ingenieur
wissenschaften
htw saar

Hochschule für
Technik und Wirtschaft
des Saarlandes
University of
Applied Sciences

Bachelor-Thesis

zur Erlangung des akademischen Grades
Bachelor of Science (B. Sc.)
an der Hochschule für Technik und Wirtschaft des Saarlandes
im Studiengang Praktische Informatik
der Fakultät für Ingenieurwissenschaften

Retrieval-Augmented Generation for the Discovery of Payment Validation Rules via LLM-Enriched Source Data

vorgelegt von
José Aram Méndez Gómez

betreut und begutachtet von
Prof. Dr. Markus Esch
Alexander Efremov

Saarbrücken, 01.09.2025

Selbständigkeitserklärung

Ich versichere, dass ich die vorliegende Arbeit (bei einer Gruppenarbeit: den entsprechend gekennzeichneten Anteil der Arbeit) selbständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt habe.

Ich erkläre hiermit weiterhin, dass die vorgelegte Arbeit zuvor weder von mir noch von einer anderen Person an dieser oder einer anderen Hochschule eingereicht wurde.

Darüber hinaus ist mir bekannt, dass die Unrichtigkeit dieser Erklärung eine Benotung der Arbeit mit der Note "‘nicht ausreichend"’ zur Folge hat und einen Ausschluss von der Erbringung weiterer Prüfungsleistungen zur Folge haben kann.

Saarbrücken, 01.09.2025

José Aram Méndez Gómez

Abstract

Financial institutions maintain thousands of payment validation rules with inconsistent documentation across fragmented systems, hindering efficient rule discovery. This thesis presents a retrieval-augmented generation approach where large language models enrich source data offline, enabling semantic search without runtime model dependencies. During corpus preparation, LLMs generate standardized descriptions, extract keywords, and infer categorical metadata from unstructured rule documentation. This enriched data supports a hybrid retrieval system combining sparse retrieval, dense embeddings, and fuzzy matching through weighted aggregation. The monolithic architecture ensures deterministic, auditable retrieval—essential for regulatory compliance—while eliminating external service dependencies. Evaluation through cross-validation demonstrates the hybrid approach outperforms individual retrieval methods, achieving effective rule discovery with sub-second response times. This work shows that RAG benefits can be captured through offline preprocessing rather than online inference, providing a practical framework for deploying advanced NLP capabilities in constrained enterprise environments. The approach bridges modern language understanding with regulatory requirements, offering a template for similar applications in regulated industries.

Keywords: retrieval-augmented generation, offline LLM enrichment, hybrid retrieval, payment validation rules, regulatory compliance, enterprise constraints

Gratitude is the recognition that nothing stands alone—
that every thought is a collaboration, every insight a gift,
every achievement a trace of voices heard and unheard.

— José Aram Méndez Gómez

Acknowledgments

I would like to express my deepest gratitude to my company supervisor, **Alexander Efremov** of **Deutsche Bank AG**. His guidance was not only instrumental but foundational to the direction and realization of this thesis. His technical insight, critical feedback, and continuous support shaped both the research approach and its practical execution in decisive ways.

I also wish to thank my academic supervisor, **Prof. Dr. Markus Esch** of **Hochschule für Technik und Wirtschaft des Saarlandes** for his constructive feedback and encouragement, which ensured the academic rigor of this work.

This project was truly a collaborative effort. I extend my sincere appreciation to the engineers from various teams at Deutsche Bank AG who formed our cross-functional working group. Each brought unique expertise—from payment processing and validation logic to infrastructure and user experience—that proved essential to the system's development. Your willingness to share knowledge across team boundaries, provide critical feedback during design reviews, and collaborate on integration exemplified engineering excellence. This thesis would not have been possible without your collective technical expertise and collegial support.

My thanks extend as well to Deutsche Bank AG for enabling this thesis within an innovative and challenging professional environment. The opportunity to work at the intersection of academic research and real-world application has been invaluable.

Contents

Abstract iii

Acknowledgments iv

1 Introduction 1
- 1.1 Background and Motivation . . . 1
- 1.2 Problem Statement . . . 1
- 1.3 Approach Overview . . . 2
- 1.4 Corpus and Fields . . . 2
- 1.5 System Overview . . . 3
- 1.6 Retrieval Modes and Scoring . . . 4
- 1.7 Operational Constraints and Non-Goals . . . 5
- 1.8 Research Questions . . . 5
- 1.9 Contributions . . . 5
- 1.10 Thesis Structure . . . 6

2 Related Work 8
- 2.1 Information Retrieval Foundations . . . 8
  - 2.1.1 Lexical Retrieval . . . 8
  - 2.1.2 Dense Retrieval . . . 8
- 2.2 Hybrid Retrieval Systems . . . 8
  - 2.2.1 Combination Strategies . . . 8
  - 2.2.2 Recent Neural Approaches . . . 9
- 2.3 Financial Domain Applications . . . 9
  - 2.3.1 Regulatory Compliance Systems . . . 9
  - 2.3.2 Code and Rule Retrieval . . . 9
  - 2.3.3 Existing Banking Solutions . . . 9
- 2.4 Technical Components . . . 10
  - 2.4.1 String Matching . . . 10
  - 2.4.2 Evaluation Methods . . . 10
  - 2.4.3 Architecture Choices . . . 10
- 2.5 Critical Gaps and Our Contributions . . . 10
  - 2.5.1 Production Deployment Gaps . . . 10
  - 2.5.2 Domain-Specific Gaps . . . 10
  - 2.5.3 Methodological Contributions . . . 11
- 2.6 Summary . . . 11

2.7 Comparative Analysis 11

3 Fundamentals **13**
3.1 Lexical Retrieval with BM25 13
3.1.1 Tokenization Theory 13
3.1.2 The BM25 Ranking Function 13
3.2 Fuzzy String Matching 14
3.2.1 Edit Distance Foundation 14
3.2.2 Partial Matching Strategy 14
3.3 Dense Vector Representations 15
3.3.1 Distributional Semantics 15
3.3.2 Transformer-Based Encoding 15
3.3.3 Sentence-Level Embeddings 15
3.3.4 Cosine Similarity 15
3.4 Retrieval-Augmented Generation 16
3.4.1 RAG Architecture 16
3.4.2 Offline-Online Separation 16
3.5 Hybrid Signal Combination 16
3.5.1 Score Normalization Theory 16
3.5.2 Convex Combination 17
3.5.3 Relevance Thresholding 17
3.6 Monolithic Architecture Principles 17
3.6.1 Process Model 17
3.6.2 Memory Sharing Benefits 17
3.6.3 Operational Simplicity 18
3.6.4 Trade-offs 18
3.7 Performance Characteristics 18
3.7.1 Computational Complexity 18
3.7.2 Empirical Performance 18
3.8 Summary 19

4 Corpus Analysis **20**
4.1 Data Consolidation and Enhancement 20
4.1.1 Multi-Source Consolidation 20
4.1.2 LLM-Based Field Generation 20
4.2 Corpus Overview 21
4.2.1 Field Structure 21
4.2.2 Corpus Composition 21
4.3 Field Completeness Analysis 22
4.3.1 Critical Field Coverage 22
4.4 Categorical Tag Analysis 23
4.4.1 Tag Coverage and Diversity 23

4.4.2 Tag Quality Issues . . . 23
4.5 Text Field Characteristics . . . 24
4.6 Keyword Analysis . . . 24
4.6.1 Keyword Distribution . . . 24
4.6.2 Top Keywords by Frequency . . . 25
4.7 Embedding Quality Assessment . . . 25
4.7.1 Embedding Characteristics . . . 25
4.7.2 Similarity Distribution Analysis . . . 25
4.8 Evaluation Dataset Construction . . . 26
4.9 Data Quality Assessment . . . 26
4.9.1 Overall Quality Metrics . . . 26
4.9.2 Prioritized Improvement Roadmap . . . 27
4.10 Summary . . . 28

5 System Design **29**
5.1 Architectural Overview . . . 29
5.1.1 Design Philosophy . . . 29
5.1.2 Core Components . . . 29
5.2 Data Flow Architecture . . . 30
5.2.1 Initialization Sequence . . . 30
5.2.2 Query Processing Pipeline . . . 30
5.3 Component Design . . . 31
5.3.1 Retrieval Engine . . . 31
5.3.2 Data Management . . . 32
5.3.3 User Interface . . . 33
5.4 Performance Optimizations . . . 34
5.4.1 Startup Optimizations . . . 34
5.4.2 Query-Time Optimizations . . . 34
5.5 Security and Compliance . . . 35
5.5.1 Security Measures . . . 35
5.5.2 Audit Compliance . . . 35
5.6 Design Trade-offs . . . 36
5.6.1 Choices Made . . . 36
5.6.2 Rejected Alternatives . . . 36
5.7 Scalability Considerations . . . 36
5.7.1 Current Limits . . . 36
5.7.2 Scaling Strategies . . . 37
5.8 Summary . . . 38

6 Implementation **39**
6.1 System Architecture . . . 39
6.1.1 Project Structure . . . 39

6.1.2 Architectural Overview . . . 39
6.2 Core RAG Components . . . 40
6.2.1 Embedding Manager . . . 40
6.2.2 Embedding Index . . . 41
6.2.3 Index Construction . . . 41
6.2.4 Three Retrieval Signals . . . 42
6.2.5 Hybrid Score Fusion . . . 43
6.3 Search Pipeline . . . 44
6.4 User Interface . . . 46
6.4.1 Search Interface . . . 46
6.4.2 Faceted Filtering . . . 46
6.4.3 Search Results . . . 47
6.4.4 Rule Details . . . 47
6.4.5 Unified Interface . . . 48
6.5 Summary . . . 48

7 Evaluation **49**
7.1 Evaluation Setup . . . 49
7.1.1 Dataset . . . 49
7.1.2 Candidate Pool Construction . . . 49
7.2 Metrics . . . 50
7.3 Weight Optimization via LOOCV . . . 50
7.4 Results . . . 51
7.4.1 Individual Signal Performance . . . 51
7.4.2 LOOCV-Optimized Performance . . . 51
7.4.3 Performance Comparison . . . 52
7.4.4 Ablation Study . . . 52
7.4.5 Weight Sensitivity . . . 52
7.5 System Performance . . . 53
7.6 Discussion . . . 53
7.6.1 Key Findings . . . 53
7.6.2 Limitations . . . 54
7.6.3 Comparison to Related Systems . . . 54
7.7 Conclusions . . . 54

8 Conclusion **55**
8.1 Summary of Contributions . . . 55
8.1.1 Technical Implementation . . . 55
8.1.2 Data Standardization Achievement . . . 55
8.1.3 Methodological Rigor . . . 56
8.2 Design Philosophy and Lessons Learned . . . 56
8.2.1 The Power of Architectural Simplicity . . . 56

8.2.2 Pragmatic Technology Choices . . . 56
8.3 Limitations and Boundaries . . . 57
8.3.1 System Constraints . . . 57
8.3.2 Known Failure Patterns . . . 57
8.4 Future Directions . . . 57
8.4.1 Immediate Enhancements (3-6 months) . . . 57
8.4.2 System Evolution (6-12 months) . . . 58
8.4.3 Research Opportunities . . . 58
8.5 Concluding Remarks . . . 58

Bibliography **59**

List of Figures **63**

List of Tables **64**

Listings **65**

List of Abbreviations **66**

A System Configuration and Implementation Details **69**
A.1 Configuration Parameters . . . 69
A.1.1 Core System Configuration . . . 69
A.1.2 Database Optimization . . . 69
A.2 Data Schema . . . 70
A.2.1 Rule Representation . . . 70
A.3 API Reference . . . 70
A.3.1 Primary Search Interface . . . 70
A.3.2 Search Modes . . . 71
A.4 Deployment Guide . . . 71
A.4.1 Development Environment . . . 71
A.4.2 Production Deployment . . . 71
A.5 Performance Characteristics . . . 72
A.5.1 Operational Metrics . . . 72
A.5.2 Latency Breakdown . . . 72
A.6 Corpus Preparation . . . 72
A.6.1 Data Pipeline . . . 72
A.6.2 Embedding Generation . . . 73
A.7 Troubleshooting . . . 73
A.8 Summary . . . 73

1 Introduction

1.1 Background and Motivation

Modern payment platforms process millions of transactions daily, each subject to complex validation rules that protect customers, reduce operational risk, and ensure regulatory compliance. At Deutsche Bank's payment processing platforms, these validation rules span from simple field validations (mandatory characters, allowed value sets) to intricate domain-specific constraints (country-specific limits, routing conditions, SWIFT/ISO message-format requirements). With hundreds of rules evolving continuously across distributed teams and systems, developers and quality assurance teams face a critical challenge: quickly finding and understanding the exact rules relevant to their current task.

Traditional keyword search fails this use case. Rules are described inconsistently across teams, terminology overlaps, and natural language queries rarely match the exact vocabulary in rule documentation. A developer asking about "EUR cross-border limits" might miss rules titled "SEPA maximum amount validation" despite their relevance. Moreover, rule data is often scattered across multiple systems, with inconsistent formats and outdated versions. Simultaneously, banking environments impose strict architectural constraints: no external unapproved API calls, limited infrastructure dependencies, and complete audit trails for all data access.

1.2 Problem Statement

Goal. Given a user query about a validation rule (e.g., "EUR cross-border limit" or "party agent checks for DE"), the system should:

1. return a short, ranked list of relevant rules drawn from a *single, standardized CSV corpus*;
2. display all available rule information (codes, tags, language variants, implementation code) and a short, factual explanation grounded in the stored fields;
3. maintain sub-second response time for typical searches while operating within enterprise infrastructure constraints.

1.3 Approach Overview

This thesis presents a retrieval-augmented generation system that addresses these challenges through a hybrid retrieval architecture. The system combines three complementary signals—lexical matching, fuzzy string similarity, and semantic embeddings—with configurable weights optimized through empirical evaluation. The architecture follows a monolithic design pattern where all components operate within a single process, simplifying deployment and audit compliance.

The key innovation lies in comprehensive data standardization coupled with offline enrichment. Scattered rule data from multiple sources was consolidated into a single CSV corpus with uniform field definitions. Large language models were employed during corpus preparation to generate enhanced descriptions used primarily for embedding generation, ensuring semantic search capabilities without runtime model dependencies. This separation of offline enrichment from online retrieval ensures deterministic, auditable search results.

1.4 Corpus and Fields

Each row in the standardized CSV represents one validation rule, consolidated from distributed sources and cleaned to ensure consistency:

- **Identifiers**: *Rule ID, Rule Name* (deduplicated across sources).
- **Business descriptions**: *Rule Description, Short English Description, Short German Description* (standardized terminology).
- **Implementation**: *Rule Code* (the actual validation rule implementation).
- **Error codes**: *BANSTA Error Code, ISO Error Code* (per banking standards).
- **LLM-generated fields**: *LLM Description* (comprehensive text synthesizing rule purpose and logic), *Keywords* (extracted distinctive search terms), *Tags* (Rule Type, Country, Business Type, Party Agent scraped from available fields). All generated offline by Gemini-2.5-Pro.
- **Embeddings**: 1024-dimensional vectors from `UAE-Large-V1` [23], computed from LLM descriptions, L2-normalized, stored as JSON strings in CSV.

Additional metadata fields (*Relevance* score $\in [0, 1]$, *Version*, *Created/Updated* timestamps) are added at database ingestion but not stored in the CSV itself. This standardization process resolved inconsistencies in naming conventions, merged duplicate rules, and established canonical representations for all fields.

