gerenciar a complexidade distribuída (ADR-005). O componente **OpenTelemetry** atua como agente coletor unificado, interceptando rastros (*traces*), métricas e *logs* de todos os contêineres (OpenTelemetry Community, 2025; BLANCO, 2023).

Esses dados são exportados para uma *stack* especializada:

- **Loki:** Agregação e consulta de logs estruturados (ADR-013).

- **Prometheus:** Armazenamento de métricas de séries temporais — CPU, RAM, latência (ADR-014).
- **Tempo / Jaeger:** Rastreamento distribuído, permitindo visualizar a jornada completa de uma requisição através dos múltiplos microsserviços e tópicos do Kafka (ADR-012).
- **Grafana:** Visualização unificada e monitoramento de custos (FinOps).

Esta camada de suporte endereça diretamente as dificuldades de depuração encontradas na versão monolítica, fornecendo visibilidade granular sobre o comportamento do sistema.

5.2 Decisões Arquiteturais (ADRs)

Os *Architecture Decision Records* (ADRs) são artefatos cruciais para a longevidade técnica e a transparência de um sistema. Um ADR é um documento conciso que registra uma decisão arquitetural significativa, capturando o *porquê* de uma escolha ter sido feita em um contexto específico, as alternativas consideradas e as consequências resultantes.

Em alinhamento com a filosofia *Docs-as-Code*, os ADRs do Hermeneutics 2.0 são mantidos em arquivos Markdown no repositório de controle de versão (Git), próximos à base de código relacionada à decisão. Essa abordagem cria uma **memória viva do sistema**, acelerando o *onboarding* de novos desenvolvedores e evitando que decisões antigas sejam reexaminadas sem o contexto adequado.

5.2.1 ADRs Estruturantes

A Tabela 5.1 apresenta os seis ADRs fundamentais que definem a base arquitetural da transição para microsserviços.

5.2.1.1 ADR-001: Adoção da Arquitetura de Microsserviços (DDD)

Contexto e Problema: O monólito original (v1) foi construído com alto acoplamento entre os módulos (*crawlers*, web, ML). Esse acoplamento impedia a escalabilidade

Tabela 5.1: ADRs Fundamentais da Arquitetura Hermeneutics 2.0

ID	Título	Decisão	Consequência Principal
ADR-001	Adoção de Microsserviços (DDD)	Decompor o sistema por *Bounded Contexts*.	Aumento da complexidade operacional.
ADR-002	Mensageria com Kafka e Saga	Adotar Kafka para coreografia de Sagas.	Garantia de entrega e resiliência.
ADR-003	Database per Service	Cada serviço possui seu próprio banco PostgreSQL.	Necessidade de consistência eventual.
ADR-004	API Gateway	Implementar Nginx como ponto único de entrada.	Desacoplamento Frontend/Backend.
ADR-005	Observabilidade OpenTelemetry	Instrumentar todos os serviços com OTel.	Rastreabilidade total das transações.
ADR-006	Stack Tecnológica	FastAPI + Python 3.12 LTS para novos serviços.	Curva de aprendizado para a equipe.

Fonte: Autor.

isolada do módulo de Processamento de Linguagem Natural (NLP) e dificultava a manutenção.

Decisão: O sistema será decomposto em microsserviços por meio da aplicação do *Domain-Driven Design* (DDD) para definir *Bounded Contexts* (Ingestão, Anotação, Persistência, Orquestração).

Consequências:

- **[+]** Maior escalabilidade granular — possibilidade de escalar o módulo de Anotação Semântica separadamente.
- **[+]** Facilidade de manutenção e evolução independentes dos módulos.
- **[-]** Aumento da complexidade operacional (deploy, monitoramento).
- **[-]** Necessidade de orquestração de serviços distribuídos.

5.2.1.2 ADR-002: Transição para Mensageria Assíncrona com Apache Kafka

Contexto e Problema: O modelo monolítico utilizava o Celery e o RabbitMQ para orquestrar o fluxo assíncrono, resultando em uma fila única de processamento que gerava gargalos e potencial perda de mensagens em picos de carga.

Decisão: Adotar Apache Kafka como o barramento de eventos central e implementar o Padrão Saga Coreografada para transações distribuídas.

Consequências:

- **[+]** Garantia de entrega de mensagens e resiliência para os eventos de domínio.
- **[+]** Maior elasticidade no processamento de cargas de trabalho.

- **[-]** Necessidade de lidar com consistência eventual dos dados, visto que as sagas operam sob o modelo BASE.

5.2.1.3 ADR-003: Adoção do Padrão Database per Service

Contexto e Problema: O sistema monolítico utilizava um banco de dados relacional compartilhado para todos os módulos. Este banco compartilhado criava dependências ocultas e dificultava a evolução do esquema de forma isolada.

Decisão: Cada microsserviço (Maître, Research, Annotation, Persistence) terá seu próprio esquema ou instância isolada do PostgreSQL (Amazon Web Services, 202X).

Consequências:

- **[+]** Eliminação de dependências ocultas entre serviços.
- **[+]** Isolamento de persistência, permitindo a escolha de tecnologias adequadas para cada caso.
- **[+]** Facilitação da escalabilidade e governança de dados.
- **[-]** O isolamento dos dados exige garantia de consistência eventual através do Padrão Saga.

5.2.1.4 ADR-004: Implementação de API Gateway

Contexto e Problema: Na versão monolítica, o cliente (*Frontend*) acessava os serviços diretamente, o que, na transição para microsserviços, exporia a topologia interna e criaria potenciais falhas de segurança.

Decisão: Implementar uma API Gateway (Nginx) que atua como o ponto de entrada único para todos os clientes, sendo responsável pelo roteamento, autenticação e agregação de respostas.

Consequências:

- **[+]** Desacoplamento entre o *Frontend* e a topologia de microsserviços do *Backend*.
- **[+]** Segurança centralizada (AuthN/Z).
- **[+]** Abstração da complexidade interna para os clientes.
- **[-]** Ponto único de falha se não for bem arquitetado (necessita de alta disponibilidade).

5.2.1.5 ADR-005: Adoção de Observabilidade com OpenTelemetry

Contexto e Problema: A depuração de problemas em um ambiente distribuído (microsserviços, filas Kafka) é inviável apenas com logs locais, sendo necessária uma visão holística e correlacionada do fluxo de requisições (THAKUR; CHANDAK, 2022).

Decisão: Instrumentar todos os serviços com o *framework* OpenTelemetry, coletando os três pilares da observabilidade: Logs, Métricas e Traces.

Consequências:

- **[+]** Rastreabilidade total das transações (Saga) através de serviços.
- **[+]** Redução drástica do tempo de resolução de incidentes (MTTK).
- **[+]** Suporte à criação de indicadores FinOps como "custo por artigo processado".

5.2.1.6 ADR-006: Padronização da Stack Tecnológica

Contexto e Problema: Com a decomposição do monólito em múltiplos microsserviços, surge a necessidade de definir uma *stack* tecnológica homogênea para os novos serviços, evitando uma "torre de babel" tecnológica.

Decisão: Padronizar novos serviços com o *framework* FastAPI sobre a versão Long-Term Support (LTS) mais recente do Python (3.12).

Justificativa: FastAPI alinha-se perfeitamente aos requisitos de uma arquitetura de microsserviços moderna. Sua natureza assíncrona é fundamental para construir serviços reativos e eficientes que interagem com o Kafka. A validação de dados via Pydantic e a documentação automática reduzem a complexidade acidental.

5.2.2 Catálogo Completo de ADRs

Além dos ADRs estruturantes, o projeto Hermeneutics 2.0 documenta 32 decisões arquiteturais organizadas por área de atuação. A Tabela 5.2 apresenta o catálogo completo.

A evolução dessas decisões e suas implicações práticas serão discutidas no Capítulo 6, que apresenta os resultados da avaliação experimental.

A estrutura de repositório adota o modelo **Monorepo**, onde todos os microsserviços e bibliotecas compartilhadas coexistem em um único repositório Git. Aplicando o

Tabela 5.2: Catálogo Completo de ADRs do Hermeneutics 2.0

Categoria	ADRs
Arquitetura Core	ADR-001 (Microsserviços/DDD), ADR-002 (Kafka/Saga), ADR-003 (Database per Service), ADR-004 (API Gateway), ADR-005 (OpenTelemetry), ADR-006 (Stack Tecnológica)
Infraestrutura	ADR-007 (Kafka Init Container), ADR-008 (Dockerfile Template), ADR-019 (Docker Compose Naming), ADR-020 (Makefile Convention), ADR-024 (Single Postgres Container)
Shared Kernels	ADR-009 (Kafka Shared Kernel), ADR-022 (FastAPI App Factory), ADR-025 (Postgres Connectivity), ADR-026 (Redis Cache), ADR-027 (Connection Mixin Pattern)
Observabilidade	ADR-010 (MinIO Object Storage), ADR-011 (Thanos Long-Term Metrics), ADR-012 (Tempo Tracing), ADR-013 (Loki Log Aggregation), ADR-014 (Golden Signals Monitoring), ADR-015 (OTel Instrumentation), ADR-016 (OTel Hybrid Approach)
DevOps/DX	ADR-017 (Django 4.2 LTS Upgrade), ADR-021 (Subdomínios Localhost), ADR-023 (SCM Padronização)
E2E/FinOps	ADR-018 (Facade Maitre Mock), ADR-028 (Feature Toggles), ADR-029 (BFF Redis Sessions), ADR-030 (Kafka Batch Publishing), ADR-031 (ETL Legacy Migration), ADR-032 (Delays com Jitter)

Fonte: Autor.

princípio **DRY** (*Don't Repeat Yourself*), foram criadas bibliotecas compartilhadas (*Shared Kernels*) na pasta `libs/`, incluindo `hermeneutics_kafka` para abstração de produção e consumo de eventos, `hermeneutics_postgres` para conectividade com bancos de dados, e `hermeneutics_otel` para instrumentação padronizada de telemetria. Essa organização elimina duplicação de código entre serviços e garante consistência nas implementações de infraestrutura.

5.3 Padrões Implementados

Esta seção detalha os padrões arquiteturais implementados na prática, com foco no Padrão Saga com Apache Kafka e no ambiente de avaliação baseado em Docker Compose.

5.3.1 Padrão Saga Coreografada com Apache Kafka

O Padrão Saga, cuja fundamentação teórica foi apresentada na Seção 2.4.2 da fundamentação, é implementado no Hermeneutics 2.0 através de uma abordagem coreografada, onde os serviços reagem a eventos publicados no barramento Kafka sem a necessidade de um coordenador central (RICHARDSON, 2025; ŠTEFANKO et al., 2019).

5.3.1.1 Topologia de Tópicos Kafka

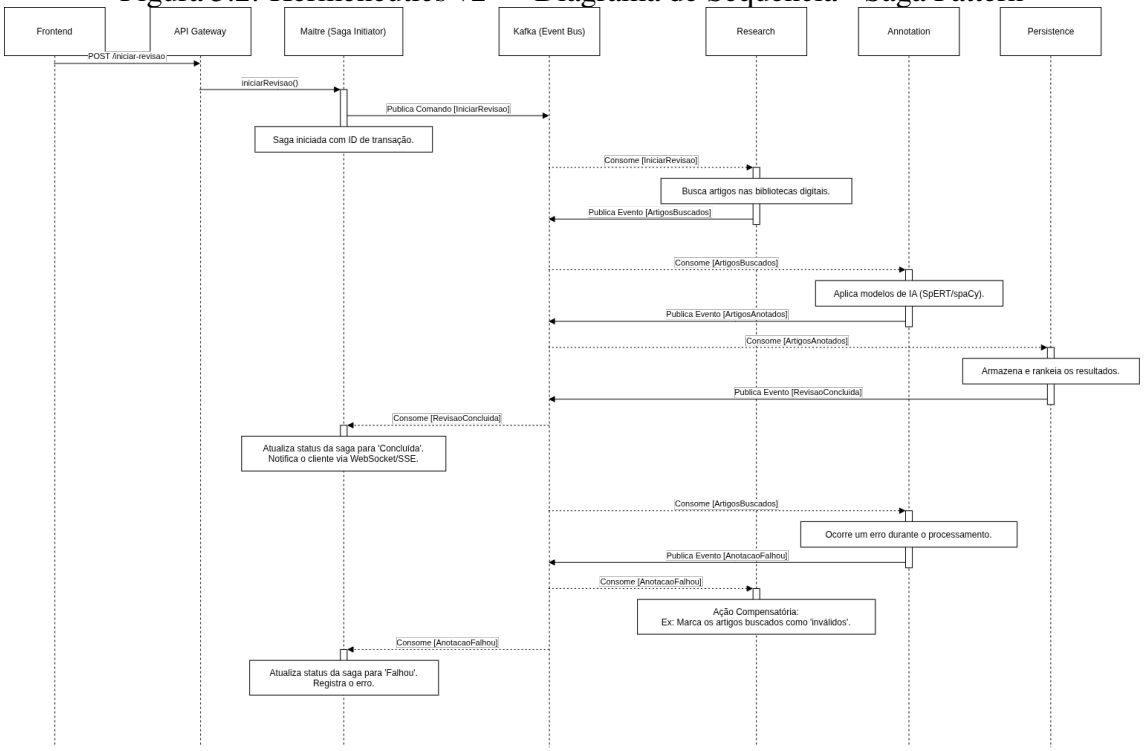
A implementação define quatro tópicos principais que orquestram o fluxo da Saga:

- `saga.commands`: Recebe comandos de iniciação de processos enviados pelo Maître.
- `research.events`: Publica eventos de artigos buscados pelo serviço Research.
- `annotation.events`: Publica eventos de artigos anotados pelo serviço Annotation.
- `saga.status`: Canal de fechamento do ciclo de feedback para o orquestrador.

5.3.1.2 Coreografia da Saga

O fluxo da saga coreografada define a comunicação assíncrona entre os microsserviços. A Figura 5.2 ilustra o diagrama de sequência da implementação.

Figura 5.2: Hermeneutics v2 — Diagrama de Sequência - Saga Pattern



Passos do Fluxo de Sucesso:

1. O microsserviço **Maître** (Iniciador da Saga) recebe a requisição do Web UI, atribui um ID de transação e publica o comando `[IniciarRevisao]` no Kafka.

2. O serviço **Research** consome o comando, busca os artigos nas bibliotecas digitais e publica o evento `[ArtigosBuscados]`.

3. O serviço **Annotation** consome `[ArtigosBuscados]`, aplica os modelos de PLN e publica o evento `[ArtigosAnotados]`.

4. O serviço **Persistence** consome `[ArtigosAnotados]`, armazena os dados definitivamente e publica `[RevisaoConcluida]`.

5. O **Maître** consome `[RevisaoConcluida]`, atualiza o status da saga e notifica o cliente.

Fluxo de Falha e Compensação: Se ocorrer um erro, por exemplo, no serviço de **Annotation**, ele publicará um evento de falha como `[AnotacaoFalhou]`. Os serviços anteriores consomem este evento para executar ações compensatórias, e o **Maître** atualiza o status final da saga, garantindo que o sistema alcance um estado consistente conforme o modelo BASE.

5.3.2 Ambiente de Avaliação: Docker Compose

Conforme documentado no ADR-018 e ADR-028, o ambiente de avaliação do Hermeneutics 2.0 foi implementado utilizando Docker Compose para simular o ambiente distribuído localmente. Esta decisão foi guiada pelos princípios KISS (*Keep It Simple, Stupid*) e OCCAM, priorizando a avaliação do fluxo da Saga sobre a complexidade de uma infraestrutura de nuvem real.

5.3.2.1 Stack de Serviços

O ambiente Docker Compose orquestra 15+ contêineres que compõem o ecossistema completo:

- **Aplicação:** Web UI (Django), Maître, Research, Annotation, Persistence (FastAPI)
- ***Mensageria*:** Apache Kafka[1] + Zookeeper[2]
- **Dados:** PostgreSQL (com bancos isolados por serviço), Redis[3]

[1] Kafka <https://www.confluent.io/what-is-apache-kafka/>
[2] Zookeeper <https://zookeeper.apache.org/>
[3] Redis Cache <https://aws.amazon.com/pt/elasticache/what-is-redis/>

- **Observabilidade:** OpenTelemetry Collector, Prometheus[4], Grafana[5], Loki[6], Tempo[7], Jaeger[8]

- **Infraestrutura:** Nginx[9] (API Gateway), MinIO[10] (Object Storage)

5.3.2.2 Feature Toggles para Modos de Operação

O ADR-028 define *Feature Toggles* via variáveis de ambiente que permitem alternar entre modos de operação sem modificar código:

- **Modo Memory (default):** Adapters *fake/in-memory* para desenvolvimento rápido e testes.

- **Modo Database:** Adapters reais PostgreSQL para demonstração E2E e avaliação FinOps.

As variáveis `SEARCH_MODE` e `PERSISTENCE_MODE` controlam qual *adapter* é utilizado em cada serviço, garantindo *backward compatibility* com o fluxo de mocks existente.

5.3.2.3 Delays com Jitter para Métricas FinOps

O ADR-032 implementa delays configuráveis com *jitter* (variação aleatória) para simular latências realistas de processamento. Isso permite que as métricas de observabilidade reflitam padrões de uso próximos ao ambiente de produção, habilitando análises de FinOps como "custo por artigo processado".

5.3.2.4 Limitações Explícitas do Ambiente

É importante documentar as limitações do ambiente de avaliação:

- **Ambiente Local:** O Docker Compose executa em máquina local, não em infraestrutura de nuvem real. Métricas de latência e escalabilidade são aproximações.

[4]Prometheus <https://prometheus.io/docs/introduction/overview/>
[5]Grafana <https://www.redhat.com/en/topics/data-services/what-is-grafana>
[6]Loki <https://grafana.com/oss/loki/>
[7]Tempo <https://grafana.com/oss/tempo/>
[8]Jaeger <https://www.jaegertracing.io/>
[9]Nginx <https://nginx.org/en/>
[10]Minio <https://github.com/minio/minio>

- **LLM Simulado:** O serviço Annotation utiliza *mocks* para simular o processamento de PLN. Aplicando o princípio **YAGNI** (*You Ain't Gonna Need It*), a integração custosa com APIs de LLM foi postergada até que o *pipeline* de *mensageria* (Saga) estivesse totalmente observável e resiliente. A integração real com modelos de linguagem (e.g., Google Gemini, SpERT) é escopo de trabalhos futuros.
- **APIs de Bibliotecas Digitais Simuladas:** O serviço Research pode operar com dados *mock*, não realizando chamadas reais às APIs de bibliotecas digitais em todos os cenários de teste.

Essas limitações são aceitáveis para o escopo deste trabalho, cujo objetivo principal é avaliar a arquitetura orientada a eventos e os padrões de engenharia de software, não a funcionalidade completa de produção.

6 IMPLEMENTAÇÃO E ANÁLISE DE RESULTADOS

Este capítulo apresenta a metodologia de desenvolvimento adotada para a modernização do Hermeneutics, o ambiente de avaliação configurado e as evidências obtidas através da observabilidade implementada.

