NAZ Order Papers System: A Questions and Motions Management System with Hybrid AI-Assisted Duplicate and Similarity Detection for the National Assembly of Zambia

By

Bruce Mainza (2022038479)

Dalitso Zulu (2022026047)

Maliseni Chavula (2022056205)

Supervisors:

Dr Lighton Phiri[1]

Mr Alinani Sichone[2]

Mr Paul Chola[2]

Mr David Mulenga[2]

Dr Mwaula Solopi[2]

A report submitted to the Department of Computing and Informatics, The University of Zambia, in partial fulfilment of the requirements of the degree of Bachelor of Science in Computer Science (Software Engineering)

THE UNIVERSITY OF ZAMBIA

LUSAKA

2026

[1] Department of Computing and Informatics School of Natural and Applied Sciences, University of Zambia
[2] Information and Communications Technology, National Assembly of Zambia

DECLARATION

I declare that this final year project report is my own work and has been prepared in accordance with the academic requirements of the University of Zambia. All sources of information, ideas, data, software, images, and other material that are not my own have been appropriately acknowledged and referenced. Where this project was undertaken as part of a group, I have clearly identified my individual contributions and have not presented the work of other group members as my own

Student Name(s):.....

Computer Number: ……

Signature: Date: ……..

Student Name(s):.....

Computer Number: ……

Signature: Date: ……..

Student Name(s):.....

Computer Number: ……

Signature: Date: ……..

APPROVAL

This final year project report has been submitted for examination with the approval of the project supervisor.

Supervisor Name:Dr Lighton Phiri

Signature:

Date:23rd October, 2026

Departmental/Examiner Approval (if applicable):

Abstract

The National Assembly of Zambia (NAZ) handles parliamentary questions and notices of motion through largely manual procedures. Clerks check each new submission against earlier records by hand to decide whether the matter is a duplicate or has already been addressed, and then place approved items on the Order Paper. The process is slow and difficult to audit, and it misses repeated matters that are worded differently. This project designed, built and evaluated the NAZ Order Papers System, a web application that digitises the full lifecycle of a question or motion, from submission and clerk review to scheduling, Order Paper generation and archiving. It adds locally hosted artificial intelligence that helps clerks detect duplicate and previously addressed matters. Following a design science approach and an iterative, test-driven development process, the team implemented a Next.js interface, a FastAPI service and a PostgreSQL 16 database with the pgvector extension. Access is controlled by fifteen permissions grouped into configurable roles, and a nine-state lifecycle governs every record. Retrieval combines weighted PostgreSQL full-text search with 768-dimensional EmbeddingGemma vectors through Reciprocal Rank Fusion, run entirely on locally hosted infrastructure through Ollama, and a clerk always makes the final decision. This hybrid lexical-semantic retrieval, used for record search and for two-stage exact and near-duplicate detection, is the project's delivered artificial intelligence contribution; a further generative, retrieval-augmented explanation layer is scaffolded in the codebase but is deliberately treated as future work and is not evaluated in this report. All 224 backend tests and 21 frontend source tests pass. Lexical search over a 100,000-record corpus returned 20 ranked results with a 95th-percentile latency of 266.5 ms, against a 3,000 ms budget. In a controlled evaluation with reworded queries, hybrid retrieval ranked the correct earlier record higher than keyword search alone (Chapter 7). The results show that a locally deployed, human-supervised hybrid retrieval system is a feasible way to modernise Order Paper preparation at NAZ. Formal user acceptance testing, calibration of the similarity thresholds on real NAZ records, and the future generative explanation phase remain for subsequent work.

Keywords: parliamentary information systems; Order Paper; duplicate detection; hybrid retrieval; reciprocal rank fusion; pgvector; local embedding models; FastAPI

Acknowledgements

First and foremost, we thank God for the opportunity, strength and wisdom to undertake and complete this project.

We are sincerely grateful to our supervisors, Dr Lighton Phiri, Mr Alinani Sichone, Mr Paul Chola, Mr David Mulenga and Dr Mwaula Solopi for their guidance, encouragement and technical insight throughout the project.

We are deeply grateful to the National Assembly of Zambia, in particular the ICT Director, Dr Solopi, and the staff of the ICT Directorate and the Table Office. They shared their time and expertise, gave us a prioritised list of requirements, and gave feedback that shaped the system.

We also thank the Department of Computing and Informatics at the University of Zambia for the academic environment and resources that made this work possible, and our families for their prayers and support.

Table of Contents

DECLARATION 1
APPROVAL 2
Abstract 3
Acknowledgements 4
Table of Contents 5
List of Tables 8
List of Figures 9
List of Acronyms and Abbreviations 10
1. Introduction 11
1.1. Study Background 12
1.2. Problem Statement 13
1.3. Study Objectives 13
1.3.1. Broad Objective 13
1.3.2. Specific Objectives 13
1.3.3. Research Questions 14
1.4. Study Rationale 14
1.5. Scope and Limitations 15
1.5.1. Specific Scope 15
1.5.2. Limitations 15
1.5.3. Addressing Limitations 15
1.6. Ethical Considerations 16
1.7. Report Organisation 16
2. Literature Review 17
2.1. Parliamentary Questions, Motions and the Order Paper 17
2.2. Digital Parliaments and Existing Systems 17
2.3. Lexical Information Retrieval 18
2.4. Near-Duplicate Detection 19
2.5. Dense Semantic Retrieval 19
2.6. Vector Storage and Approximate Nearest-Neighbour Search 20
2.7. Hybrid Retrieval and Rank Fusion 20
2.8. Domain Adaptation of Embedding Models 20
2.9. Summary and Research Gap 21
3. Methodology 22
3.1. Introduction 22
3.2. Research Approach 22
3.3. Development Methodology 23
3.3.1. Agile Scrum Development 23
3.3.2. Test-Driven Development 23
3.4. Requirements Elicitation 25
3.5. Design Principles 25
3.6. Tools and Technologies 26
3.7. Data Strategy 27
3.8. Project Management and Version Control 28
3.9. Evaluation Design 28
4. System Analysis and Design 29
4.1. Analysis of the Current Process 29
4.2. Actors and Permissions 29

4.3. Functional Requirements 31
4.4. Non-Functional Requirements 32
4.6. Business Process Model 34
4.7. System Architecture 34
4.8. Data Design 36
4.9. Submission Lifecycle Design 39
4.10. Interface (API) Design 40
4.11. Security Design 40
4.12. Deployment Design 41
5. Implementation 43
5.1. Codebase Structure 43
5.2. Backend Application Framework 44
5.3. Authentication and Authorisation 44
5.4. Submission and Review Workflow 46
5.4.1. Creating a Submission 46
5.4.2. Lifecycle Enforcement 47
5.4.3. Scheduling, Order Paper, Responses and Archiving 48
5.5. Duplicate and Previously-Addressed Detection 48
5.6. Hybrid Retrieval 51
5.6.1. Lexical Retrieval 51
5.6.2. Chunking and Semantic Indexing 51
5.6.3. Semantic Retrieval 52
5.6.4. Reciprocal Rank Fusion 52
5.7. Background Processing and Operational Readiness 53
5.8. Document and Data Ingestion 54
5.9. Frontend Implementation 54
5.10. Deployment and Operations 55
5.11. Design Evolution 55
5.12. Implementation Challenges and Solutions 56
6. Testing 58
6.1. Testing Strategy 58
6.2. Test Environment 58
6.3. Backend Test Suite 59
6.4. Representative Test Cases 61
6.5. Frontend and Browser Tests 63
6.6. Non-Functional Testing 64
6.7. Summary of Results 65
6.8. Testing Limitations 65
7.1. Functional Completeness Against Requirements 65
7.2. Retrieval Effectiveness 66
7.2.1. Method 66
7.2.2. Results 66
7.3. Duplicate and Previously-Addressed Threshold Analysis 68
7.4. Performance Benchmark 69
7.5. Usability Evaluation 70
7.6. Self-Critique and Honest Assessment 71
7.7. Comparison with the Original Project Proposal 73
8. Conclusion and Recommendations 76
8.1. Conclusion 76
8.3. Future Work 77
8.3.1. Completing the Generative Explanation (RAG) Phase 77

8.3.2. Other Future Work 78
9. References **80**
APPENDICES **85**
APPENDIX A: INDIVIDUAL CONTRIBUTION STATEMENT (GROUP PROJECTS) 86
APPENDIX B: USER MANUAL / INSTALLATION GUIDE 89
APPENDIX C: ADDITIONAL SYSTEM DESIGNS / DIAGRAMS 92
APPENDIX D: DATA COLLECTION INSTRUMENTS 95

List of Tables

Table 2.1: Comparison of existing parliamentary systems with the NAZ Order Papers System
Table 3.1: Tools and technologies used in the project
Table 4.1: Default role–permission matrix
Table 4.2: Functional requirements and implementation status
Table 4.3: Non-functional requirements and design measures
Table 4.4: Primary Use Case Descriptions Summary
Table 4.5: Main architectural decisions and alternatives considered
Table 4.6: Summary of the main API resources
Table 5.1: Repository structure
Table 5.2: Implementation challenges and solutions
Table 6.1: Backend test suite by area (224 tests, all passing)
Table 6.2: Representative test cases and results
Table 6.3: Summary of verification results
Table 7.1: Retrieval effectiveness by mode, 18 paraphrased queries against the 24-record corpus
Table 7.2: Cosine similarity of paraphrased queries to their true target versus the best non-target match
Table 7.3: Lexical search benchmark, 100,000-record corpus, 20 results per query
Table 7.4: The delivered system compared against the March 2026 project proposal
Table B.1: Demo login accounts for the installation guide

List of Figures

Figure 3.1: The iterative test-driven development cycle applied to each prioritised requirement
Figure 4.1: Use case diagram for the Clerk and Administrator
Figure 4.2: System architecture (component view)
Figure 4.3: Entity-relationship diagram: identity, access control and parliamentary business
Figure 4.4: Entity-relationship diagram: retrieval, reliability and audit
Figure 4.5: Submission lifecycle state machine
Figure 4.6: Production deployment diagram for the NAZ handover
Figure 5.1: Sequence diagram: login, protected navigation, API calls, logout and unlock
Figure 5.2: Sequence diagram: creating a question or notice of motion
Figure 5.3: submission screen
Figure 5.4: Sequence diagram: duplicate and previously-addressed detection
Figure 5.5: Duplicate detection on the front end
Figure 5.6: Sequence diagram: hybrid retrieval with lexical fallback
Figure 5.7: State machine of a durable background job
Figure 6.1: Verification results by test suite
Figure 7.1: Retrieval effectiveness (Recall@1 and MRR@10) by retrieval mode
Figure 7.2: Retrieval latency (median and maximum) by retrieval mode
Figure 7.3: Lexical search benchmark latency against the 3,000 ms budget
Figure 7.4: SUS results, distribution of responses to each statement
Figure C.1: Conceptual ERD Diagram
Figure C.2: Hybrid Retrieval Class Diagram
Figure C.3: Duplicate Package Class Diagram
Figure D.1: Google Form used to record SUS responses (first screenshot)
Figure D.2: Google Form used to record SUS responses (second screenshot)