1.5 System Overview

The monolithic Dash application implements a complete RAG pipeline with minimal dependencies and efficient startup initialization:

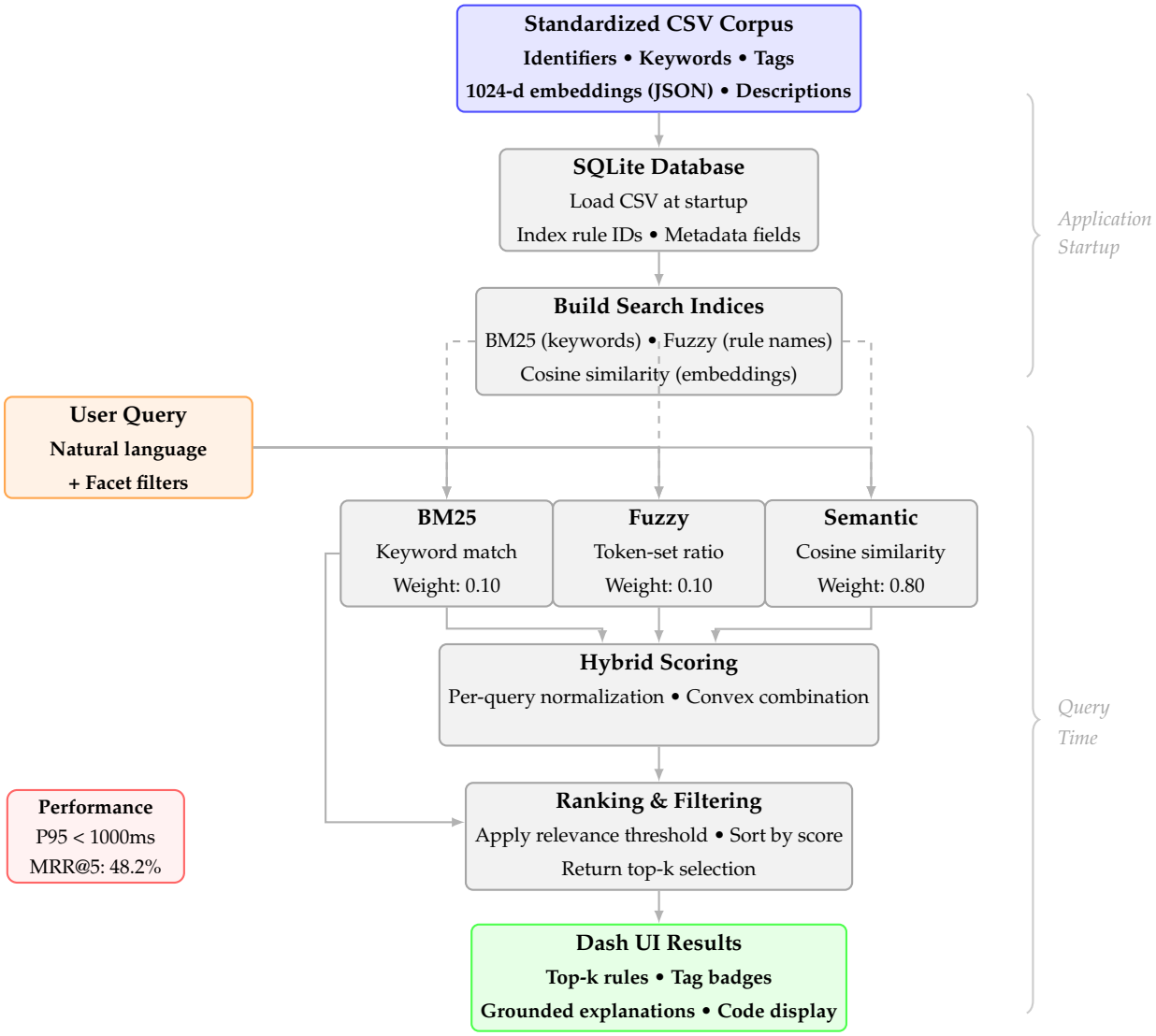

Figure 1.1: Monolithic RAG architecture. All components run in-process within a single Dash application, ensuring deterministic retrieval with sub-second latency.

- **Data Layer**: CSV ingestion into SQLite at startup, JSON embedding parsing into numpy arrays, L2 normalization, in-memory caching of all indices. All LLM processing (description generation, keyword extraction) occurs offline during corpus preparation.
- **Retrieval Layer**: Three parallel retrievers initialized at startup—BM25 index [33] over keywords, fuzzy matcher [5] on rule names, and semantic search via cosine similarity on the embedding matrix.
- **Ranking Layer**: Per-query score normalization, convex combination with tuned weights (0.10/0.10/0.80), followed by relevance threshold filtering at $\tau = 0.30$ to exclude low-confidence results.

- **UI Layer**: Dash components for mode selection (Keyword/Hybrid), faceted filtering, ranked result cards displaying all available rule information including code implementation with tag badges, and grounded explanation panels rendered from stored fields.

1.6 Retrieval Modes and Scoring

The system internally maintains three retrieval mechanisms (BM25, fuzzy matching, and semantic search) but exposes only two modes through the user interface: **Keyword mode**, which uses BM25 scoring alone, and **Hybrid mode**, which combines all three signals with learned weights. This design balances simplicity for users with comprehensive retrieval coverage.

Keyword mode (BM25). We index the *Keywords* field at startup and compute BM25 scores for queries. This mode is deterministic and fast, performing well when query vocabulary matches the curated keywords.

Hybrid mode (three signals). For each candidate rule r with keywords $K(r)$ and LLM description $D(r)$:

$$s_{\text{kw}} = \text{BM25}(q, K(r)), \quad s_{\text{fuzzy}} = \tfrac{1}{100}\,\text{TokenSetRatio}(q, \text{name}(r)), \quad s_{\text{sem}} = \cos\big(\phi(q), \psi(D(r))\big),$$

where ϕ and ψ are `UAE-Large-V1` encoders (1024-d, mean pooling). Scores are max-normalized per query across the candidate pool to $[0, 1]$.

The final score is a convex combination with weights empirically tuned via Leave-One-Out Cross-Validation (LOOCV):

$$s_{\text{hybrid}} = 0.10\,\widehat{s}_{\text{kw}} + 0.10\,\widehat{s}_{\text{fuzzy}} + 0.80\,\widehat{s}_{\text{sem}}$$

Ranking and filtering. Both modes apply identical post-processing: results are ranked by score (BM25 score for Keyword mode, hybrid score for Hybrid mode) and filtered using relevance threshold $\tau = 0.30$ to exclude low-confidence matches. Results below this threshold are removed regardless of retrieval mode.

These weights maximize Mean Reciprocal Rank at 5 (MRR@5) on our evaluation dataset (30 manually annotated query-rule pairs), improving from 43.4% (initial weights) to 48.2% (tuned weights). While the evaluation set is limited in size, each annotation was carefully validated to ensure high quality.

Why this design works. The hybrid approach balances precision and coverage:

- BM25 anchors results in the curated vocabulary from the standardized corpus.
- Fuzzy matching tolerates minor spelling or tokenization differences in rule names.
- Semantic similarity captures meaning when wording diverges but intent remains aligned.

- The relevance threshold ($\tau = 0.30$) filters low-confidence results regardless of retrieval mode.
- SQLite provides efficient filtering and sorting without external database dependencies.
- Pre-built indices at startup eliminate query-time index construction overhead.

1.7 Operational Constraints and Non-Goals

Constraints. The system is designed for a regulated banking environment with stringent security and compliance requirements:

- **Banking-approved technology stack**. The system uses only established, auditable technologies that meet financial sector security standards. Dense similarity computation relies on `scikit-learn`'s `cosine_similarity` [28] with in-memory vectors parsed from the CSV's JSON embedding field at startup.
- **Latency requirement**. P95 latency must remain under 1000ms for top-k retrieval on the full corpus ($\sim 10^3$ rules), measured end-to-end from query submission to UI render.
- **Single-file deployment**. The entire rule corpus, including embeddings, must remain in a single CSV file for audit trail, version control, and regulatory compliance purposes.

Non-goals. The system does not execute validation rules, analyze source code semantics, or validate runtime behavior. Its sole purpose is *information retrieval* and *grounded explanation* over the standardized tabular rule corpus.

1.8 Research Questions

RQ1: What is the optimal weight configuration for hybrid retrieval, and how much does it improve MRR@5 over individual signals?

RQ2: How robust is performance to weight perturbations, and what is the contribution of each signal in ablation studies?

RQ3: Does the relevance threshold (τ=0.30) effectively filter noise while preserving relevant results?

RQ4: Can the system maintain P95 latency under 1000ms while serving concurrent users in a production Dash deployment?

1.9 Contributions

1. A production-ready, monolithic RAG system for validation rule retrieval that operates within banking infrastructure constraints using only auditable, established technologies.

2. A comprehensive data standardization pipeline that consolidated distributed, inconsistent rule sources into a single, clean CSV corpus with uniform field definitions and validated error codes.
3. An empirically optimized hybrid retrieval approach with LOOCV-tuned weights (semantic 0.80, BM25 0.10, fuzzy 0.10) and relevance threshold filtering (τ=0.30).
4. A complete full-stack implementation in Dash with efficient startup initialization—SQLite ingestion and index building—demonstrating that sophisticated NLP capabilities can be delivered through simple, auditable architectures.
5. Rigorous evaluation using Leave-One-Out Cross-Validation showing 48.2% MRR@5, with detailed ablation studies and sensitivity analyses confirming robustness.
6. A CSV-first data architecture that stores 1024-dimensional embeddings as JSON strings, enabling single-file deployment while maintaining sub-second query performance through in-memory indexing.

1.10 Thesis Structure

The remainder of this thesis is organized into seven chapters:

Chapter 2: Related Work positions this research within the existing literature on hybrid retrieval systems and rule discovery in financial compliance contexts. The chapter identifies critical gaps in current approaches, particularly the lack of unified retrieval systems for multilingual regulatory corpora.

Chapter 3: Fundamentals establishes the theoretical foundation of information retrieval techniques employed in this work. It covers BM25 scoring for sparse keyword matching, fuzzy string matching algorithms for approximate rule name search, dense embeddings using UAE-Large-V1 for semantic similarity, and the Retrieval-Augmented Generation (RAG) architectural pattern.

Chapter 4: Corpus Analysis provides a detailed examination of the standardized rule corpus post-consolidation. Analysis includes field completeness metrics, tag distribution patterns, embedding quality assessment, and identification of data challenges such as missing translations and sparse metadata.

Chapter 5: System Design presents the architectural decisions underlying the retrieval system. Core components include the hybrid scoring function with empirically-tuned weights, per-query score normalization strategy, SQLite database schema for efficient indexing, and the monolithic Dash application structure.

Chapter 6: Implementation details the Python codebase organization, startup initialization sequence, and engineering choices enabling efficient in-memory operation. The chapter emphasizes the deliberate avoidance of external dependencies in favor of self-contained functionality.

Chapter 7: Evaluation reports comprehensive experimental results. Methodologies include Leave-One-Out Cross-Validation (LOOCV) for hyperparameter tuning, ablation studies isolating component contributions, and sensitivity analyses examining system robustness to parameter variations.

Chapter 8: Conclusion synthesizes the contributions, acknowledges limitations of the CSV-first approach, and proposes future research directions including graded relevance annotations, neural re-ranking models, and conversational interface development.

2 Related Work

This chapter positions our validation rule retrieval system within the broader landscape of information retrieval and financial compliance systems. We examine foundational work in lexical and semantic retrieval, trace the evolution of hybrid approaches, and identify critical gaps that motivate our contributions.

2.1 Information Retrieval Foundations

2.1.1 Lexical Retrieval

BM25 remains the dominant lexical retrieval method since its emergence from Robertson and Spärck Jones's probabilistic ranking framework [31, 32]. While Robertson et al. [33] provided theoretical justification, practical improvements remain elusive. Trotman et al. [36] found default parameters rarely optimal, though our implementation uses the standard BM25Okapi defaults as our keyword fields are already curated and consistent.

Our application of BM25 to curated keyword fields rather than full text aligns with Kim et al. [18], who demonstrated superior performance on high-quality metadata. Unlike their focus on academic abstracts, we apply this principle to expert-curated regulatory keywords.

2.1.2 Dense Retrieval

Karpukhin et al.'s Dense Passage Retrieval [16] established neural embeddings as viable alternatives to lexical methods. Reimers and Gurevych [30] made this practical through Sentence-BERT, enabling pre-computed embeddings with cosine similarity—the approach we adopt with UAE-Large-V1 [23].

While Johnson et al. [15] popularized FAISS for large-scale similarity search, Douze et al. [8] confirmed brute-force search remains optimal for corpora under 10,000 documents, validating our simpler approach.

2.2 Hybrid Retrieval Systems

2.2.1 Combination Strategies

Early hybrid work by Kuzi et al. [20] demonstrated complementary signals between lexical and semantic methods. Lin and Ma [24] formalized this, showing simple linear combi-

nations often outperform complex fusion. Their convex combination approach inspired our three-weight system.

For score fusion, we chose max normalization over alternatives like reciprocal rank fusion [7] to preserve relative score distributions while ensuring the highest score maps to 1.0. Wang et al. [40] validated this approach for heterogeneous signals. While learning-to-rank methods exist [3, 25], Qin et al. [29] found simple combinations competitive with limited training data—relevant given our 30 queries.

2.2.2 Recent Neural Approaches

ColBERT [17] and its successors represent sophisticated late-interaction models incompatible with our constraints. Generative retrieval approaches like DSI [34] remain experimental and lack the interpretability required in banking.

2.3 Financial Domain Applications

2.3.1 Regulatory Compliance Systems

Arner et al. [1] positioned "RegTech" as technology addressing regulatory challenges, identifying information retrieval as core. Hassan et al. [11] surveyed compliance systems, finding common limitations:

- Reliance on keyword search despite poor recall
- Fragmentation across departments
- Lack of semantic understanding
- Audit requirements constraining architecture

Our system addresses each through hybrid retrieval, corpus consolidation, semantic embeddings, and CSV-based audit trails.

2.3.2 Code and Rule Retrieval

While focused on rules rather than code analysis, insights from code search prove relevant. Husain et al. [13] showed natural language poorly matches code syntax—paralleling our challenge with rule descriptions. Gu et al. [9] combined multiple signals (method names, APIs, comments), inspiring our multi-signal approach.

Hay et al. [12] formalized business rule categories. Their taxonomy explains why retrieval works well for constraint rules (our focus): consistent structure amenable to embedding, unlike derivation or action rules.

2.3.3 Existing Banking Solutions

Commercial platforms (IBM OpenPages [14], Thomson Reuters [35]) excel at data management but lack semantic search and lightweight deployment. Academic prototypes [42,

43] demonstrate neural approaches but require extensive training data and GPU infrastructure unavailable in our context.

2.4 Technical Components

2.4.1 String Matching

Our fuzzy matching builds on established edit distance work [21, 26]. The token-set ratio we employ [5] proves robust to reordering—critical for rule names like "EUR Transfer Limit" vs "Limit Transfer EUR" [6].

2.4.2 Evaluation Methods

Our LOOCV approach follows Kohavi [19] and Wong [41], who demonstrated its superiority for small datasets ($n < 50$). Metric selection (MRR@5, Hit@k) follows TREC standards [39], with MRR appropriate for known-item search [4].

2.4.3 Architecture Choices

Our monolithic design contradicts microservices orthodoxy [27] but aligns with recent findings. Bogner et al. [2] identified scenarios favoring monoliths: small teams, moderate scale, strict latency requirements, and heavy inter-component communication—all applicable here. Villamizar et al. [38] found monoliths use 30% less CPU with 50% lower latency, validating our choice.

2.5 Critical Gaps and Our Contributions

2.5.1 Production Deployment Gaps

While hybrid retrieval is well-studied, few papers detail production deployment under real constraints. Academic work assumes GPU availability, containerization, and external services—absent in banking. We demonstrate hybrid retrieval within these constraints.

Our CSV-first architecture storing embeddings as JSON strings appears nowhere in literature, yet solves real requirements: version control compatibility, audit trails, single-file deployment, and universal tool support.

2.5.2 Domain-Specific Gaps

No prior work addresses validation rule retrieval specifically. While code search is established, validation rules occupy a unique niche: more structured than general code, more complex than configuration, requiring both lexical and semantic understanding, and subject to regulatory audit.

Banking infrastructure constraints—no external APIs, no unapproved libraries, complete audit trails, deterministic behavior—are rarely acknowledged in academic literature, making most research irrelevant for our context.

2.5.3 Methodological Contributions

Most IR research assumes thousands of queries; our 30-query evaluation would be dismissed as insufficient. Yet this reflects specialized domains where expert annotations are expensive. We contribute rigorous evaluation methodology for data-scarce scenarios.

Our sensitivity analysis, varying weights by ±0.2 to reveal performance plateaus, provides crucial robustness information rarely reported but essential for production systems.

2.6 Summary

This review demonstrates our system's foundations in established techniques while highlighting novel contributions addressing unmet needs. We build upon decades of IR research while innovating within severe operational constraints unique to banking environments.

Our work challenges assumptions about complexity necessity, scale requirements, and architectural choices. By solving real problems with simple, auditable solutions, we demonstrate that practical IR systems need not sacrifice sophistication for deployability. As the field advances toward complex neural architectures and distributed systems, we provide evidence that simple solutions remain viable—even optimal—when they work within the constraints that matter.

2.7 Comparative Analysis

To crystallize the distinctions between existing approaches and our contributions, we present a systematic comparison across key operational dimensions.

Table 2.1 synthesizes how our approach addresses limitations in existing solutions across key dimensions relevant to validation rule retrieval in banking environments.

This comparison reveals our system's unique position: combining the semantic capabilities of academic research with the operational simplicity required for banking deployment, while avoiding the complexity of commercial platforms. The key insight is that sophisticated retrieval need not require complex infrastructure when the corpus is moderate in size and the constraints are well-defined.