6.1 Metodologia de Desenvolvimento

A metodologia deste trabalho fundamenta-se em práticas contemporâneas de Engenharia de Software, combinando princípios de *Domain-Driven Design* (DDD), Arquitetura Hexagonal e desenvolvimento iterativo guiado por documentação viva (*Docs-as-Code*). O enfoque é aplicado, priorizando a entrega de uma arquitetura funcional e observável sobre a completude de funcionalidades de inteligência artificial.

6.1.1 Etapas do Projeto

O desenvolvimento foi organizado em ciclos iterativos, seguindo um fluxo estruturado que permitiu entregas incrementais:

1. **Análise e Documentação Inicial**: Levantamento do contexto legado (Hermeneutics v1), identificação de *Bounded Contexts* e definição de Decisões Arquiteturais em formato ADR (*Architectural Decision Record*), conforme recomendado por (NYGARD, 2011). Foram documentadas mais de 30 ADRs ao longo do projeto, registrando decisões como a escolha do Apache Kafka para mensageria e a adoção do padrão *Database-per-Service*.

2. **Planejamento em EPICs e Stories**: Os requisitos foram decompostos em Épicos e Histórias de Usuário, seguindo práticas ágeis inspiradas no *framework* SAFe (Scaled Agile, Inc., 2023). Cada EPIC representa uma entrega coesa, e cada *Story* detalha critérios de aceite verificáveis.

3. **Implementação Iterativa**: Codificação dos microsserviços em Python 3.12, utilizando FastAPI para APIs REST e Apache Kafka para *mensageria* assíncrona. A estrutura de código seguiu a Arquitetura Hexagonal, separando domínio, aplicação e infraestrutura.

4. **Validação Contínua**: A cada ciclo, os serviços foram validados por testes unitários automatizados, executados tanto localmente quanto em pipeline de integração contínua. A instrumentação com OpenTelemetry permitiu verificar o comportamento em tempo de execução.

5. **Consolidação e Documentação**: Registro das decisões e resultados em arquivos Markdown versionados no repositório, permitindo rastreabilidade e reprodutibilidade. Esta abordagem de *Docs-as-Code* garante que a documentação evolua junto com o código.

6.1.2 Organização do Trabalho em EPICs e Stories

Para garantir a rastreabilidade e a gestão eficiente do escopo, o desenvolvimento foi estruturado seguindo o modelo de Épicos e Histórias de Usuário. A Tabela 6.1 apresenta os principais Épicos desenvolvidos durante o projeto:

Tabela 6.1: Principais EPICs do projeto Hermeneutics v2.

EPIC	**Nome**	**Descrição**
EPIC-001	Infraestrutura Base	Configuração do Docker Compose, rede virtual e bancos de dados isolados
EPIC-002	Mensageria Kafka	Implementação do padrão Saga coreografada com eventos assíncronos
EPIC-003	Observabilidade	Instrumentação OpenTelemetry, *dashboards* Grafana e rastreamento Jaeger
EPIC-004	FinOps	Métricas de custo por operação e simulação de consumo
EPIC-005	Integração BFF	*Frontend* Django com sessões Redis e *gateway* Nginx
EPIC-006	Migração Legado	ETL de dados da versão v1 para v2 com validação de integridade

Cada EPIC foi decomposto em múltiplas Stories, com critérios de aceite claros e testes associados. Essa organização permitiu entregas incrementais e facilitou a continuidade do trabalho, sendo toda a documentação disponibilizada no repositório do projeto.

6.1.3 Aceleração do Desenvolvimento com *Prompt Engineering*

Para mitigar restrições de cronograma e maximizar a produtividade individual, foi adotada uma estratégia sistemática de uso de Inteligência Artificial Generativa, especificamente *Large Language Models* (LLMs), integrada ao fluxo de desenvolvimento.

O *framework* de *Prompt Engineering* adotado baseia-se no padrão CRISPE (*Capacity*, *Role*, *Insight*, *Statement*, *Personality*, *Experiment*), conforme catalogado por (NIGH, 2023). A aplicação desse *framework* permitiu estruturar as interações com o modelo de forma reprodutível e auditável:

- **Geração de ADRs**: A partir de discussões técnicas e decisões de *design*, o LLM gerava rascunhos iniciais de *Architectural Decision Records*, seguindo um *template* padronizado. Os documentos eram então revisados e refinados pelo autor.
- **Refatoração Assistida**: *Prompts* detalhados, enriquecidos com o contexto do repositório através de *Retrieval-Augmented Generation* (RAG) (LEWIS et al., 2020), guiavam a refatoração de código legado para a nova arquitetura hexagonal.
- **Documentação Automatizada**: Geração de documentação técnica em formato Markdown, incluindo *walkthroughs* de implementação e planos de execução para cada Story.
- **Revisão de Código**: O LLM foi utilizado como revisor auxiliar, identificando potenciais problemas de design e sugerindo melhorias alinhadas aos padrões do projeto.

Essa abordagem transforma a IA de ferramenta auxiliar em componente integral do fluxo de engenharia, alinhada às práticas emergentes de *Promptware Engineering* (WHITE et al., 2023). A documentação dos *prompts* utilizados foi versionada no repositório, garantindo reprodutibilidade.

6.2 Ambiente de Desenvolvimento e DevOps

A infraestrutura de desenvolvimento do Hermeneutics v2 foi projetada para simular um ambiente de nuvem em máquina local, utilizando contêineres Docker orquestrados via Docker Compose. Esta seção detalha os componentes de ambiente, os mecanismos de garantia de qualidade e as práticas de *Developer Experience* (DX) adotadas.

6.2.1 Orquestração com Docker Compose

O ambiente completo da versão v2 é definido em um arquivo `docker-compose.yml` que orquestra mais de 15 serviços, incluindo:

- **Microsserviços de negócio**: Maitre (orquestrador de saga), Research (busca em bibliotecas), Annotation (processamento de anotações) e Persistence (persistência de dados), cada um com banco de dados PostgreSQL dedicado, seguindo o padrão *Database-per-Service* (Amazon Web Services, 202X).

- **Infraestrutura de *mensageria***: Apache Kafka e Zookeeper para comunicação assíncrona entre serviços, implementando o padrão Saga Coreografada (RICHARDSON, 2025).

- **Stack de observabilidade**: OpenTelemetry Collector para coleta unificada de telemetria, Prometheus para armazenamento de métricas, Grafana para visualização, Loki para agregação de *logs* e Tempo para armazenamento de traces distribuídos.

- **Serviços auxiliares**: Redis para cache e gerenciamento de sessões, MinIO como Object Storage S3-compatível para armazenamento de traces e logs, e Nginx como API Gateway com roteamento baseado em subdomínios.

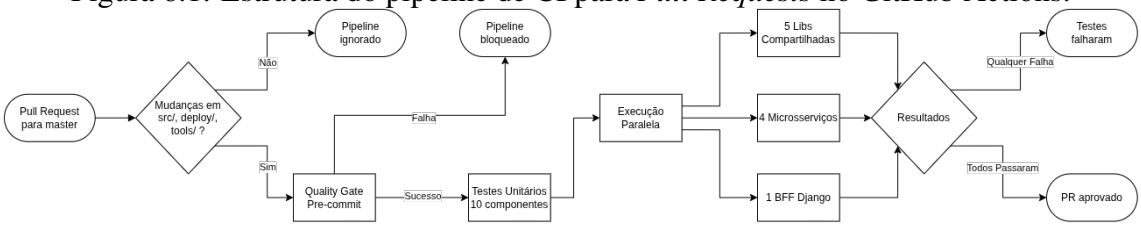
A escolha por Docker Compose, em detrimento de Kubernetes, segue o princípio KISS (*Keep It Simple, Stupid*) e a Navalha de Ockham: evitou-se a complexidade acidental de um *cluster* Kubernetes completo para validar a arquitetura lógica antes de considerar a infraestrutura física de produção (HUNT; THOMAS, 1999). Esta decisão está documentada nas ADRs do projeto e permite que qualquer desenvolvedor replique o ambiente completo em sua máquina local.

6.2.2 Pipeline de Integração Contínua (CI)

Para garantir a qualidade do código e prevenir regressões, foi implementado um pipeline de Integração Contínua utilizando GitHub Actions. O pipeline é disparado automaticamente em cada *Pull Request* para a *branch* principal, implementando o conceito de *Quality Gates*.

O pipeline é composto por dois estágios principais, conforme ilustrado na Figura 6.1:

Figura 6.1: Estrutura do pipeline de CI para *Pull Requests* no GitHub Actions.



1. **Quality Gate (Pre-commit)**: Executa verificadores de qualidade apenas nos arquivos modificados no PR, utilizando a ferramenta `pre-commit` com a *flag* `--from-ref`. São aplicados:

   - Linter e *formatter* Ruff para código Python;
   - Validadores de sintaxe YAML e arquivos de configuração;
   - Hadolint para verificação de melhores práticas em Dockerfiles;
   - djLint para formatação de *templates* Django;
   - Validação de sintaxe de arquivos `docker-compose.yml`.

2. **Testes Unitários (*Matrix Strategy*)**: Executa testes unitários em paralelo para cada componente do sistema, utilizando `pytest` com cobertura de código. A estratégia de matriz (*matrix strategy*) do GitHub Actions permite que falhas em um componente não bloqueiem a execução dos demais, acelerando o *feedback*.

Esta abordagem de validação automatizada garante que todo o código integrado à *branch* principal passou por verificações de qualidade e testes, reduzindo a probabilidade de regressões.

6.2.3 Developer Experience (DX) e Pre-commit Hooks

A experiência do desenvolvedor foi priorizada por meio da automação de tarefas repetitivas e da padronização do ambiente de trabalho. Os principais mecanismos adotados incluem:

- **Pre-commit Hooks**: Conjunto de verificações automáticas executadas localmente antes de cada commit, configuradas no arquivo `.pre-commit-config.yaml`. As verificações incluem:
  - Validação de sintaxe Python (AST) e YAML
  - Formatação automática com Ruff (linter e formatter unificados)
  - Modernização de sintaxe para Python 3.12+ via `pyupgrade`
  - Validação de Dockerfiles com Hadolint e docker-compose com verificador dedicado
  - Validação de formato de mensagens de *commit* via *hook* customizado
  - Correção automática de espaços em branco e quebras de linha
- **Makefile Centralizado**: Abstração de comandos complexos em *targets* semânticos, utilizando prefixos padronizados (`dc-*` para Docker Compose, `test-*` para testes, `rest-*` para chamadas HTTP). Exemplos incluem `make dc-up` para iniciar a *stack* e `make test-maitre` para executar testes do serviço Maitre.
- **EditorConfig**: Padronização de indentação, charset (UTF-8) e quebras de linha (LF) entre diferentes editores e sistemas operacionais, garantindo consistência no repositório.
- **Exclusão de Código Legado**: Configuração explícita para excluir os diretórios `src/legacy/v1/` e `deploy/docker/v1/` das verificações automatizadas, evitando ruído causado por código que não será modificado.

A adoção dessas práticas segue o princípio DRY (*Don't Repeat Yourself*) (HUNT; THOMAS, 1999): as regras de qualidade são definidas uma única vez nos arquivos de configuração e aplicadas automaticamente em todos os ambientes (local e CI), evitando inconsistências e reduzindo a carga cognitiva do desenvolvedor.

6.2.4 Testes Unitários e Cobertura

O projeto adota uma estratégia de testes em múltiplos níveis, com ênfase em testes unitários para validação das regras de negócio isoladamente. A Tabela 6.2 apresenta os componentes cobertos pelo pipeline de CI:

Tabela 6.2: Componentes cobertos por testes unitários no pipeline CI.

Tipo	**Componente**	**Framework**
Biblioteca Compartilhada	hermeneutics_kafka	pytest
Biblioteca Compartilhada	hermeneutics_otel	pytest
Biblioteca Compartilhada	hermeneutics_fastapi	pytest
Biblioteca Compartilhada	hermeneutics_db	pytest
Biblioteca Compartilhada	hermeneutics_cache	pytest
Microsserviço	Maitre	pytest
Microsserviço	Research	pytest
Microsserviço	Annotation	pytest
Microsserviço	Persistence	pytest
BFF	Web UI (Django)	Django TestCase

Os testes são executados com relatório de cobertura via `pytest-cov`, permitindo identificar áreas do código sem validação. A estrutura de bibliotecas compartilhadas (*Shared Kernels*) segue o princípio DRY, centralizando a lógica comum entre microsserviços — como configuração de Kafka, instrumentação OpenTelemetry e conexão com o banco de dados — em pacotes Python instaláveis.

Esta abordagem de *Shared Kernels* permite que alterações em funcionalidades comuns sejam testadas uma única vez e propagadas para todos os serviços dependentes, reduzindo duplicação de código e garantindo consistência comportamental.

6.3 Evidências de Observabilidade

A avaliação da arquitetura proposta não se limita a testes funcionais, mas estende-se à capacidade de observar o comportamento interno do sistema distribuído em tempo de execução. Esta seção apresenta as evidências de observabilidade coletadas através de um cenário de teste de carga controlado.

6.3.1 Fundamentação e Infraestrutura como Código

A implementação da observabilidade no Hermeneutics v2 é resultado de decisões arquiteturais deliberadas, documentadas em ADRs (*Architectural Decision Records*), e segue o paradigma de Infraestrutura como Código.

Decisões Arquiteturais (ADRs): A adoção do padrão OpenTelemetry (OpenTelemetry Community, 2025) para instrumentação neutra foi definida na **ADR-005**, garantindo que a coleta de telemetria seja desacoplada do *backend* de armazenamento. A

escolha da *stack* Grafana (Tempo para *traces* e Loki para logs) baseou-se na integração nativa visual, documentada nas **ADRs 012** e **013**. A estratégia de monitoramento foca nos *Golden Signals* (Latência, Tráfego, Erros e Saturação), conforme estabelecido na **ADR-014**.

Artefatos de Dashboard (IoC): As visualizações apresentadas nesta seção são geradas a partir de definições em código (JSON), versionadas no repositório do projeto no diretório `deploy/docker/v2/observability/dashboards/`. Isso garante a reprodutibilidade do ambiente de monitoramento:

- `infrastructure.json`: Monitoramento de recursos de contêineres e *host*.
- `database-health.json`: Métricas específicas de conexões e operações do PostgreSQL.
- `microservices.json`: Visão agregada de latência e *throughput* dos serviços (RED Method).
- `traces-overview.json`: Análise de dados de rastreamento distribuído.

6.3.2 Metodologia do Experimento

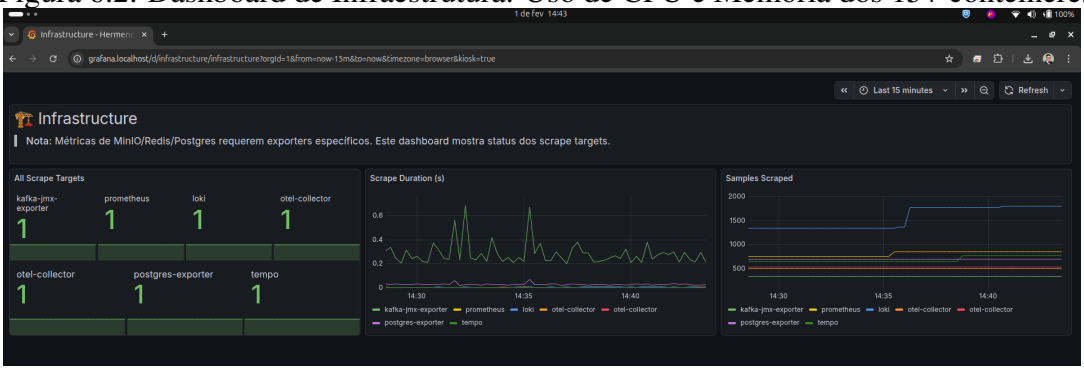
Para gerar as evidências de observabilidade, foi executado um teste de carga utilizando o *script* `tools/load_test.sh`, que simula um padrão de tráfego misto durante 1000 iterações. O cenário de teste contempla:

1. **Tráfego Baseline (Happy Path)**: Requisições contínuas aos endpoints de *health check* (`/hello`) de todos os microsserviços (Maitre, Research, Annotation, Persistence).
2. **Fluxo de Negócio (Saga)**: Disparo do fluxo de demonstração da saga (`/demo`) a cada 5 iterações, exercitando a comunicação assíncrona via Kafka.
3. **Injeção de Falhas**: Simulação de erro interno (HTTP 500) via *endpoint* `/debug/error` a cada 7 iterações para validar a detecção e o registro de exceções.
4. **Stress Test de Mensageria**: Injeção incremental de mensagens no barramento Kafka (`/debug/produce`) a cada 11 iterações, validando o comportamento de filas e consumidores sob carga crescente.

6.3.3 Monitoramento de Infraestrutura

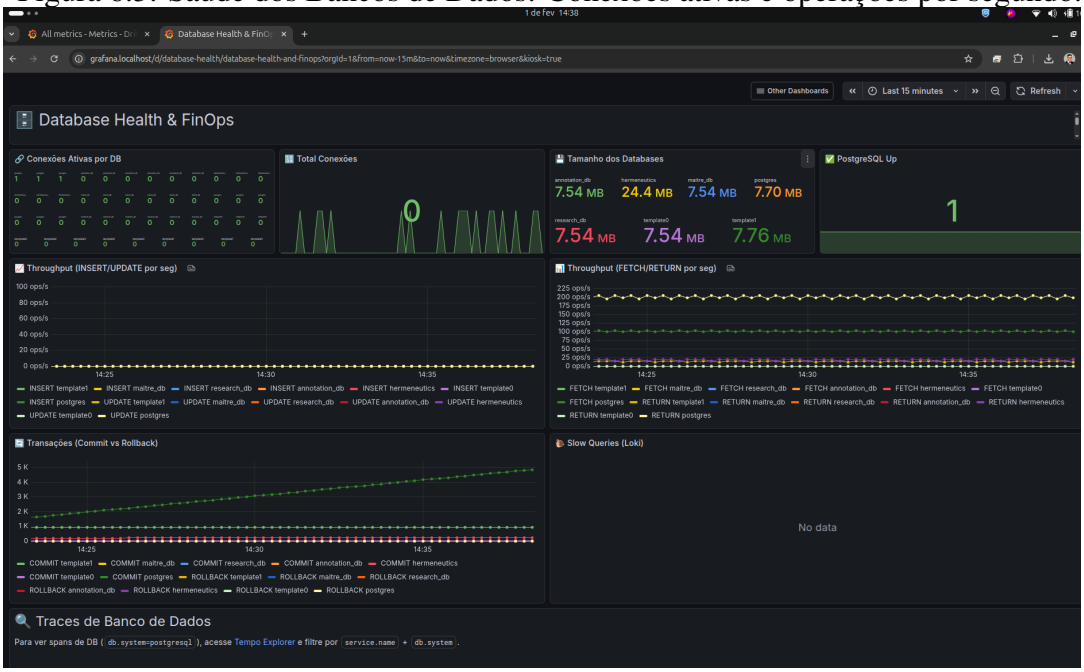
O primeiro nível de observabilidade valida a saúde dos recursos computacionais. A Figura 6.2 demonstra o monitoramento de CPU e Memória dos contêineres Docker, permitindo identificar gargalos de recursos. Observa-se que, mesmo sob carga, o consumo de memória (painel inferior) mantém-se estável, indicando ausência de *memory leaks*.