List of Acronyms and Abbreviations

Abbreviation	Description
ABC	Abstract Base Class (Python)
AI	Artificial Intelligence
ANN	Approximate Nearest Neighbour
API	Application Programming Interface
BFF	Backend-for-Frontend
BM25	Best Matching 25 (probabilistic ranking function)
CI	Continuous Integration
CRUD	Create, Read, Update and Delete
CSV	Comma-Separated Values
DOCX	Office Open XML Word document
DSR	Design Science Research
EC2	Amazon Elastic Compute Cloud
ER	Entity-Relationship
FR	Functional Requirement
FTS	Full-Text Search
GIN	Generalised Inverted Index (PostgreSQL)
HNSW	Hierarchical Navigable Small World
HTTP(S)	Hypertext Transfer Protocol (Secure)
IPU	Inter-Parliamentary Union
IVFFlat	Inverted File index with flat (uncompressed) vectors
JSON	JavaScript Object Notation
JWT	JSON Web Token
LLM	Large Language Model
LoRA	Low-Rank Adaptation
MRR	Mean Reciprocal Rank
NAZ	National Assembly of Zambia
NFR	Non-Functional Requirement
ORM	Object-Relational Mapping
PDF	Portable Document Format
RAG	Retrieval-Augmented Generation (planned future work; not implemented in this report)
RBAC	Role-Based Access Control
REST	Representational State Transfer
RQ	Research Question
RRF	Reciprocal Rank Fusion
SOLID	Single responsibility,Open-closed,Liskov substitution, Interface segregation, Dependency inversion
SQL	Structured Query Language
SUS	System Usability Scale
TDD	Test-Driven Development
TLS	Transport Layer Security
UAT	User Acceptance Testing
UML	Unified Modelling Language

1. Introduction

The National Assembly of Zambia (NAZ) is the country's legislature. Under the Constitution it makes laws, oversees the Executive and approves public expenditure [1]. Two of the most direct oversight tools available to Members of Parliament (MPs) are the parliamentary question, which asks a Minister for information or action, and the notice of motion, which puts a proposition before the House for debate [1]. Admitted questions and motions reach the floor of the House through the Order Paper, the official agenda that the Journals and Table Office prepares for each sitting and that NAZ publishes [3].

Before an item is placed on the Order Paper, staff must check that it follows procedure and that it does not repeat a matter already before the House or already answered in an earlier session. At NAZ this check is done by hand. Clerks read through earlier Order Papers and records, interpret differences in wording, and decide whether a new item is new, a duplicate, or substantially similar to an earlier one. The work is repetitive and slow, and two items can be missed as related when they describe the same problem in different words.

This report presents the design, implementation and evaluation of the **NAZ Order Papers System**, developed by Team CMZ for handover to NAZ. The system covers the whole journey of a question or motion: submission, validation, clerk review, scheduling for a sitting, generation of a structured Order Paper, recording of the response, and archiving. On top of this workflow it adds a locally hosted artificial intelligence (AI) capability: **hybrid lexical-semantic retrieval**, combining PostgreSQL full-text search with locally generated embeddings, used both for general record search and for two-stage exact and near-duplicate detection. All model inference runs on NAZ-controlled infrastructure through Ollama, and every procedural decision stays with a human clerk. A further generative explanation capability, in which a locally hosted language model would summarise why retrieved records are related (a retrieval-augmented generation, or RAG, pattern), is scaffolded in the codebase as an early prototype but is deliberately treated as future work throughout this report; it is not evaluated here and is discussed only in Section 8.3.

The rest of this chapter sets out the background (Section 1.1), the problem (Section 1.2), the objectives and research questions (Section 1.3), the rationale (Section 1.4), the scope and limitations (Section 1.5), the ethical considerations (Section 1.6) and the organisation of the report (Section 1.7).

1.1. Study Background

Parliamentary questions let MPs obtain information from the Government and hold Ministers to account for matters affecting their constituencies. Questions may be for oral answer on the floor of the House or for written answer[1] . Private members' motions let MPs propose that the House urge the Government to take a particular course of action. Both kinds of business are governed by the Standing Orders and are published on the Order Paper for the sitting at which they will be considered [3].

Parliaments worldwide are digitising these processes. The Inter-Parliamentary Union (IPU) reports that parliaments are investing in document management, open data and digital tools that give members and staff faster access to parliamentary information [4], [5]. Mature examples include the UK Parliament's written questions service, which publishes every question and answer as searchable structured data [6], and the Bungeni parliamentary information system developed for African legislatures under the Africa i-Parliaments Action Plan [7].

At NAZ, submissions still move largely on paper. A question or motion is written out, received and recorded by staff, checked against the rules and against earlier business, and then scheduled. The consultations held with the NAZ ICT Directorate and the Table Office (Section 3.4) identified four recurring difficulties:

- **Volume and traceability.** Each session produces many submissions, and without a central register it is difficult to know which items are pending, approved, scheduled or answered, or how long each stage has taken.
- **Duplicate and repeated matters.** The Standing Orders expect that the same matter is not raised repeatedly [2]. Checking this means searching earlier records by hand, which is slow and error-prone.
- **Paraphrase.** Different members describe the same issue in different words (for example, "shortage of teachers" and "too few educators"). Exact-word searching cannot reliably connect such items.
- **Accountability.** Decisions made on paper leave a weak audit trail, which makes it difficult to show who reviewed, approved or scheduled an item, and when.

Recent advances in text embeddings [8], [9] and hybrid retrieval that fuses lexical and semantic ranking [10], [11] make it practical to search by meaning rather than exact words. Because parliamentary material can be sensitive and Zambia's Data Protection Act regulates

personal data [12], such capability should run inside the institution's own infrastructure rather than on external AI services. Open-weight embedding models served locally through Ollama allow this [13], [14].

1.2. Problem Statement

The National Assembly of Zambia manages parliamentary questions and private members' motions through manual procedures based on paper submissions, physical registers and human verification. These procedures make it difficult to track submissions through their lifecycle, to detect duplicate submissions within a session, and to establish whether a matter has already been addressed in an earlier session, especially when the new item is worded differently. As a result, the process is slow, labour-intensive, weakly auditable and prone to oversight, which affects procedural compliance, record management and retrieval efficiency.

NAZ therefore needs a centralised, secure system that supports the submission, validation, review, scheduling, Order Paper preparation, tracking and archiving of questions and motions. The system must also give clerks reliable hybrid retrieval that helps them detect duplicate and previously addressed matters, without sending parliamentary data outside the institution and without removing the final decision from a human officer.

1.3. Study Objectives

1.3.1. Broad Objective

To design, implement and evaluate a secure, web-based system that digitises the lifecycle of parliamentary questions and notices of motion at the National Assembly of Zambia and uses locally hosted hybrid retrieval to help clerks detect duplicate and previously addressed matters.

1.3.2. Specific Objectives

1) To analyse the existing Order Paper verification workflow and identify the data sources, business rules and user requirements for question and motion submission.
2) To design and implement an end-to-end workflow covering submission, clerk review, scheduling, Order Paper generation, response recording and archiving, secured by permission-based access control and a complete audit trail.
3) To design and implement an ingestion and indexing pipeline for parliamentary records, covering text extraction, normalisation, chunking, embedding and full-text indexing.

4) To integrate a locally hosted embedding model through Ollama into a hybrid lexical-semantic retrieval pipeline, fused by Reciprocal Rank Fusion, that flags duplicate and previously addressed matters for human review, and to establish what groundwork this lays for a future generative explanation phase.

5) To verify and evaluate the system in terms of functional correctness, retrieval accuracy, response time and readiness for handover to NAZ.

1.3.3. Research Questions

1) What workflows, rules and data does NAZ use to accept, verify and schedule questions and motions, and which requirements follow from them?

2) How can the submission-to-archive lifecycle be implemented so that every state change is valid, authorised and auditable?

3) To what extent does hybrid lexical and semantic retrieval with a local embedding model find earlier related records that keyword search alone misses?

4) What architecture and data groundwork does the delivered hybrid retrieval system need to put in place now, so that a future generative, retrieval-augmented explanation layer can be added later without redesign?

1.4. Study Rationale

The project matters in four ways.

- **Institutional efficiency.** Moving submissions, reviews and scheduling into one system removes repeated paper handling, gives the Table Office a live view of pending business, and lets a clerk generate the items for a sitting in one step.
- **Procedural integrity.** An explicit state machine, permission checks and an audit trail make every decision traceable to an authorised officer. This supports the accountability NAZ expects of its own processes.
- **Better detection of repeated matters.** Hybrid retrieval helps clerks find related earlier questions and motions even when they are worded differently, reducing the chance that a repeated matter goes unnoticed.
- **Sovereign AI.** Running open embedding models on NAZ infrastructure keeps records within the institution, in line with the Data Protection Act [12].

For the University, the project applies software engineering practice (requirements engineering, modular design, test-driven development and containerised deployment) to a real

public-sector problem, and it produces a reusable reference design for AI-assisted parliamentary information retrieval in the region, with an explicit, honestly scoped path towards a future generative phase.

1.5. Scope and Limitations

This study designed, implemented and evaluated a working system for question and motion management with hybrid AI-assisted similarity detection, to be handed over to NAZ. The following subsections define the boundaries of the work.

1.5.1. Specific Scope

- **Geographical scope:** The system was developed for the National Assembly of Zambia in Lusaka. Requirements came from the NAZ ICT Directorate and Table Office.
- **Temporal scope:** The project ran from April 2026 (the repository was initialised on 28 April 2026) to September 2026.
- **Topical scope:** Submission of questions (oral and written) and notices of motion; clerk review; hybrid-retrieval-assisted similarity and duplicate review; scheduling; Order Paper generation as structured data; response recording; archiving; search; reporting; and audit.
- **Technical scope:** A Next.js web client with a Backend-for-Frontend (BFF), a FastAPI service, PostgreSQL 16 with pgvector, a durable background worker, and Ollama serving the embeddinggemma:300m embedding model for hybrid retrieval. The codebase additionally provisions a durable job type and a database audit table for a future generative-explanation phase; these are not part of this report's evaluated scope (Section 7.6).
- **Methodological scope:** Design science research with iterative, test-driven development, evaluated through automated tests, a controlled retrieval experiment and performance benchmarks (Chapter 3).