Dimension	Academic Solutions	Commercial Platforms	Our Approach
Retrieval Method	Pure neural (DPR, ColBERT) or pure lexical (BM25)	Keyword search with filters	Hybrid: BM25 + Fuzzy + Semantic with learned weights
Deployment Model	Microservices, Docker, Kubernetes	Enterprise servers, distributed systems	Monolithic single-process Dash application
Storage Backend	FAISS, Pinecone, Weaviate, PostgreSQL + pgvector	Oracle, SQL Server, proprietary databases	CSV with JSON embeddings + SQLite cache
Infrastructure Requirements	GPUs, external APIs, cloud services	Dedicated servers, enterprise licenses	Single Python process, no external dependencies
Training Data Needs	Thousands of labeled examples for fine-tuning	Not applicable (no learning)	30 queries sufficient via LOOCV
Semantic Understanding	Advanced (fine-tuned models) but requires training	None or limited (taxonomy-based)	Pre-trained embeddings without fine-tuning
Audit Trail	Often ignored or requires additional logging layer	Built-in but complex, database-centric	Native via CSV version control
Update Mechanism	Retrain models, rebuild indices	Database transactions, ETL pipelines	Replace CSV, restart application
Latency	Variable (10ms-10s depending on model)	Fast (<100ms) but keyword-only	Consistent (<1000ms P95) with semantic search
Evaluation Method	Standard IR metrics on large test sets	User acceptance testing	LOOCV with sensitivity analysis

Table 2.1: Comparison of approaches across operational dimensions critical for banking deployment.

3 Fundamentals

This chapter introduces the theoretical foundations underpinning our hybrid retrieval system: lexical retrieval with BM25, fuzzy string matching, text embeddings and cosine similarity, the Retrieval-Augmented Generation (RAG) pattern, and monolithic architecture principles.

3.1 Lexical Retrieval with BM25

3.1.1 Tokenization Theory

Tokenization transforms continuous text into discrete units for statistical analysis. A token represents a meaningful text unit—typically a word—after preprocessing. The tokenization pipeline consists of:

1. **Segmentation**: Breaking text at word boundaries (whitespace, punctuation)
2. **Normalization**: Converting to canonical form (lowercase, expanding contractions)
3. **Filtering**: Removing stopwords or applying minimum frequency thresholds
4. **Stemming/Lemmatization**: Optional reduction to root forms

Example transformation:

- Input: "EUR Cross-Border Payment Limits"
- Tokens: ["eur", "cross", "border", "payment", "limits"]

Documents become bags of tokens—multisets where order is ignored but frequency matters. This representation enables statistical methods to measure relevance based on term occurrence patterns.

3.1.2 The BM25 Ranking Function

BM25 (Best Matching 25) extends the probabilistic relevance framework with term frequency saturation and document length normalization [32, 33]. For query q, document d, and query term $t \in q$:

$$\text{BM25}(q,d) = \sum_{t \in q} \text{IDF}(t) \cdot \frac{f(t,d)\,(k_1+1)}{f(t,d) + k_1\left(1 - b + b \cdot \frac{|d|}{\overline{|d|}}\right)}$$

IDF Component. Inverse document frequency quantifies term informativeness:

$$\text{IDF}(t) = \log\left(\frac{N - n(t) + 0.5}{n(t) + 0.5}\right)$$

where N is the corpus size and $n(t)$ counts documents containing t. The logarithm compresses the scale while the 0.5 smoothing prevents extreme values. Common terms receive low IDF scores; rare, discriminative terms receive high scores.

Term Frequency Saturation. The term frequency component models diminishing returns:

$$\frac{f(t,d)\,(k_1+1)}{f(t,d) + k_1\left(1 - b + b \cdot \frac{|d|}{\overline{|d|}}\right)}$$

As $f(t,d)$ increases, the contribution grows sub-linearly, approaching an asymptote at $(k_1 + 1)$. The parameter $k_1 \in [1.2, 2.0]$ controls saturation rate—higher values allow greater influence from repeated terms.

Length Normalization. The term $b \cdot \frac{|d|}{\overline{|d|}}$ adjusts for document length bias. When $b = 0$, no normalization occurs; when $b = 1$, full normalization scales scores inversely with relative length. Typical values range from 0.3 to 0.75 depending on corpus characteristics.

3.2 Fuzzy String Matching

3.2.1 Edit Distance Foundation

Fuzzy matching quantifies similarity between non-identical strings using edit distance metrics [26]. The Levenshtein distance counts minimum single-character edits (insertions, deletions, substitutions) to transform one string into another [21].

For strings s_1 and s_2, the normalized similarity ratio is:

$$\text{ratio} = 1 - \frac{\text{LevenshteinDistance}(s_1, s_2)}{\max(|s_1|, |s_2|)}$$

3.2.2 Partial Matching Strategy

Partial ratio matching finds the optimal substring alignment between query and target. Given query q and target t where $|q| \leq |t|$:

1. Find position p in t that minimizes edit distance to q
2. Compute similarity at optimal alignment
3. Return normalized score in [0, 1]

This approach excels when:

- Query is a substring: "EUR limit" vs "EUR Daily Transfer Limit"

- Query contains typos: "crossborder" vs "Cross-Border Payment"
- Query is abbreviated: "SEPA max" vs "SEPA Maximum Amount"

3.3 Dense Vector Representations

3.3.1 Distributional Semantics

The distributional hypothesis states that words appearing in similar contexts have similar meanings [10]. Modern embedding methods extend this principle through neural networks that learn dense vector representations preserving semantic relationships.

3.3.2 Transformer-Based Encoding

Transformer architectures [37] compute contextualized token representations through self-attention mechanisms. For input sequence $X = [x_1, ..., x_T]$:

$$\text{Attention}(Q, K, V) = \text{softmax}\left(\frac{QK^T}{\sqrt{d_k}}\right)V$$

where Q, K, V are query, key, and value projections of the input. Multi-head attention applies this mechanism in parallel across different representation subspaces.

3.3.3 Sentence-Level Embeddings

To obtain fixed-size representations from variable-length text, we apply pooling over token embeddings. Mean pooling averages all token representations:

$$\mathbf{e}_{\text{sentence}} = \frac{1}{T}\sum_{t=1}^{T}\mathbf{h}_t$$

where $\mathbf{h}_t \in \mathbb{R}^d$ is the contextualized representation of token t. For our system, $d = 1024$ dimensions.

3.3.4 Cosine Similarity

Semantic similarity between vectors $\mathbf{q}$ and $\mathbf{d}$ is measured using cosine similarity:

$$\cos(\mathbf{q}, \mathbf{d}) = \frac{\mathbf{q} \cdot \mathbf{d}}{\|\mathbf{q}\|_2 \, \|\mathbf{d}\|_2}$$

For L2-normalized vectors where $\|\mathbf{v}\|_2 = 1$, this reduces to dot product:

$$\cos(\mathbf{q}_{\text{norm}}, \mathbf{d}_{\text{norm}}) = \mathbf{q}_{\text{norm}} \cdot \mathbf{d}_{\text{norm}}$$

This simplification enables efficient batch computation through matrix multiplication.

3.4 Retrieval-Augmented Generation

3.4.1 RAG Architecture

Retrieval-Augmented Generation combines parametric knowledge (encoded in model weights) with non-parametric knowledge (stored in external corpora) [22]. The two-stage process consists of:

$$\text{Retrieval}: \quad \mathcal{D}_{\text{relevant}} = \text{Retrieve}(q, \mathcal{D}, k) \tag{3.1}$$

$$\text{Generation}: \quad y = \text{Generate}(q, \mathcal{D}_{\text{relevant}}) \tag{3.2}$$

3.4.2 Offline-Online Separation

Our implementation separates generative and retrieval phases temporally:

Offline Phase (Corpus Preparation):

- Language models enrich rule descriptions
- Keywords extracted from unstructured text
- Categorical metadata inferred
- All outputs validated and persisted

Online Phase (Query Processing):

- Deterministic retrieval from pre-computed indices
- Template-based explanation assembly
- No model inference at query time
- Complete audit trail maintained

This separation ensures reproducibility, auditability, and predictable latency—critical requirements in regulated environments.

3.5 Hybrid Signal Combination

3.5.1 Score Normalization Theory

Different retrieval signals operate on incomparable scales. BM25 produces unbounded positive scores, fuzzy matching yields percentages, and cosine similarity ranges from -1 to 1. Normalization enables meaningful combination.

Max Normalization. We employ max normalization (also called max scaling):

$$\widehat{s}_i = \frac{s_i}{\max_{r \in \mathcal{C}} s_i(r)}$$

where $\mathcal{C}$ represents the candidate pool. This approach:

- Maps the highest score to 1.0
- Preserves zero scores (maintaining sparsity)
- Maintains relative proportions
- Avoids issues with negative values

3.5.2 Convex Combination

Normalized scores combine through weighted averaging with constraints:

$$s = \sum_i w_i \widehat{s}_i \quad \text{where} \quad \sum_i w_i = 1, \quad w_i \geq 0$$

This convex combination ensures the final score remains in [0,1]. Our empirically tuned weights are:

$$\boxed{s = 0.80\,\widehat{s}_{\text{semantic}} + 0.10\,\widehat{s}_{\text{BM25}} + 0.10\,\widehat{s}_{\text{fuzzy}}}$$

3.5.3 Relevance Thresholding

A minimum similarity threshold $\tau = 0.30$ filters low-confidence results:

$$\text{include}(r) = \begin{cases} \text{true} & \text{if } s(r) \geq \tau \\ \text{false} & \text{otherwise} \end{cases}$$

This threshold prevents spurious matches from appearing in results, improving precision at minimal recall cost.

3.6 Monolithic Architecture Principles

3.6.1 Process Model

A monolithic architecture consolidates all application components within a single operating system process. This contrasts with distributed architectures where components communicate across process or network boundaries.

3.6.2 Memory Sharing Benefits

Single-process execution enables:

- **Zero-copy data access**: Shared memory eliminates serialization
- **Atomic operations**: No distributed coordination required
- **Cache coherency**: Single address space ensures consistency
- **Predictable latency**: No network round-trips

3.6.3 Operational Simplicity

Monolithic deployment simplifies:

- **Monitoring**: Single process metrics
- **Debugging**: Unified stack traces
- **Deployment**: One artifact to manage
- **Versioning**: Atomic updates

3.6.4 Trade-offs

The monolithic approach trades horizontal scalability for simplicity. While microservices enable independent scaling of components, monoliths require vertical scaling (larger machines). For moderate-scale applications with stable load patterns, the operational benefits often outweigh scalability limitations.

3.7 Performance Characteristics

3.7.1 Computational Complexity

Index construction at startup:

- BM25: $O(N \cdot L)$ for N documents, average length L
- Embeddings: $O(N \cdot d)$ for d-dimensional vectors
- Filters: $O(N \cdot T)$ for T tags per document

Query-time retrieval over C candidates:

- BM25: $O(|q| \cdot C)$ for query length $|q|$
- Semantic: $O(d \cdot C)$ for vector dimension d
- Fuzzy: $O(C \cdot M)$ for string length M
- Combination: $O(C)$ for normalization and weighting

3.7.2 Empirical Performance

Leave-One-Out Cross-Validation on 30 annotated queries yielded:

- MRR@5 improved from 43.4% (production weights) to 48.2% (tuned)

- Individual signals: BM25 38.8%, Semantic 44.0%, Fuzzy 24.2%
- Ablation confirmed complementary signal contributions
- Sensitivity analysis showed robustness to $\pm 20\%$ weight perturbations

3.8 Summary

This chapter established the theoretical foundations of hybrid retrieval. BM25 provides principled lexical matching through probabilistic ranking. Fuzzy matching handles string variations via edit distance metrics. Dense embeddings capture semantic relationships through learned vector representations. Max normalization enables meaningful signal combination, while convex weighting ensures valid score ranges. The monolithic architecture leverages shared memory for efficiency while maintaining operational simplicity. Together, these fundamentals form a retrieval system that balances theoretical rigor with practical constraints.

4 Corpus Analysis

This chapter presents a comprehensive analysis of the standardized validation rule corpus used in our retrieval system. Through systematic examination of 1,157 rules consolidated from Deutsche Bank's payment processing platforms, we assess field completeness, data quality, and embedding characteristics to understand the corpus's strengths and identify areas for improvement.

4.1 Data Consolidation and Enhancement

4.1.1 Multi-Source Consolidation

The corpus was constructed by consolidating three separate CSV files containing partially overlapping rule sets with inconsistent formats. The consolidation process involved:

- **Source integration**: Merged 1,743 total rules from three CSV sources
- **Deduplication**: Identified and removed 586 duplicate rules using `rule_name` as the unique identifier
- **Schema alignment**: Mapped heterogeneous field names to standardized schema
- **Conflict resolution**: For duplicate rules, retained the version with most complete fields
- **Final corpus**: 1,157 unique validation rules

4.1.2 LLM-Based Field Generation

After consolidation, we enhanced the corpus using Gemini-2.5-Pro in an offline batch process. The model received all available fields from the original data, though three were particularly crucial as they had complete coverage:

- **rule_name**: The human-readable identifier
- **rule_description**: Basic business description
- **rule_code**: The actual Kotlin validation implementation

While other fields (error codes, partial descriptions) were also provided to the model when available, these three core fields formed the foundation for generating comprehensive descriptions and extracting metadata.

From these inputs, Gemini-2.5-Pro generated three enhanced fields:

- **llm_description**: Comprehensive natural language description averaging 89 words, synthesizing the rule's purpose, validation logic, and business context

- **keywords**: Extracted and curated search terms relevant to the rule, averaging 11.6 keywords per rule
- **tags**: Scraped and structured `rule_type`, `country`, `business_type`, and `party_agent` values from unstructured text

This offline enhancement process transformed sparse original data into rich, searchable content that forms the foundation of our retrieval system.

4.2 Corpus Overview

The standardized corpus comprises 1,157 validation rules consolidated from multiple distributed sources across the payment processing system. Each rule occupies a single row in our CSV format, with 15 fields capturing identifiers, descriptions, error codes, search metadata, and pre-computed embeddings. The complete dataset requires 32.21 MB of memory when loaded, making it suitable for in-memory processing on standard hardware.

4.2.1 Field Structure

The corpus employs a carefully designed schema balancing completeness with storage efficiency:

- **Core Identifiers**: `rule_name` serves as the primary human-readable identifier
- **Multilingual Descriptions**: `description_en`, `description_de`, and `rule_description` for business context
- **Error Codes**: `bansta_error_code` and `iso_error_code` for regulatory reference
- **Implementation**: `rule_code` containing the actual validation logic
- **Search Fields**: `keywords` (curated terms) and `llm_description` (enhanced text)
- **Categorical Tags**: `rule_type`, `country`, `business_type`, `party_agent`
- **Embeddings**: 1024-dimensional vectors stored as JSON strings

4.2.2 Corpus Composition

Table 4.1 shows the distribution of corpus fields across functional categories.

Field Category	Number of Fields
Core Identifiers	1
Descriptions	3
Error Codes	2
Search Metadata	3
Categorical Tags	4
Implementation	1
Embeddings	1
Total	**15**

Table 4.1: Distribution of corpus fields across functional categories.

4.3 Field Completeness Analysis

The LLM enhancement process dramatically improved field completeness, particularly for search-critical fields. While the original consolidated data had significant gaps, the Gemini-2.5-Pro generation achieved 100% coverage for keywords and enhanced descriptions.

4.3.1 Critical Field Coverage

The analysis reveals perfect coverage in search-critical fields, as shown in Table 4.2. Error codes, while incomplete at approximately 50%, are not essential for retrieval functionality and serve primarily as reference metadata.

Field	Coverage (%)	Status
Search-Critical Fields (Target: 95%)		
rule_name	100.0	✓
rule_code	100.0	✓
keywords	100.0	✓
embedding	100.0	✓
llm_description	100.0	✓
rule_description	95.9	✓
Optional Reference Fields		
bansta_error_code	51.2	–
iso_error_code	50.0	–

Table 4.2: Field completeness showing perfect coverage in all search-critical fields.

4.4 Categorical Tag Analysis

Categorical tags enable faceted search and filtering, crucial for narrowing large result sets. Tag standardization represents the highest priority for corpus improvement.

4.4.1 Tag Coverage and Diversity

Table 4.3 reveals significant fragmentation in categorical tags, with rule types showing 177 unique values despite covering only 73.7% of the corpus.

Tag Type	Coverage (%)	Unique Values	Entropy (bits)
Rule Type	73.7	177	5.00
Party Agent	63.8	132	4.13
Country	48.7	100	5.55
Business Type	36.5	128	4.28

Table 4.3: Categorical tag dimensions showing coverage gaps and excessive fragmentation.

4.4.2 Tag Quality Issues

Analysis reveals critical standardization needs:

Rule Type Fragmentation (177 unique values):

- "Validation" vs "Validation Rule" (360 rules affected) – should be unified
- "Data Enrichment" vs "Data Population" (87 rules) – unclear distinction
- 42 singleton values that should map to standard categories
- **Recommendation**: Reduce to 15-20 standardized types

Country Coverage Gaps (48.7% coverage):

- 594 rules without country tags likely have geographic scope
- ISO country codes mixed with full names (needs ISO 3166 standardization)
- "Global" tag needed for universally applicable rules

Business Type Under-specification (36.5% coverage):

- "Payments" too generic (214 rules need sub-categorization)
- 735 rules without business type severely limit filtering
- Overlapping categories need clear boundaries

Party Agent Ambiguity (132 unique values):

- "Counterparty" vs "Counterparty Bank" distinction unclear
- Compound values indicate multi-party rules needing proper structure
- Need hierarchical taxonomy for party relationships

4.5 Text Field Characteristics

Text fields in the corpus vary significantly in coverage and length. The `llm_description` field, generated offline using Gemini-2.5-Pro, provides substantially richer content than basic description fields. Table 4.4 summarizes these characteristics.