Figura 6.2: Dashboard de Infraestrutura: Uso de CPU e Memória dos 15+ contêineres.



Complementarmente, a Figura 6.3 detalha a saúde dos bancos de dados PostgreSQL. O padrão *Database-per-Service* é evidenciado pelas métricas independentes para cada instância. O gráfico de "Active Connections" ilustra o comportamento do pool de conexões durante os picos de carga do teste.

Figura 6.3: Saúde dos Bancos de Dados: Conexões ativas e operações por segundo.



6.3.4 Métricas de Aplicação (*Golden Signals*)

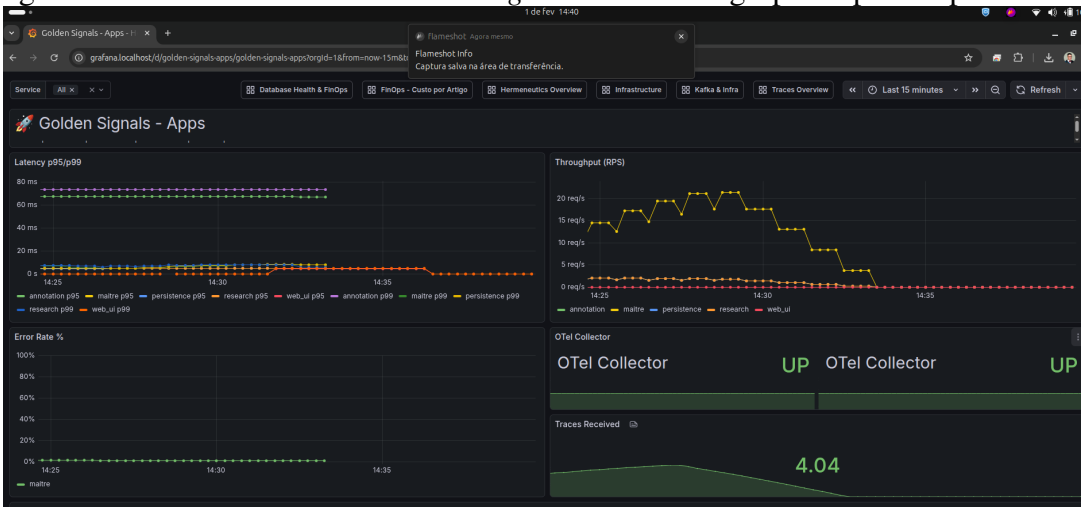
No nível da aplicação, a observabilidade foca nos *Golden Signals* instrumentados via OpenTelemetry. A Figura 6.4 apresenta o *dashboard* unificado de microsserviços.

Figura 6.4: Visão Geral OpenTelemetry: Latência, Taxa de Requisições e Taxa de Erros.



A Figura 6.5 oferece uma visão detalhada das métricas RED (*Rate, Errors, Duration*) por operação, permitindo isolar a performance de *endpoints* específicos como `POST /demo` versus `GET /hello`.

Figura 6.5: Detalhamento dos *Golden Signals*: Métricas agrupadas por endpoint HTTP.



É possível correlacionar diretamente os eventos do *script* de teste com as métricas visuais:

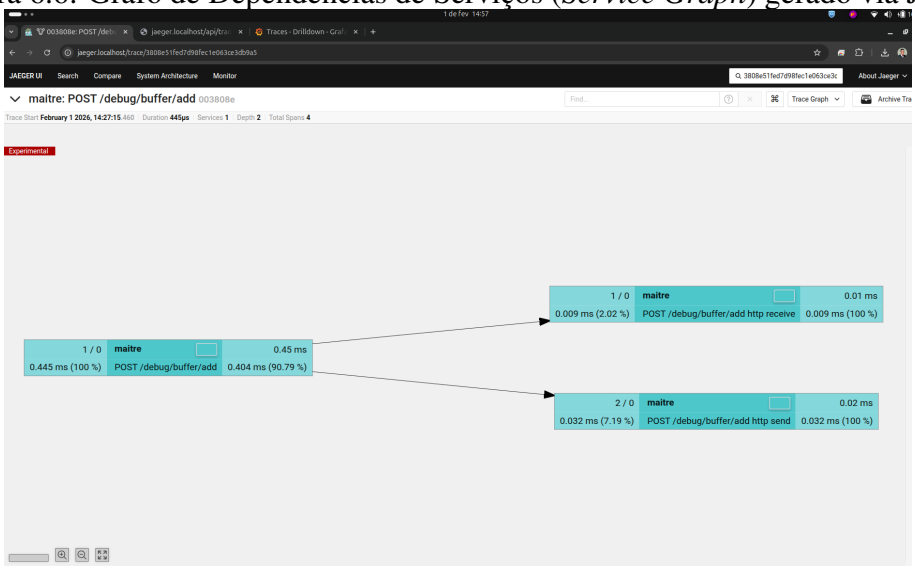
- **Picos de Erro**: O gráfico de "Error Rate" exibe picos periódicos correspondentes à injeção de falhas (erro 500 a cada 7 iterações).
- **Latência de Cauda**: O *heatmap* de latência permite visualizar os *outliers* (P95 e P99), fundamentais para análise de performance em sistemas distribuídos.

6.3.5 Rastreamento Distribuído (Distributed Tracing)

A capacidade de rastrear uma transação que atravessa múltiplos serviços é a evidência definitiva da observabilidade em microsserviços. A instrumentação implementada propaga o contexto (Trace Context) através de headers HTTP e mensagens Kafka.

A Figura 6.6 exibe o Grafo de Dependências gerado automaticamente a partir dos *traces* coletados. Ele confirma a arquitetura implementada: o `maitre-service` (orquestrador) comunica-se com o Kafka, que por sua vez aciona os consumidores nos serviços `research`, `annotation` e `persistence`.

Figura 6.6: Grafo de Dependências de Serviços (*Service Graph*) gerado via Jaeger.



Aprofundando a análise, a Figura 6.7 apresenta a visualização de Timeline (Gráfico de Gantt) de uma transação completa.

É possível identificar:

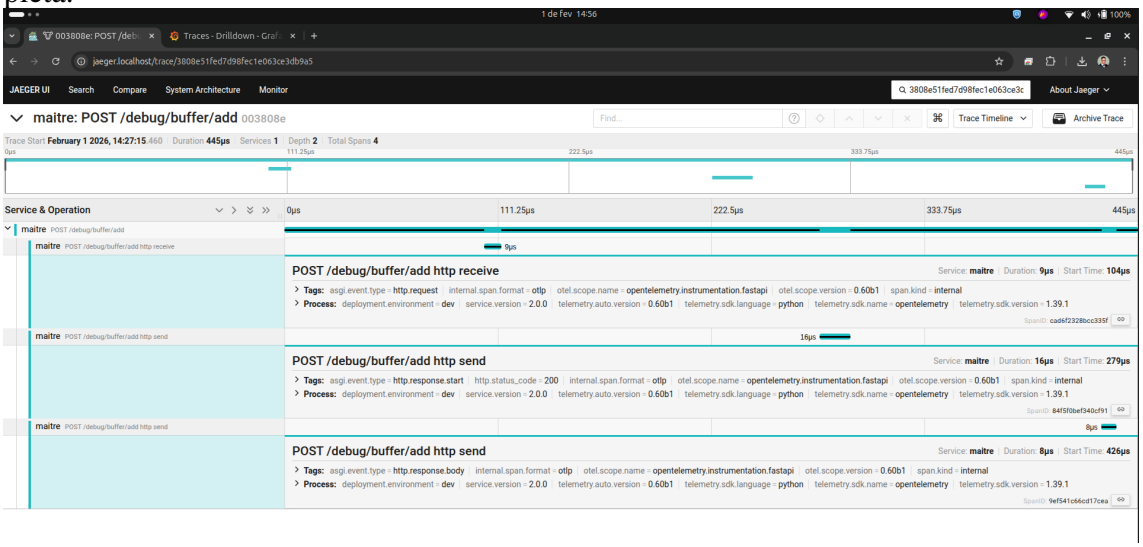
1. O início da requisição no API Gateway/Maitre.

2. O tempo de processamento assíncrono nas mensagens Kafka (representado pelos spans desconectados na *timeline* visual, mas ligados logicamente pelo ID).

3. A latência individual de cada operação de banco de dados.

A integração com o Grafana Tempo permite buscar esses *traces* baseados em critérios complexos, como mostrado na Figura 6.8, facilitando o *troubleshooting* de transações lentas ou com erro.

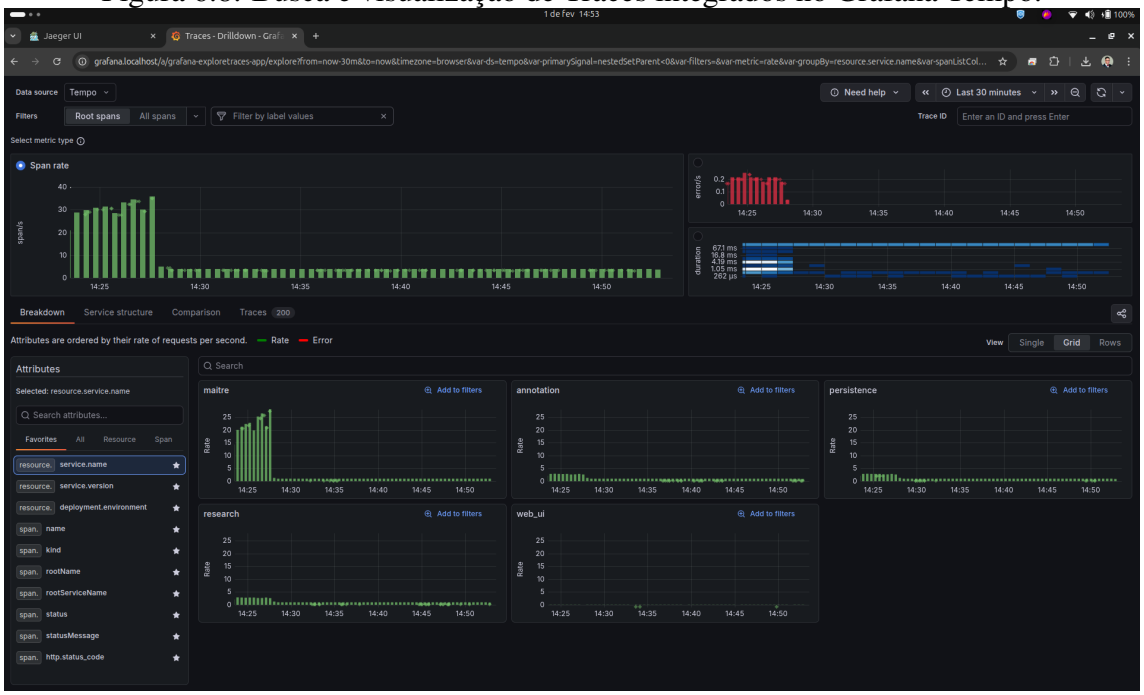
Figura 6.7: Trace Timeline: Visualização Waterfall de uma transação distribuída completa.



6.3.6 Correlação de Logs

Por fim, a Figura 6.9 demonstra a correlação de logs estruturados armazenados no Loki. Através da injeção automática do `trace_id` e `span_id` nos logs da aplicação, é possível filtrar todas as mensagens de log associadas a uma transação específica, eliminando a necessidade de buscar manualmente em múltiplos arquivos de log.

Figura 6.8: Busca e visualização de Traces integrados no Grafana Tempo.



Essa capacidade de "pular" de um gráfico de métrica (indicando um problema) para um trace (localizando o serviço culpado) e finalmente para os *logs* (identificando a causa raiz) constitui o pilar central da observabilidade do Hermeneutics v2.

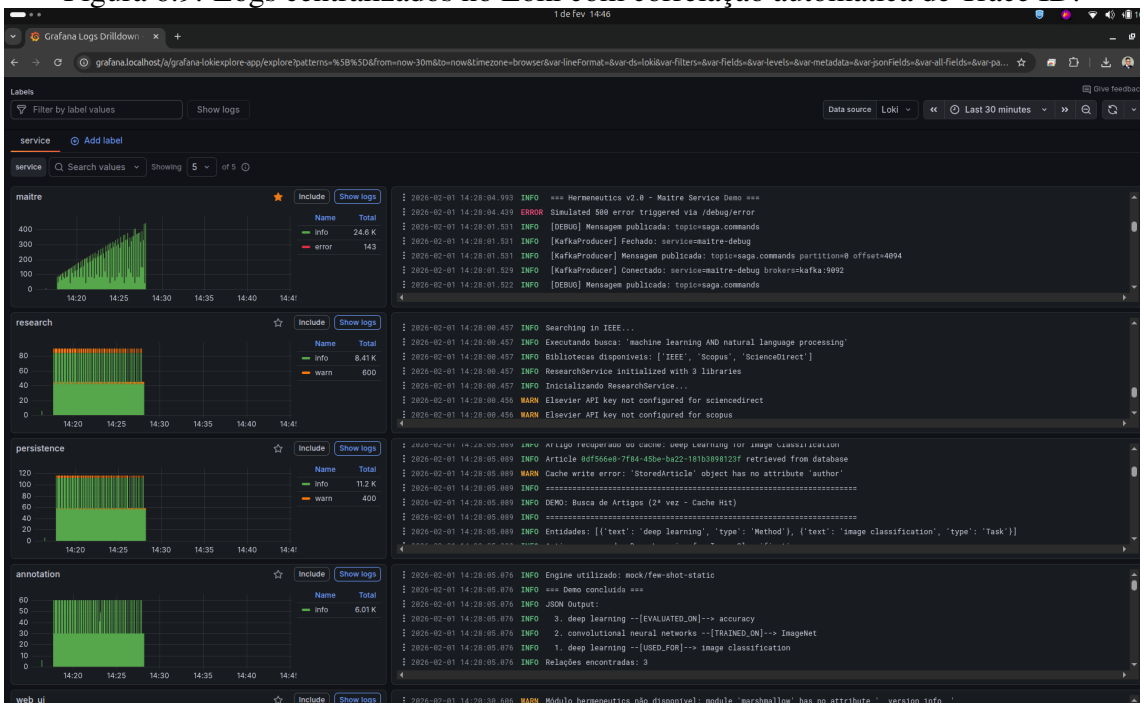
6.4 Análise de Custos e Demonstração E2E

Além da observabilidade técnica, a arquitetura modernizada permite a demonstração antecipada de custos (FinOps) e fluxos de negócio através de uma estratégia robusta de *mocking* e simulação. Esta seção detalha como a Arquitetura Hexagonal foi explorada para acelerar o ciclo de *feedback*.

6.4.1 O Papel dos Mocks e Adapters na Estratégia de Testes

A adoção da Arquitetura Hexagonal (Ports & Adapters) permitiu o desacoplamento total entre a lógica de negócio e os mecanismos de persistência e busca externa. Conforme documentado na **ADR-028**, foram implementadas *Feature Toggles* (`SEARCH_MODE`, `PERSISTENCE_MODE`) que permitem alternar dinamicamente entre implementações reais (PostgreSQL, APIs externas) e simuladas em memória.

Figura 6.9: Logs centralizados no Loki com correlação automática de Trace ID.



Para viabilizar a demonstração End-to-End (E2E) da funcionalidade de Saga sem a complexidade de um API Gateway completo nesta fase, o serviço `Maitre` assumiu o papel de *Facade* (**ADR-018**), atuando como ponto único de entrada para a interface de usuário durante os testes. Isso possibilitou a demonstração da coreografia de eventos Kafka mesmo em ambientes de desenvolvimento locais com recursos limitados.

6.4.2 Metodologia da Simulação de Custos (FinOps)

Um dos requisitos não funcionais críticos do Hermeneutics v2 é a previsibilidade de custos de operação, especialmente considerando o uso intensivo de LLMs e APIs pagas. Para validar as métricas de FinOps sem incorrer em custos reais durante o desenvolvimento, foi desenvolvida uma metodologia de simulação baseada em *delays* artificiais com *jitter* (variação aleatória).

Conforme a **ADR-032**, uma função centralizada no *Shared Kernel* aplica atrasos configuráveis em cada etapa do pipeline (ex: 300ms para simular uma chamada de LLM, 100ms para persistência), injetando uma variância de 25% para simular a latência real de rede e processamento. O custo é então projetado multiplicando-se o tempo simulado pelo custo unitário do recurso configurado nas métricas do Prometheus.

6.4.3 Evidências do Experimento (Modo Memória)

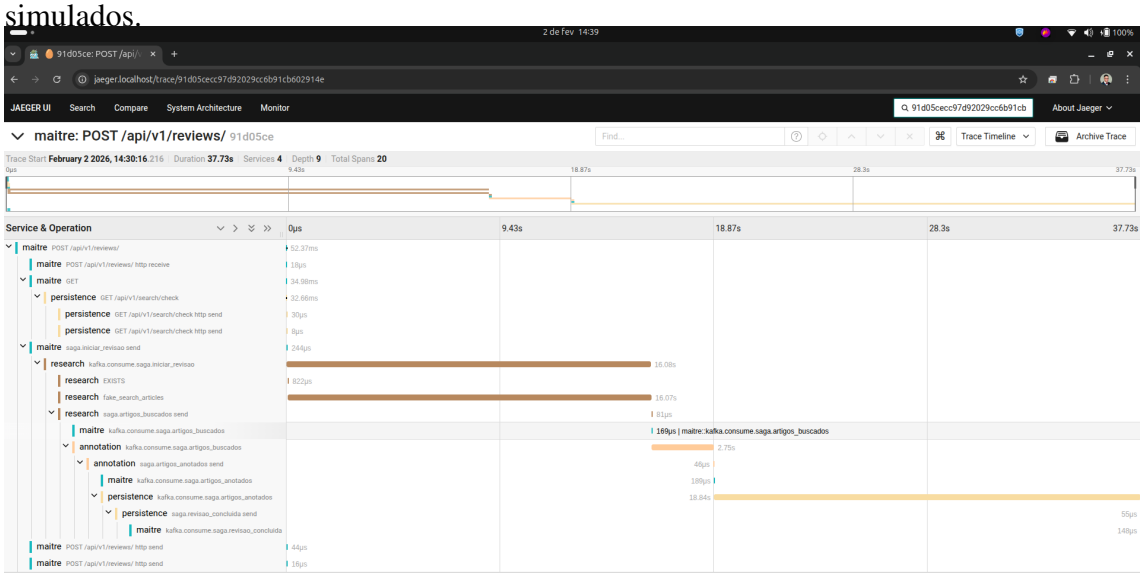
Para demonstrar essa capacidade, foi executado o cenário de teste `make mock-saga-repeat`, que dispara a Saga de revisão para 10 artigos em modo puramente em memória (sem banco de dados ou chamadas externas).

6.4.3.1 Demonstração de Fluxo e Rastreabilidade

Mesmo sem persistência física, a observabilidade do fluxo é mantida. A análise via Jaeger permite visualizar a estrutura de dependências entre os serviços e identificar gargalos lógicos.

A Figura 6.10 apresenta a *Timeline* de execução da Saga. É possível observar claramente os *delays* artificiais inseridos (representados pelas barras de duração nos spans de `process_article`), confirmando que a simulação de carga está funcionando conforme projetado pela ADR-032.