1.5.2. Limitations

The limitations of this study are discussed in Section 7.6.

1.5.3. Addressing Limitations

The system was built so that each limitation can be closed after handover without redesign. The CSV importer and document parser accept real records as soon as the MoU allows data transfer. A threshold-evaluation tool (scripts/evaluate_similarity_thresholds.py) derives duplicate and linking thresholds from clerks' historical decisions. Model names, digests and

preprocessing versions are configuration, so an adapted embedding model can be swapped in and the corpus re-indexed through the durable job queue. Chapter 8 sets out a handover plan covering user acceptance testing, threshold calibration and production hardening, and a separate roadmap for completing and evaluating the generative explanation phase.

1.6. Ethical Considerations

The project was carried out with the knowledge and cooperation of the NAZ ICT Directorate, and the team observed the following safeguards.

- **Data governance.** No personal or confidential NAZ data was used during development. All records in the test suites and evaluation corpus are synthetic. A Memorandum of Understanding governing the transfer and use of real parliamentary records had been negotiated and was scheduled for signature at the time of writing. Real data will be loaded only under its terms and in line with the Data Protection Act No. 3 of 2021 [12].
- **Data sovereignty.** All AI inference runs on the institution's own infrastructure. No record or query is sent to an external AI service.
- **Human oversight.** Hybrid retrieval and duplicate detection are advisory only: every result carries its ranking evidence (lexical rank, semantic rank, cosine similarity), and only an authorised clerk can record a similarity decision or change a record's state.
- **Security and accountability.** Passwords are stored as bcrypt hashes, sessions are revocable, and access, searches and decisions are written to an audit log (Section 4.11).
- **Participants.** Consultations with NAZ staff were professional requirement discussions. No personal data about participants was collected.

1.7. Report Organisation

Chapter 2 reviews related work on parliamentary information systems, information retrieval, dense embeddings and hybrid retrieval. Chapter 3 describes the research and development methodology. Chapter 4 presents the analysis and design of the system, including requirements, use cases, architecture and data design. Chapter 5 describes the implementation, with emphasis on the duplicate-detection and hybrid retrieval algorithms. Chapter 6 covers testing. Chapter 7 evaluates the system against the objectives. Chapter 8 concludes and makes recommendations for the handover to NAZ and for future work, including the generative explanation phase. The references and appendices (source code excerpts, user manual, API reference and deployment guide) follow.

2. Literature Review

This project sits where two fields meet: parliamentary information management and information retrieval. This chapter reviews each field in turn, compares existing parliamentary systems with the proposed system, and identifies the gap the project addresses.

2.1. Parliamentary Questions, Motions and the Order Paper

In Westminster-derived legislatures such as the National Assembly of Zambia, the Order Paper is the authoritative agenda for a sitting. It lists the business the House will take, including questions for oral answer and notices of motion, in an order set by the Standing Orders [2], [3]. Items reach the Order Paper only after the Clerk's office has checked that they are admissible: correctly addressed, within the responsibility of the Minister concerned, properly formulated, and not a repetition of business already dealt with. The Order Papers that NAZ publishes show a consistent structure of sitting date, question number, member, Minister and question text [3]. That structure is the minimum data model a digital system must capture.

The admissibility check is fundamentally a *retrieval* problem. For each new item, the clerk must find the earlier items that are most similar to it and judge whether they concern the same matter. As the archive grows across sessions, this search becomes the bottleneck of the process.

2.2. Digital Parliaments and Existing Systems

The IPU's *World e-Parliament Report* documents a steady shift towards digital document management and open parliamentary data in parliaments [4]. Its Centre for Innovation in Parliament supports parliaments in sharing such practices [5]. Four relevant systems are compared below.

- **UK Parliament Written Questions service.** Every written question, answer and ministerial statement is published as structured, searchable data with filters for member, department, date and answering body [6]. It is a strong example of lifecycle tracking and open publication, but its search is based on keywords and metadata, and duplicate checking remains a procedural task for clerks.
- **Bungeni.** Bungeni is an open-source parliamentary and legislative information system developed under the United Nations' Africa i-Parliaments Action Plan. It manages the workflow of bills, questions and motions and uses XML-based document models [7]. It

showed that African parliaments can run configurable workflow systems, but it predates modern semantic retrieval.

- **Akoma Ntoso.** Akoma Ntoso is an XML standard for parliamentary, legislative and judicial documents that supports long-term interoperability and machine readability [16]. It is a data standard rather than a workflow system, and it is relevant as a future export format for Order Papers.
- **Parliamentary Monitoring Group (South Africa).** This civil-society platform tracks parliamentary committee activity and publishes questions and replies for public search [17]. It improves transparency, but it operates outside Parliament's internal admission workflow.

Table 2.1 summarises the comparison.

Table 2.1: Comparison of existing parliamentary systems with the NAZ Order Papers System

Capability	**UK Written Questions**	**Bungeni**	**PMG(South Africa)**	**NAZ Order Papers System**
Internal submission and review workflow	Partial (publication-focused)	Yes	No	Yes, nine-state lifecycle
Configurable role and permission model	Not public	Yes	Not applicable	Yes, 15 permissions and 3 user roles
Keyword search with structured filters	Yes	Yes	Yes	Yes (weighted PostgreSQL full-text search)
Semantic (meaning-based) search	Not documented	No	No	Yes (768-dimensional embeddings, pgvector)
Automated duplicate / previously-addressed checks	No (manual)	No	No	Yes, exact hash plus semantic thresholds
Order Paper generation	Separate publication process	Yes	No	Yes (structured JSON)
Deployment model	Hosted by Parliament	Self-hosted	Hosted service	Self-hosted, containerised, on-premises AI

The comparison shows that existing systems handle workflow and publication well, but none combines an internal admission workflow with hybrid, semantic-aware duplicate detection that runs entirely inside the institution.

2.3. Lexical Information Retrieval

Classical information retrieval represents documents and queries as bags of terms and ranks documents by weighted term overlap [18]. The TF-IDF scheme weights a term by how often

it appears in a document and how rare it is across the collection [19]. BM25, derived from the probabilistic relevance framework, adds term-frequency saturation and document-length normalisation, and it remains a strong baseline [20]. Lexical methods are fast, interpretable and precise for names, constituencies and ministries, but they miss relevant documents that use different vocabulary, which is known as the vocabulary mismatch problem [18].

PostgreSQL provides built-in full-text search. Documents are reduced to a `tsvector` of normalised lexemes that can be weighted by field (A to D). Queries are parsed into `tsquery` expressions, for example with `websearch_to_tsquery`, and results are ranked by `ts_rank` or the cover-density variant `ts_rank_cd`. A Generalised Inverted Index (GIN) makes matching efficient [21]. Keeping lexical retrieval inside the system-of-record database removes the need for a separate search server, a design choice this project adopted (Section 5.6.1).

2.4. Near-Duplicate Detection

Detecting documents that are almost identical is a long-studied problem. Broder's shingling and min-wise hashing estimate the overlap between sets of word sequences [22], and SimHash fingerprints were used at web scale to detect near-duplicate pages [23]. These techniques work well for copies with small edits, but they measure *surface* similarity. Two parliamentary questions can share almost no word sequences yet concern the same matter. Conversely, questions on different matters to the same Ministry can share much of their boilerplate ("To ask the Minister of Health..."). This motivates combining a cheap exact-match check on normalised text with semantic similarity, which is the two-stage design used in this project (Section 5.5).

2.5. Dense Semantic Retrieval

Distributed representations place words with similar meanings near each other in a vector space [24]. Transformer encoders such as BERT produce contextual representations [25], and Sentence-BERT fine-tuned such encoders so that whole sentences can be compared efficiently with cosine similarity [8]. Dense Passage Retrieval showed that learned dense embeddings can outperform BM25 for open-domain question answering [26]. The Massive Text Embedding Benchmark (MTEB) now compares embedding models across retrieval, clustering and semantic-similarity tasks [9].

EmbeddingGemma is a 300-million-parameter open embedding model from Google that produces 768-dimensional vectors and is designed to run on modest hardware [14]. Its small

size makes it suitable for CPU-only deployment inside NAZ, which is why this project uses it (Section 3.6). Dense retrieval complements lexical retrieval: it captures paraphrase, but it can rank documents that are only broadly on-topic highly, and it is less precise for exact identifiers.

2.6. Vector Storage and Approximate Nearest-Neighbour Search

Searching every vector exhaustively becomes slow as collections grow, so practical systems use approximate nearest-neighbour (ANN) indexes. Hierarchical Navigable Small World (HNSW) graphs give high recall with logarithmic search behaviour [27]. The pgvector extension adds a `vector` type, distance operators (including cosine distance, `<=>`) and HNSW and IVFFlat indexes to PostgreSQL [28]. Because pgvector keeps vectors in the same transactional database as the domain data, visibility rules, filters and joins can be applied in the same SQL statement as the similarity search. For a system with strict row-level visibility rules, this is a significant advantage over a separate vector store.

2.7. Hybrid Retrieval and Rank Fusion

Because lexical and dense retrieval fail in different ways, many systems combine them. Reciprocal Rank Fusion (RRF) merges ranked lists by summing 1 / (k + rank) for each document across lists. It ignores the incomparable raw scores of the underlying retrievers and has been shown to outperform more complex fusion methods [10]. Bruch *et al.* analysed fusion functions for hybrid retrieval and found RRF a robust default, although it discards score magnitudes [11]. Retrieval effectiveness is usually reported with rank-based measures such as Recall@k and Mean Reciprocal Rank (MRR) [18], [29], which this project uses in Chapter 7. Hybrid retrieval, as implemented here, is a complete and well-established retrieval technique in its own right; it does not require a generative component to be useful, and evaluating it on its own terms is the focus of this report.

2.8. Domain Adaptation of Embedding Models

General-purpose embedding models are trained mostly on web text and may not reflect specialised vocabulary. Continued pre-training on in-domain text improves downstream retrieval performance [30], and domain-specific encoder models such as LEGAL-BERT show clear gains on legal text [31]. Parameter-efficient methods make such adaptation affordable: LoRA trains small low-rank matrices instead of all weights [32], and QLoRA adds 4-bit quantisation so that adaptation fits on a single GPU [33]. Adaptation requires representative,

labelled in-domain data. For NAZ, such data will be the historical archive and clerks' recorded similarity decisions, which is why this project deferred embedding fine-tuning and instead built the data capture (review decisions, versioned thresholds and model digests) that adaptation will need later.

A brief note on scope: the literature on retrieval-augmented generation (RAG), in which a language model drafts a natural-language answer conditioned on retrieved evidence [34], [35], and on the risks of deploying such models safely [36], [37], is relevant to this project's longer-term direction rather than to what is delivered and evaluated here. It informed the forward design of the future generative explanation phase discussed in Section 8.3, but that phase is not implemented or evaluated in this report, and it is not discussed further in this chapter.