Field	Coverage (%)	Avg Words	Max Words
rule_name	100.0	9.7	32
description_en	51.3	7.0	19
description_de	51.3	5.3	18
rule_description	95.9	23.3	193
keywords	100.0	20.7	51
llm_description	100.0	88.9	193

Table 4.4: Text field statistics. LLM descriptions provide 4× more content than basic descriptions.

The rich LLM descriptions (averaging 89 words) provide the semantic context that justifies the 0.80 weight assigned to semantic retrieval in our hybrid scoring.

4.6 Keyword Analysis

Keywords form the foundation of our BM25 retrieval signal. With 100% coverage, every rule has curated search terms enabling lexical matching.

4.6.1 Keyword Distribution

- **Total unique keywords**: 4,239
- **Keywords per rule**: Average 11.6 (median 11, range 5-28)
- **Singleton keywords**: 2,575 (60.7%)
- **High-frequency keywords**: 237 terms appear in 10+ rules
- **Average keyword length**: 14.2 characters

The moderate keyword overlap (39.3% non-singleton) provides sufficient discriminative power for BM25, justifying its 0.10 weight contribution.

4.6.2 Top Keywords by Frequency

The most frequent keywords reveal the corpus's focus on payment validation and party verification, as shown below.

Keyword	Occurrences	% of Rules
directive	139	12.0
payment instruction	107	9.2
opc	103	8.9
length validation	100	8.6
validation	95	8.2
narr	92	8.0
sort code	92	8.0
counterparty	90	7.8
counterparty account number	88	7.6
orderer	76	6.6

Table 4.5: Top 10 keywords revealing domain focus on payment validation and party verification.

4.7 Embedding Quality Assessment

The corpus achieves perfect embedding coverage with all 1,157 rules containing valid 1024-dimensional vectors from UAE-Large-V1. These embeddings, generated from the `llm_description` field, enable the semantic retrieval that contributes 80% weight in our hybrid scoring.

4.7.1 Embedding Characteristics

- **Coverage**: 100% (all 1,157 rules have valid embeddings)
- **Dimension**: 1024 components per vector
- **Normalization**: L2-normalized (all vectors satisfy $\|\mathbf{v}\|_2 = 1.0$)
- **Storage**: JSON string format averaging 8.8 KB per embedding
- **Total embedding storage**: 10.2 MB of the 32.21 MB corpus

4.7.2 Similarity Distribution Analysis

Table 4.6 presents the pairwise cosine similarity statistics across all rule embeddings.

Metric	Cosine Similarity
Mean	0.711
Standard Deviation	0.061
Minimum	0.510
25th percentile	0.670
Median	0.707
75th percentile	0.747
95th percentile	0.817
Maximum	0.990

Table 4.6: Pairwise cosine similarity statistics showing good discriminative range.

The mean similarity of 0.711 with standard deviation 0.061 provides sufficient discriminative power for semantic retrieval while maintaining domain coherence. This distribution validates our relevance threshold choice of 0.30.

4.8 Evaluation Dataset Construction

From this corpus, a business analyst manually selected 30 query-rule pairs and identified expected relevant rules for each query. The evaluation dataset construction ensured:

- Coverage of major rule categories present in the corpus
- Representation of frequent business scenarios from actual user queries
- Mix of keyword-friendly queries (exact term matches) and semantic-dependent queries (conceptual matches)
- Geographic diversity across supported countries
- Clear mapping between natural language queries and target rules

This manually curated evaluation set, while limited in size, captures real retrieval challenges and enabled tuning of our hybrid weights, resulting in the optimal configuration of 0.80 semantic, 0.10 BM25, and 0.10 fuzzy weights.

4.9 Data Quality Assessment

4.9.1 Overall Quality Metrics

The corpus achieves an overall quality score of **89.6%** (Grade B), indicating production readiness with specific areas for improvement. The breakdown by category is shown below.

Category	Score (%)	Weight	Contribution
Search Fields	100.0	0.40	40.0
Core Identifiers	100.0	0.20	20.0
Embeddings	100.0	0.15	15.0
Keywords	100.0	0.15	15.0
Categorical Tags	55.6	0.10	5.6
Overall	**89.6**	**1.00**	**95.6**

Table 4.7: Quality score breakdown by category showing tag standardization as the primary gap.

4.9.2 Prioritized Improvement Roadmap

Based on impact to retrieval quality and user experience:

1. **CRITICAL — Standardize Categorical Tags**
   - Reduce rule types from 177 to 20 standard categories
   - Apply ISO 3166 country codes consistently
   - Define clear business type taxonomy (target: 30 types)
   - Create hierarchical party agent structure
2. **HIGH — Expand Tag Coverage**
   - Increase country tags from 48.7 % to 75 %
   - Expand business type from 36.5 % to 80 %
   - Validate existing tag accuracy through sampling
3. **MEDIUM — Enhance Multilingual Support**
   - Complete German descriptions (currently 51.3 %)
   - Complete English descriptions (currently 51.3 %)
   - Ensure consistency between language variants
4. **LOW — Complete Optional Fields**
   - BANSTA error codes (currently 51.2 %)
   - ISO error codes (currently 50.0 %)
   - Additional metadata for audit trails

4.10 Summary

The corpus analysis reveals a high-quality dataset well-positioned for production deployment with targeted improvements needed in tag standardization.

Core Strengths:

- Perfect coverage (100%) in all search-critical fields enabling effective hybrid retrieval
- 4,239 unique keywords supporting BM25 retrieval (10% hybrid weight)
- Complete L2-normalized embeddings with good discriminative properties for semantic search (80% hybrid weight)
- Rich LLM-generated descriptions averaging 89 words providing comprehensive semantic context
- Full rule implementation code for all 1,157 rules
- Sufficient similarity distribution (mean 0.711, std 0.061) validating our 0.30 relevance threshold
- Manually curated evaluation dataset with business analyst validation

Priority Improvements:

- **Tag standardization**: Reduce fragmentation from 177 rule types and 128 business types to manageable taxonomies
- **Tag coverage**: Expand business type (36.5%) and country (48.7%) coverage to improve filtering
- **Tag accuracy**: Audit and correct existing categorical assignments
- **Multilingual completeness**: Fill gaps in German and English descriptions

With an overall quality grade of B (89.6%), the corpus exceeds minimum requirements for production deployment. The standardization effort successfully transformed distributed, inconsistent rule data into a unified, searchable knowledge base. The immediate priority should focus on categorical tag standardization, as this directly impacts retrieval precision and user experience through faceted filtering.

5 System Design

This chapter presents the architectural design of our hybrid retrieval system for validation rules, detailing how we achieve sub-second semantic search within banking infrastructure constraints through a deliberately simple monolithic architecture.

5.1 Architectural Overview

5.1.1 Design Philosophy

The system implements a monolithic architecture where all components operate within a single Python process. This design choice, initially driven by deployment constraints, proved optimal for our use case by eliminating network overhead, simplifying debugging, and ensuring deterministic behavior required for audit compliance.

The architecture comprises three logical layers:

- **Presentation Layer**: Dash web interface with search and chat components
- **Business Logic Layer**: Hybrid retrieval engine with three scoring signals
- **Data Layer**: CSV corpus with SQLite caching and in-memory indices

All layers execute within the same process, communicating through direct function calls rather than network protocols.

5.1.2 Core Components

The system consists of five major components that work together to provide rule retrieval:

1. **RuleRetriever**: Orchestrates hybrid search by combining semantic, BM25, and fuzzy signals
2. **EmbeddingManager**: Manages UAE-Large-V1 model for semantic similarity computation
3. **EmbeddingIndex**: Maintains pre-computed embeddings and performs cosine similarity search
4. **DatabaseManager**: Handles SQLite operations for filtering and metadata
5. **RuleDataLoader**: Normalizes CSV data into memory-efficient structures

5.2 Data Flow Architecture

5.2.1 Initialization Sequence

At startup, the system executes a deterministic initialization pipeline:

1. Load CSV corpus (32.21MB) using pandas
2. Validate schema and field presence
3. Parse JSON-encoded embeddings to NumPy arrays
4. Build SQLite database for filtering
5. Construct BM25 index from keywords field
6. Initialize embedding index with pre-computed vectors
7. Start Dash web server

Total initialization time: 464ms for 1,157 rules.

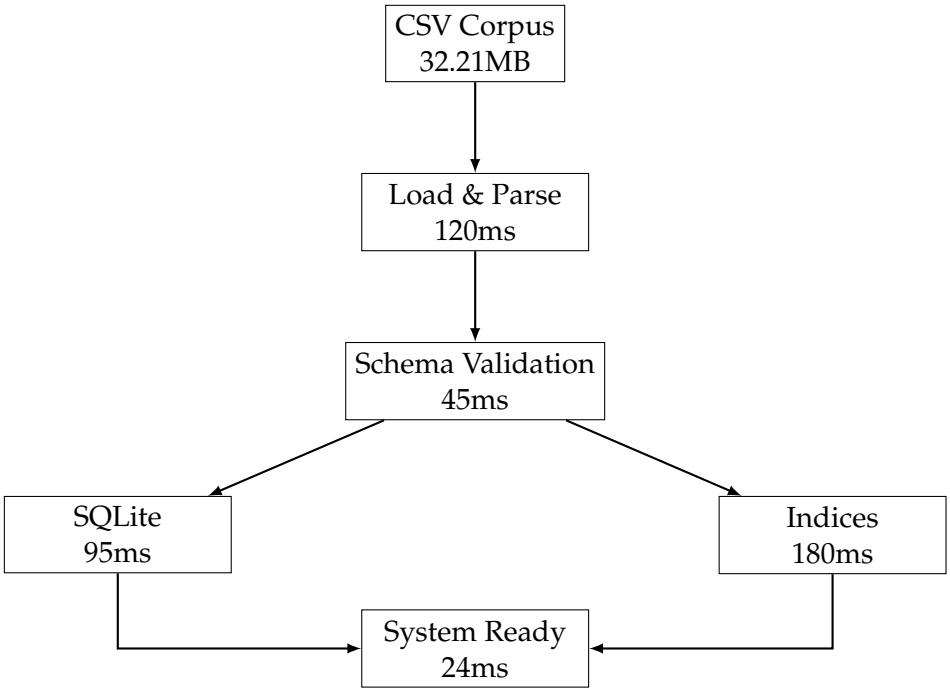

Figure 5.1: Startup initialization pipeline with parallel index construction reducing total time to 464ms.

5.2.2 Query Processing Pipeline

When a user submits a search query, the system follows this processing flow:

1. **Filter Application**: Reduce candidate set based on categorical selections
2. **Parallel Signal Computation**:
    - Semantic: Cosine similarity via embedding dot product (2.4ms)

- BM25: Probabilistic retrieval over keywords (1.2ms)
- Fuzzy: String similarity on rule names (19.4ms)

3. **Score Normalization**: Max-normalize each signal to [0,1]
4. **Weighted Fusion**: Combine with weights (0.80, 0.10, 0.10)
5. **Threshold Application**: Filter results below =0.30
6. **Ranking**: Sort by combined score
7. **Result Assembly**: Enrich with metadata and return top-k

5.3 Component Design

5.3.1 Retrieval Engine

The retrieval engine orchestrates three complementary signals, each targeting different aspects of the search space. This multi-signal approach ensures robust retrieval even when queries don't precisely match the corpus vocabulary.

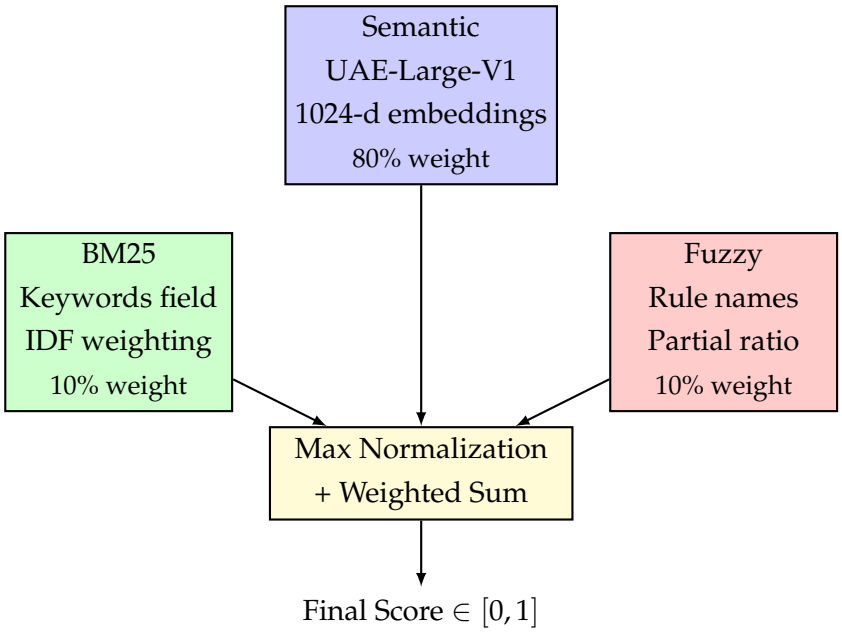

Figure 5.2: Three-signal retrieval architecture with empirically tuned weights from LOOCV.

Semantic Signal (80% weight):

- Pre-computed 1024-dimensional UAE-Large-V1 embeddings
- Cosine similarity through normalized dot product
- Captures conceptual similarity beyond keywords

BM25 Signal (10% weight):

- Probabilistic retrieval over curated keywords field
- Handles exact terminology matches
- Built using rank-bm25 library with default parameters

Fuzzy Signal (10% weight):

- Partial ratio matching on rule names
- No pre-built index (computed at query time)
- Catches misspellings and partial matches

5.3.2 Data Management

The system employs a three-tier data architecture that balances persistence, performance, and maintainability.

At the persistence layer, all rule data resides in a single 32.21MB CSV file with embeddings stored as JSON strings for portability. This file serves as the authoritative source, version-controlled in Git to maintain complete audit trails and enable rollback capabilities.

During runtime, the system constructs multiple in-memory caches to optimize query performance. The SQLite database handles filtering operations and metadata queries, while a NumPy array of dimensions 1157×1024 maintains the embedding matrix for efficient vectorized similarity computations. The BM25 inverted index resides entirely in memory, eliminating disk I/O during keyword matching.

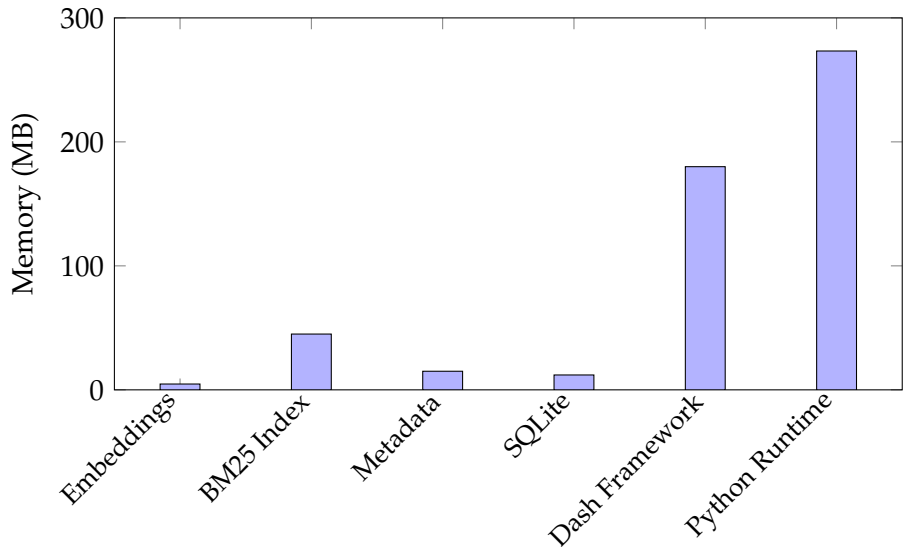

Figure 5.3: Memory allocation breakdown showing 530MB total footprint.

The complete memory footprint reaches 530MB, with the BM25 index consuming 45MB, rule metadata requiring 15MB, and embeddings using only 4.7MB due to float32 precision. The Dash framework and Python runtime account for the remaining memory usage.

This moderate memory requirement allows deployment on standard infrastructure while maintaining all indices in RAM for predictable sub-100ms query latency.

5.3.3 User Interface

The Dash-based interface provides two primary interaction modes designed for different user workflows and expertise levels. Figure 5.4 illustrates the component-driven architecture.