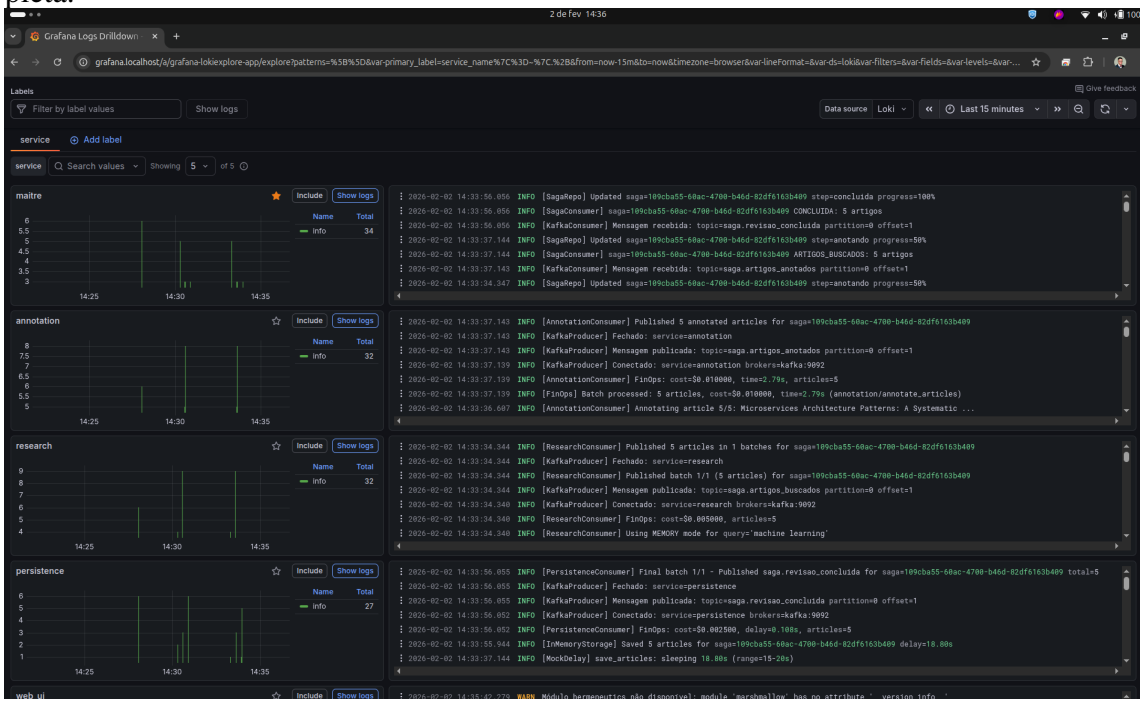
Figura 6.10: Timeline de execução da Saga em modo Memória, evidenciando os delays simulados.



6.4.3.2 Correlação de Logs

A Figura 6.11 demonstra que a correlação de *logs* via Loki permanece funcional independentemente do modo de operação. A *tag* `search_mode=memory` é visível, garantindo que operadores possam distinguir entre execuções reais e simuladas durante auditorias.

Figura 6.11: Logs no Loki durante execução em memória, mantendo rastreabilidade completa.

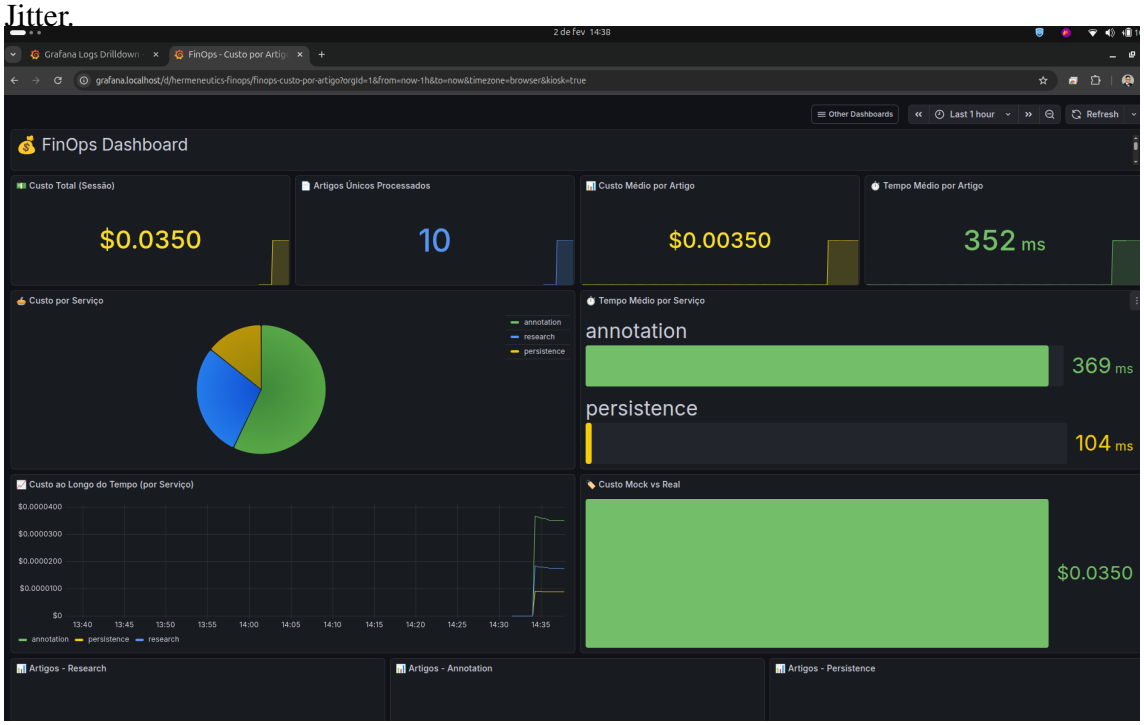


6.4.3.3 Dashboard FinOps e Análise de Valor

Finalmente, a Figura 6.12 exibe o Dashboard FinOps gerado a partir da simulação. O sistema calculou o custo estimado e o tempo médio de processamento para os 10 artigos, validando a lógica de monitoramento de custos antes mesmo do *deploy* em produção.

Esta capacidade de simular cenários de custo (“E se o modelo demorar 2s em vez de 300ms?”) confere ao Hermeneutics v2 uma vantagem significativa de planejamento de capacidade e gestão orçamentária.

Figura 6.12: Dashboard FinOps projetando custos e tempos baseados na simulação com Jitter.



6.4.4 Evidências do Experimento (Modo *Database*)

Enquanto a demonstração em memória evidencia a correção funcional do sistema, a demonstração com persistência real é essencial para evidenciar o comportamento em escala. Esta subseção apresenta os resultados do processamento de aproximadamente 500 artigos acadêmicos, utilizando a mesma *query* de busca empregada na versão 1.0 do Hermeneutics.

6.4.4.1 Cenário de Demonstração E2E

A execução foi realizada com a seguinte configuração:

- **Query de Busca:** `("systematic literature review" OR "systematic review" OR "evidence-based") AND (clustering OR classification OR "information extraction" OR "information retrieval" OR "information visualization" OR "summarization" OR "text mining")`
- **Modo:** `PERSISTENCE_MODE=database` (PostgreSQL real)
- **Volume:** ∼500 artigos de múltiplas bibliotecas digitais

A arquitetura de Saga Coreografada utiliza **batching** para processar grandes volumes sem sobrecarregar os serviços individuais. A Tabela 6.3 detalha a estratégia adotada:

Tabela 6.3: Estratégia de Batching na Saga Coreografada.

Etapa	**Serviço**	**Lote**	**Justificativa**
Busca	Research	100 artigos/batch	Simula latência de busca (dataset legado v1)
Anotação	Annotation	25 artigos/batch	Evita timeout de modelos LLM/SpaCy
Persistência	Persistence	10 artigos/batch	Transações menores, rollback seguro

Esta estratégia implica que o total de interações entre microsserviços é significativamente maior que o número bruto de artigos, gerando *traces* complexos conforme demonstrado adiante.

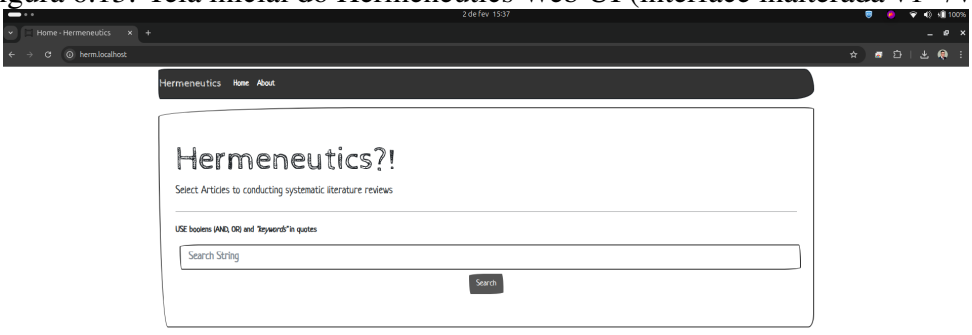
Os artigos processados neste experimento provêm do **dataset legado v1**, importado para o ambiente v2 conforme a **ADR-031**. O banco de dados original conta com 3.779 artigos únicos anotados, distribuídos em cinco *search strings* distintas, incluindo *datasets* de treino para modelos SpERT e a query acadêmica real utilizada neste trabalho. O *schema* original (DER) foi **preservado** (princípio YAGNI), com adição apenas da coluna `saga_id` para rastreabilidade das execuções v2.

Adicionalmente, o **Web UI** foi refatorado para utilizar **Redis** como *backend* de sessões (ao invés do SQLite padrão do Django), permitindo que a camada de apresentação seja *stateless* e escalável horizontalmente — um requisito fundamental para ambientes Kubernetes.

6.4.4.2 Interface do Usuário (Web UI)

A experiência do usuário final permanece praticamente inalterada entre as versões v1 e v2, evidenciando o sucesso da refatoração transparente. A Figura 6.13 apresenta a tela inicial do sistema.

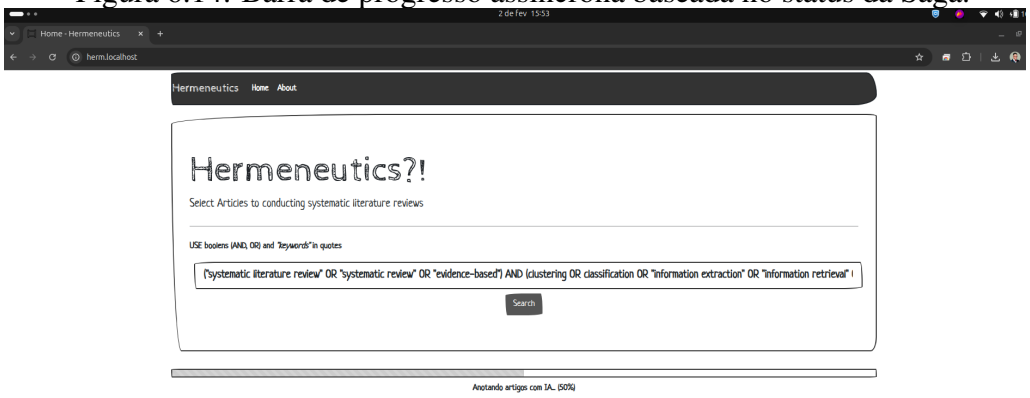
Figura 6.13: Tela inicial do Hermeneutics Web UI (interface inalterada v1→v2).



A v2 processa artigos de forma assíncrona via Saga. A Figura 6.14 demonstra

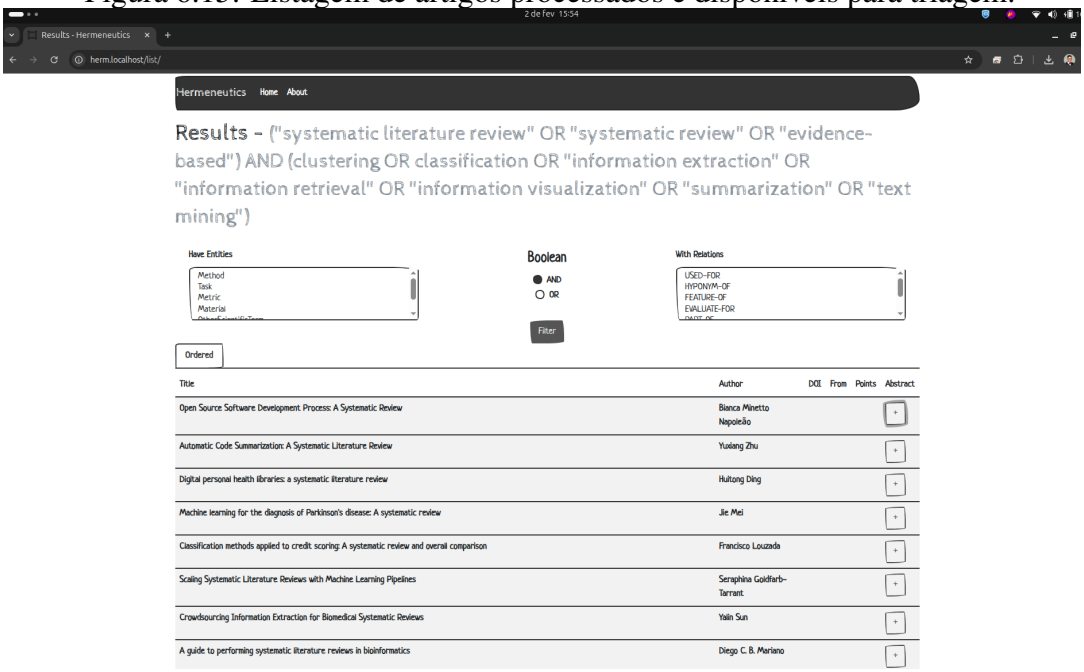
a barra de progresso que atualiza via *polling* AJAX ao *endpoint* de status do Maitre, refletindo o estado atual da coreografia Kafka.

Figura 6.14: Barra de progresso assíncrona baseada no status da Saga.



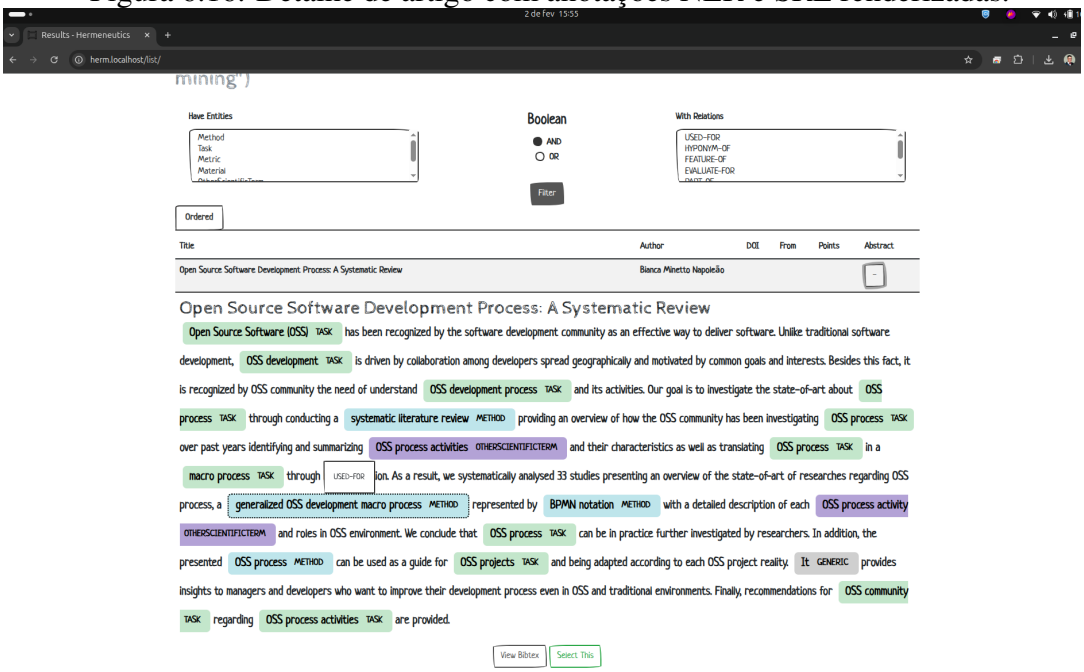
Após a conclusão, a listagem de resultados (Figura 6.15) exibe todos os artigos processados, agora armazenados no PostgreSQL e disponíveis para triagem.

Figura 6.15: Listagem de artigos processados e disponíveis para triagem.



O detalhe de um artigo (Figura 6.16) exibe as anotações de Reconhecimento de Entidades Nomeadas (NER) e Rotulação de Papel Semântico (SRL) aplicadas ao *abstract*, demonstrando a integração do pipeline de anotação.

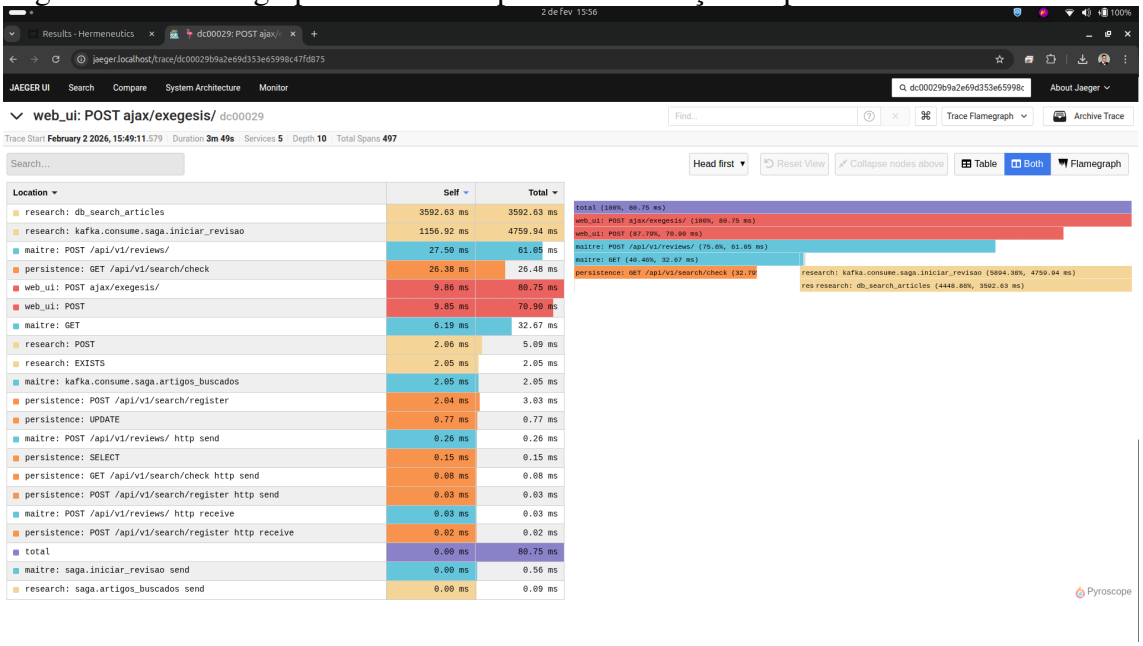
Figura 6.16: Detalhe de artigo com anotações NER e SRL renderizadas.