2.9. Summary and Research Gap

The literature shows that (i) parliaments are digitising their business processes [4]; (ii) lexical and dense retrieval are complementary and can be fused effectively without a generative component [10], [11]; and (iii) small open embedding models can now run locally on modest hardware [13], [14]. The systems reviewed in Section 2.2, however, do not combine an internal question-and-motion admission workflow with hybrid, semantic-aware duplicate detection that runs entirely on the institution's own infrastructure, and no such system has been documented for the National Assembly of Zambia.

This project addresses that gap. It integrates a permission-controlled parliamentary workflow with two-stage duplicate detection and hybrid lexical-semantic retrieval, fully deployable within NAZ, and lays the architectural groundwork for a future generative explanation phase without attempting to deliver or evaluate that phase now.

3. Methodology

3.1. Introduction

This chapter explains how the project was carried out and why each approach was chosen. It covers the overall research approach (Section 3.2), the development process (Section 3.3), how requirements were gathered (Section 3.4), the design principles that guided the build (Section 3.5), the tools used (Section 3.6), the data strategy (Section 3.7), project management (Section 3.8) and how the system was evaluated (Section 3.9). Together these methods connect the objectives in Section 1.3 to the design in Chapter 4, the implementation in Chapter 5 and the evidence in Chapters 6 and 7.

3.2. Research Approach

The project followed Design Science Research (DSR), which treats building and evaluating an artefact as a way of solving an identified organisational problem [38]. DSR suited this project for three reasons: the problem came from a real institution (NAZ); the main output is a working system rather than a survey or model; and the value of that system has to be shown through evaluation. Hevner *et al.*'s guidelines map onto the project as follows:

- **Design as an artefact:** the NAZ Order Papers System, together with its architecture, data model and hybrid retrieval pipeline.
- **Problem relevance:** the manual verification bottleneck described by NAZ (Section 1.2).
- **Design evaluation:** automated verification, a controlled retrieval experiment and performance benchmarks (Chapters 6 and 7).
- **Research contribution:** a locally deployable, human-supervised design for detecting duplicate parliamentary business using hybrid retrieval.
- **Research rigour:** grounding in the retrieval literature (Chapter 2) and test-driven engineering.
- **Design as a search process:** successive iterations that replaced weaker designs, for example the move from in-process BM25 to PostgreSQL full-text search (Section 5.12).
- **Communication:** this report, the diagrams and the handover documentation.

3.3. Development Methodology

3.3.1. Agile Scrum Development

Because requirements were refined through continuing engagement with NAZ, the team adopted Scrum, an Agile framework, rather than a fixed sequential plan [39]. Requirements were maintained as a prioritised product backlog, and work proceeded in time-boxed sprints. Each sprint took one or more backlog items through analysis, design, implementation and testing, and ended with a sprint review in which a working increment was demonstrated and feedback was folded back into the backlog. The repository history reflects this sprint rhythm: an initial application skeleton (April–May 2026); workflow and permission requirements implemented requirement by requirement (FR-001 to FR-029, 23–24 July 2026); similarity, responses and reporting features (24 July 2026); a production hybrid-retrieval and reliability sprint (5 August 2026); and deployment readiness (16–26 August 2026).

3.3.2. Test-Driven Development

Within each iteration the team used test-driven development (TDD) [40]. For each requirement, the acceptance criteria were first written as automated tests. Those tests were run to confirm that they failed for the expected reason, the minimum implementation was then written to make them pass, and the code was refactored with the full suite still passing. Figure 3.1 shows the cycle.

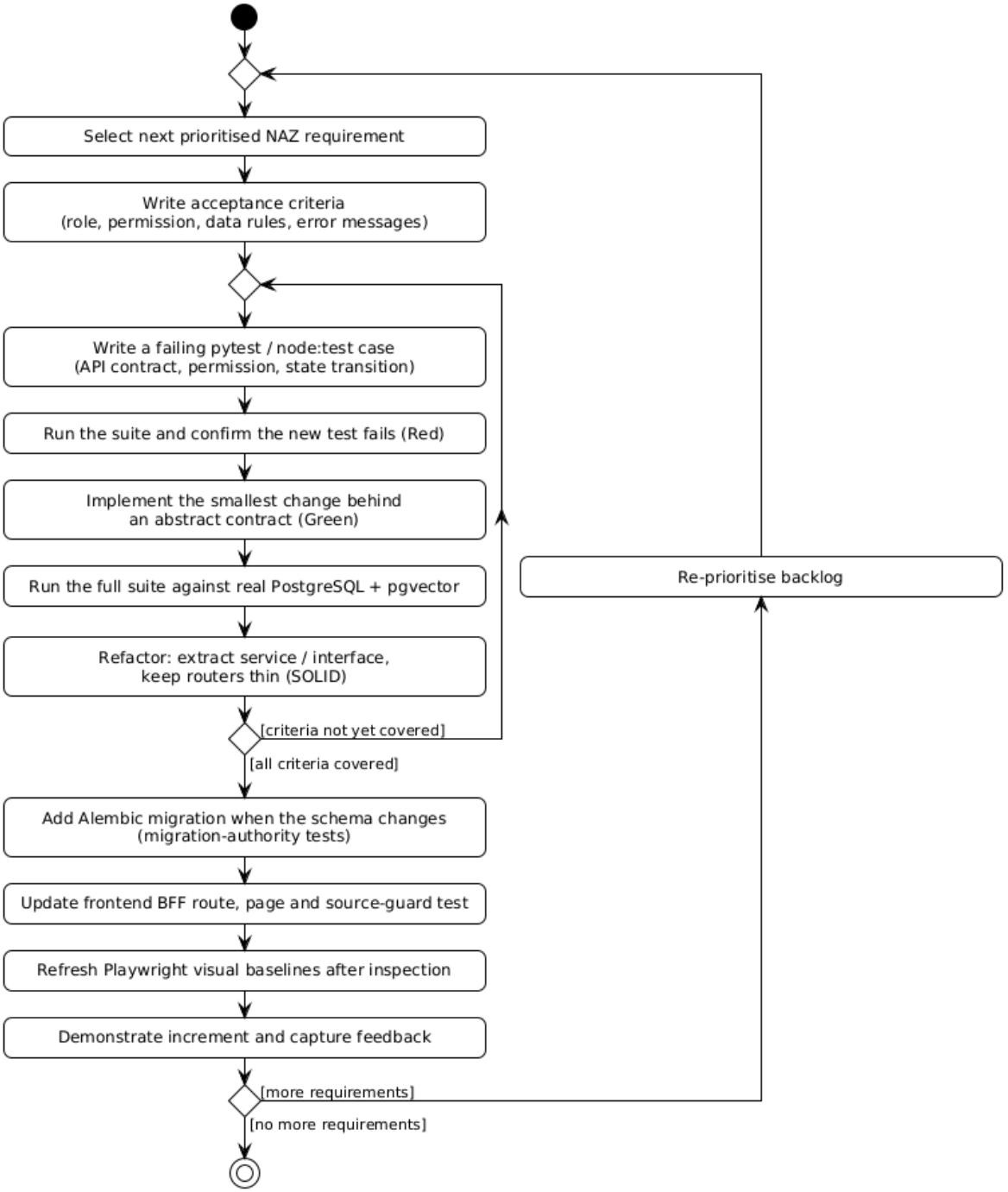


Figure 3.1: The iterative test-driven development cycle applied to each prioritised requirement

The project's implementation report records this red–green evidence: missing endpoints returned HTTP 404 before implementation, missing service modules failed test collection before the service existed, and custom roles without permissions exposed access before guards were added. Several test modules still contain the guard that failed first, for example the assertion message "FR-012 duplicate-detection service has not been implemented" in backend/tests/test_duplicate_detection.py. Each requirement was committed atomically with its

tests, for example commit 2fa1c16 (FR-003 account lockout) and commit 3e9b545 (FR-020 submission lifecycle).

TDD was chosen for three reasons. First, parliamentary procedure is rule-heavy, and rules such as "only Approved items may be scheduled" translate directly into executable tests. Second, the system was expected to change substantially (new retrieval back ends, a new job queue), and a comprehensive suite made such refactoring safe. Third, the system is being handed over to another team, and the tests serve as living documentation of the intended behaviour.

3.4. Requirements Elicitation

Requirements were gathered through three complementary activities.

1) **Consultative engagements with NAZ.** The team met the ICT Directorate, which maintains the Assembly's information systems, and staff of the Table Office, which prepares the Order Paper. NAZ provided a prioritised list of functional requirements, numbered FR-001 onwards, which became the project backlog. The priorities set the order of implementation, so the most operationally important functions (login, roles, submission, review, lifecycle and scheduling) were built first.
2) **Document analysis.** Published Order Papers [3] and the Standing Orders [2] were studied to identify the data an item must carry (session, member, Ministry, answer type, subject and text), the grouping of business on the Order Paper, and the kinds of admissibility check clerks make.
3) **Baseline assessment of the prototype.** Before the main build iteration, the team recorded the state of the early prototype and its gaps against each requirement (docs/implementation-baseline.md, 23 July 2026). This baseline turned vague needs into precise gaps, for example "records contain no submitter/owner identifier" for FR-019.

From these activities, the team identified three direct user roles: Administrator, Clerk and Viewer. Members of Parliament (MPs) are indirect stakeholders whose questions and motions are processed and managed within the system by Table Office Clerks on their behalf, rather than direct system users. The early prototype had three fixed roles (Admin, Senior Clerk and Clerk). Requirement FR-002 replaced them with configurable role data, and the legacy names are now mapped onto the new roles at start-up. The resulting requirements are presented in Section 4.3.

3.5. Design Principles

The following principles guided design decisions and are referred to throughout Chapters 4 and 5.

- **SOLID and dependency inversion.** Business services depend on abstract contracts (Python abstract base classes) rather than concrete technologies [41]. For example, duplicate detection depends on a SimilarityBackend, not on pgvector directly, and FastAPI's dependency injection supplies the concrete implementation. Swapping a model or storage engine therefore does not change business logic, and tests can substitute deterministic fakes.
- **Thin HTTP layer, explicit domain rules.** Routers validate input, enforce permissions and orchestrate calls, while rules such as lifecycle transitions live in dedicated services (submission_status.py).
- **Authorise capabilities, not role names.** Code checks atomic permission codes such as schedule_item, so NAZ can change its roles as data without changing code.
- **Database as the source of truth.** PostgreSQL holds the domain data, the full-text index, the vectors and the job queue, which keeps data consistent through ordinary transactions [42].
- **Human in the loop.** Hybrid retrieval output is advisory, and only authorised clerks can record decisions.
- **Fail safe.** If the embedding service is unavailable, search falls back to lexical retrieval and submissions are still accepted, with indexing queued for later.