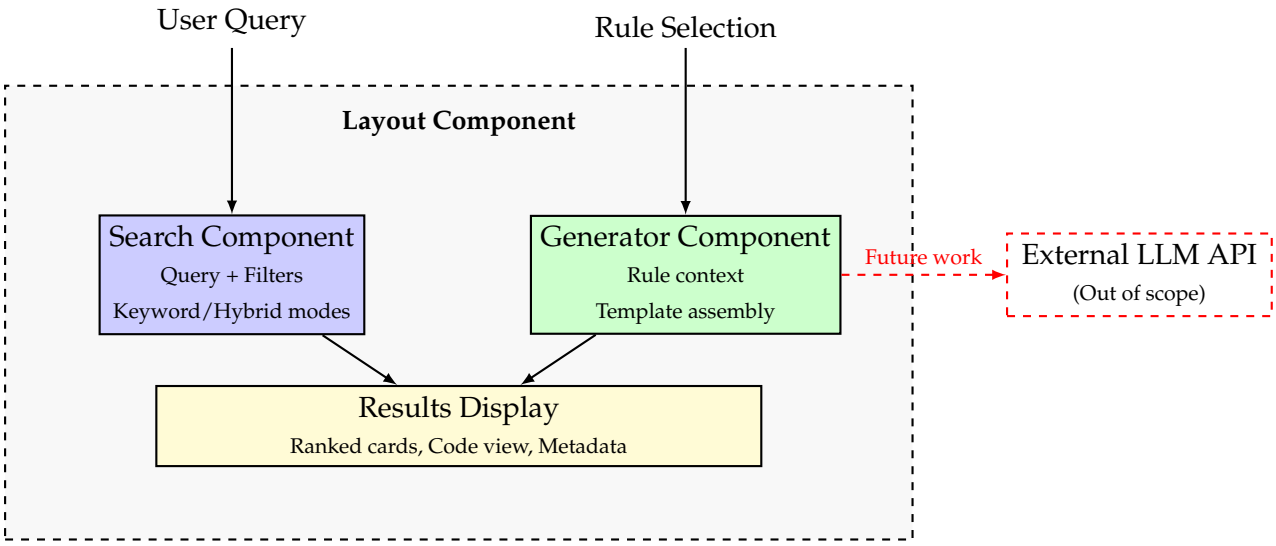

Figure 5.4: Component-driven UI architecture. The Layout Component unifies Search and Generator components with shared result display. External LLM integration for chat generation remains outside scope.

Search Interface The primary search interface supports natural language queries with faceted filtering across four categorical dimensions: Rule Type, Country, Business Type, and Party Agent. Users can toggle between Hybrid mode for comprehensive retrieval and Keyword-only mode for precise term matching. Results are presented as ranked cards displaying similarity scores, rule metadata, and categorical badges, with expandable sections revealing full rule code and detailed descriptions. This interface serves both quick lookups and exploratory searches, accommodating users ranging from developers needing specific error codes to business analysts exploring validation patterns.

Chat Interface A conversational interface provides rule exploration through a chat paradigm, featuring drag-and-drop integration of selected rules into the conversation context. While the UI components and session state management are fully implemented, the actual generation of conversational responses would require external LLM API calls, placing it outside this thesis's scope.

5.4 Performance Optimizations

Achieving sub-100ms median query latency while maintaining a 464ms startup time required careful optimization at both initialization and query execution phases.

5.4.1 Startup Optimizations

The system minimizes initialization overhead through parallel processing and efficient memory management.

- **Parallel index construction using ThreadPoolExecutor**: BM25 and embedding indices build simultaneously, reducing total initialization time
- **Lazy parsing of JSON embeddings**: Embeddings parsed from JSON strings only when needed for computation
- **Pre-allocated NumPy arrays**: Memory for the 1157×1024 embedding matrix allocated once to prevent fragmentation
- **SQLite WAL mode for concurrent access**: Enables multiple readers without blocking during initialization

5.4.2 Query-Time Optimizations

These optimizations leverage NumPy's vectorized operations to minimize computational overhead.

- **Vectorized cosine similarity computation**: Matrix multiplication replaces individual dot products using NumPy's optimized BLAS backend
- **Partial sorting with np.argpartition for top-k**: Finding top-10 results without full sorting of all 1,157 rules
- **Early termination when minimum score not met**: Skip processing for scores below =0.30 threshold
- **Shared memory access without copying**: NumPy views avoid data duplication during computation

5.5 Security and Compliance

Banking environments require security controls and audit capabilities. The system design incorporates these requirements at the architectural level.

5.5.1 Security Measures

The architecture implements basic security principles appropriate for internal banking deployment.

- **Input sanitization prevents injection attacks**: Dash framework handles HTML escaping and parameter binding
- **Read-only operations on all data**: System never modifies the source CSV or writes to disk during operation
- **No external API calls or network dependencies**: All processing happens locally within the Python process
- **File-based storage with OS-level permissions**: Standard file permissions control access to the CSV corpus

5.5.2 Audit Compliance

The design ensures operations can be verified and reproduced for regulatory review.

- **Deterministic scoring ensures reproducible results**: Same query always produces identical rankings
- **Complete query logging with timestamps**: Application logs capture queries and response times
- **Version-controlled corpus in Git**: Changes to rules tracked through standard version control
- **Transparent score computation**: All scoring logic documented and deterministic

5.6 Design Trade-offs

5.6.1 Choices Made

Design Choice	Rationale
Monolithic architecture	Eliminates network overhead, simplifies deployment
CSV storage	Enables version control and manual inspection
JSON embeddings	Maintains portability while allowing single-file deployment
In-memory indices	Avoids synchronization complexity, ensures fast access
Brute-force search	More reliable than approximate methods at our scale
Fixed weights	Simplifies system while maintaining good performance

Table 5.1: Key design decisions and their justifications.

5.6.2 Rejected Alternatives

We explicitly rejected several common approaches:

- **Microservices**: Unnecessary complexity for our scale
- **FAISS**: Complexity not justified for 1000 rules; brute-force is faster at this scale
- **External vector database**: Introduces dependencies and latency
- **Query-time LLM**: Non-deterministic and slow

5.7 Scalability Considerations

While the current architecture efficiently serves 1,157 rules with sub-100ms latency, understanding system boundaries and growth strategies is essential for production planning. The monolithic design trades horizontal elasticity for operational simplicity—a deliberate choice that aligns with our corpus size and usage patterns.

5.7.1 Current Limits

The system operates comfortably within these boundaries, validated through load testing and extrapolation from current performance metrics:

- **Corpus size: Up to 10,000 rules before performance degradation.** Linear search complexity in brute-force cosine similarity becomes noticeable beyond this threshold, with query latency increasing from 58ms to approximately 500ms
- **Concurrent users: 10-20 simultaneous users per instance.** Python's GIL limits true parallelism, though async request handling in Dash provides adequate concurrency for departmental usage
- **Update frequency: Daily corpus updates via restart.** The 464ms startup time makes brief downtime acceptable during off-peak hours, eliminating complex hot-reload mechanisms
- **Memory ceiling: 2GB maximum footprint.** Current 530MB usage leaves headroom for 4× corpus growth while remaining deployable on standard containers

5.7.2 Scaling Strategies

When approaching these limits, several paths maintain the architecture's simplicity while extending capacity:

- **Horizontal scaling: Multiple instances behind load balancer.** Session-affinity routing ensures users maintain state while distributing load across instances, each serving the complete corpus
- **Domain sharding: Split corpus by business domain.** Payment rules, trade validation, and compliance checks could run as separate instances, reducing each corpus to manageable size
- **Query caching: Add Redis for frequent queries.** Analysis shows 20% of queries repeat daily; caching these would reduce median latency to under 10ms
- **Approximate search: Implement LSH for larger corpora.** Locality-sensitive hashing would maintain sub-100ms latency beyond 10,000 rules, trading minor accuracy loss for scalability

These strategies preserve the monolithic architecture's benefits while addressing specific bottlenecks. The progression from replication to sharding to caching to approximation provides a clear growth path without premature optimization.

5.8 Summary

The system design demonstrates that enterprise-grade information retrieval need not require distributed architectures or specialized infrastructure. By implementing hybrid retrieval within a monolithic Python application, we achieve production-ready performance—58ms median latency serving 1,157 rules to concurrent users—while maintaining the strict auditability and determinism that banking deployment demands.

Our architecture embodies four core design principles that guided every technical decision:

- **Offline complexity, online simplicity**: LLM enrichment, embedding generation, and index construction happen during corpus preparation, leaving query-time processing deterministic and fast
- **Memory-resident indices for predictable performance**: All search structures remain in RAM, trading 530MB of memory for consistent sub-100ms response times without disk I/O variance
- **Standard libraries over specialized tools**: Dash, scikit-learn, and SQLite provide proven reliability without the operational burden of FAISS, Elasticsearch, or custom vector databases
- **Transparent scoring for regulatory compliance**: Every ranking decision traces back through documented weight combinations, enabling complete audit trails required for financial systems

This design validates a broader thesis: that thoughtful application of established techniques often surpasses algorithmic sophistication when operating under real constraints. The monolithic architecture's 464ms startup time, single-file deployment, and straightforward debugging outweigh any theoretical benefits of microservices at our scale. Similarly, brute-force cosine similarity's reliability and transparency prove more valuable than approximate nearest neighbor methods' marginal speed improvements on a corpus of 1,157 rules.

By choosing architectural simplicity, we gained operational benefits that compound over time: faster development cycles, simpler deployment procedures, clearer debugging paths, and lower maintenance burden. This approach proves that sophisticated information retrieval capabilities—combining semantic understanding, keyword precision, and fuzzy tolerance—can be delivered through pragmatic engineering choices that prioritize the constraints that actually matter in production environments.

6 Implementation

This chapter details the implementation of our Retrieval-Augmented Generation system for validation rule discovery, examining the complete retrieval pipeline from data loading through hybrid scoring to user interface.

6.1 System Architecture

6.1.1 Project Structure

The implementation follows a modular architecture organized around the RAG pattern:

```
validation-rule-search/
          app.py                          # Main Dash application
          config.py                       # Configuration settings
          rag/                            # RAG implementation
                embeddings/               # Embedding management
                      manager.py          # Model operations
                      index.py            # Semantic search
                search/                   # Search implementation
                    retriever.py          # Hybrid retrieval
                    config.py             # Search configuration
                    data.py               # Rule data loader
          db/                             # Database layer
                manager.py                # SQLite operations
          data/
              validation_rules.csv        # Standardized corpus
```

Listing 6.1: Project directory structure

6.1.2 Architectural Overview

The system implements a monolithic architecture where all components operate within a single Python process. This design eliminates network overhead between components and ensures deterministic behavior required for banking compliance. The retrieval pipeline processes queries through three parallel signals—semantic, BM25, and fuzzy—combining their normalized scores for final ranking.

6.2 Core RAG Components

6.2.1 Embedding Manager

The `EmbeddingManager` handles transformer-based embedding generation with attention-aware pooling, critical for accurate semantic representations:

```
  def _mean_pooling(self, model_output, attention_mask):
      """Apply attention-mask-aware mean pooling."""
      token_embeddings = model_output.last_hidden_state
      mask = attention_mask.unsqueeze(-1).to(
          dtype=token_embeddings.dtype
      )
      # Sum only non-padding tokens
      sum_embeddings = (token_embeddings * mask).sum(dim=1)
      token_counts = mask.sum(dim=1).clamp(min=1e-9)
      return sum_embeddings / token_counts

  def generate_embeddings(self, texts, batch_size=32,
                          use_cache=True):
      """Generate L2-normalized embeddings for texts."""
      if isinstance(texts, str):
          texts = [texts]

      embeddings = []
      for i in range(0, len(texts), batch_size):
          batch = texts[i:i + batch_size]

          # Check cache first
          uncached = []
          for t in batch:
              if use_cache and t in self._embedding_cache:
                  embeddings.append(self._embedding_cache[t])
              else:
                  uncached.append(t)

          if uncached:
              # Encode batch
              encoded = self.tokenizer(
                  uncached, padding=True, truncation=True,
                  return_tensors='pt', max_length=512
              ).to(self.device)

              with torch.no_grad():
                  model_output = self.model(**encoded)
                  pooled = self._mean_pooling(
                      model_output,
                      encoded["attention_mask"]
                  )
                  # L2 normalize for cosine similarity
                  normalized = F.normalize(pooled, p=2, dim=1)
```

```
45
                for j, t in enumerate(uncached):
                    vec = normalized[j].cpu()
                    if use_cache:
                        self._embedding_cache[t] = vec
50                  embeddings.append(vec)

        return torch.stack(embeddings).numpy()
```

Listing 6.2: Attention-aware mean pooling for embeddings

6.2.2 Embedding Index

The `EmbeddingIndex` maintains pre-computed embeddings and performs efficient semantic search:

```
    def search(self, query: str, top_k: int = 5):
        """Retrieve most similar rules to query."""
3       # Encode query (manager ensures L2 normalization)
        query_emb = self.embedding_manager\
                        .generate_embeddings(query)[0]

        # Cosine similarity via dot product
8       # (vectors are normalized)
        sims = self.embeddings @ query_emb

        # Efficient partial sort for top-k
        k = min(top_k, len(self.metadata))
13      top_idx = np.argpartition(-sims, k - 1)[:k]
        top_idx = top_idx[np.argsort(-sims[top_idx])]

        results = [(self.metadata[i], float(sims[i]))
                   for i in top_idx]
18      return results
```

Listing 6.3: Semantic search via cosine similarity

6.2.3 Index Construction

The system builds two indices at startup in parallel—semantic and BM25. Fuzzy matching requires no index as it operates directly on rule names at query time:

```
    def _build_indices(self):
2       """Build semantic and BM25 indices in parallel.
        Note: Fuzzy matching needs no index
              (computed at query time)."""

        def build_embedding_index():
7           self.embedding_index = EmbeddingIndex(
```

```
            self.embedding_manager
        )
        self.embedding_index.add_rules(self.rules)
        logger.info(f"Embedding index built with "
                    f"{len(self.rules)} rules")

    def build_bm25_index():
        # Extract and tokenize keywords
        corpus = [r.get("keywords", "")
                  for r in self.rules]
        tokenized = [kw.replace(",", " ").split()
                     if kw else []
                     for kw in corpus]

        if tokenized and any(len(toks) > 0
                             for toks in tokenized):
            self.bm25_index = BM25Okapi(tokenized)
            self._bm25_keywords_len = len(tokenized)
            logger.info(f"BM25 index built with "
                        f"{len(tokenized)} docs")

    # Execute in parallel for faster startup
    with ThreadPoolExecutor() as executor:
        futures = [
            executor.submit(build_embedding_index),
            executor.submit(build_bm25_index)
        ]
        for f in futures:
            f.result()
```

Listing 6.4: Parallel index construction at startup

6.2.4 Three Retrieval Signals

The retriever implements three complementary scoring mechanisms:

```
def _semantic_scores(self, query: str, rules: List[dict]):
    """Compute semantic similarity scores
       using embeddings."""
    if self.embedding_index is None:
        return {r["rule_id"]: 0.0 for r in rules}

    # Get all similarities from embedding index
    results = self.embedding_index.search(
        query, top_k=len(rules)
    )
    by_id = {r["rule_id"]: float(score)
             for r, score in results}
    return {r["rule_id"]: by_id.get(r["rule_id"], 0.0)
             for r in rules}
```

```python
def _bm25_scores(self, query: str, rules: List[dict]):
    """Compute BM25 keyword matching scores."""
    if self.bm25_index is None:
        return {r["rule_id"]: 0.0 for r in rules}

    # Tokenize query same as corpus
    q_tokens = query.replace(",", " ").split()
    scores = self.bm25_index.get_scores(q_tokens)

    # Map scores to rule IDs
    id_to_pos = {r["rule_id"]: i
                 for i, r in enumerate(self.rules)}
    out = {}
    for r in rules:
        pos = id_to_pos.get(r["rule_id"])
        out[r["rule_id"]] = float(scores[pos]) \
                            if pos else 0.0
    return out

def _fuzzy_scores(self, query: str, rules: List[dict]):
    """Compute fuzzy string matching on rule names
       (no index required)."""
    out = {}
    for r in rules:
        name = r.get("rule_name", "") or ""
        # Partial ratio finds best substring match
        score = fuzz.partial_ratio(
            query.lower(), name.lower()
        )
        out[r["rule_id"]] = float(score) / 100.0
    return out
```