6.4.4.3 Análise de Rastreabilidade em Escala

O processamento de 500 artigos em lotes gera *traces* significativamente mais complexos que os cenários de teste. A Figura 6.17 apresenta o *Flamegraph* de uma saga completa, evidenciando a profundidade das chamadas entre serviços e os "platôs"correspondentes ao processamento em batch.

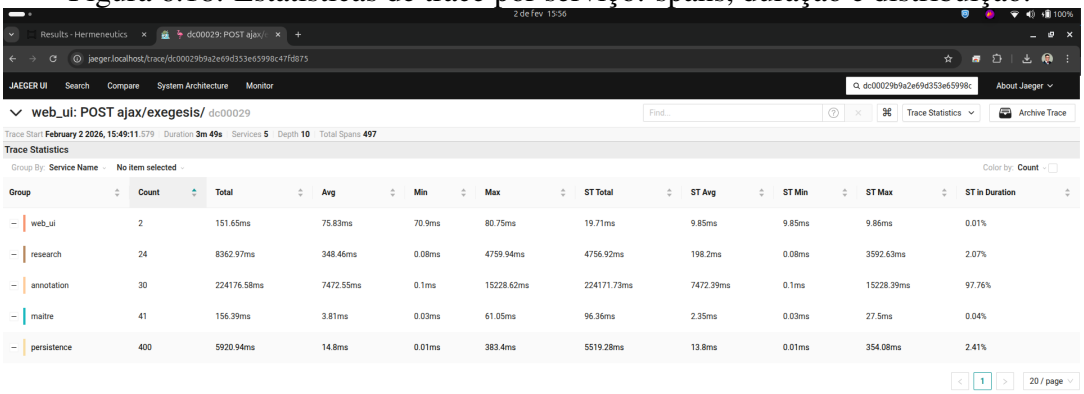
Figura 6.17: Flamegraph de trace completo: visualização da profundidade de chamadas.



A Figura 6.18 complementa a análise com estatísticas tabulares por serviço, indi-

cando o número de spans, a duração média e a distribuição de tempo entre os componentes.

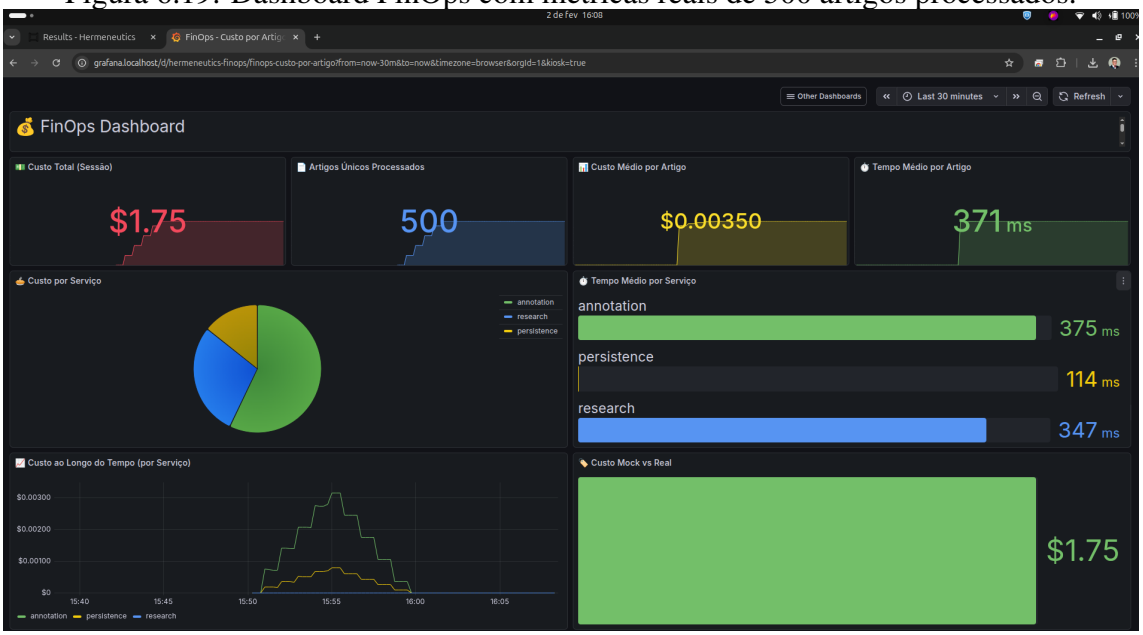
Figura 6.18: Estatísticas de trace por serviço: spans, duração e distribuição.



6.4.4.4 Dashboard FinOps (Escala Real)

A Figura 6.19 apresenta o Dashboard FinOps com métricas reais do processamento de 500 artigos. Diferentemente da simulação em memória (10 artigos), este cenário permite validar a **linearidade dos custos** e identificar oportunidades de otimização.

Figura 6.19: Dashboard FinOps com métricas reais de 500 artigos processados.



6.4.4.5 Otimização: Cache Hit

Uma otimização fundamental para FinOps é evitar o reprocessamento de dados já conhecidos. O Hermeneutics v2 implementa um protótipo de *cache* baseado na *query* de busca: se uma *string* já foi processada anteriormente, os resultados são recuperados do banco sem acionar a Saga novamente.

A Figura 6.20 evidencia nos logs do Loki a detecção do *cache hit*, enquanto a Figura 6.21 demonstra o *trace* extremamente curto resultante — sem chamadas a Research, Annotation.

Figura 6.20: Logs no Loki evidenciando detecção de Cache Hit.

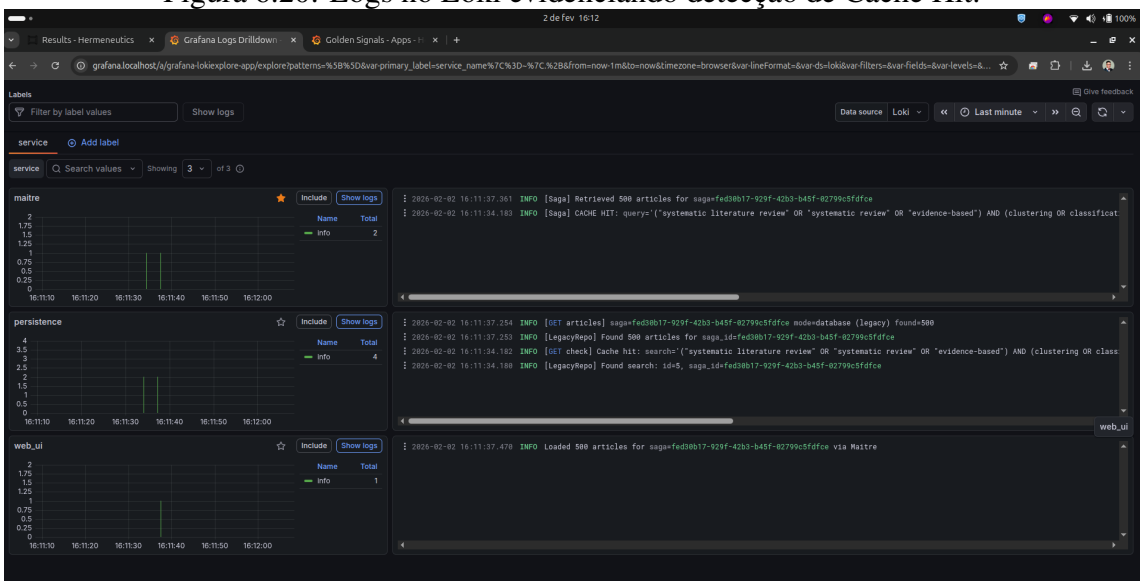
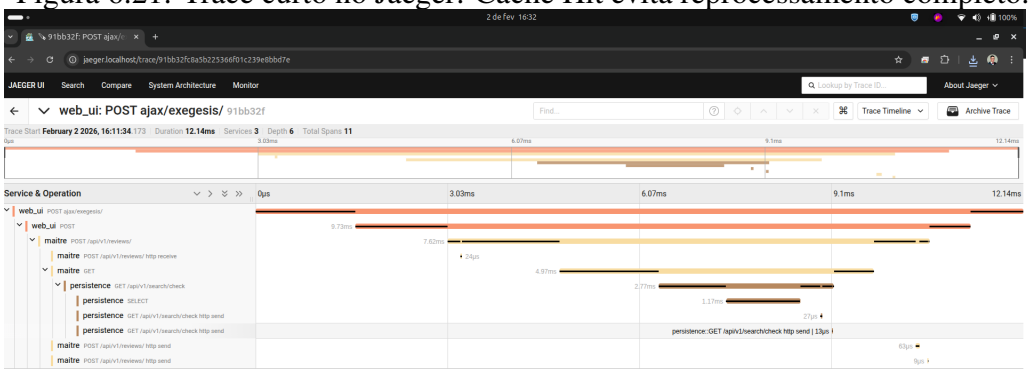


Figura 6.21: Trace curto no Jaeger: Cache Hit evita reprocessamento completo.



Esta otimização resulta em:

- **Carregamento instantâneo** (<100ms vs minutos) para *queries* repetidas.
- **Custo zero adicional** de processamento, impactando diretamente as métricas FinOps.

- **Redução de carga** nos serviços *downstream*, liberando capacidade para novas requisições.

6.5 Análise Comparativa: Do Monólito à Arquitetura Distribuída

A avaliação da proposta deste trabalho não se dá apenas pela execução funcional, mas pelo contraste qualitativo entre a arquitetura legada (*Hermeneutics v1*) e a nova arquitetura proposta (*Hermeneutics v2*). Esta seção analisa os ganhos obtidos sob a ótica dos Requisitos Não Funcionais (NFRs) estabelecidos no planejamento do projeto.

6.5.1 Persistência e Estratégia de Dados

Na v1, toda a aplicação compartilhava um único banco PostgreSQL monolítico, violando o princípio de isolamento de dados entre domínios. Na v2, adotou-se o padrão **Database per Service** (**ADR-024**), onde cada microsserviço possui seu próprio *database* lógico (`maitre_db`, `research_db`, `persistence_db`).

Entretanto, seguindo os princípios **KISS** e **YAGNI**, optou-se por uma implementação pragmática para o ambiente de desenvolvimento e TCC:

- **Single Container Strategy:** Ao invés de múltiplos *containers* PostgreSQL (um por serviço), utilizou-se um único *container* hospedando múltiplos *databases* lógicos. Isso economiza recursos significativos sem violar o isolamento arquitetural, pois *queries cross-database* exigem configuração explícita (`dblink`).

- **Reuso do DER Legado:** O banco `hermeneutics` da v1 foi importado integralmente via **ADR-031**, preservando o *schema* original (DER). A única alteração foi a adição da coluna `saga_id` (UUID) para associar registros a execuções da Saga v2.

- **Schemas Simples:** Os bancos dos novos serviços (Maitre, Research) possuem estruturas mínimas, evitando a criação de *schemas* complexos que não seriam utilizados nesta fase.

6.5.2 Acoplamento, Coesão e Manutenibilidade

Na v1, a lógica de extração de dados (*crawlers*), processamento de NLP (SpERT) e interface de usuário compartilhavam o mesmo ciclo de vida de memória e banco de dados, violando o princípio da responsabilidade única (SRP). Uma falha na biblioteca `torch` (modelo de NER) causava indisponibilidade da interface web.

Na v2, a adoção da **Arquitetura Hexagonal** resultou em desacoplamento efetivo:

- **Isolamento de Falhas:** O serviço WebUI permanece operacional mesmo se o Annotation falhar.
- **Evolução Independente:** É possível atualizar a versão do Python do Research sem impactar o Maitre.
- **Adoção de DRY:** A criação de bibliotecas compartilhadas (`hermeneutics_kafka`, `hermeneutics_db`, `hermeneutics_otel`) centralizou a lógica de infraestrutura, evitando duplicação de código boilerplate.

6.5.3 Estratégia de Deploy e Escalabilidade

Enquanto a v1 exigia escalabilidade **vertical** (aumentar recursos da máquina inteira), a v2 permite escalabilidade **horizontal granular**. O serviço de Anotação, intensivo em CPU/GPU, pode ter múltiplas réplicas, enquanto o Gateway permanece com recursos mínimos.

A implementação seguiu princípios de engenharia parcimoniosa:

- **KISS:** Optou-se por orquestrar via **Docker Compose** para demonstração local, evitando a complexidade acidental de um *cluster* Kubernetes completo nesta fase.
- **YAGNI:** A integração com LLMs comerciais (OpenAI/Gemini) foi abstraída via interfaces (Ports), mas implementada com **Mocks** controlados. Isso permitiu focar na robustez do pipeline de dados (Saga) sem custos prematuros de API.

6.5.4 Consistência de Dados e Mensageria

A transição de chamadas síncronas (v1) para arquitetura orientada a eventos (v2) introduziu o desafio da consistência eventual. A implementação do **Padrão Saga Coreografada** via Apache Kafka demonstrou ser superior ao modelo anterior de filas simples (Celery) para fluxos longos e complexos.

Os *traces* distribuídos apresentados na Seção 6.4.4 evidenciam que o sistema mantém a integridade transacional sem bloquear recursos, cumprindo o NFR de **Resiliência**.

6.5.5 Observabilidade e Suporte a FinOps

A maior lacuna da v1 era sua "caixa-preta": erros silenciosos nos crawlers eram difíceis de diagnosticar. A v2, instrumentada com **OpenTelemetry**, fornece transparência total:

- **Dados como Ativo:** A telemetria gerada não serve apenas para *debug*, mas constitui um dataset valioso de comportamento de sistema.
- **Fundação para FinOps:** Através dos *spans* de rastreamento, é possível mensurar o tempo exato de processamento de cada artigo, habilitando o cálculo do "Custo por Artigo Processado" – algo impossível na arquitetura anterior onde recursos eram compartilhados indiscriminadamente.

6.5.6 Síntese Comparativa

A Tabela 6.4 sumariza as mudanças arquiteturais realizadas para atender aos requisitos de Engenharia de Software para aplicações *Data-Intensive*.

Tabela 6.4: Comparativo Arquitetural: Hermeneutics v1 (AS-IS) vs. Hermeneutics v2 (TO-BE).

Critério/NFR	**Hermeneutics v1 (Legado)**	**Hermeneutics v2 (Modernizado)**
Estilo Arquitetural	Monolito Modular (Django MTV)	Microsserviços (Hexagonal/DDD)
Comunicação	Chamadas locais e Filas Simples (Celery)	Orientada a Eventos (Kafka) + REST Gateway
Persistência	Banco Relacional Único	Database per Service (lógico)
Consistência	Transações ACID (commit atômico)	Consistência Eventual (Saga Coreografada)
Inteligência Artificial	SpERT acoplado ao código (Hard-coded)	Abstração via Ports/Adapters (Mockável)
Observabilidade	Logs em arquivo texto	Full-stack: Traces, Métricas, Logs (OTEL)
Escalabilidade	Vertical (todo o sistema)	Horizontal (por domínio/serviço)
Sessões Web	SQLite/Django Sessions	Redis (stateless, escalável)
Engenharia	Scripts manuais	CI/CD Pipelines, Pre-commit, Monorepo

7 CONCLUSÃO

Este capítulo apresenta as considerações finais do trabalho, sintetizando as contribuições alcançadas, explicitando as limitações de escopo e delineando um *roadmap* de trabalhos futuros para a continuidade do projeto Hermeneutics.

7.1 Síntese das Contribuições

O objetivo geral deste trabalho consistiu em projetar e implementar a evolução arquitetural da ferramenta Hermeneutics (v2), migrando de um monólito legado para uma arquitetura distribuída baseada em microsserviços, orientada a eventos e observabilidade. Conforme demonstrado ao longo dos capítulos anteriores, esse objetivo foi atingido através de uma abordagem sistemática de engenharia de software.

A decomposição do monólito original foi realizada seguindo os princípios de *Domain-Driven Design* (DDD), resultando na identificação de quatro *Bounded Contexts* distintos: Ingestão (Research), Anotação Semântica (Annotation), Persistência (Persistence) e Orquestração (Maître). Cada contexto foi implementado como um microsserviço autônomo, comunicando-se através de eventos assíncronos publicados no Apache Kafka.

A consistência eventual entre os serviços foi garantida pela implementação do padrão **Saga Coreografada**, conforme detalhado no Capítulo 5. Essa abordagem permitiu a execução de transações de longa duração – como o fluxo completo de busca, anotação e persistência de artigos – sem bloquear recursos ou comprometer a resiliência do sistema.

A camada de observabilidade implementada com **OpenTelemetry** representa uma das contribuições centrais deste trabalho. Diferentemente da versão anterior, onde erros silenciosos nos *crawlers* eram difíceis de diagnosticar, a arquitetura v2 fornece transparência total através da correlação de Logs, Métricas e Traces. Conforme evidenciado no Capítulo 6, é possível rastrear uma transação desde sua origem no Web UI até a persistência final, identificando gargalos e pontos de falha com precisão.

7.1.1 Observabilidade como Ativo de Dados

Um resultado emergente da instrumentação implementada é a transformação do comportamento do sistema em dados analisáveis. Os dados de telemetria (traces e mé-

tricas) gerados pela v2 constituem, por si só, um *dataset* valioso para análise de desempenho e otimização. A engenharia implementada não apenas habilita o monitoramento operacional, mas também fornece a fundação para análises quantitativas do pipeline de processamento.

7.1.2 Fundação para FinOps

Com o rastreamento distribuído, é possível medir exatamente quanto tempo de computação cada artigo consome em cada etapa do pipeline (busca, anotação, persistência). Essa granularidade permite, em trabalhos futuros, calcular o custo unitário por revisão sistemática – o conceito de *FinOps Unit Economics* (FinOps Foundation Technical Advisory Council, 2025). Essa capacidade é essencial para a viabilidade econômica de RSLs assistidas por LLMs pagos, onde cada chamada de API representa um custo mensurável.

7.1.3 Extensibilidade para *Machine Learning*

A adoção da Arquitetura Hexagonal (Ports & Adapters) confere ao Hermeneutics v2 uma extensibilidade natural para integração com modelos de *Machine Learning*. A implementação de novos *Adapters* permite a instanciação de modelos personalizados, acoplando práticas de **MLOps** ao pipeline existente sem modificar a lógica de domínio.

Essa arquitetura abre possibilidades para a implementação de:

- Algoritmos de ranqueamento de artigos por relevância, similaridade semântica ou *clustering* temático;
- Técnicas de sumarização automática de *abstracts* e textos completos;
- Detecção de duplicatas e artigos similares através de embeddings;
- Classificação temática automatizada para triagem inicial.

A separação clara entre portas (interfaces de domínio) e adaptadores (implementações de infraestrutura) garante que a evolução dos modelos de ML não impacte a estabilidade do pipeline de dados.

7.2 Limitações do Trabalho

Em consonância com o rigor acadêmico, esta seção explicita as limitações de escopo e as decisões de implementação que delimitam as fronteiras do presente trabalho.

7.2.1 Uso de *Mocks* para Modelos de Linguagem

Aplicando o princípio **YAGNI** (*You Ain't Gonna Need It*) (HUNT; THOMAS, 1999), a integração com LLMs comerciais (OpenAI, Google Gemini) foi abstraída através de interfaces (*Ports*), porém implementada com *mocks* controlados. Essa decisão permitiu focar na robustez do pipeline de *mensageria* (Saga) antes de incorrer em custos prematuros de APIs pagas.