3.6. Tools and Technologies

Table 3.1 lists the main tools and the reason each was chosen.

Table 3.1: Tools and technologies used in the project

Layer	Technology (version)	Reason for selection
Web client	Next.js 15.5.22, React 19.2.8 [43]	App Router with server components [57] and route handlers used as a same-origin Backend-for-Frontend.
Client validation and state	Zod 3, React Hook Form 7, Zustand 5, jose 5	Declarative form validation [58], [59], a lightweight authentication store [60] and JWT verification in middleware [61].
Styling	Plain CSS (Tailwind [62] removed)	Fewer dependencies and full control; protected by visual regression tests.

Layer	Technology (version)	Reason for selection
API service	Python 3.11 [63], FastAPI 0.141.1, Pydantic 2 [44]	Typed request/response contracts [64], dependency injection and automatic OpenAPI documentation.
Data access	SQLAlchemy 2.0, psycopg 3, Alembic 1.18 [45], [46]	Mature ORM with PostgreSQL dialect features and the psycopg driver [66]; versioned schema migrations.
Database	PostgreSQL 16 with pgvector [28]	Relational integrity, built-in full-text search and vector search in one engine [67].
Local AI runtime	Ollama 0.32.3 [13]	Serves the open embedding model over HTTP on the institution's own hardware.
Embedding model	embeddinggemma:300m (768 dimensions) [14]	Small, CPU-friendly and strong on retrieval tasks; the only model this report's evaluated features depend on.
Document parsing	pypdf 6.14, python-docx	Text extraction from uploaded PDF [68] and DOCX [69] submissions.
Security	bcrypt 4.2 (cost 12), PyJWT 2.8 (HS256)	Adaptive password hashing and signed, expiring session tokens [47], [48].
Testing	pytest 8.3, Node test runner [70], Playwright 1.61 [49], [50]	API integration tests on real PostgreSQL, source-level guards and browser visual tests.
Packaging	Docker and Docker Compose [51]	Reproducible multi-service environment and a single-host deployment path.
Version control	Git and GitHub	Branch-based collaboration [72] and an auditable history of changes [71].

The Ollama runtime is also capable of serving a local instruction-following language model (qwen3.5:4b was provisioned in the development stack for this purpose). That model supports only the early-stage generative explanation prototype described in Section 7.6, which is future work; it is not required by, and is not used by, any feature this report evaluates.

3.7. Data Strategy

At the time of development, historical NAZ records had not been transferred because the data-sharing MoU was still pending (Section 1.6). The team therefore followed a synthetic-data strategy.

- **Seed data.** An optional development seed (backend/db/seed.sql) creates sample parliamentary sessions, staff accounts and example records, and the API start-up routine seeds the default roles and permissions. Together they allow the full workflow to be demonstrated.

- **Test fixtures.** Every automated test builds its own records inside an isolated test database, so tests never depend on shared data.
- **Evaluation corpus.** For the retrieval experiment in Chapter 7, the team wrote 24 synthetic records in the style of NAZ questions and motions. The corpus includes deliberately similar "hard negatives" that share a Ministry or vocabulary with a target record but concern a different matter.
- **Real-data path.** Real records will enter through the CSV importer (backend/scripts/ingest_csv.py), which expects the columns item_type, session_code, member, ministry, subject, full_text and optionally status, or through the document upload endpoint. Imported records are indexed by the background worker without any change to the architecture.

3.8. Project Management and Version Control

The team, Team CMZ, worked in a shared Git repository with feature branches (for example feat/mainza) that were merged into main. Between 28 April and 26 August 2026 the repository recorded 126 commits from the three team members. Commit messages follow a conventional format (feat:, fix:, refactor:, test:, docs:) and cite requirement identifiers, which keeps the history traceable to the backlog. Progress was reviewed with the supervisors and the NAZ ICT Directorate at the end of iterations. Design and verification documents were kept in the repository (docs/) next to the code, so documentation changed together with the implementation.

3.9. Evaluation Design

The artefact was evaluated along the dimensions set out in specific objective 5 (Section 1.3.2).

1) **Functional correctness.** Automated pytest integration tests run against a real PostgreSQL and pgvector database, source-level guards check the frontend, and Playwright checks the browser views (Chapter 6).
2) **Retrieval effectiveness.** A controlled experiment compared lexical, semantic and hybrid retrieval on 18 reworded queries against the 24-record synthetic corpus, using Recall@k and MRR@10 [29]. The same corpus was used to examine the cosine-similarity values that the duplicate and previously-addressed thresholds act on (Section 7.3).

3) **Performance.** The project's benchmark script measured PostgreSQL lexical retrieval over 100,000 synthetic records against a 3,000 ms 95th-percentile budget, in an isolated database with all inserted data rolled back (Section 7.4).

4) **Usability.** A System Usability Scale (SUS) study [52] with Table Office staff is planned for the handover phase. It could not be carried out before the MoU was signed (Section 7.5).

All measurements were taken on the development workstation: a four-core CPU, 16 GB RAM, no GPU, Linux, and PostgreSQL 16 in Docker.

4. System Analysis and Design

This chapter turns the understanding gained in Chapter 3 into a specification and design. It analyses the current process (Section 4.1), defines the actors and permissions (Section 4.2) and the functional and non-functional requirements (Sections 4.3 and 4.4), models use cases (Figure 4.1 and Table 4.4) and the business process (Sections 4.5 and 4.6), and presents the architecture, data, lifecycle, interface, security and deployment designs (Sections 4.7 to 4.12). All UML diagrams were written in PlantUML and are kept with the project in diagrams/plantuml/.

4.1. Analysis of the Current Process

In the current manual process, an MP drafts a question or notice of motion and submits it on paper. Table Office staff record its receipt, check that it is admissible, and search earlier Order Papers and registers to see whether the same matter has already been raised. Admitted items are placed on the Order Paper for a sitting, and answered items are eventually filed. The analysis identified four weaknesses, each of which maps to a design response:

- **No single register of business.** *Design response:* one database of records with an explicit lifecycle state and an owner (Sections 4.8 and 4.9).
- **Slow, wording-dependent duplicate checks.** *Design response:* automatic exact and semantic similarity checks at submission time, plus hybrid search for clerks (Sections 5.5 and 5.6).
- **Weak accountability.** *Design response:* permission-controlled actions and an append-only audit log (Section 4.11).
- **Manual assembly of the Order Paper.** *Design response:* scheduling against a sitting date and automatic generation of the sitting's business (Section 5.4).

4.2. Actors and Permissions

Three internal human user roles and one external system actor interact directly with the system. Members of Parliament (MPs) do not interact with or log into the system directly; instead, Table Office Clerks receive drafts and physical submissions from MPs and process, submit, review, and track them within the system on their behalf.

- **Clerk:** Table Office staff who enter questions and notices of motion on behalf of MPs, review submissions, record similarity decisions, approve, reject or request changes, edit

and resubmit returned drafts, schedule approved items, record responses, and manage parliamentary sessions.

- **Administrator:** has every permission across the system, manages users and roles, and can unlock locked user accounts.
- **Viewer:** has read-only access to view reports, generated Order Papers, and the parliamentary archive.
- **Local AI service (Ollama):** provides embeddings to the system's hybrid retrieval pipeline.

Following the design principle of authorising capabilities rather than role names (Section 3.5), access is defined through 15 atomic permissions grouped into roles. Table 4.1 shows the default matrix seeded by backend/app/services/permissions.py. Roles are ordinary database records, so NAZ can create a custom role (for example, a Senior Clerk with scheduling rights but no audit access) without changing code.

Table 4.1: Default role–permission matrix

Permission	Administrator	Clerk	Viewer
submit_question, submit_motion	Yes	Yes	No
review_submission	Yes	Yes	No
approve_motion, reject_submission, request_changes	Yes	Yes	No
schedule_item, record_response	Yes	Yes	No
manage_sessions	Yes	Yes	No
view_audit	Yes	Yes	No
view_reports, search_archive, view_archive	Yes	Yes	Yes
manage_users, manage_roles	Yes	No	No

4.3. Functional Requirements

Table 4.2 lists the functional requirements from NAZ's prioritised list that were in scope for this phase, using NAZ's identifiers. The final column records the implementation status, which is examined in Chapter 7.

Table 4.2: Functional requirements and implementation status

ID	Requirement	Status
FR-001	Staff log in with employee ID and password; invalid credentials give a generic message.	Implemented
FR-002	The system supports Administrator, Clerk, and Viewer roles, with MPs served indirectly as stakeholders.	Implemented (configurable)
FR-003	An account locks after five failed logins; an administrator can unlock it.	Implemented (unlock via API)
FR-005	Authorised users submit questions for oral or written answer.	Implemented
FR-006	Authorised users submit notices of motion.	Implemented
FR-007	Required fields are validated with clear error messages.	Implemented
FR-009	New submissions enter a clerk review queue.	Implemented
FR-010	Clerks approve, reject or request changes.	Implemented
FR-012	Possible duplicates are detected and blocked unless confirmed.	Implemented (confirmation not yet exposed in the web form)
FR-014	Similarity results show session, date, member, source record and score.	Implemented
FR-016	Matters previously addressed in earlier sessions are identified.	Implemented
FR-017	High-confidence related earlier items are linked to the new record.	Implemented
FR-019	Clerks track the status of submissions entered on behalf of MPs.	Implemented
FR-020	A defined set of lifecycle statuses and transitions is enforced.	Implemented (nine states)
FR-022	Approved items are scheduled for a sitting date.	Implemented
FR-023	An Order Paper is generated for a sitting date.	Implemented (structured JSON)
FR-025	Records of ended or closed sessions are archived automatically.	Implemented (runs at API start-up)
FR-026	Archive viewing and searching are permission-controlled.	Implemented
FR-028	Keyword search covers subject and full text.	Implemented (hybrid by default)
FR-029	Records can be filtered by session, date, member, Ministry, status and type.	Implemented
FR-031	Users are notified by email of status changes.	Built on 24 July 2026 and later retired (Section 5.11)
FR-037	A report can be produced for a parliamentary session.	Implemented (JSON)
FR-040	Clerks record the House's response to a scheduled question.	Implemented

ID	Requirement	Status
FR-041	Recording a response marks the question as answered in the same transaction.	Implemented

4.4. Non-Functional Requirements

Table 4.3 lists the non-functional requirements derived from the consultations and from the institutional context (sensitive data, a small ICT team, modest hardware).