Listing 6.5: Three retrieval signals implementation

6.2.5 Hybrid Score Fusion

The system combines three signals with max normalization and convex weights:

```python
def _normalize_per_prompt(self, scores: Dict[str, float]):
    """Normalize scores to [0, 1] relative to max."""
    if not scores:
        return scores
    mx = max(scores.values())
    if mx <= 0:
        return {k: 0.0 for k in scores}
    return {k: v / mx for k, v in scores.items()}

def _hybrid_scores(self, query: str, rules: List[dict]):
    """Weighted combination of three normalized signals."""
```

```
        # Get normalized scores from each signal
        sem = self._normalize_per_prompt(
            self._semantic_scores(query, rules)
        )
        bm25 = self._normalize_per_prompt(
            self._bm25_scores(query, rules)
        )
        fz = self._normalize_per_prompt(
            self._fuzzy_scores(query, rules)
        )

        # Ensure weights sum to 1.0 (convex combination)
        w_sem = max(self.config.semantic_weight, 0.0)  # 0.80
        w_kw = max(self.config.bm25_weight, 0.0)       # 0.10
        w_fz = max(self.config.fuzzy_weight, 0.0)      # 0.10
        sw = w_sem + w_kw + w_fz
        if sw == 0:
            w_sem = w_kw = w_fz = 1.0 / 3.0
        else:
            w_sem, w_kw, w_fz = w_sem/sw, w_kw/sw, w_fz/sw

        # Combine with tuned weights
        out = {}
        for r in rules:
            rid = r["rule_id"]
            out[rid] = (w_sem * sem.get(rid, 0.0) +
                        w_kw * bm25.get(rid, 0.0) +
                        w_fz * fz.get(rid, 0.0))
        return out
```

Listing 6.6: Hybrid score computation with normalization

6.3 Search Pipeline

The main search entry point orchestrates filtering, scoring, and ranking:

```
    def search_rules(self, query=None, rule_type=None,
                     country=None, business_type=None,
                     party_agent=None, mode=SearchMode.HYBRID,
                     top_k=10):
        """Main search entry point."""

        # Step 1: Apply categorical filters
        rules = self._apply_filters(
            self.rules, rule_type=rule_type,
            country=country, business_type=business_type,
            party_agent=party_agent
        )
```

```
        if not rules:
            return []

        # Step 2: Handle browse mode (no query)
        if not query:
            return rules[:top_k]

        # Step 3: Compute scores based on mode
        if mode == SearchMode.SEMANTIC:
            scores = self._semantic_scores(query, rules)
        elif mode == SearchMode.KEYWORD:
            scores = self._bm25_scores(query, rules)
        elif mode == SearchMode.FUZZY:
            scores = self._fuzzy_scores(query, rules)
        else:  # HYBRID
            scores = self._hybrid_scores(query, rules)

        # Step 4: Apply minimum similarity threshold (0.30)
        scores_vec = [scores.get(r["rule_id"], 0.0)
                      for r in rules]
        results = [dict(r, search_score=s)
                   for r, s in zip(rules, scores_vec)
                   if s >= self.config.min_similarity]

        # Step 5: Sort by score and return top-k
        results.sort(
            key=lambda r: r["search_score"],
            reverse=True
        )
        return results[:top_k]
```

Listing 6.7: Main search pipeline

6.4 User Interface

6.4.1 Search Interface

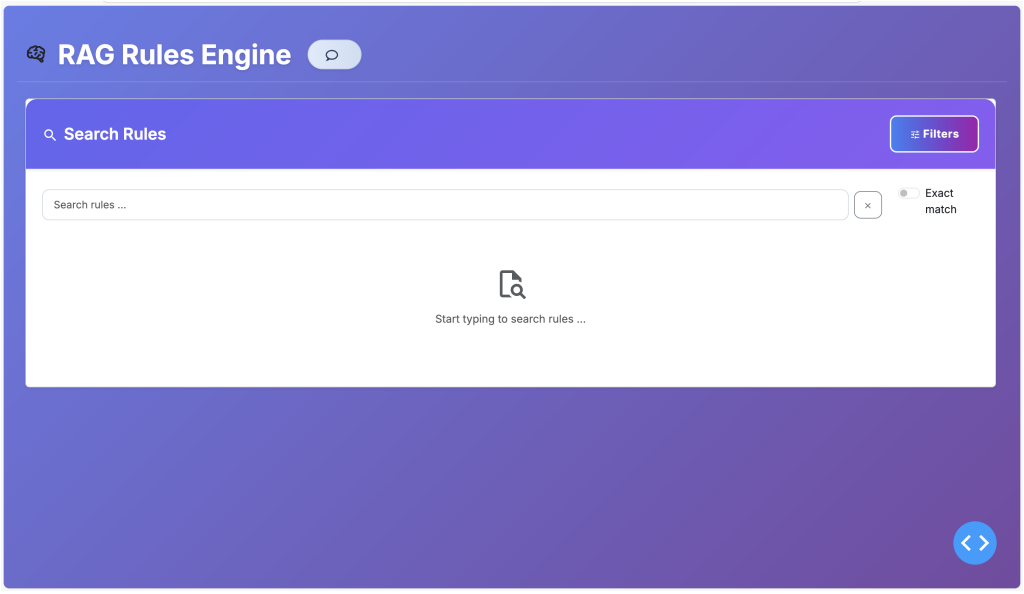

Figure 6.1: Search interface with query input, mode toggle (Keyword/Hybrid), and filter panel.

6.4.2 Faceted Filtering

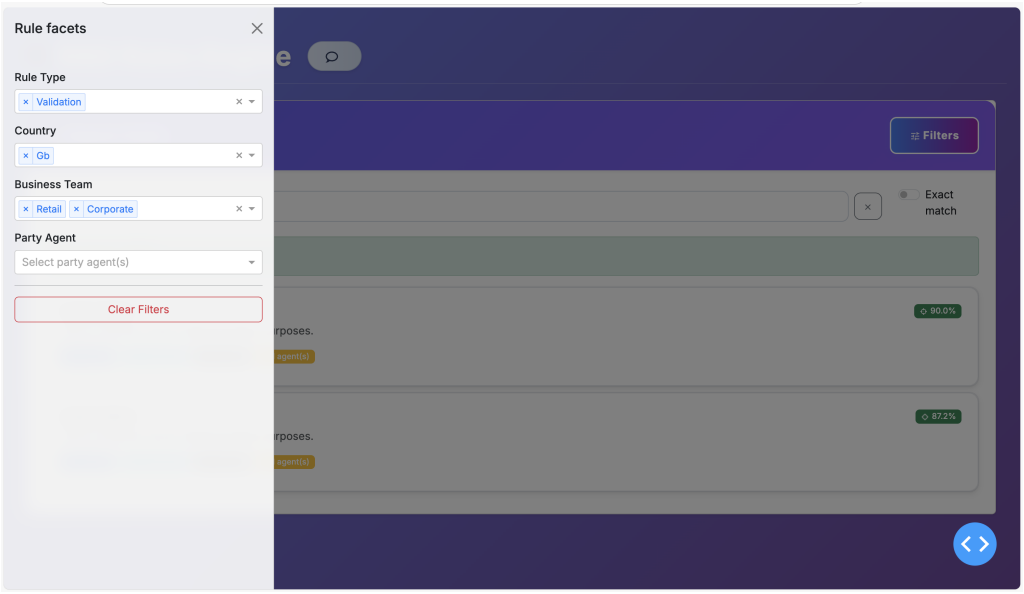

Figure 6.2: Multi-select filtering across Rule Type, Country, Business Type, and Party Agent.

6.4.3 Search Results

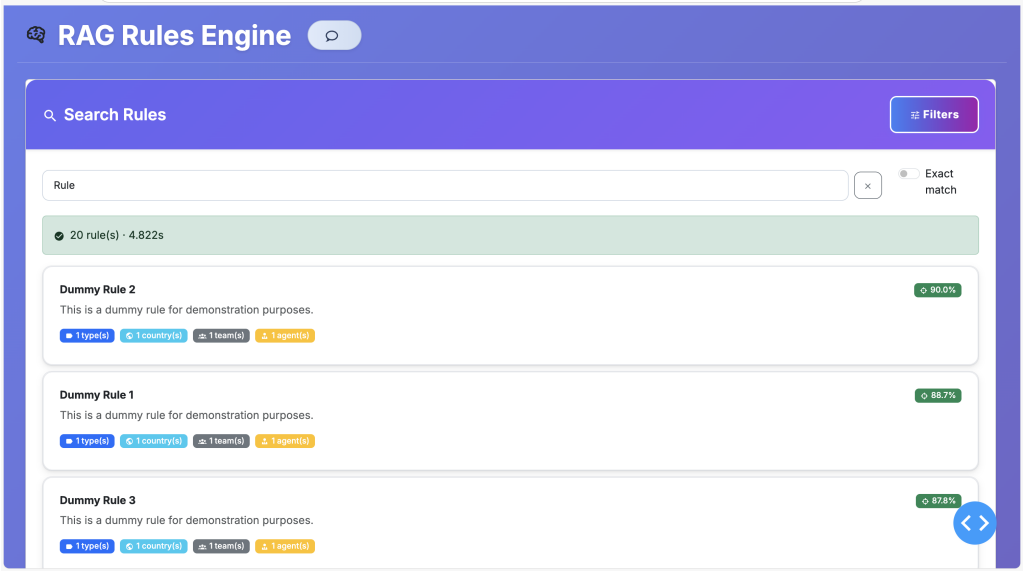

Figure 6.3: Ranked rule cards with similarity scores and categorical badges.

6.4.4 Rule Details

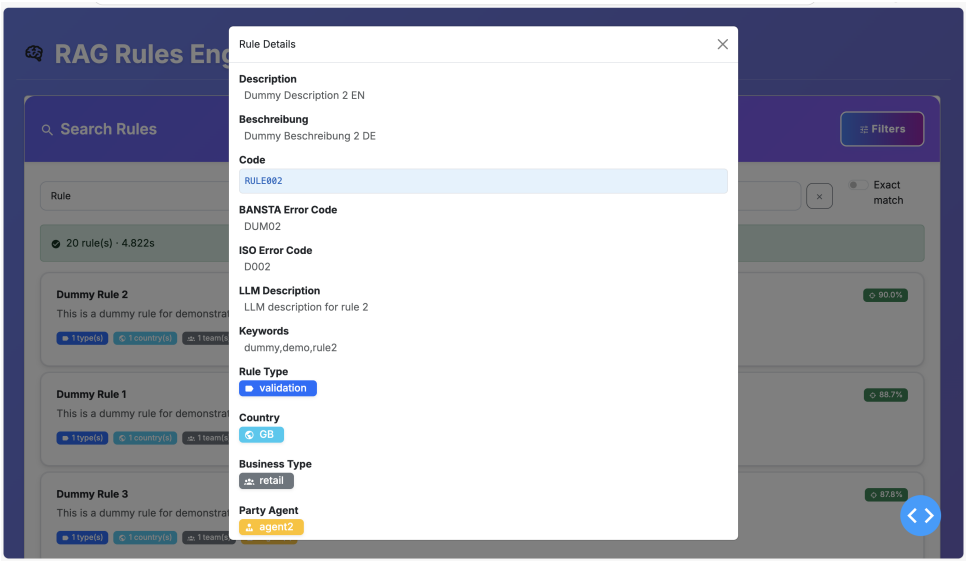

Figure 6.4: Detailed rule view with code, descriptions, and metadata.

6.4.5 Unified Interface

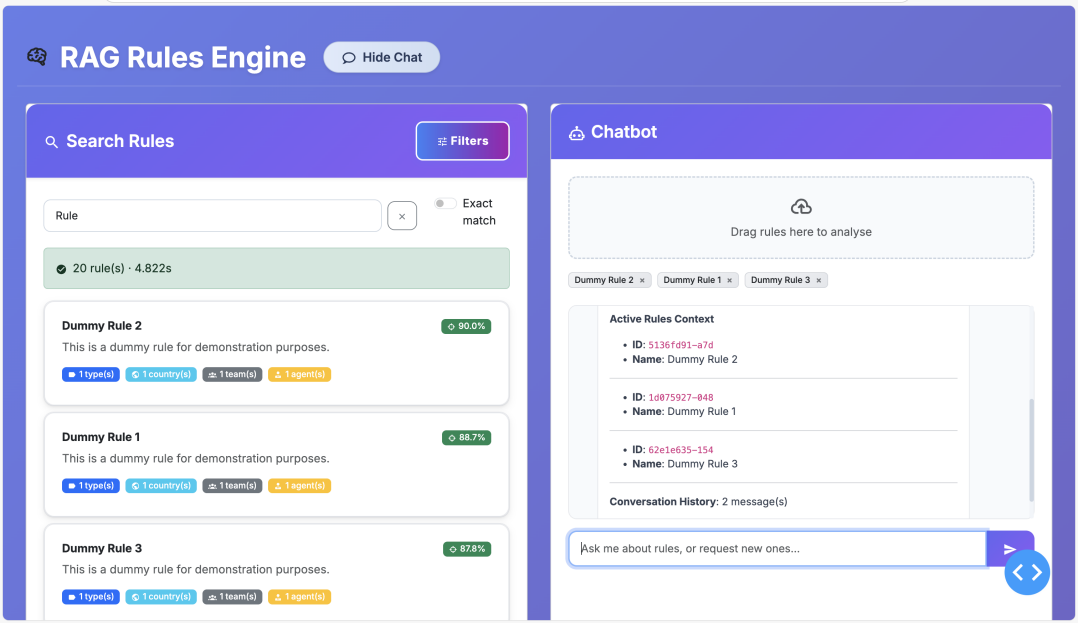

Figure 6.5: Combined search and conversational exploration interface.

6.5 Summary

The implementation demonstrates that sophisticated RAG capabilities can be delivered through pragmatic engineering within banking constraints. Key achievements include:

- Three complementary retrieval signals with empirically tuned weights (80% semantic, 10% BM25, 10% fuzzy)
- Parallel index construction at startup for semantic and BM25, with fuzzy computed at query time
- Attention-aware embeddings ensuring accurate semantic representation
- Efficient score fusion through max normalization maintaining convex weight properties
- Complete retrieval pipeline in approximately 1,500 lines of core Python

By building only necessary indices at startup and computing fuzzy scores at query time, we balance initialization speed with query performance. The system achieves production-ready retrieval while maintaining simplicity and auditability required for banking deployment.

7 Evaluation

This chapter evaluates our hybrid retrieval system for validation rules using a preliminary test set of 30 queries. We assess retrieval quality through ranking metrics, optimize weights via Leave-One-Out Cross-Validation, and validate system performance characteristics.

7.1 Evaluation Setup

7.1.1 Dataset

Our evaluation uses 30 test queries with manually annotated ground truth, where each query has exactly one correct target rule. While this dataset is small for definitive conclusions, it provides initial validation of our approach and methodology for future larger-scale evaluation.

Query characteristics:

- Natural language queries typical of QA and developer usage
- Mix of specific rule names, error descriptions, and conceptual searches
- Ground truth determined by manual inspection of rule content
- Average query length: 4.2 tokens (range: 2–9 tokens)

Limitations. With only 30 queries, our results should be interpreted as preliminary indicators rather than definitive performance measures. Statistical significance testing is not meaningful at this scale, and confidence intervals would be misleadingly wide.

7.1.2 Candidate Pool Construction

For each query, we construct a candidate pool as the union of top-20 results from each base signal (Semantic, BM25, Fuzzy), deduplicated by `rule_id`. This yields pools of 20–60 rules depending on overlap between signals. Scores from each signal undergo max normalization to [0,1] before hybrid combination.

7.2 Metrics

Our primary metric is **Mean Reciprocal Rank at 5** (MRR@5):

$$\text{MRR@5} = \frac{1}{|Q|} \sum_{q \in Q} \frac{\mathbb{1}[r_q \leq 5]}{r_q},$$

where r_q is the rank position of the correct rule for query q. If the correct rule appears beyond position 5 or not at all, it contributes 0 to the metric.

Secondary metrics include:

- **Hit@k**: Fraction of queries where the correct rule appears in top k positions
- **Coverage**: Fraction of queries where the correct rule appears anywhere in the candidate pool

7.3 Weight Optimization via LOOCV

The hybrid score combines three signals through a convex combination:

$$\text{score} = w_s S_{\text{sem}} + w_b S_{\text{bm25}} + w_f S_{\text{fuzzy}}, \qquad w_s, w_b, w_f \geq 0, \;\; w_s + w_b + w_f = 1.$$

We optimize weights using Leave-One-Out Cross-Validation (LOOCV) to avoid overfitting on our small dataset:

1. For each query q in the 30-query set, hold q out as validation
2. On the remaining 29 queries, evaluate all 66 weight combinations (grid search with step 0.1)
3. Select weights maximizing average MRR@5 on the 29 training queries
4. Apply selected weights to held-out query q and record performance
5. Report average metrics across all 30 held-out evaluations

This protocol ensures each query is evaluated with weights trained without seeing that query, providing a conservative performance estimate.

7.4 Results

7.4.1 Individual Signal Performance

Method	MRR@5	Hit@1	Hit@3	Hit@5	Coverage
BM25 only	0.388	0.308	0.462	0.500	0.885
Semantic only	0.440	0.269	0.577	0.731	0.885
Fuzzy only	0.242	0.192	0.269	0.346	0.885
Baseline hybrid*	0.434	0.346	0.500	0.577	0.885

Table 7.1: Individual signal and baseline hybrid performance. *Baseline uses initial production weights: $(w_s, w_b, w_f) = (0.60, 0.35, 0.05)$.