7.2.1.1 Complexidade do Modelo SpERT

O modelo SpERT (EBERTS; ULGES, 2019), utilizado na versão v1 para Reconhecimento de Entidades Nomeadas (NER) e Rotulação de Papel Semântico (SRL), é baseado na arquitetura BERT (2018) e requer TensorFlow ou PyTorch para inferência. A portabilidade deste modelo para a arquitetura v2 apresenta desafios significativos:

- **Peso computacional elevado:** A inferência de modelos BERT-based requer GPU para execução em tempo hábil, inviabilizando a avaliação em ambientes de desenvolvimento com recursos limitados;

- **Dependências conflitantes:** As versões de TensorFlow/PyTorch frequentemente conflitam com outras bibliotecas do ecossistema Python, dificultando a containerização;

- **Manutenção do modelo:** O *fine-tuning* e a atualização do modelo para novos domínios científicos exigem expertise em *deep learning* e acesso a *datasets* anotados.

A decisão de postergar essa integração permitiu validar completamente o *pipeline* de *mensageria* antes de adicionar a complexidade de inferência de modelos *deep learning*.

7.2.2 Ausência de Deploy em Cloud Real

O ambiente de demonstração foi implementado localmente via Docker Compose, não em infraestrutura de nuvem real. Seguindo os princípios **KISS** (*Keep It Simple, Stupid*) e a Navalha de Ockham, evitou-se a complexidade acidental de um *cluster* Kubernetes completo nesta etapa, optando-se por demonstrar a arquitetura lógica antes da infraestrutura física (HUNT; THOMAS, 1999).

Consequentemente, as métricas de latência e escalabilidade apresentadas são aproximações obtidas em ambiente controlado, não representando o comportamento exato em produção com carga real de usuários concorrentes.

7.2.3 APIs de Bibliotecas Digitais Simuladas

O serviço Research opera com dados do dataset legado v1, não realizando chamadas reais às APIs de bibliotecas digitais (IEEE Xplore, Scopus, ACM) em todos os cenários de teste. Essa limitação decorre de dependências externas:

- Renovação periódica de *tokens* de acesso;
- Necessidade de VPN institucional para acesso a bases restritas;
- Políticas de *rate limiting* das APIs comerciais.

A implementação da classe `DigitalLibraryAdapter` na v1 foi preservada e pode ser reativada quando as credenciais e conectividade estiverem disponíveis.

7.2.4 Ausência de Avaliação Experimental com Usuários

O escopo deste trabalho focou na avaliação técnica da arquitetura, não incluindo avaliação de usabilidade com usuários finais (pesquisadores). A interface web foi mantida funcionalmente equivalente à v1, priorizando a demonstração do pipeline distribuído sobre melhorias de experiência do usuário.

7.3 Trabalhos Futuros

A arquitetura implementada estabelece uma fundação sólida para a evolução contínua da ferramenta Hermeneutics. Esta seção apresenta um *roadmap* técnico estruturado em categorias, indicando os próximos passos viáveis para a continuidade do projeto.

7.3.1 Infraestrutura e *Deploy*

- **Implementação em Kubernetes com Argo CD:** Migração do ambiente Docker Compose para um *cluster* Kubernetes gerenciado, utilizando GitOps através do Argo CD para *deploys* declarativos e auditáveis. Essa evolução habilitaria escalabilidade horizontal real dos microsserviços.

- **Deploy em Cloud Real com Terraform:** Provisionamento de infraestrutura em provedores *cloud* (AWS, GCP ou Azure) utilizando *Infrastructure as Code* (IaC) via Terraform, garantindo reprodutibilidade e versionamento do ambiente de produção.

- **Alertas e SRE:** Implementação de políticas de *alerting* baseadas nos *Golden Signals* (latência, tráfego, erros, saturação) e estabelecimento de práticas de *Site Reliability Engineering* para operação sustentável.

7.3.2 Integração com Modelos de Linguagem

- **Substituição dos Mocks por LLMs Reais:** Implementação de *Adapters* para integração com APIs comerciais (OpenAI GPT, Google Gemini) ou modelos *open-source* (LLaMA, Mistral), substituindo os *mocks* atuais por inferência real.

- **Portabilidade do SpERT ou Modelos Alternativos:** Containerização do modelo SpERT com suporte a GPU, ou avaliação de modelos mais leves, como NuNER (VARDA et al., 2024) para NER em ambientes com recursos limitados.

7.3.3 Integração com Bibliotecas Digitais

- **APIs Reais de Bibliotecas Digitais:** Reativação da classe `DigitalLibraryAdapter` da v1 com implementação de renovação automática de *tokens*, *proxy* para ambientes universitários e tratamento de *rate limiting*.

7.3.4 Evolução de FinOps

- **Cálculo de Custo de Tokens LLM:** Expansão do *dashboard* FinOps para incluir anotação e cálculo de consumo de *tokens* em modelos pagos, permitindo projeção de custos por revisão sistemática completa.

7.3.5 Qualidade e Testes

- **Cobertura de Testes End-to-End:** Implementação de testes de carga com Grafana K6, testes de interface com Cypress e expansão da cobertura de testes unitários com Pytest.
- **Relatórios e Implementação de Login:** Adição de autenticação de usuários e extração de relatórios analíticos sobre as revisões sistemáticas realizadas.

7.3.6 Evolução da Interface

- **Ajustes no Frontend:** Implementação de melhorias de UX pendentes da v1, incluindo responsividade para dispositivos móveis e modernização da *stack frontend* (possível migração para Next.js ou tecnologia equivalente).

7.3.7 Evolução Metodológica

- **Suporte Completo ao Protocolo PRISMA:** Evolução da ferramenta para suporte integral às diretrizes PRISMA 2020 (PAGE et al., 2021) e PRISMA-S (RETHLEFSEN et al., 2021), incluindo geração automática de diagramas de fluxo e *checklists* de conformidade.

7.3.8 Perspectivas de Continuidade

- **Versão 3 e Grupo de Pesquisa:** Disponibilização da ferramenta para uso real pelo grupo de pesquisa do Instituto de Informática, com possível coorientação de trabalhos de graduação para evolução contínua.
- **Monetização e Registro INPI:** Avaliação de possibilidades de licenciamento comercial e registro de propriedade intelectual junto ao Instituto Nacional da Propriedade Industrial (INPI), considerando o potencial de aplicação em contextos corporativos de gestão do conhecimento.

Considerações Finais

Este trabalho demonstrou que a modernização de software científico legado é viável através da aplicação sistemática de padrões de engenharia de software consolidados. A migração do Hermeneutics para uma arquitetura de microsserviços, orientada a eventos e instrumentada com observabilidade, estabelece as bases para que a ferramenta evolua de um protótipo acadêmico para uma plataforma robusta de apoio à Revisão Sistemática da Literatura.

A transparência sobre as limitações — notadamente o uso de *mocks* para modelos de IA e a ausência de *deploy* em *cloud* real — não diminui a contribuição técnica, mas delimita honestamente o escopo entregue. Os trabalhos futuros delineados constituem um *roadmap* acionável que pode guiar a continuidade do projeto, seja em contexto acadêmico ou comercial.

A arquitetura implementada não é apenas funcional; é evolutiva. E essa capacidade de evolução, garantida pela separação de preocupações e pela observabilidade intrínseca, representa a verdadeira herança técnica deste trabalho.

REFERÊNCIAS

Amazon Web Services. **Pattern database-per-service**. 202X. Online. Available from Internet: <https://microservices.io/patterns/data/database-per-service.html>.

BLANCO, D. G. **Practical OpenTelemetry: Adopting Open Observability Standards Across Your Organization**. [S.l.]: Apress, 2023. ISBN 978-1-4842-9074-3 (pbk); 978-1-4842-9075-0 (electronic).

BROWN, T. et al. Language models are few-shot learners. **arXiv preprint arXiv:2005.14165**, 2020.

EBERTS, M.; ULGES, A. **Span-based Joint Entity and Relation Extraction with Transformer Pre-training**. 2019.

EVANS, E. **DomainDriven Design: Atacando as Complexidades no Coração do Software**. 3. ed. São Paulo: Alta Books, 2016. 528 p. Tradução em português, edição brasileira da Alta Books.

FinOps Foundation Technical Advisory Council. **What is FinOps?** 2025. <https://www.finops.org/introduction/what-is-finops/>. Acessado em 28 de agosto de 2024. Atualizado em: Janeiro de 2025.

HUNT, A.; THOMAS, D. **The Pragmatic Programmer: From Journeyman to Master**. [S.l.]: Addison-Wesley Professional, 1999. Fonte original dos princípios DRY e KISS aplicados a software. ISBN 978-0201616224.

JUNIOR, L. C. S. **Migração de monolito para microsserviços: relatório da experiência do projeto integrador**. São Paulo: [s.n.], 2022. Monografia de Especialização (Engenharia de Software) – Faculdade de Ciências Exatas e Tecnologia, Pontifícia Universidade Católica de São Paulo. Available from Internet: <https://repositorio.pucsp.br/jspui/handle/handle/43864>.

KITCHENHAM, B. Procedures for performing systematic reviews. **Keele, UK, Keele Univ.**, v. 33, 08 2004.

KITCHENHAM, B.; CHARTERS, S. Guidelines for performing systematic literature reviews in software engineering. v. 2, 01 2007.

LEWIS, P. et al. Retrieval-augmented generation for knowledge-intensive nlp tasks. In: **Proceedings of the 34th International Conference on Neural Information Processing Systems**. Red Hook, NY, USA: Curran Associates Inc., 2020. (NIPS '20). ISBN 9781713829546.

LIMA, P. H. G. **Avaliação prática da abordagem Monolith First no desenvolvimento de aplicações com arquitetura de microsserviços**. Quixadá: [s.n.], 2025. Trabalho de Conclusão de Curso (Graduação em Engenharia de Software) – Universidade Federal do Ceará, Campus de Quixadá. 72f. Available from Internet: <http://repositorio.ufc.br/handle/riufc/80688>.

LUAN, Y. et al. Multi-task identification of entities, relations, and coreferencefor scientific knowledge graph construction. In: **Proc. Conf. Empirical Methods Natural Language Process. (EMNLP)**. [S.l.: s.n.], 2018.

MARSHALL, I. J.; WALLACE, B. C. Toward systematic review automation: a practical guide to using machine learning tools in research synthesis. **Systematic Reviews**, v. 8, n. 1, p. 163, 2019.

MARTIN, R. C. **Arquitetura Limpa: O Guia do Artesão para Estrutura e Design de Software**. 1. ed. [S.l.]: Alta Books, 2020. Tradução de "Clean Architecture: A Craftsman's Guide to Software Structure and Design"(2018). ISBN 978-85-508-1600-5.

NADEAU, D.; SEKINE, S. A survey of named entity recognition and classification. **Lingvisticae Investigationes**, v. 30, 01 2007.

NEWMAN, S. **Migrando sistemas monolíticos para microsserviços: Padrões evolutivos para transformar seu sistema monolítico**. 1. ed. [S.l.]: Novatec Editora, 2020. 288 p. Tradução da obra original: Monolith to Microservices. ISBN 978-6586057042.

NEWMAN, S. **Building Microservices: Designing Fine-Grained Systems**. 2nd. ed. O'Reilly Media, 2021. ISBN 978-1-492-07754-3. Available from Internet: <https://www.oreilly.com/library/view/building-microservices-2nd/9781492037099/>.

NIGH, M. **ChatGPT-Free-Prompt-List: The CRISPE Framework**. 2023. GitHub Repository. Framework estruturado para engenharia de prompts com LLMs. Available from Internet: <https://github.com/mattnigh/ChatGPT-Free-Prompt-List>.

NYGARD, M. T. **Documenting Architecture Decisions**. 2011. Blog post. Introdução do formato ADR (Architectural Decision Record). Available from Internet: <https://cognitect.com/blog/2011/11/15/documenting-architecture-decisions>.

OpenTelemetry Community. **Observability primer**. 2024. Acessado em [data atual de acesso]. Available from Internet: <https://opentelemetry.io/docs/concepts/observability-primer/>.

OpenTelemetry Community. **What is OpenTelemetry?** 2025. Acessado em [data atual de acesso]. Available from Internet: <https://opentelemetry.io/docs/what-is-opentelemetry/>.

OUZZANI, M. et al. Rayyan — a web and mobile app for systematic reviews. **Systematic Reviews**, v. 5, n. 1, p. 210, 2016. Available from Internet: <https://doi.org/10.1186/s13643-016-0384-4>.

OZKAYA, I. **Managed Evolution: A Strategy for Very Large Information Systems**. Addison-Wesley, 2023. ISBN 978-0-13-749033-2. Available from Internet: <https://www.pearson.com/en-us/subject-catalog/p/managed-evolution/P200000009239>.

PAGE, M. J. et al. The prisma 2020 statement: an updated guideline for reporting systematic reviews. **BMJ**, v. 372, n. n71, 2021.

RETHLEFSEN, M. L. et al. Prisma-s: an extension to the prisma statement for reporting literature searches in systematic reviews. **Systematic Reviews**, v. 10, n. 1, p. 39, 2021.

RICHARDSON, C. **Pattern: Saga**. 2025. Microservice Architecture (Microservices.io). Acesso em: [Data de Acesso]. Available from Internet: <https://microservices.io/patterns/data/saga.html>.

Scaled Agile, Inc. **SAFe 6.0 Framework**. 2023. Online. Framework ágil escalado para organização de EPICs e Features. Available from Internet: <https://scaledagileframework.com/>.

SEBASTIANI, F. Machine learning in automated text categorization. **ACM Computing Surveys**, v. 34, p. 1–47, 04 2001.

SILVA, D. L. d. S. P. **Hermeneutics: Ferramenta de auxílio para realização de revisões sistemáticas da literatura**. Porto Alegre, RS, Brasil, 2021. Orientador: Leandro Krug Wives; Coorientador: Vinicius Woloszyn. Available from Internet: <https://lume.ufrgs.br/handle/10183/235003>.

ŠTEFANKO, M. et al. The saga pattern in a reactive microservices environment. In: SCITEPRESS PRAGUE, CZECH REPUBLIC. **Proc. 14th Int. Conf. Softw. Technologies (ICSOFT 2019)**. [S.l.], 2019. p. 483–490.

STRADOLINI, C. J. **Migração de sistemas monolíticos para microsserviços: estudo de caso de migração de um módulo de pagamentos de e-Commerce**. Porto Alegre: [s.n.], 2022. Trabalho de Conclusão de Curso – Universidade Federal do Rio Grande do Sul. Available from Internet: <http://hdl.handle.net/10183/243216>.

THAKUR, A.; CHANDAK, M. B. A review on opentelemetry and http implementation. **International Journal of Health Sciences**, v. 6, n. S2, p. 15013–15023, 2022.

VARDA, J. de et al. NuNER: A compact foundation model for named entity recognition. **arXiv preprint arXiv:2402.17611**, 2024.

WEI, J. et al. Finetuned language models are zero-shot learners. **arXiv preprint arXiv:2109.01652**, 2022.

WHITE, J. et al. A prompt pattern catalog to enhance prompt engineering with chatgpt. **arXiv preprint arXiv:2302.11382**, 2023. Catálogo de padrões de design para prompts de LLM. Available from Internet: <https://arxiv.org/abs/2302.11382>.

APÊNDICE A — REGISTRO DE DECISÕES ARQUITETURAIS (ADRS)

Os *Architecture Decision Records* (ADRs) são documentos que registram decisões arquiteturais significativas, seu contexto e consequências. Eles formam a memória técnica do sistema, seguindo a filosofia *Docs-as-Code* (DaC) adotada no projeto Hermeneutics v2.0.

A Tabela A.1 apresenta o índice completo das 32 ADRs documentadas durante a evolução arquitetural do projeto.

Tabela A.1: Índice de Decisões Arquiteturais (ADRs) do Projeto Hermeneutics v2.0

ID	**Título**	**Status**	**Categoria**
ADR-001	Adoção da Arquitetura de Microsserviços (DDD)	Aceito	Arquitetura Core
ADR-002	Mensageria Assíncrona com Kafka e Saga	Aceito	Arquitetura Core
ADR-003	Padrão Database per Service	Aceito	Arquitetura Core
ADR-004	Implementação de API Gateway	Aceito	Arquitetura Core
ADR-005	Observabilidade com OpenTelemetry	Aceito	Arquitetura Core
ADR-006	Padronização da Stack Tecnológica	Aceito	Stack e Infraestrutura
ADR-007	Contêiner Inicializador para Tópicos Kafka	Aceito	Stack e Infraestrutura
ADR-008	Estratégia de Build Unificado (Dockerfile)	Aceito	Stack e Infraestrutura
ADR-009	Shared Kernel para Infraestrutura Kafka	Aceito	Stack e Infraestrutura
ADR-010	Object Storage Local com MinIO	Aceito	Stack e Infraestrutura

Continua na próxima página

Tabela A.1 – continuação da página anterior

ID	Título	Status	Categoria
ADR-011	Métricas de Longo Prazo com Thanos	Aceito	Observabilidade
ADR-012	Rastreamento Distribuído com Tempo	Aceito	Observabilidade
ADR-013	Agregação de Logs com Loki	Aceito	Observabilidade
ADR-014	Monitoramento de Golden Signals	Aceito	Observabilidade
ADR-015	Estratégia de Instrumentação OpenTelemetry	Aceito	Observabilidade
ADR-016	Abordagem Híbrida OpenTelemetry	Aceito	Observabilidade
ADR-017	Upgrade do Django para 4.2 LTS	Aceito	Padronização
ADR-018	Uso do Padrão Facade no Maître (Mock E2E)	Aceito	Padronização
ADR-019	Docker Compose Naming Convention	Aceito	Padronização
ADR-020	Makefile Naming Convention	Aceito	Padronização
ADR-021	API Gateway com Subdomínios Localhost	Aceito	Padronização
ADR-022	Shared Kernel para FastAPI (App Factory)	Aceito	Padronização
ADR-023	Padronização SCM (.editorconfig/.gitattributes)	Aceito	Padronização
ADR-024	Estratégia de Banco de Dados (Single Container)	Aceito	Persistência
ADR-025	Shared Kernel para Conectividade PostgreSQL	Aceito	Persistência

Continua na próxima página

Tabela A.1 – continuação da página anterior

ID	Título	Status	Categoria
ADR-026	Shared Kernel para Cache Redis	Aceito	Persistência
ADR-027	ConnectionMixin Pattern para Repositórios	Aceito	Persistência
ADR-028	Feature Toggles para Modos de Operação	Aceito	Feature Toggles e FinOps
ADR-029	BFF Stateless com Redis Sessions	Aceito	Feature Toggles e FinOps
ADR-030	Kafka Batch Publishing para Mensagens Grandes	Aceito	Feature Toggles e FinOps
ADR-031	ETL Strategy para Migração de Legados	Aceito	Feature Toggles e FinOps
ADR-032	Delays Centralizados com Jitter (FinOps)	Aceito	Feature Toggles e FinOps

A.1 Decisões de Arquitetura Core

Esta seção apresenta as decisões fundamentais que definem a estrutura arquitetural do sistema Hermeneutics v2.0.