Table 4.3: Non-functional requirements and design measures

ID	Quality	Requirement	Design measure
NFR-01	Security	Only authenticated, authorised staff can act; sessions can be revoked.	bcrypt, HS256 JWT with a server-side session row, permission guards, row visibility rules.
NFR-02	Privacy and sovereignty	Parliamentary text never leaves NAZ infrastructure.	Local Ollama embedding model; no external AI API calls.
NFR-03	Accountability	Significant actions are traceable to a user, time and IP address.	audit_logs written in the same transaction as the action.
NFR-04	Integrity	Records cannot enter invalid states, even under concurrent requests.	Central transition graph, database CHECK constraints, row locks and idempotency keys.
NFR-05	Performance	Search returns ranked results quickly on a production-sized archive.	`GIN full-text index, HNSW vector index; budget of 95th percentile ≤ 3 s for 100,000 records.`
NFR-06	Availability and resilience	Loss of the embedding service does not stop submission or search.	Retries, circuit breaker, lexical fallback, durable job queue with retry and dead-lettering.
NFR-07	Interpretability and oversight	Retrieval results are interpretable and advisory, and a human makes the final call.	Every result reports its lexical rank, semantic rank and cosine similarity; only an authorised

ID	Quality	Requirement	Design measure
			clerk can record a similarity decision.
NFR-08	Maintainability	NAZ can change models, roles and thresholds without code changes.	Abstract contracts, dependency injection, configuration via environment variables, versioned migrations.
NFR-09	Operability	The system is easy to deploy, monitor and recover.	Docker Compose, health-gated start-up, /livez and /readyz, structured JSON logs with request IDs.
NFR-10	Usability	The interface works on desktop and mobile screens and hides actions a user cannot perform.	Responsive plain CSS, permission-aware navigation, visual regression baselines.

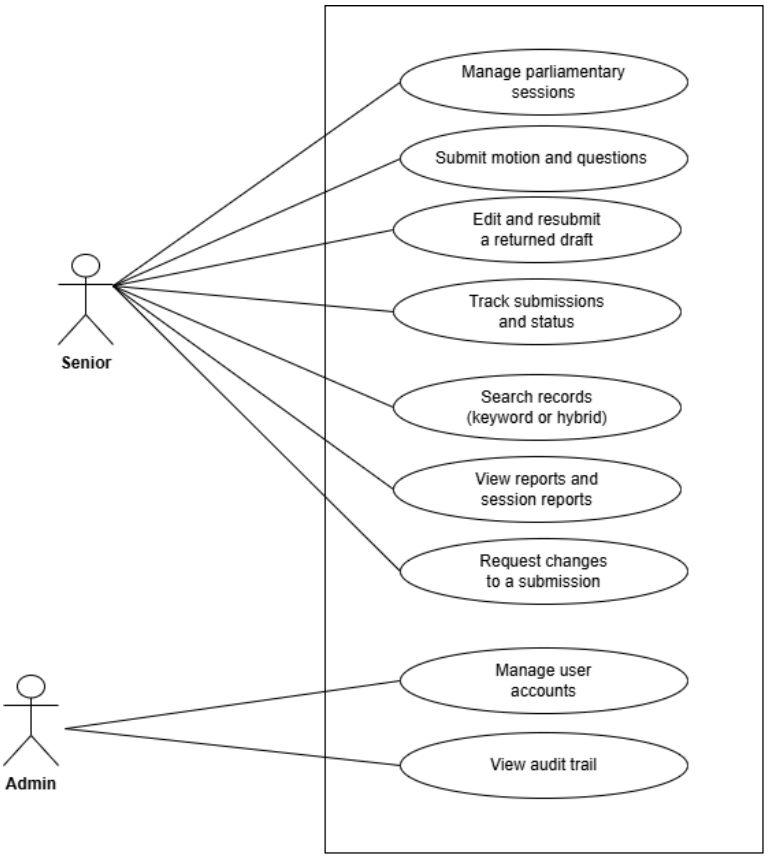


Figure 4.1: Use case diagram for the Clerk and Administrator

Table 4.4: Primary Use Case Descriptions Summary

Use Case	Primary Actor	Triggers	Preconditions	Postconditions
Submit Question	Clerk	Clerk enters question details on behalf of an MP	User logged in; active/upcoming session	Record stored as Submitted, moved to Under Review, indexing queued
Submit Notice of Motion	Clerk	Clerk enters motion details on behalf of an MP	User logged in; active/upcoming session	Record stored as Submitted, moved to Under Review
Log In / Log Out	All Users	User submits credentials or clicks Log Out	Active account (not locked)	Authenticated session created (login) or terminated (logout)
Manage Parliamentary Sessions	Clerk / Admin	User accesses Session Management	User logged in with manage_sessions permission	Session created/updated; Active/Upcoming sessions can receive submissions
Edit & Resubmit Returned Draft	Clerk	Clerk re-opens returned item to apply requested revisions	Item returned with requested changes	Status updated to Under Review for re-review
Track Submissions & Status	All Users	User views submissions list or status tracker	User logged in	Current status and submission details displayed
Record Similarity Decision	Clerk	Clerk reviews ranked matches	Submission has similarity check candidates	Decision (Clear/Duplicate/ Substantially Similar) logged and linked
Schedule Approved Item	Clerk	Clerk selects approved item for sitting	Item status is Approved; valid sitting date	Item assigned to sitting date; status updated to Scheduled
Generate Order Paper	Clerk	User triggers Order Paper generation	At least one item scheduled for sitting date	Structured Order Paper compiled and saved
View Order Paper	All Users	User selects sitting date	Order Paper generated for sitting date	Order Paper content displayed
Record Response to Question	Clerk	Clerk enters response for scheduled question	Question scheduled for a sitting	Response saved; question marked as Answered
Search Records	All Users	User enters query in search interface	User logged in with search permissions	Ranked search results returned
View & Search Archived Records	All Users	User searches or browses archive	User logged in with view_archive permission	Historical/archived records displayed

4.6. Business Process Model

Figure 4 models the end-to-end process that the system supports, with swimlanes illustrating the administrative workflow managed by Clerks on behalf of Members of Parliament alongside the automated system actions. Compared with the manual process, the similarity checks happen automatically at submission time, and the Order Paper is derived from the scheduled records instead of being assembled by hand.

4.7. System Architecture

The system uses a layered, service-oriented architecture with three runtime tiers plus supporting processes (Figure 4.2; see also Figure C.4 in Appendix C).

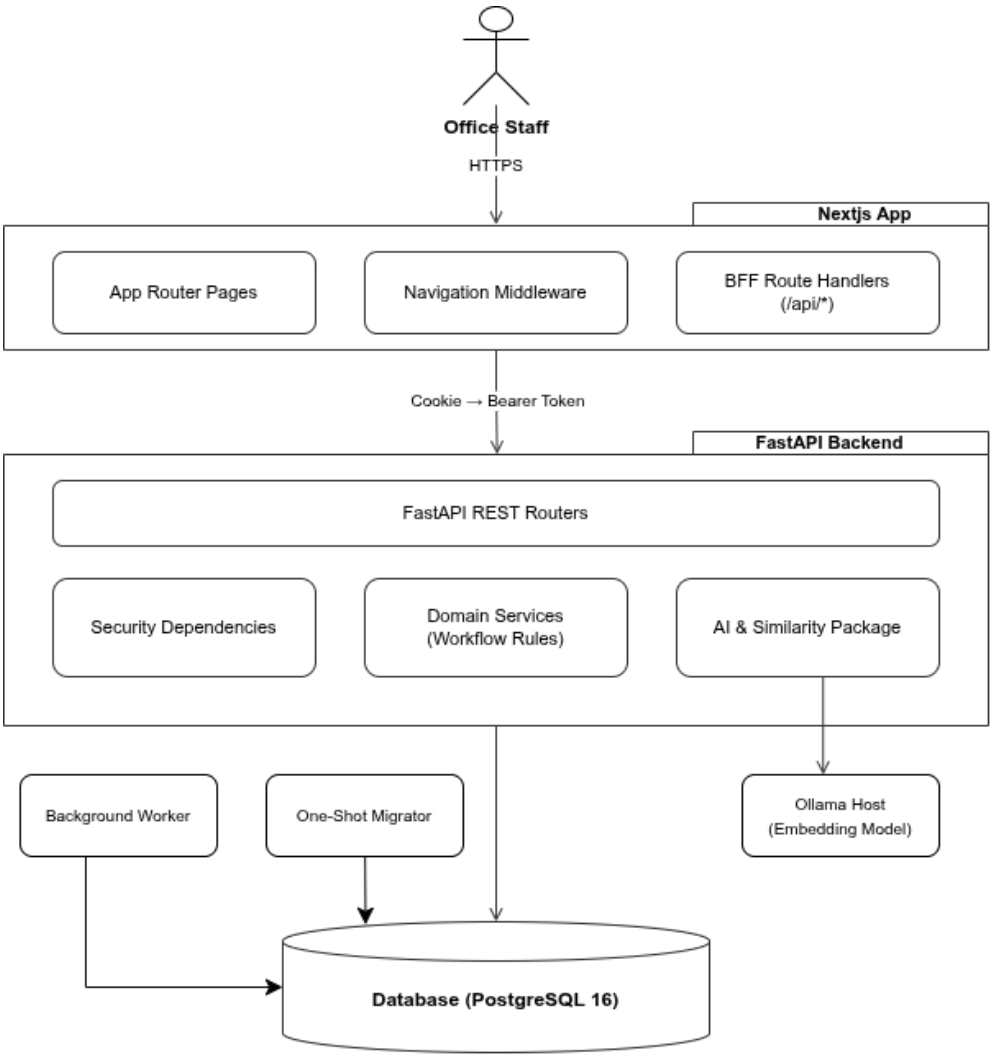


Figure 4.2: System architecture (component view)

- **Presentation tier.** Next.js App Router pages render the interface. A navigation middleware verifies the session token before protected routes load. Route handlers under

/api/* act as a Backend-for-Frontend (BFF) [53]: the browser only calls its own origin, and the BFF reads the HTTP-only session cookie on the server and forwards it to the API as a bearer token.

- **Application tier.** FastAPI routers expose the REST API. Security dependencies resolve the current user, check permissions and apply row visibility. Domain services hold the workflow rules. The similarity package performs duplicate and previously-addressed checks, and the AI package implements hybrid lexical-semantic retrieval.
- **Data tier.** PostgreSQL 16 stores the domain, access-control and audit data, the weighted full-text vectors, the embedding chunks and the durable job queue.
- **Supporting processes.** A one-shot migrator applies Alembic migrations under an advisory lock. A background worker processes indexing jobs. Ollama serves the embedding model on the host.

Table 4.5 records the main architectural decisions and the alternatives considered.