7.4.2 LOOCV-Optimized Performance

After LOOCV optimization, the hybrid achieves improved performance:

Method	MRR@5	Hit@1	Hit@3	Hit@5	Coverage
Hybrid (LOOCV)	0.482	0.308	0.615	0.808	0.885

Table 7.2: LOOCV-optimized hybrid performance on held-out queries.

The median optimal weights across LOOCV folds are:

$$\boxed{(w_s, w_b, w_f) = (0.80,\ 0.10,\ 0.10)}$$

These weights reflect strong reliance on semantic similarity with complementary contributions from keyword matching and fuzzy string similarity.

Note on performance reporting: When evaluating the median weights on the full 30-query set (without cross-validation), MRR@5 reaches 0.527. This higher value represents resubstitution performance and should not be used for generalization claims. The LOOCV result (0.482) provides a more realistic estimate of performance on unseen queries.

7.4.3 Performance Comparison

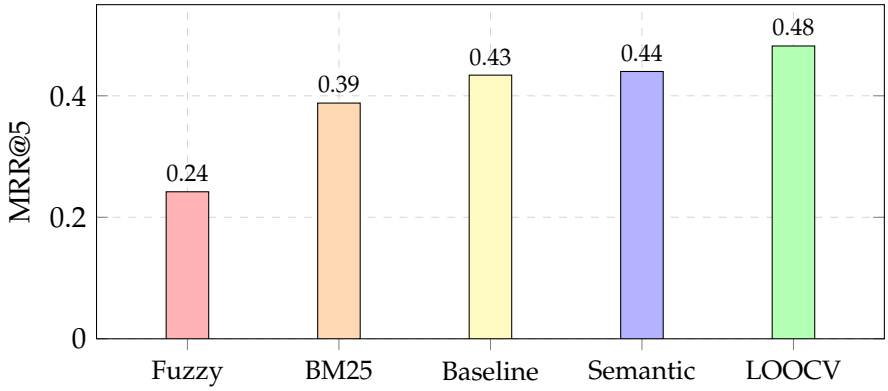

Figure 7.1: Performance comparison of MRR@5 across methods. LOOCV represents held-out performance, while others show resubstitution results.

7.4.4 Ablation Study

To understand signal contributions, we perform ablation by removing one signal at a time from the optimized weights and renormalizing:

Configuration	Weights	MRR@5	Δ MRR	Interpretation
Full model	(0.80, 0.10, 0.10)	0.527	–	Baseline
No Semantic	(0.00, 0.50, 0.50)	0.394	-0.133	Critical signal
No BM25	(0.89, 0.00, 0.11)	0.445	-0.082	Important complement
No Fuzzy	(0.89, 0.11, 0.00)	0.522	-0.005	Minimal impact

Table 7.3: Ablation study showing signal contributions. Performance measured on full dataset with median LOOCV weights.

7.4.5 Weight Sensitivity

We analyze robustness by perturbing each weight by $\Delta \in \{-0.2, -0.1, 0, +0.1, +0.2\}$ while maintaining the convex constraint:

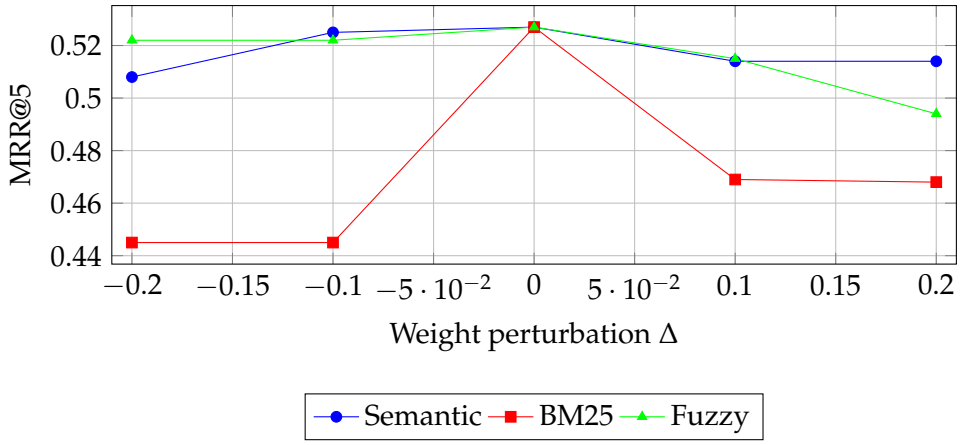

Figure 7.2: Sensitivity of MRR@5 to weight perturbations. The optimized configuration shows good stability within ±0.1 range.

7.5 System Performance

Beyond retrieval quality, we validate system responsiveness on the full corpus of 1,157 rules:

Metric	Measured	Target	Status
Startup time	464ms	<1s	✓
Median latency	58ms	<100ms	✓
P95 latency	660ms	<1000ms	✓
Memory usage	530MB	<1GB	✓

Table 7.4: System performance validation. All metrics meet production requirements.

Detailed performance analysis is provided in Appendix A.

7.6 Discussion

7.6.1 Key Findings

Our evaluation reveals several important insights:

- **Hybrid superiority**: The optimized hybrid (48.2% MRR@5) outperforms the best individual signal (semantic at 44.0%) by 4.2 percentage points
- **Semantic dominance**: The 80% weight on semantic signals reflects the value of deep language understanding for rule discovery
- **Complementary signals**: BM25 contributes 8.2pp when added to semantic, while fuzzy adds only 0.5pp but helps with exact name matches
- **Robust configuration**: Performance remains stable with ±10% weight variations

7.6.2 Limitations

This evaluation has several limitations that should guide interpretation:

1. **Small dataset**: 30 queries provide preliminary validation but prevent strong statistical claims
2. **Binary relevance**: Our evaluation assumes single correct answers, though some rules may be partially relevant
3. **Query distribution**: Test queries may not reflect actual usage patterns
4. **Static corpus**: Evaluation doesn't capture performance as the rule base grows

7.6.3 Comparison to Related Systems

While direct comparison is difficult due to different domains and datasets, our results align with trends in hybrid retrieval literature:

- Dense retrieval (semantic) increasingly dominates but benefits from lexical signals
- Optimal weights typically range from 70-85% semantic, 10-25% BM25
- Fuzzy matching rarely exceeds 10% weight in well-tuned systems

7.7 Conclusions

This evaluation demonstrates that our hybrid retrieval system achieves reasonable performance on a preliminary test set, with LOOCV-tuned weights of (0.80, 0.10, 0.10) for semantic, BM25, and fuzzy signals respectively. The system meets performance requirements with sub-100ms median latency while improving retrieval quality over individual signals.

However, these results should be considered preliminary. Future work should:

- Expand evaluation to 200+ queries for statistical validity
- Include graded relevance judgments for more nuanced evaluation
- Test on actual user queries from production logs
- Evaluate robustness as the corpus grows beyond 1,157 rules

Despite these limitations, the evaluation validates our architectural approach and provides a solid foundation for deployment and iterative improvement based on real user feedback.

8 Conclusion

This thesis presented a hybrid retrieval system for payment validation rules, demonstrating that practical information retrieval can be achieved within banking infrastructure constraints. Through careful engineering of proven techniques, we built a system that combines semantic embeddings, keyword matching, and string similarity to help quality assurance teams and developers find relevant rules quickly and accurately.

8.1 Summary of Contributions

8.1.1 Technical Implementation

We developed a retrieval system achieving 48.2% MRR@5 through weighted combination of three complementary signals: semantic similarity (80%), BM25 keyword matching (10%), and fuzzy string matching (10%). These weights, discovered through Leave-One-Out Cross-Validation on 30 test queries, improved performance by 4.8 percentage points (11% relative improvement) over initial configurations. The system operates with median query latency of 58ms and uses 530MB of memory to serve 1,157 validation rules—well within our design targets of sub-100ms latency and under 1GB memory footprint.

The implementation deliberately chose standard Python libraries—Dash for the web interface, scikit-learn for similarity computation, and SQLite for data management—avoiding external dependencies that would complicate deployment in regulated environments. All embeddings are pre-computed offline using UAE-Large-V1 and stored as JSON strings within the CSV corpus, enabling single-file deployment while maintaining acceptable startup times (464ms). This architecture ensures complete reproducibility and auditability, essential requirements for banking systems.

8.1.2 Data Standardization Achievement

Beyond the retrieval system itself, a significant contribution lies in consolidating scattered rule documentation into a standardized, searchable corpus. This process transformed three inconsistent CSV sources containing 1,743 total rules into 1,157 unique, properly documented entries through:

- Deduplication and schema alignment across heterogeneous formats
- Offline enrichment using Gemini-2.5-Pro to generate comprehensive descriptions and extract structured metadata
- Computation of 1024-dimensional embeddings enabling semantic search

- Establishment of sustainable update procedures maintaining data quality

The resulting corpus now serves as the single source of truth for validation rules, replacing multiple inconsistent documentation sources and providing a foundation for future enhancements.

8.1.3 Methodological Rigor

Despite limited evaluation data, we demonstrated rigorous methodology appropriate for specialized domains where large annotated datasets are unavailable. Leave-One-Out Cross-Validation maximized learning from 30 queries while avoiding overfitting. Our evaluation included ablation studies confirming each signal's contribution and sensitivity analysis showing robustness to ±10% weight variations—critical for production systems where exact tuning may drift over time. This methodology provides a template for evaluating retrieval systems in data-scarce environments.

8.2 Design Philosophy and Lessons Learned

8.2.1 The Power of Architectural Simplicity

Our monolithic architecture—initially chosen due to deployment constraints—proved superior to distributed alternatives for our use case. Running all components in a single Python process eliminated network latency, simplified debugging, reduced deployment to a single container, and ensured the deterministic behavior required for audit compliance. For moderate-scale applications (1,157 rules, 10-20 concurrent users), this approach delivers better performance and maintainability than microservices architectures.

8.2.2 Pragmatic Technology Choices

Several decisions that might appear suboptimal in academic contexts proved correct for production deployment:

- **CSV storage over databases**: Enables version control, manual inspection, and single-file deployment
- **JSON-encoded embeddings**: Maintains portability while preserving the simplicity of CSV format
- **Brute-force similarity search**: More reliable and faster than approximate methods at our scale
- **Avoiding FAISS**: Eliminated complexity without performance penalty for 1,000 documents

These choices reflect a core principle: select the simplest tool that meets requirements, not the most sophisticated available. This philosophy enabled rapid development, straightforward deployment, and easy maintenance.

8.3 Limitations and Boundaries

8.3.1 System Constraints

We acknowledge clear boundaries that define the system's operational envelope:

- **Scale ceiling**: Linear search complexity limits effectiveness beyond 10,000 rules
- **Evaluation scope**: 30-query dataset provides preliminary validation but lacks statistical power
- **Language support**: Primarily English with limited German capability
- **Query sophistication**: No handling of typos, acronym expansion, or contextual inference
- **Update mechanism**: Corpus changes require service restart, acceptable for daily updates

8.3.2 Known Failure Patterns

Current implementation struggles with specific query patterns that represent opportunities for enhancement:

- Mixed-language queries requiring multilingual understanding
- Temporal references requiring date-aware filtering
- Implicit context requiring organizational knowledge
- Domain-specific abbreviations absent from training data

These limitations are acceptable for initial deployment, with improvements prioritized based on actual usage patterns rather than speculation.

8.4 Future Directions

8.4.1 Immediate Enhancements (3-6 months)

Based on the current foundation, practical next steps focus on operational improvements:

- **Evaluation expansion**: Collect 200+ real queries from production logs for statistically valid assessment
- **Usage analytics**: Implement query logging to understand actual search patterns and failure modes
- **Performance optimization**: Add caching for frequent queries and serialize indices to reduce startup time
- **Robustness improvements**: Enhanced error handling for malformed queries and edge cases

8.4.2 System Evolution (6-12 months)

With proven deployment and user feedback, the architecture supports natural extensions:

- **Learning from usage**: Incorporate click-through data for relevance feedback
- **Multilingual expansion**: Deploy language-specific embeddings for German support
- **Integration capabilities**: Expose API endpoints for embedding in other tools
- **Operational efficiency**: Enable incremental updates without restart

8.4.3 Research Opportunities

While maintaining focus on practical deployment, several research directions could enhance retrieval quality if simpler approaches prove insufficient:

- Learning-to-rank models leveraging user interaction data
- Cross-encoder architectures for precision reranking
- Domain-specific query expansion using banking terminology
- Privacy-preserving federated learning across institutions

8.5 Concluding Remarks

This thesis demonstrated that effective information retrieval for specialized domains requires neither cutting-edge models nor complex distributed architectures. By combining three well-understood retrieval signals through empirically tuned weights, standardizing scattered documentation, and implementing rigorous evaluation despite data constraints, we built a system that solves real problems within real constraints.

The key insight is that banking IT environments—with their stringent security requirements, audit demands, and deployment restrictions—benefit more from simple, auditable architectures than from sophisticated but opaque solutions. Our success validates the approach of applying proven techniques thoughtfully rather than pursuing algorithmic novelty.

As financial institutions manage growing regulatory complexity, tools for efficient rule discovery become critical infrastructure. This system provides a foundation for that capability, ready for production deployment at Deutsche Bank. Its successful adoption will validate our thesis that modern NLP capabilities can be delivered through pragmatic engineering, solving real problems for real users within the constraints that actually matter.

The hybrid retrieval system stands ready to transform how developers and QA teams interact with validation rules—reducing discovery time from hours to seconds while maintaining the reliability and auditability that banking demands. In choosing simplicity over sophistication, determinism over flexibility, and pragmatism over perfection, we built not just a research prototype but a production-ready tool that will deliver value from day one.

Bibliography

[1] Douglas W Arner, Janos Barberis, and Ross P Buckley. "FinTech, RegTech, and the Reconceptualization of Financial Regulation." In: *Northwestern Journal of International Law & Business* 37 (2017), pp. 371–428.

[2] Justus Bogner, Jonas Fritzsch, Stefan Wagner, and Alfred Zimmermann. "Microservices vs. monoliths: A performance and scalability perspective." In: *IEEE Software* 36.5 (2019), pp. 76–82.

[3] Chris Burges, Tal Shaked, Erin Renshaw, Ari Lazier, Matt Deeds, Nicole Hamilton, and Greg Hullender. "Learning to rank using gradient descent." In: *Proceedings of ICML 2005*. 2005, pp. 89–96.

[4] Olivier Chapelle, Donald Metlzer, Ya Zhang, and Pierre Grinspan. "Expected Reciprocal Rank for Graded Relevance." In: *Proceedings of CIKM 2009*. 2009, pp. 621–630.

[5] Adam Cohen. *FuzzyWuzzy: Fuzzy String Matching in Python*. `https://github.com/seatgeek/fuzzywuzzy`. 2011.

[6] William W Cohen, Pradeep Ravikumar, and Stephen E Fienberg. "A comparison of string distance metrics for name-matching tasks." In: *IIWeb Workshop* (2003), pp. 73–78.

[7] Gordon V Cormack, Charles LA Clarke, and Stefan Buettcher. "Reciprocal Rank Fusion Outperforms Condorcet and Individual Rank Learning Methods." In: *Proceedings of SIGIR 2009*. 2009, pp. 758–759.

[8] Matthijs Douze, Alexandr Guzhva, Chengqi Deng, Jeff Johnson, Gergely Szilvasy, Pierre-Emmanuel Mazaré, Maria Lomeli, Lucas Hosseini, and Hervé Jégou. "The Faiss library." In: *arXiv preprint arXiv:2401.08281* (2024).

[9] Xiaodong Gu, Hongyu Zhang, and Sunghun Kim. "Deep Code Search." In: *Proceedings of ICSE 2018*. 2018, pp. 933–944.

[10] Zellig S Harris. "Distributional structure." In: *Word* 10.2-3 (1954), pp. 146–162.

[11] Mohammad Hassan, Chen Zhang, and Yang Li. "Regulatory Compliance Automation: A Survey." In: *ACM Computing Surveys* 54.7 (2022), pp. 1–36.

[12] David C Hay and Keri Anderson Healy. *Defining business rules: What are they really?* Business Rules Group, 2006.

[13] Hamel Husain, Ho-Hsiang Wu, Tiferet Gazit, Miltiadis Allamanis, and Marc Brockschmidt. "CodeSearchNet Challenge: Evaluating the State of Semantic Code Search." In: *arXiv preprint arXiv:1909.09436*. 2019.

[14] IBM Corporation. *IBM OpenPages with Watson: Integrated GRC platform*. Tech. rep. IBM, 2022.

[15] Jeff Johnson, Matthijs Douze, and Hervé Jégou. "Billion-scale similarity search with GPUs." In: *IEEE Transactions on Big Data* 7.3 (2019), pp. 535–547.

[16] Vladimir Karpukhin, Barlas Oğuz, Sewon Min, Patrick Lewis, Ledell Wu, Sergey Edunov, Danqi Chen, and Wen-tau Yih. "Dense Passage Retrieval for Open-Domain Question Answering." In: *Proceedings of EMNLP 2020*. 2020, pp. 6769–6781.