A.1.1 ADR-001: Adoção da Arquitetura de Microsserviços (DDD)

Status: Aceito **Data:** 2025-01-01 **Tags:** arquitetura, DDD, microsserviços

Contexto e Problema: A ferramenta original (v1.0) foi construída como um monólito com alto acoplamento entre os módulos (crawlers, web, ML). Este acoplamento impedia a escalabilidade isolada do módulo de Processamento de Linguagem Natural (NLP) e dificultava a manutenção.

Decisão: O sistema será decomposto em microsserviços por meio da aplicação do *Domain-Driven Design* (DDD) para definir *Bounded Contexts* (Ingestão, Anotação, Persistência).

Consequências:

[+] Maior escalabilidade granular (escalar o módulo de Anotação Semântica separadamente).

[+] Facilidade de manutenção e evolução independente dos módulos.

[-] Aumento da complexidade operacional (deploy, monitoramento).

[-] Necessidade de orquestração de serviços distribuídos.

A.1.2 ADR-002: Mensageria Assíncrona com Apache Kafka e Saga Pattern

Status: Aceito **Data:** 2025-01-01 **Tags:** Kafka, Saga, assíncrono

Contexto e Problema: O modelo monolítico utilizava Celery e RabbitMQ para orquestrar o fluxo assíncrono (busca e anotação), resultando em uma fila única de processamento, o que gerava gargalos e potencial perda de mensagens em picos de carga.

Decisão: Adotar Apache Kafka como barramento de eventos (*Event Bus*) e implementar o Padrão Saga Coreografada para transações distribuídas.

Consequências:

[+] Garantia de entrega e resiliência para os eventos de domínio (como `ArtigosBuscados`).

[+] Maior elasticidade no processamento de cargas de trabalho.

[-] Necessidade de lidar com consistência eventual dos dados, visto que as Sagas operam sob o modelo BASE.

A.1.3 ADR-003: Adoção do Padrão Database per Service

Status: Aceito **Data:** 2025-01-01 **Tags:** database, PostgreSQL, isolamento

Contexto e Problema: O sistema monolítico (v1.0) utilizava um banco de dados relacional compartilhado (PostgreSQL) para todos os módulos. Este banco compartilhado criava dependências ocultas e dificultava a evolução do esquema de forma isolada.

Decisão: Cada microsserviço (Maître, Research, Annotation) terá seu próprio esquema ou instância PostgreSQL isolada (*Database per Service*).

Consequências:

[+] Eliminação de dependências ocultas entre serviços.

[+] Isolamento de persistência, permitindo escolha de tecnologias adequadas para cada caso.

[+] Facilitação da escalabilidade e governança de dados.

[-] O isolamento exige garantia de consistência eventual através do Padrão Saga (ADR-002).

A.1.4 ADR-004: Implementação de API Gateway

Status: Aceito **Data:** 2025-01-01 **Tags:** API Gateway, segurança, roteamento

Contexto e Problema: O cliente (Frontend) acessava serviços diretamente na versão monolítica (v1.0), o que, na transição para microsserviços, exporia a topologia interna e criaria potenciais falhas de segurança e complexidade de roteamento.

Decisão: Implementar um API Gateway que atuará como o ponto de entrada único para todos os clientes, sendo responsável pelo roteamento, autenticação (AuthN/Z) e agregação de respostas.

Consequências:

[+] Desacoplamento entre o Frontend e a topologia de microsserviços do Backend.

[+] Segurança centralizada (AuthN/Z).

[+] Abstração da complexidade interna para os clientes.

[-] Ponto único de falha se não for bem arquitetado (necessita alta disponibilidade).

A.1.5 ADR-005: Adoção de Observabilidade com OpenTelemetry

Status: Aceito **Data:** 2025-01-01 **Tags:** observabilidade, OpenTelemetry, tracing

Contexto e Problema: A depuração de problemas em um ambiente distribuído (microsserviços, filas Kafka) é inviável apenas com logs locais, sendo necessária uma visão holística e correlacionada do fluxo de requisições.

Decisão: Instrumentar todos os serviços com o framework OpenTelemetry, coletando os três pilares da observabilidade: Logs, Métricas e Traces.

Consequências:

[+] Rastreabilidade total das transações (Saga) através de serviços.

[+] Redução drástica do tempo de resolução de incidentes (MTTK).

[+] Suporte à criação de indicadores FinOps como "custo por artigo processado".

A.2 Decisões de Stack e Infraestrutura

Esta seção documenta as decisões relacionadas à padronização tecnológica e configuração de infraestrutura.

A.2.1 ADR-006: Padronização da Stack Tecnológica

Status: Aceito **Data:** 2025-12-13 **Tags:** Python, FastAPI, stack

Contexto e Problema: Com a decomposição do monólito em múltiplos microsserviços, surge a necessidade de definir uma stack tecnológica homogênea. Sem padronização, cada serviço poderia usar diferentes versões ou frameworks, criando uma "torre de babel" tecnológica.

Decisão: Adotar FastAPI com Python 3.12 LTS como padrão para todos os novos microsserviços.

Justificativa: FastAPI alinha-se perfeitamente com os requisitos de uma arquitetura de microsserviços moderna. Sua natureza assíncrona é fundamental para construir serviços reativos e eficientes que interagem com o Kafka. A validação de dados via Pydantic e a documentação automática reduzem a complexidade acidental.

Consequências:

[+] Performance otimizada para operações de I/O, cruciais na comunicação entre serviços.

[+] Redução drástica de código *boilerplate* para validação e documentação de APIs.

[+] Facilita a adesão a contratos de API claros entre os serviços e o API Gateway.

[-] Exige que a equipe, acostumada com Django, aprenda os padrões do FastAPI e Pydantic.

A.2.2 ADR-007: Contêiner Inicializador para Tópicos Kafka

Status: Aceito **Tags:** infraestrutura, Kafka, Docker

Decisão: Implementar um contêiner efêmero (`kafka-init`) que cria automaticamente os tópicos necessários na inicialização do ambiente, garantindo idempotência e eliminando a necessidade de criação manual.

A.2.3 ADR-008: Estratégia de Build Unificado (Dockerfile Template)

Status: Aceito **Tags:** Docker, CI/CD, DRY

Decisão: Padronizar os Dockerfiles dos microsserviços usando *multi-stage build*, usuário não-root e um template base comum para garantir consistência, segurança e redução do tamanho das imagens.

A.2.4 ADR-009: Shared Kernel para Infraestrutura Kafka

Status: Aceito **Data:** 2025-12-27 **Tags:** Shared Kernel, biblioteca, DRY

Contexto e Problema: Durante a implementação dos Ports e Adapters para eventos, surgiu a questão de como estruturar o código de infraestrutura Kafka equilibrando DRY (não repetir código), KISS (simplicidade), DDD (*Bounded Contexts*) e Arquitetura Hexagonal.

Decisão: Implementar Shared Kernel na forma de biblioteca `src/libs/hermeneutics_kafk` contendo apenas mecânica de infraestrutura (conexão, serialização, retries, headers). Nenhuma lógica de negócio, DTOs específicos ou nomes de tópicos hardcoded são permitidos na biblioteca.

Consequências:

[+] Manutenção centralizada de infraestrutura Kafka.

[+] Facilita adoção de Schema Registry (Avro) no futuro.

[+] Menor *surface area* para bugs de configuração.

[-] Requer disciplina: proibido adicionar lógica de negócio na biblioteca.

A.2.5 ADR-010: Object Storage Local com MinIO

Status: Aceito **Tags:** MinIO, S3, storage

Decisão: Utilizar MinIO como *Object Storage* S3-compatível para armazenamento local de traces (Tempo), logs (Loki) e métricas de longo prazo (Thanos), garantindo portabilidade para AWS/GCP/Azure.

A.3 Decisões de Observabilidade

Esta seção documenta as decisões relacionadas ao pipeline de observabilidade e monitoramento.

A.3.1 ADR-011: Métricas de Longo Prazo com Thanos

Status: Aceito **Tags:** Thanos, Prometheus, observabilidade

Decisão: Implementar Thanos Sidecar junto ao Prometheus para enviar métricas históricas ao MinIO (Object Storage), permitindo manter meses de histórico sem consumir disco local excessivamente.

A.3.2 ADR-012: Rastreamento Distribuído com Tempo

Status: Aceito **Tags:** Tempo, tracing, observabilidade

Decisão: Adotar Grafana Tempo como backend de traces distribuídos, configurado com armazenamento S3 (MinIO), permitindo visualização de *Gantt Charts* do fluxo completo das Sagas.

A.3.3 ADR-013: Agregação de Logs com Loki

Status: Aceito **Tags:** Loki, logs, observabilidade

Decisão: Utilizar Grafana Loki para agregação centralizada de logs, configurado para receber logs do OpenTelemetry Collector e permitir correlação por TraceID.

A.3.4 ADR-014: Monitoramento de Golden Signals

Status: Aceito **Data:** 2026-01-01 **Tags:** Grafana, dashboards, Golden Signals

Decisão: Implementar dashboards padronizados no Grafana seguindo os 4 *Golden Signals* do Google SRE:

- **Latency**: Tempo de resposta (p95, p99)
- **Traffic**: Throughput (RPS)
- **Errors**: Taxa de erros (5xx/Total)
- **Saturation**: Uso de recursos (CPU, RAM)

A.3.5 ADR-015: Estratégia de Instrumentação OpenTelemetry

Status: Aceito **Tags:** OpenTelemetry, instrumentação, Python

Decisão: Utilizar auto-instrumentação do OpenTelemetry para bibliotecas padrão (HTTP, SQLAlchemy, Kafka) combinada com instrumentação manual para spans de negócio específicos.

A.3.6 ADR-016: Abordagem Híbrida OpenTelemetry

Status: Aceito **Tags:** OpenTelemetry, híbrido, migração

Decisão: Adotar abordagem híbrida que permite coexistência do legado (Django v1) com novos serviços (FastAPI v2) no mesmo pipeline de observabilidade, facilitando a migração gradual.

A.4 Decisões de Padronização

Esta seção documenta as decisões relacionadas a convenções de nomenclatura, configuração e boas práticas.

A.4.1 ADR-017: Upgrade do Django para 4.2 LTS

Status: Aceito **Tags:** Django, upgrade, segurança

Decisão: Atualizar o BFF Django (legado v1) para a versão 4.2 LTS para garantir suporte de segurança até abril de 2026 e compatibilidade com Python 3.12.

A.4.2 ADR-018: Uso do Padrão Facade no Maître (Mock E2E)

Status: Aceito **Tags:** Facade, Maître, mock, TCC

Decisão: Implementar o padrão Facade no serviço Maître para permitir simulação E2E da Saga com dados mockados, facilitando testes e demonstrações para a banca avaliadora.

A.4.3 ADR-019: Docker Compose Naming Convention

Status: Aceito **Tags:** Docker, naming, DevOps

Decisão: Padronizar nomes de serviços e contêineres no Docker Compose usando o prefixo `herm-` seguido do padrão `<serviço>-<tipo/tecnologia>` (ex: `herm-maitre-servi` `herm-db-postgres`).

A.4.4 ADR-020: Makefile Naming Convention

Status: Aceito **Tags:** Makefile, DRY, DevOps

Decisão: Introduzir prefixos padronizados no Makefile: `dc-*` para comandos Docker Compose e `rest-*` para comandos curl, eliminando targets redundantes e parametrizando por variável.

A.4.5 ADR-021: API Gateway com Subdomínios Localhost

Status: Aceito **Tags:** API Gateway, Nginx, subdomínios

Decisão: Configurar o API Gateway (Nginx) para rotear subdomínios localhost (ex: `maitre.localhost`, `grafana.localhost`), facilitando testes locais e pre-

parando para migração para domínio próprio em Kubernetes.

A.4.6 ADR-022: Shared Kernel para FastAPI (App Factory)

Status: Aceito **Tags:** Shared Kernel, FastAPI, factory, DRY

Decisão: Criar factory `build_app` em Shared Kernel que instancia FastAPI, aplica metadata e inclui router compartilhado de rotas básicas (`/health`, `/hello`), permitindo injeção de rotas de domínio via `extra_routes`.

A.4.7 ADR-023: Padronização SCM (.editorconfig e .gitattributes)

Status: Aceito **Tags:** SCM, Git, padronização

Decisão: Publicar `.gitattributes` (normalização EOL, marcação de binários) e `.editorconfig` (charset UTF-8, indentação por espaços, newline final) para garantir consistência entre desenvolvedores e sistemas operacionais.

A.5 Decisões de Persistência

Esta seção documenta as decisões relacionadas a estratégias de banco de dados e cache.

A.5.1 ADR-024: Estratégia de Banco de Dados (Single Container)

Status: Aceito **Tags:** PostgreSQL, Docker, infraestrutura

Decisão: Centralizar os bancos de dados em um único contêiner PostgreSQL com múltiplos databases lógicos (`maitre_db`, `research_db`, `annotation_db`, `hermeneutics`), economizando recursos sem violar o isolamento lógico do padrão *Database per Service*.

A.5.2 ADR-025: Shared Kernel para Conectividade PostgreSQL

Status: Aceito **Tags:** Shared Kernel, PostgreSQL, DRY

Decisão: Evoluir Shared Kernel com configuração base do SQLAlchemy (Async) e pool de conexões otimizado, reutilizável pelos serviços Persistence, Maître, Research e Annotation.

A.5.3 ADR-026: Shared Kernel para Cache Redis

Status: Aceito **Tags:** Shared Kernel, Redis, cache

Decisão: Implementar `RedisAdapter` nos serviços Research (cache de buscas), Maître (idempotência de Saga) e Persistence (cache de leitura para BFF), com configuração centralizada via Shared Kernel.

A.5.4 ADR-027: ConnectionMixin Pattern para Repositórios

Status: Aceito **Tags:** Mixin, Repository, testabilidade

Decisão: Adotar padrão ConnectionMixin para repositórios, permitindo injeção de conexão de banco de dados e facilitando testes unitários com mocks.

A.6 Decisões de Feature Toggles e FinOps

Esta seção documenta as decisões relacionadas a configuração dinâmica e análise de custos.

A.6.1 ADR-028: Feature Toggles para Modos de Operação

Status: Aceito **Tags:** toggles, E2E, FinOps

Decisão: Implementar *feature toggles* via variáveis de ambiente (`MOCK_MODE`, `E2E_MODE`, `FINOPS_MODE`) para alternar entre modos de operação sem necessidade de redeploy.

A.6.2 ADR-029: BFF Stateless com Redis Sessions

Status: Aceito **Tags:** Django, Redis, sessions, scaling

Decisão: Configurar o BFF Django para usar Redis como backend de sessões e cache, tornando-o *stateless* e permitindo escalabilidade horizontal em Kubernetes.

A.6.3 ADR-030: Kafka Batch Publishing para Mensagens Grandes

Status: Aceito **Tags:** Kafka, batch, reliability

Decisão: Implementar publicação em lote no Kafka para processamento de grandes volumes de artigos, respeitando limites de tamanho de mensagem e garantindo atomicidade.

A.6.4 ADR-031: ETL Strategy para Migração de Legados

Status: Aceito **Tags:** ETL, migração, legado

Decisão: Criar script Python (`migrate_v1_to_v2.py`) para migração ETL dos dados legados (SQLite e PostgreSQL v1) para o PostgreSQL v2, garantindo idempotência e deduplicação.

A.6.5 ADR-032: Delays Centralizados com Jitter (FinOps)

Status: Aceito **Tags:** FinOps, delays, OpenTelemetry, Shared Kernel

Decisão: Centralizar lógica de cálculo de delays com jitter no Shared Kernel para simular tempos de processamento realistas em modo mock, permitindo visualização precisa de métricas FinOps nos dashboards Grafana. Os delays são configuráveis via variáveis de ambiente e registrados no histograma `hermeneutics_article_processing_time`.

APÊNDICE B — DOCUMENTO DE REQUISITOS DO PRODUTO (PRD)

O Documento de Requisitos do Produto (PRD) é o artefato central que funciona como a **fonte única da verdade (SSOT) estratégica** para o projeto Hermeneutics 2.0. Ele comunica a visão, os objetivos de negócio e os requisitos do produto, definindo os limites de cada *Bounded Context* identificado via *Domain-Driven Design* (DDD).

Na versão 2.0, o PRD é um documento vivo e sua geração e refinamento são acelerados pelo uso de *Large Language Models* (LLMs), que geram rascunhos de seções e garantem que o documento contenha os detalhes essenciais, seguindo a filosofia *Docs-as-Code* (DaC).

B.1 Atributos do Produto

A Tabela B.1 apresenta os atributos gerais do produto Hermeneutics 2.0.

Tabela B.1: Atributos Gerais do Produto Hermeneutics 2.0

Atributo	**Detalhe**
Nome do Produto	Hermeneutics 2.0 - Evolução Arquitetural
Versão	2.0 (Microsserviços em Nuvem)
Público-alvo	Pesquisadores e acadêmicos que realizam Revisões Sistemáticas da Literatura (RSL)
Propósito Principal	Reestruturar a ferramenta para torná-la escalável, modular e resiliente, facilitando a integração de técnicas avançadas de rotulação semântica
Arquitetura Base	Microsserviços, Arquitetura Hexagonal (Ports & Adapters)

B.2 Catálogo de PRDs Estratégicos

O Hermeneutics 2.0 está estruturado em cinco PRDs principais, cada um correspondendo a um domínio de negócio ou camada tecnológica da nova arquitetura. A Tabela B.2 apresenta o catálogo completo.

Tabela B.2: Catálogo de PRDs Estratégicos do Hermeneutics 2.0

ID	Título do PRD	Propósito e Escopo	NFRs Críticos
PRD-001	Modernização da Arquitetura Core	Reestruturar o monólito legado em microsserviços para permitir a escalabilidade independente das funcionalidades de busca e anotação	Escalabilidade, Manutenibilidade
PRD-002	Motor de Ingestão de Artigos	Criar um serviço resiliente para busca em múltiplas bibliotecas digitais (IEEE, Scopus), isolando a aplicação de falhas externas	Resiliência, Tolerância a Falhas
PRD-003	Pipeline de Anotação Semântica	Desenvolver um módulo de processamento assíncrono capaz de aplicar modelos de PLN (SpERT) em grandes volumes de texto sem bloquear a Interface do Usuário (UI)	Performance, Throughput
PRD-004	Persistência e Gestão de Dados	Implementar uma estratégia de dados distribuída que garanta consistência eventual entre os serviços e a integridade das revisões	Consistência Eventual, Disponibilidade
PRD-005	Painel de Triagem Inteligente	Prover uma interface web reativa que permita ao pesquisador filtrar e selecionar artigos com feedback visual instantâneo das anotações	Usabilidade, Tempo de Resposta

B.3 Requisitos Funcionais e Objetivos

Os objetivos estratégicos da versão 2.0 traduzem a decomposição do monólito v1.0. A Tabela B.3 apresenta os requisitos funcionais identificados.