Table 4.5: Main architectural decisions and alternatives considered

Decision	Alternatives considered	Rationale
Keep vectors and full-text search inside PostgreSQL (pgvector).	Separate search engine or vector database.	One transactional store; visibility filters and joins in a single query; fewer services for NAZ to operate.
Use a BFF with HTTP-only cookies.	Browser calls the API directly with a stored token.	Keeps the token out of browser storage and gives same-origin requests and one place to handle errors.
Run the embedding model locally with Ollama.	Cloud embedding APIs.	Data sovereignty and Data Protection Act compliance; no 0per-request cost.
Hybrid retrieval with RRF.	Lexical only; semantic only; weighted score blending.	Combines complementary signals without calibrating incompatible score scales [10].
Durable PostgreSQL job queue with a transactional outbox.	In-process background tasks; external message broker.	Jobs survive restarts and are committed atomically with the domain change [54]; no broker to run.
Alembic as the sole schema authority.	Runtime DDL at start-up (earlier prototype).	Versioned, reviewable, repeatable upgrades; readiness checks can verify the schema head.
Permission-based RBAC stored as data.	Hard-coded role names.	NAZ can reconfigure roles without code changes.

4.8. Data Design

The logical data model is implemented as PostgreSQL tables with UUID primary keys. It is presented in two parts. Figure 4.3 shows identity, access control and parliamentary business, and Figure 4.4 shows retrieval, reliability and audit. A conceptual overview of the same data is given in Figure C.1 (Appendix C).

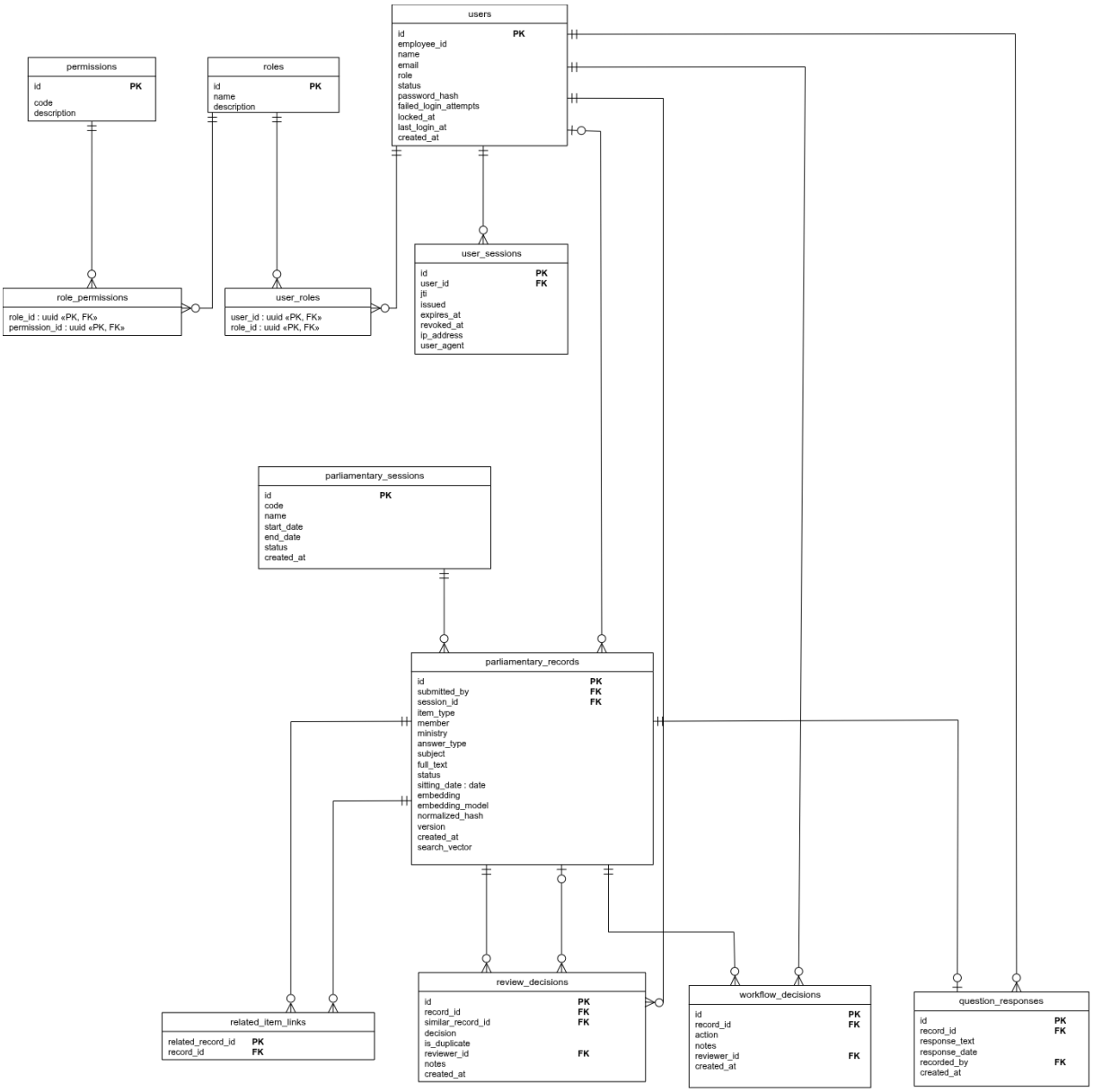


Figure 4.3: Entity-relationship diagram: identity, access control and parliamentary business

The central entity is parliamentary_records, which stores a question or motion with its session, member, Ministry, answer type, subject, full text, owner, lifecycle status and sitting date. Four columns support similarity work:

- normalized_hash: a SHA-256 digest of the normalised type, subject and text, used for exact-duplicate detection.
- version: incremented whenever the content changes, so that stale indexing jobs can be recognised.
- search_vector: a generated tsvector that weights the subject as A, the full text as B, and the member and Ministry as C, with a GIN index.
- embedding: a record-level vector(768) with an IVFFlat cosine index, used by duplicate detection.

Workflow history (workflow_decisions) and similarity decisions (review_decisions) are deliberately kept in separate tables, so classifying an item as a duplicate never changes its workflow state by itself. A CHECK constraint on review_decisions requires a related record for Duplicate and Substantially Similar decisions and forbids one for Clear (New).

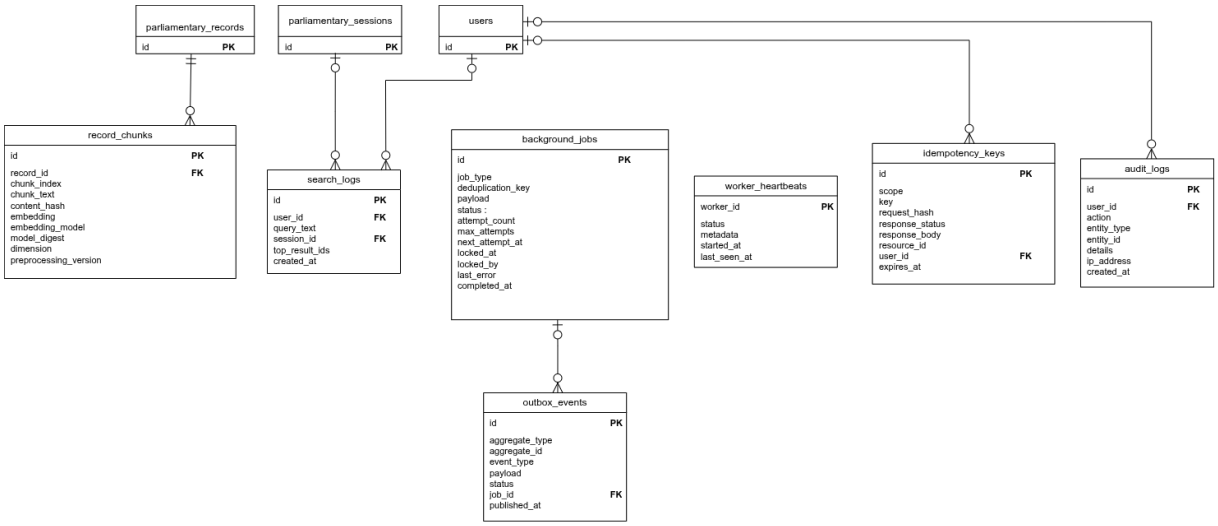


Figure 4.4: Entity-relationship diagram: retrieval, reliability and audit

record_chunks stores overlapping passages of each record with their embeddings, identified by chunk index, content hash, model name, exact model digest, dimension and preprocessing version, and served by an HNSW cosine index. background_jobs and outbox_events implement the durable queue. idempotency_keys stores replayable responses for write operations, and audit_logs keeps the accountability trail. The schema is created and evolved by 12 Alembic migrations, from 20260701_0000 (baseline) to 20260805_0011 (worker heartbeats). This section, and Figure 4.4, document the schema supporting the hybrid-retrieval and workflow features evaluated in this report; the database additionally provisions an audit table and a durable job type for the future generative-explanation phase described in Section 7.6, which are outside this report's evaluated scope and are not diagrammed here.

4.9. Submission Lifecycle Design

Requirement FR-020 is realised as an explicit state machine (Figure 4.5), defined once in ALLOWED_TRANSITIONS and enforced by transition_submission(). The database check constraint accepts exactly these nine values. The original seven-state design was extended with Answered and Discussed so that addressed matters can be recognised in later sessions (FR-016). Any transition not in the graph raises InvalidStatusTransition, which the API reports as HTTP 409.