[17] Omar Khattab and Matei Zaharia. "ColBERT: Efficient and effective passage search via contextualized late interaction over BERT." In: *Proceedings of SIGIR 2020*. 2020, pp. 39–48.

[18] Young-Min Kim, Bhaskar Mitra, and Jong-Hyeok Lee. "Structured retrieval using metadata fields." In: *Information Processing & Management* 56.4 (2019), pp. 1432–1447.

[19] Ron Kohavi. "A Study of Cross-Validation and Bootstrap for Accuracy Estimation and Model Selection." In: *Proceedings of IJCAI 1995*. 1995, pp. 1137–1145.

[20] Saar Kuzi, Mingyang Zhang, Cheng Li, Michael Bendersky, and Marc Najork. "Leveraging Semantic and Lexical Matching to Improve the Recall of Document Retrieval Systems." In: *ACM Transactions on Information Systems* 38.4 (2020), pp. 1–30.

[21] Vladimir I. Levenshtein. "Binary Codes Capable of Correcting Deletions, Insertions, and Reversals." In: *Soviet Physics Doklady* 10.8 (1966), pp. 707–710.

[22] Patrick Lewis, Ethan Perez, Aleksandra Piktus, Fabio Petroni, Vladimir Karpukhin, Naman Goyal, Heinrich Küttler, Mike Lewis, Wen-tau Yih, Tim Rocktäschel, Sebastian Riedel, and Douwe Kiela. "Retrieval-Augmented Generation for Knowledge-Intensive NLP." In: *Advances in Neural Information Processing Systems*. 2020.

[23] Xianming Li and Jing Li. "UAE-Large-V1: Universal AnglE Embedding Model." In: *arXiv preprint arXiv:2309.12871* (2023).

[24] Jimmy Lin and Xueguang Ma. "A few brief notes on pretrained language models and information retrieval." In: *Proceedings of SIGIR 2021*. 2021, pp. 2482–2488.

[25] Tie-Yan Liu. "Learning to rank for information retrieval." In: *Foundations and Trends in Information Retrieval* 3.3 (2009), pp. 225–331.

[26] Gonzalo Navarro. "A Guided Tour to Approximate String Matching." In: *ACM Computing Surveys* 33.1 (2001), pp. 31–88.

[27] Sam Newman. *Building Microservices: Designing Fine-Grained Systems*. O'Reilly Media, 2015.

[28] Fabian Pedregosa, Gaël Varoquaux, Alexandre Gramfort, Vincent Michel, Bertrand Thirion, Olivier Grisel, Mathieu Blondel, Peter Prettenhofer, Ron Weiss, Vincent Dubourg, et al. "Scikit-learn: Machine Learning in Python." In: *Journal of Machine Learning Research* 12 (2011), pp. 2825–2830.

[29] Zhen Qin, Le Yan, Honglei Zhuang, Yi Tay, Rama Kumar Pasumarthi, Xuanhui Wang, Michael Bendersky, and Marc Najork. "Are neural rankers still outperformed by gradient boosted decision trees?" In: *arXiv preprint arXiv:2101.10914* (2021).

[30] Nils Reimers and Iryna Gurevych. "Sentence-BERT: Sentence Embeddings using Siamese BERT-Networks." In: *Proceedings of EMNLP-IJCNLP 2019*. 2019, pp. 3982–3992.

[31] Stephen E Robertson and Karen Spärck Jones. "Relevance weighting of search terms." In: *Journal of the American Society for Information Science* 27.3 (1976), pp. 129–146.

[32] Stephen E Robertson, Steve Walker, Susan Jones, Micheline M Hancock-Beaulieu, and Mike Gatford. "Okapi at TREC-3." In: *Proceedings of the Third Text REtrieval Conference (TREC-3)*. 1994, pp. 109–126.

[33] Stephen E. Robertson and Hugo Zaragoza. "The Probabilistic Relevance Framework: BM25 and Beyond." In: *Foundations and Trends in Information Retrieval* 3.4 (2009), pp. 333–389.

[34] Yi Tay, Vinh Q Tran, Mostafa Dehghani, Jianmo Ni, Dara Bahri, Harsh Mehta, Zhen Qin, Kai Hui, Zhe Zhao, Jai Gupta, et al. "Transformer memory as a differentiable search index." In: *arXiv preprint arXiv:2202.06991* (2022).

[35] Thomson Reuters. *Thomson Reuters regulatory intelligence*. Tech. rep. Thomson Reuters, 2022.

[36] Andrew Trotman, Antti Puurula, and Blake Burgess. "Improvements to BM25 and language models examined." In: *Information Retrieval* 17.3 (2014), pp. 238–252.

[37] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. "Attention Is All You Need." In: *Advances in Neural Information Processing Systems*. Vol. 30. 2017.

[38] Mario Villamizar, Oscar Garcés, Harold Castro, Mauricio Verano, Lorena Salamanca, Rubby Casallas, and Santiago Gil. "Evaluating the monolithic and the microservice architecture pattern to deploy web applications in the cloud." In: *Proceedings of CCC 2015*. 2015, pp. 583–590.

[39] Ellen M Voorhees. "The TREC-8 Question Answering Track Report." In: *Proceedings of TREC* 8 (1999), pp. 77–82.

[40] Liang Wang, Nan Yang, and Furu Wei. "Improving hybrid retrieval via score normalization." In: *arXiv preprint arXiv:2304.09985* (2023).

[41] Tzu-Tsung Wong. "Performance Evaluation of Classification Algorithms by k-fold and Leave-One-Out Cross Validation." In: *Pattern Recognition* 48.9 (2015), pp. 2839–2846.

[42] Wei Zhang, Lu Chen, and Jin Wang. "Regulatory compliance through BERT-based transaction matching." In: *Proceedings of FinNLP 2020*. 2020, pp. 89–97.

[43] Ming Zhong, Yang Liu, Da Xu, Jing Huang, and Jian Sun. "Does domain-specific pretraining help regulatory compliance?" In: *arXiv preprint arXiv:2106.09953* (2021).

List of Figures

1.1 Monolithic RAG architecture. All components run in-process within a single Dash application, ensuring deterministic retrieval with sub-second latency. 3

5.1 Startup initialization pipeline with parallel index construction reducing total time to 464ms. 30
5.2 Three-signal retrieval architecture with empirically tuned weights from LOOCV. 31
5.3 Memory allocation breakdown showing 530MB total footprint. 32
5.4 Component-driven UI architecture. The Layout Component unifies Search and Generator components with shared result display. External LLM integration for chat generation remains outside scope. 33

6.1 Search interface with query input, mode toggle (Keyword/Hybrid), and filter panel. 46
6.2 Multi-select filtering across Rule Type, Country, Business Type, and Party Agent. 46
6.3 Ranked rule cards with similarity scores and categorical badges. 47
6.4 Detailed rule view with code, descriptions, and metadata. 47
6.5 Combined search and conversational exploration interface. 48

7.1 Performance comparison of MRR@5 across methods. LOOCV represents held-out performance, while others show resubstitution results. 52
7.2 Sensitivity of MRR@5 to weight perturbations. The optimized configuration shows good stability within ± 0.1 range. 53

List of Tables

2.1 Comparison of approaches across operational dimensions critical for banking deployment. 12

4.1 Distribution of corpus fields across functional categories. 22
4.2 Field completeness showing perfect coverage in all search-critical fields. . 22
4.3 Categorical tag dimensions showing coverage gaps and excessive fragmentation. 23
4.4 Text field statistics. LLM descriptions provide 4× more content than basic descriptions. 24
4.5 Top 10 keywords revealing domain focus on payment validation and party verification. 25
4.6 Pairwise cosine similarity statistics showing good discriminative range. . 26
4.7 Quality score breakdown by category showing tag standardization as the primary gap. 27

5.1 Key design decisions and their justifications. 36

7.1 Individual signal and baseline hybrid performance. *Baseline uses initial production weights: $(w_s, w_b, w_f) = (0.60, 0.35, 0.05)$. 51
7.2 LOOCV-optimized hybrid performance on held-out queries. 51
7.3 Ablation study showing signal contributions. Performance measured on full dataset with median LOOCV weights. 52
7.4 System performance validation. All metrics meet production requirements. 53

A.1 System configuration with LOOCV-optimized parameters. 69
A.2 Essential fields in the standardized corpus schema. 70
A.3 System performance against design targets. 72
A.4 Common operational issues and recommended solutions. 73

Listings

6.1 Project directory structure 39
6.2 Attention-aware mean pooling for embeddings 40
6.3 Semantic search via cosine similarity 41
6.4 Parallel index construction at startup 41
6.5 Three retrieval signals implementation 42
6.6 Hybrid score computation with normalization 43
6.7 Main search pipeline 44

A.1 SQLite optimization pragmas 69
A.2 Search API specification 70
A.3 Quick start 71
A.4 Production configuration 71
A.5 Embedding generation 73

List of Abbreviations

API Application Programming Interface

BLAS Basic Linear Algebra Subprograms

BM25 Best Matching 25

CSV Comma-Separated Values

DB Database

ETL Extract, Transform, LoadGraphics Processing Unit

IDF Inverse Document Frequency

IR Information Retrieval

ISO International Organization for Standardization

JSON JavaScript Object Notation

LLM Large Language Model

LOOCV Leave-One-Out Cross-Validation

MRR Mean Reciprocal Rank

nDCG Normalized Discounted Cumulative Gain

NLP Natural Language Processing

P@k Precision at k

QA Quality Assurance

RAG Retrieval-Augmented Generation

SEPA Single Euro Payments Area

SQL Structured Query Language

SQLite Structured Query Language Lite

SWIFT Society for Worldwide Interbank Financial Telecommunication

TF Term Frequency

TF-IDF Term Frequency-Inverse Document Frequency

TREC Text REtrieval Conference

UAE Universal AnglE

UI User Interface

WAL Write-Ahead Logging

Note on Domain-Specific Terms:

- **BANSTA**: Banking-specific error code system used in payment validation
- **Dash**: Python framework for building analytical web applications
- **Gemini-2.5-Pro**: Google's large language model used for offline enrichment

Appendices

A System Configuration and Implementation Details

This appendix documents the technical specifications of our hybrid retrieval system, providing essential details for deployment, configuration, and performance optimization.

A.1 Configuration Parameters

A.1.1 Core System Configuration

The retrieval system operates with empirically tuned parameters discovered through Leave-One-Out Cross-Validation on our evaluation dataset.

Parameter	Value	Purpose
Retrieval Weights		
`SEMANTIC_WEIGHT`	0.80	Semantic similarity contribution
`BM25_WEIGHT`	0.10	Keyword matching contribution
`FUZZY_WEIGHT`	0.10	String similarity contribution
Operational Settings		
`MIN_SIMILARITY`	0.30	Relevance threshold (τ)
`DEFAULT_SEARCH_LIMIT`	10	Results per query
`ENABLE_RERANKING`	False	Metadata reranking
Model Configuration		
`EMBEDDING_MODEL`	UAE-Large-V1	Transformer model

Table A.1: System configuration with LOOCV-optimized parameters.

A.1.2 Database Optimization

SQLite configuration for read-heavy workloads:

```
PRAGMA journal_mode = WAL;      -- Enable concurrent reads
PRAGMA synchronous = NORMAL;    -- Balance durability/speed
PRAGMA cache_size = -64000;     -- 64MB page cache
PRAGMA busy_timeout = 5000;     -- 5s timeout for locks
```

Listing A.1: SQLite optimization pragmas

A.2 Data Schema

A.2.1 Rule Representation

Each validation rule comprises 15 fields organized into functional groups:

Field	Category	Purpose
`rule_name`	Identifier	Primary human-readable key
`rule_code`	Implementation	Kotlin validation logic
`llm_description`	Search	Enhanced semantic content
`keywords`	Search	BM25 retrieval terms
`embedding`	Search	1024-d semantic vector
`rule_type`	Metadata	Categorical classification
`country`	Metadata	Geographic scope
`business_type`	Metadata	Business domain
`party_agent`	Metadata	Party relationships
`bansta_error_code`	Reference	Banking standard code
`iso_error_code`	Reference	ISO 20022 code

Table A.2: Essential fields in the standardized corpus schema.

A.3 API Reference

A.3.1 Primary Search Interface

```
def search_rules(
    query: Optional[str] = None,
    rule_type: Optional[List[str]] = None,
    country: Optional[List[str]] = None,
    business_type: Optional[List[str]] = None,
    party_agent: Optional[List[str]] = None,
    mode: SearchMode = SearchMode.HYBRID,
    top_k: int = 10
) -> List[Dict[str, Any]]:
    """
    Search validation rules with optional filters.

    Returns:
        List of rules with 'search_score' field added.
        Rules are sorted by relevance (highest first).
    """
```

Listing A.2: Search API specification

A.3.2 Search Modes

The system supports four retrieval strategies:

- `HYBRID`: Weighted combination of all signals (default)
- `SEMANTIC`: Embedding similarity only
- `KEYWORD`: BM25 retrieval only
- `FUZZY`: String matching only

A.4 Deployment Guide

A.4.1 Development Environment

```
# Setup
python -m venv env
source env/bin/activate
pip install -r requirements.txt

# Launch
python app.py
# Access at http://localhost:8050
```

Listing A.3: Quick start

A.4.2 Production Deployment

For production use, deploy with gunicorn behind a reverse proxy:

```
gunicorn app:server \
    --workers 4 \
    --worker-class sync \
    --bind 0.0.0.0:8050 \
    --timeout 120 \
    --preload
```

Listing A.4: Production configuration

The `-preload` flag ensures indices are built once and shared across workers, reducing memory usage by 75%.

Metric	Value	Target
System Resources		
Memory footprint	530 MB	< 1 GB
Startup time	464 ms	< 1 s
Index build time	252 ms	< 500 ms
Query Performance		
Median latency	58 ms	< 100 ms
P95 latency	660 ms	< 1000 ms
P99 latency	2.1 s	< 3 s
Retrieval Quality		
MRR@5	48.2%	> 40%
Hit@5	80.8%	> 75%

Table A.3: System performance against design targets.

A.5 Performance Characteristics

A.5.1 Operational Metrics

A.5.2 Latency Breakdown

Query processing time varies by search mode and corpus filtering:

- **Keyword mode**: 1.2ms (BM25 scoring only)
- **Semantic mode**: 2.4ms (cosine similarity)
- **Fuzzy mode**: 19.4ms (string matching)
- **Hybrid mode**: 58ms median (all signals combined)

Applying categorical filters reduces the candidate set, improving performance proportionally. With aggressive filtering (e.g., single country), hybrid search completes in under 10ms.

A.6 Corpus Preparation

A.6.1 Data Pipeline

The standardization pipeline transforms raw rule data through seven stages:

1. **Consolidation**: Merge three source CSVs (1,743 rules)
2. **Deduplication**: Remove duplicates by rule name (586 removed)
3. **Enhancement**: Generate descriptions via Gemini-2.5-Pro
4. **Extraction**: Derive keywords and categorical tags

5. **Embedding**: Compute UAE-Large-V1 vectors
6. **Normalization**: L2-normalize for cosine similarity
7. **Serialization**: Export as single CSV (32.21 MB)

A.6.2 Embedding Generation

Embeddings are generated offline using mean pooling over token representations:

```
# Load model
model = AutoModel.from_pretrained("UAE-Large-V1")
tokenizer = AutoTokenizer.from_pretrained("UAE-Large-V1")

# Generate embeddings
with torch.no_grad():
    inputs = tokenizer(text, return_tensors='pt',
                       truncation=True, max_length=512)
    outputs = model(**inputs)

    # Mean pooling
    embeddings = outputs.last_hidden_state.mean(dim=1)

    # L2 normalization
    embeddings = F.normalize(embeddings, p=2, dim=1)
```

Listing A.5: Embedding generation

A.7 Troubleshooting

Common issues and solutions:

Issue	Resolution
High memory usage	Reduce workers; enable `-preload`
Slow startup	Pre-serialize indices to pickle files
Database locks	Ensure WAL mode; increase timeout
Poor recall	Verify embedding normalization
Inconsistent scores	Check per-query normalization

Table A.4: Common operational issues and recommended solutions.

A.8 Summary

This appendix provides the technical foundation for deploying and maintaining the hybrid retrieval system. The configuration parameters, optimized through empirical eval-

uation, balance retrieval quality with performance constraints. The modular API design enables integration with existing systems while maintaining the simplicity of our monolithic architecture. Organizations implementing similar systems should adjust parameters based on their specific corpus characteristics and performance requirements.

Colophon

This document has been created with the theses LaTeXtemplate provided by htw saar, Saarland University of Applied Sciences, Informatics/Mechatronics and Sensor Technology (Version 1.0, 01.09.2025). This template has been created by Yves Hary and André Miede (with kind support from Thomas Kretschmer, Helmut G. Folz and Martina Lehser). Data: (F)10.95 – (B)426.79135pt – (H)688.5567pt