Tabela B.3: Requisitos Funcionais e Objetivos do Hermeneutics 2.0

Categoria	**Requisito / Objetivo Específico**
Decomposição	Identificar e decompor os principais domínios funcionais (*Bounded Contexts*) do Hermeneutics original, utilizando princípios de DDD e modelagem contextual
Implementação Core	Implementar a arquitetura de microsserviços baseada na separação dos domínios, utilizando contêineres Docker e comunicação por meio de REST APIs e filas assíncronas
Refatoração	Refatorar os componentes existentes da ferramenta, adaptando-os ao novo modelo arquitetural e garantindo isolamento entre os serviços
RF: Interface	Incorporar melhorias de usabilidade, como exibição da quantidade de artigos, visualização de citações e destaque de itens selecionados
RF: Filtragem	Permitir ao pesquisador filtrar artigos que contenham entidades específicas, como “Deep Learning”
RF: Progresso	O sistema deve exibir o progresso da revisão em tempo real

B.4 Requisitos Não Funcionais e Arquitetura

Os Requisitos Não Funcionais (NFRs) do Hermeneutics 2.0 são definidos pela necessidade de resiliência e escalabilidade em ambiente de nuvem. A Tabela B.4 apresenta os requisitos identificados.

Tabela B.4: Requisitos Não Funcionais e Arquitetura do Hermeneutics 2.0

Categoria	**Requisito Arquitetural ou Técnico**
Persistência	Reestruturar a camada de persistência utilizando um modelo híbrido, com múltiplos bancos (relacional e NoSQL) de acordo com a responsabilidade de cada serviço. Adotar o padrão *Database per Service*
Mensageria	Transição de Celery/RabbitMQ para Apache Kafka para gerenciar o barramento de eventos, garantindo resiliência e elasticidade
Transações	Adotar o Padrão Saga Coreografada para gerenciar transações distribuídas e garantir a consistência eventual dos dados entre serviços independentes
Infraestrutura	Implantar a solução em ambiente de nuvem (GCP ou AWS), com orquestração via Kubernetes
Observabilidade	Implementar OpenTelemetry para instrumentação, coletando os três pilares (traces, logs, métricas) para monitoramento e análise de custos
FinOps	O sistema deve suportar a geração de indicadores FinOps, como o "custo por artigo processado", conectando o consumo de recursos técnicos ao valor de negócio
Manutenibilidade	A arquitetura deve facilitar a futura incorporação de modelos preditivos mais avançados (como LLMs via API)

B.5 Gestão de Documentação (Metodologia)

A gestão do PRD e dos demais artefatos é regida por práticas modernas de engenharia:

DaC/CD: A documentação segue a filosofia *Docs-as-Code* (DaC) e *Continuous Documentation* (CD), tratando a documentação com o mesmo rigor do código, utilizando Markdown e Git para versionamento.

ADRs: As decisões arquiteturais críticas, como a adoção de microsserviços e Kafka, são

documentadas em *Architecture Decision Records* (ADRs), seguindo o modelo Nygard/MADR, para registrar o contexto e as consequências das escolhas. Os ADRs são a documentação das escolhas feitas para cumprir os NFRs definidos no PRD.

O PRD é, metaforicamente, o projeto de engenharia de um navio: ele define que o navio precisa ser rápido (Performance) e à prova d'água (Resiliência), e especifica os compartimentos (Microsserviços) e os materiais (Tecnologias) para que a equipe de construção (Desenvolvimento) possa trabalhar, garantindo que o resultado final atenda ao propósito estratégico de navegar (Revisão Sistemática).

APÊNDICE C — ÉPICOS DE ENGENHARIA E USER STORIES

Este apêndice documenta os Épicos de Engenharia (*Technical Enablers*) e as *User Stories* funcionais que guiaram o desenvolvimento do Hermeneutics v2.0. As *User Stories* são descrições de funcionalidades contadas da perspectiva do usuário, seguindo o formato: "Como um [quem], eu quero [o quê], para que eu possa [por quê/valor]".

Para garantir que as *User Stories* sejam de alta qualidade e testáveis (critério INVEST), elas foram formalizadas com a sintaxe Gherkin (*Given-When-Then-And*), baseada no *Behavior-Driven Development* (BDD).

C.1 Visão Geral dos Épicos

A Tabela C.1 apresenta o resumo dos 6 épicos de engenharia implementados no projeto.

Tabela C.1: Resumo dos Épicos de Engenharia do Hermeneutics v2.0

Épico	**Título**	**Status**	**Escopo Principal**
EPIC-001	Reorganização e Engenharia de IA	Concluído	Monorepo, LaTeX, Agentes IA
EPIC-002	Infraestrutura e Refatoração Core	Concluído	Docker, Hexagonal, BFF, Kafka
EPIC-003	Integração Kafka e Padrão Saga	Concluído	Saga Coreografada, TDD
EPIC-004	Observabilidade Full-Stack e FinOps	Concluído	OpenTelemetry, Dashboards
EPIC-005	Padronização da Arquitetura	Concluído	Compose, Makefile, Gateway
EPIC-006	Persistência e Migração de Dados	Concluído	PostgreSQL, Redis, ETL

C.2 EPIC-001: Reorganização e Estruturação de Engenharia de IA

Status: Concluído **Versão Alvo:** Hermeneutics v2.0

Contexto: Preparação do terreno para a migração de Monólito para Microsserviços e adoção de práticas de Engenharia de IA (*Docs-as-Code*, Agentes).

Descrição: Este Épico engloba todas as atividades fundacionais necessárias para transformar o repositório legado (v1.0) em um monorepo moderno (v2.0), estabelecendo

a estrutura para desenvolvimento de microsserviços e a integração de agentes de IA para automação de tarefas.

C.2.1 User Stories Associadas

US-001.1: Reorganização do Repositório (Strangler Fig)

Como Arquiteto de Software, **quero** isolar o código legado da v1.0 em um diretório `src/legacy` e estabelecer uma nova estrutura de pastas para a v2.0, **para que** eu possa desenvolver os novos microsserviços sem ser bloqueado pela dívida técnica do monólito existente.

Critérios de Aceite: Branch v2.0 criada; código v1 movido para `src/legacy/v1`; estrutura `src/services`, `docs/`, `deploy/` criada.

US-001.2: Infraestrutura de Engenharia de IA

Como Líder Técnico, **quero** criar uma biblioteca de prompts e definições de agentes na pasta `docs/ia_engineering`, **para que** eu possa utilizar LLMs para gerar documentação, código e corrigir bugs de forma padronizada e acelerada.

Critérios de Aceite: Pasta `docs/ia_engineering/prompts` com templates para PRD, ADR e User Stories; pasta `docs/ia_engineering/agents` definindo personas; módulos LaTeX integrados.

US-001.3: Extração de Conhecimento (Reverse Engineering)

Como Desenvolvedor, **quero** extrair as regras de negócio implícitas na monografia e no código legado e documentá-las como ADRs e PRDs, **para que** o conhecimento tácito se torne explícito e guie a implementação da v2.0.

Critérios de Aceite: Contexto do projeto extraído para `docs/ia_engineering/context/`; plano de ação de 45 dias documentado.

C.3 EPIC-002: Infraestrutura de Microsserviços e Refatoração Core

Status: Concluído **Versão Alvo:** Hermeneutics v2.0

Contexto: Implementação física da arquitetura distribuída, containerização e refatoração dos domínios principais.

Descrição: Este Épico foca na "subida" da infraestrutura v2.0 (Docker Compose, Redes) e na implementação dos microsserviços principais (Maître, Research, Annotation, Persistence) seguindo rigorosamente a Arquitetura Hexagonal, desacoplando-os do monólito Django.

C.3.1 User Stories Associadas

US-002.1: Containerização e Orquestração Local

Como DevOps Engineer, **quero** configurar o `docker-compose.yml` e templates de Dockerfile padronizados com multi-stage build, **para que** todos os desenvolvedores possam subir o ambiente completo (v1 e v2) com um único comando `make up`.

US-002.2: Scaffolding Arquitetura Hexagonal

Como Desenvolvedor Backend, **quero** criar a estrutura de pastas dos microsserviços (`domain`, `application`, `infrastructure`) e implementar as interfaces (Ports), **para que** o código de negócio fique isolado de frameworks e banco de dados.

US-002.3: Refatoração Django para BFF

Como Desenvolvedor Fullstack, **quero** adaptar o Django (v1) para atuar temporariamente como frontend/BFF, consumindo as novas APIs v2, **para que** possamos manter a UI antiga funcionando enquanto o backend é modernizado (Estratégia Strangler).

US-002.4: Persistência Poliglota (Database per Service)

Como DBA/Arquiteto de Dados, **quero** configurar esquemas de banco de dados isolados para cada serviço, **para que** garantir que nenhum serviço acesse diretamente as tabelas de outro (evitando acoplamento).

US-002.5: Setup Inicial Kafka Saga

Como Arquiteto de Software, **quero** configurar os tópicos iniciais do Kafka e os produtores/consumidores básicos nos serviços, **para que** a comunicação entre Maître, Research e Annotation seja assíncrona e resiliente.

C.4 EPIC-003: Integração Kafka e Padrão Saga

Status: Concluído **Versão Alvo:** Hermeneutics v2.0

Contexto: Esta Epic visa estabelecer a infraestrutura de comunicação assíncrona robusta utilizando Apache Kafka, habilitando o padrão Saga para orquestração de microsserviços.

Descrição: O escopo abrange desde o setup da infraestrutura local (com persistência e observabilidade) até a implementação de adaptadores genéricos, testes e instrumentação para monitoramento.

C.4.1 User Stories Associadas

US-003.1: Configuração e Estabilização da Infraestrutura Kafka

Como Engenheiro de DevOps/Backend, **quero** ter um ambiente Kafka local estável e persistente, **para que** eu possa reiniciar containers durante o desenvolvimento sem perder tópicos e mensagens.

Critérios de Aceite: Tópicos essenciais (`saga.commands`, `research.events`, etc.) criados automaticamente; JMX Exporter funcional.

US-003.2: Implementação de Ports e Adapters Genéricos para Eventos

Como Desenvolvedor Backend, **quero** componentes padronizados de produção e consumo de mensagens, **para que** eu possa integrar qualquer microsserviço ao barramento de eventos sem duplicar código de infraestrutura.

Critérios de Aceite: Interface `EventProducerPort` definida; implementação `KafkaProducerAdapter` funcional; biblioteca compartilhada em `src/libs/hermeneutics`

US-003.3: Testes Unitários e Mocks para Produtores e Consumidores

Como Desenvolvedor, **quero** ferramentas de teste e mocks dos componentes de evento, **para que** eu possa validar a lógica da Saga sem depender da infraestrutura Kafka real durante os testes unitários.

US-003.4: Instrumentação e Observabilidade (Telemetry)

Como SRE/Desenvolvedor, **quero** visualizar o fluxo de mensagens e a saúde da infraestrutura em dashboards, **para que** eu possa identificar gargalos, falhas de processamento e monitorar a performance da Saga.

C.5 EPIC-004: Observabilidade Full-Stack e FinOps

Status: Concluído **Prioridade:** Alta (Requisito para Capítulo de Avaliação)

Contexto: Atualmente, o projeto Hermeneutics v2 possui métricas básicas (Prometheus) e visualização (Grafana). Para suportar a análise de FinOps (custo por transação) e depuração profunda de sistemas distribuídos, é necessário implementar um pipeline completo de observabilidade que correlacione Logs, Métricas e Traces.

C.5.1 User Stories Associadas

US-004.1: Infraestrutura de Armazenamento de Objetos (MinIO)

Como Engenheiro de Plataforma, **quero** ter um serviço de Object Storage (S3 compatível) rodando localmente, **para que** eu possa armazenar grandes volumes de traces e logs sem depender do disco local dos containers.

US-004.2: Implementação de Distributed Tracing (Tempo)

Como Desenvolvedor, **quero** visualizar o rastro completo de uma requisição (Saga) passando por Maître $\rightarrow$ Kafka $\rightarrow$ Research $\rightarrow$ Annotation, **para que** eu possa identificar gargalos de latência.

US-004.3: Agregação de Logs Centralizada (Loki)

Como Operador do Sistema, **quero** ver os logs de todos os microsserviços em um único lugar, correlacionados pelo TraceID, **para que** eu não precise entrar em cada container para debugar erros.

US-004.4: Instrumentação de Serviços e Dashboards (Golden Signals)

Como SRE/Desenvolvedor de Observabilidade, **quero** dashboards padronizados de Golden Signals, Health Status e Overview para todos os serviços, **para que** eu monitore rapidamente disponibilidade, performance e correlação de eventos.

US-004.5: Instrumentação OTel do web_ui (BFF Django)

Como Engenheiro de Plataforma, **quero** instrumentar o serviço `web_ui` (BFF Django extraído do v1 via Strangler Fig), **para que** eu tenha observabilidade ponta-a-ponta dos fluxos iniciados na camada de apresentação.

US-004.6: Mock E2E da Saga e Dashboard FinOps

Como Engenheiro de Software, **quero** simular o fluxo completo da saga com dados

mockados e visualizar o custo estimado por artigo/processo no Grafana, **para que** valide a coreografia Kafka e avalie a viabilidade financeira da arquitetura v2.

US-004.7: DaC e Atualização de Documentação

Como Documentador/Engenheiro de Plataforma, **quero** consolidar e linkar todos os artefatos gerados pelo EPIC-004, **para que** a documentação do projeto permaneça coerente e rastreável.

C.6 EPIC-005: Padronização da Arquitetura

Status: Concluído **Prioridade:** Alta (infra baseline para testes E2E e DaC)

Contexto: A base docker-compose v2 cresceu de forma orgânica. Existem imagens com tag `latest`, variáveis duplicadas inline e nomes heterogêneos de serviços/containers. Para acelerar os testes do TCC, é necessário padronizar compose, Makefile, gateway e configurações.

C.6.1 User Stories Associadas

US-005.1: Padronização de Serviços no docker-compose

Reordenar atributos, aplicar prefixo `herm-*`, centralizar variáveis via `env_file`, eliminar tags `latest`.

US-005.2: Padronização do Makefile (dc-*, rest-*)

Introduzir prefixos `dc-` para comandos Docker e `rest-*` para comandos curl, parametrizar por variável.

US-005.3: API Gateway com Subdomínios Localhost

Configurar nginx/gateway para mapear `*.localhost` a portas locais.

US-005.4: Centralização de Variáveis (.env único)

Criar `.env.example` com valores default e referenciar via `env_file` no compose.

US-005.5: Fixar Versões LTS das Imagens Docker

Eliminar tags `latest` e fixar versões estáveis próximas a dezembro/2025.

US-005.6: Composição de Rotas Comuns FastAPI

Extrair rotas comuns (`/health`, `/hello`) para `APIRouter` compartilhado em Shared Kernel.

US-005.7: App Builder Factory (Shared Kernel)

Criar factory `build_app` que instancia FastAPI e aplica configurações padrão.

US-005.8: Padronização SCM (.gitattributes + .editorconfig)

Publicar arquivos de configuração para normalização de EOL e estilo de edição.

US-005.9: ADRs Implícitas → DaC

Documentar decisões implícitas como ADRs formais.

C.7 EPIC-006: Persistência, Conectividade e Migração de Dados

Status: Concluído **Prioridade:** Alta (bloqueante para testes integrados reais)

Contexto: Atualmente, os microsserviços da v2 operam com mocks em memória, o que impede testes de carga reais e validação do fluxo completo (End-to-End). Além disso, existe um legado rico em dados (v1) que precisa ser migrado para validar a nova arquitetura.

Descrição: Este Epic visa implementar os "Real Adapters" de persistência (PostgreSQL) e cache (Redis), instrumentar a observabilidade de banco de dados para FinOps e migrar os dados do legado.

C.7.1 User Stories Associadas

US-006.0: Estratégia de Banco de Dados Único (Single Container)

Como Engenheiro de Plataforma, **quero** centralizar os bancos de dados em um único container Docker, **para que** eu economize recursos da máquina local sem violar o isolamento lógico dos microsserviços.

US-006.1: Infraestrutura de Conexão (Shared Kernel & Adapters)

Como Desenvolvedor Backend, **quero** adapters de banco e cache robustos e reutilizáveis, **para que** eu possa persistir dados com performance e segurança sem duplicar código de infraestrutura.

US-006.2: Migração de Dados Legados (ETL: v1 → v2)

Como Pesquisador/PO, **quero** meus dados antigos disponíveis na nova versão, **para que** eu não perca histórico de revisões passadas.

Critérios de Aceite: Script executado com sucesso; pelo menos 3000 artigos migrados sem duplicatas; Sagas antigas com status correspondente.

US-006.3: Refatoração do BFF (Web UI) para Redis

Como Arquiteto de Software, **quero** que o BFF seja stateless e use Redis para sessões, **para que** o frontend seja escalável horizontalmente e não dependa de persistência relacional própria.

US-006.4: Mock E2E com Persistência Real

Como Stakeholder, **quero** ver uma pesquisa acontecer do início ao fim com dados persistidos, **para que** eu confie na robustez do sistema distribuído.

US-006.5: Integração O11Y em Bancos de Dados

Como Engenheiro de FinOps/SRE, **quero** monitorar a performance e o custo das operações de banco, **para que** eu possa otimizar queries caras e prever custos de cloud.

US-006.6: Documentação como Código (DaC) e ADRs do EPIC-006

Como Novo Desenvolvedor, **quero** entender as decisões de persistência e como operar os bancos locais, **para que** eu não viole a arquitetura ao criar novas funcionalidades.

C.8 User Stories Técnicas (Quality Gates)

Histórias focadas na experiência do desenvolvedor (DX) e qualidade de código, servindo como ponte entre a infraestrutura (EPIC-002) e a implementação complexa (EPIC-003).

C.8.1 US-DEV-001: Configuração de Pre-Commit Hooks

Como desenvolvedor da equipe Hermeneutics, **quero** configurar validações automáticas (quality gates) no meu ambiente de desenvolvimento local, **para que** a qualidade

do código, a padronização de formato e as convenções de commit sejam garantidas antes de o código ser enviado ao repositório remoto, reduzindo a incidência de erros em CI/CD.

Regras de Negócio:

1. Toda mensagem de commit deve iniciar com um dos prefixos: `ADD`, `CREATE`, `UPDATE`, `FIX`, `TEST`, `DELETE`, `REMOVE`, `MERGE`.
2. Todos os arquivos de texto devem utilizar o padrão de final de linha LF (Line Feed).
3. O código não deve ser comitado se apresentar erros de sintaxe Python ou YAML.
4. O commit de arquivos grandes (>500KB) deve ser bloqueado.

Ferramentas: `pre-commit`, `check-ast`, `check-yaml`, `trailing-whitespace`, `end-of-file-fixer`, `prettier`.

C.8.2 US-DEV-002: Implementação de CI Pipeline (TDD)

Como Engenheiro de DevOps/Software, **quero** ter um pipeline de CI automatizado que execute verificações de qualidade e testes unitários a cada Pull Request, **para que** eu possa garantir a integridade da arquitetura, a conformidade com o estilo de código (linting) e a não-regressão das regras de negócio antes do merge para a branch master.

Regras de Negócio:

1. O job de testes unitários só deve iniciar se o job de `quality-check` (pre-commit) passar com sucesso.
2. A falha nos testes de um microsserviço não deve impedir a execução dos testes de outro, mas deve falhar o workflow geral.
3. Todos os microsserviços dependem da biblioteca `hermeneutics_kafka`, que deve ser instalada previamente no ambiente de CI.

Ferramentas: GitHub Actions, `pytest`, `pytest-cov`.