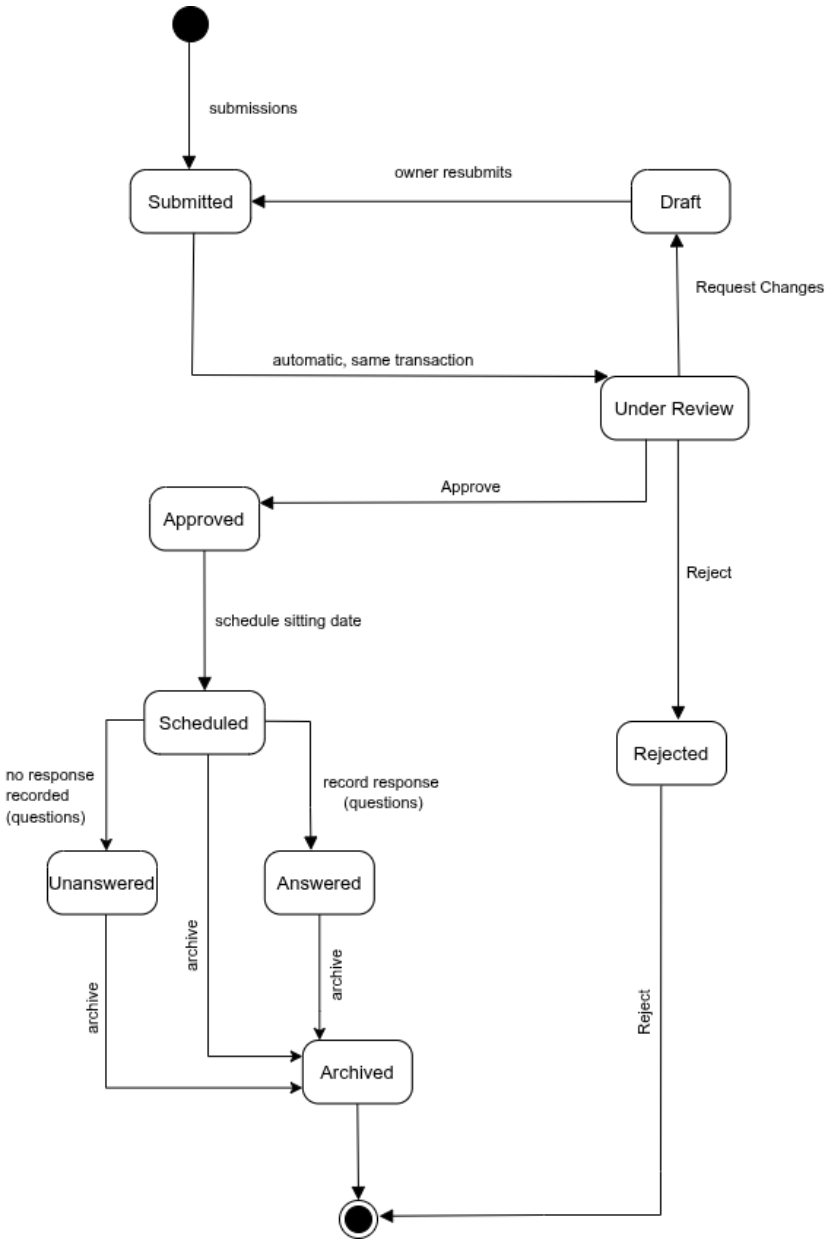


Figure 4.5: Submission lifecycle state machine

4.10. Interface (API) Design

The API follows REST conventions with JSON bodies validated by Pydantic. FastAPI publishes an OpenAPI [65] description at /docs. Table 4.6 summarises the main resources evaluated in this report; the full list is given in Appendix C.

Table 4.6: Summary of the main API resources

Resource	Main operations	Guard
/auth	POST /login, GET /me, POST /logout	Public / session
`/submissions`	create; check-similarity; upload; mine; review-queue; PATCH draft; submit	Submit permission per item type; review_submission for the queue
`/submissions/{id}/workflow-review`	Approve, Reject, Request Changes	approve_motion / reject_submission / request_changes
`/submissions/{id}/schedule`	Schedule for a sitting date	schedule_item
`/records`	list with filters; detail; similar	Session plus row visibility
`/records/{id}/reviews`, `/records/{id}/response`	Similarity decisions; question response	review_submission; record_response
`/search`	Keyword or hybrid ranked search	Session plus archive visibility
`/order-papers/{date}`	Generate a sitting's Order Paper	Session
`/reports`, `/reports/sessions/{id}`	Operational and session reports	view_reports
`/sessions`	list; create; update; delete	Public list; manage_sessions for changes
`/ai/search`, `/ai/health`, `/ai/index/*`, `/ai/reindex`	Hybrid search, model health, chunk indexing	Session; review_submission
`/audit`, `/notifications`, `/users/{id}/unlock`	Audit trail, activity feed, account unlock	view_audit; session; manage_users
`/livez, /health, /readyz`	Liveness and readiness	Public

Write endpoints that could be retried by a client (submission, workflow review, scheduling, similarity decision and response recording) accept an optional Idempotency-Key header. A

repeated request with the same key and payload returns the original response, and the same key with a different payload is rejected with HTTP 409.

4.11. Security Design

Security controls follow the OWASP Top 10 [55] and NIST authentication guidance [56]:

- **Authentication.** Passwords are verified against bcrypt hashes with cost factor 12 [47]. Invalid employee IDs, wrong passwords, inactive accounts and locked accounts all return the same message, which prevents account enumeration. The fifth consecutive failure locks the account.
- **Sessions.** A successful login issues an HS256 JWT [48] with subject, permissions, a unique jti and an eight-hour expiry, and stores a matching user_sessions row. Every API call checks that this row exists, has not been revoked and has not expired, so logout takes effect immediately even though the token itself has not expired.
- **Authorisation.** Permission dependencies guard each route. Row-level rules hide other users' drafts from anyone without review_submission and hide archived records from anyone without the archive permissions. Unauthorised list queries omit rows rather than reveal that they exist.
- **Transport and browser.** The cookie is HttpOnly and SameSite=Lax, and it is Secure in production behind TLS. CORS is limited to the configured frontend origin.
- **Input handling.** All inputs are validated by Pydantic, and all SQL is parameterised through SQLAlchemy. Uploaded documents are size-limited (10 MB), type-checked by extension, declared MIME type and magic bytes, and parsed in a separate process with CPU-time and memory limits (Section 5.8).
- **Accountability.** Logins, logouts, unlocks, submissions, edits, workflow actions, scheduling, similarity decisions, searches, record access, and report and audit access are all written to audit_logs.

4.12. Deployment Design

For the handover, the system is designed to run on a single host (Figure 4.6), as documented in the project's EC2 runbook (docs/operations/ec2-deployment.md). The same topology applies to an on-premises NAZ server. Only the frontend is published, bound to the loopback interface behind a TLS reverse proxy. The API, database and Ollama ports are never exposed publicly. Services start in a health-gated order: the database, then migrations, then the API and worker, then the frontend.

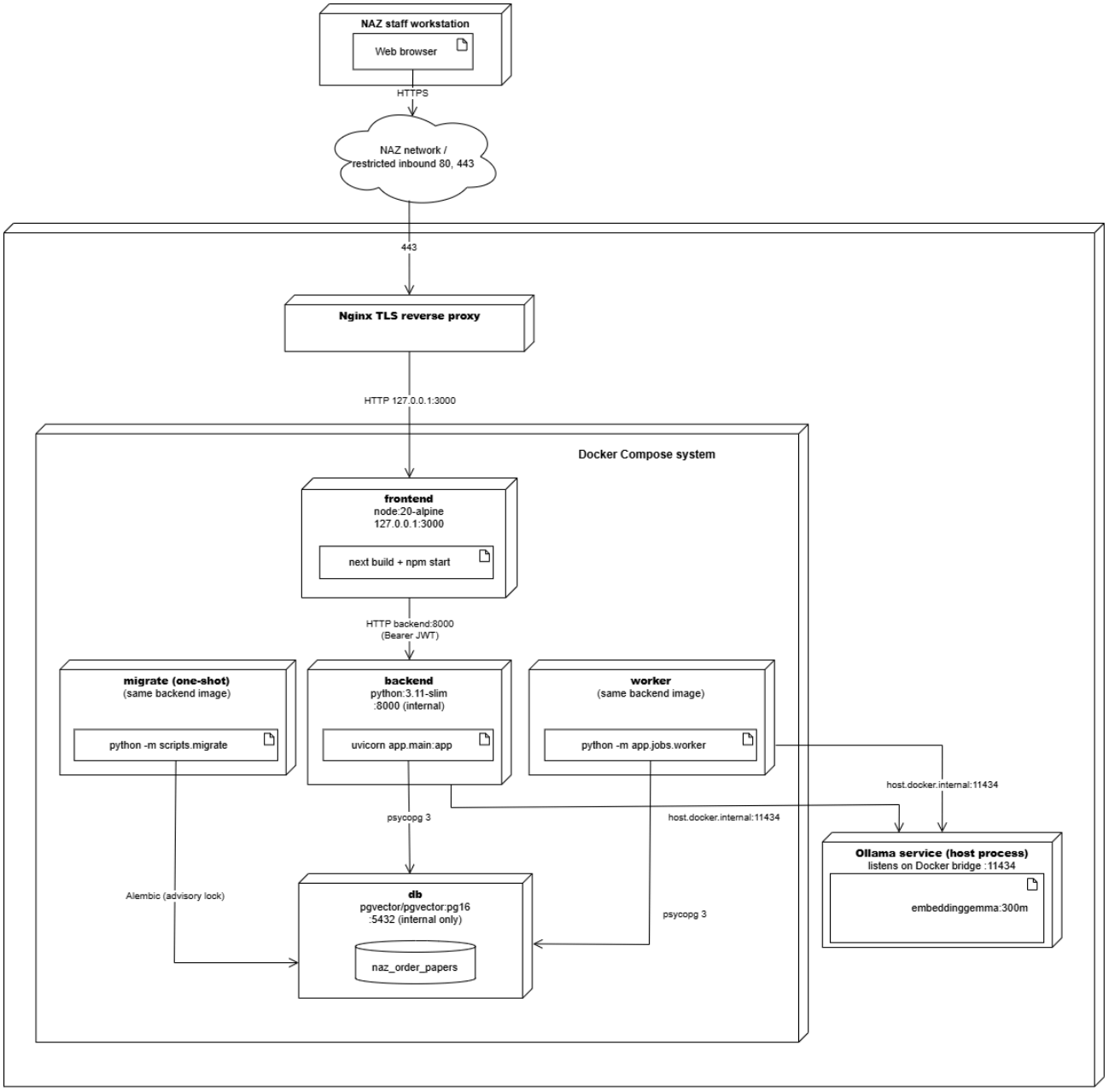


Figure 4.6: Production deployment diagram for the NAZ handover

5. Implementation

This chapter explains how the design in Chapter 4 was built. It starts with the codebase structure (Section 5.1) and the backend application framework (Section 5.2), then covers security (Section 5.3), the submission workflow (Section 5.4), the duplicate-detection algorithm (Section 5.5), hybrid retrieval (Section 5.6), background processing (Section 5.7), document ingestion (Section 5.8), the frontend (Section 5.9) and deployment (Section 5.10). It closes with how the design evolved (Section 5.11) and the main implementation challenges (Section 5.12). The backend application package contains 93 Python modules (about 8,400 lines), a portion of which implements the future-phase generative explanation prototype described in Section 7.6 and not otherwise discussed in this chapter.

5.1. Codebase Structure

Table 5.1 summarises how the repository is organised.

Table 5.1: Repository structure

Path	Contents
`backend/app/main.py`	FastAPI application factory: middleware, router registration and start-up tasks.
`backend/app/routers/`	HTTP routers (auth, submissions, workflow reviews, scheduling, responses, reviews, records, search, Order Papers, reports, session reports, sessions, audit, notifications, users, health).
`backend/app/services/`	Domain services: lifecycle, permissions, visibility, archiving, idempotency, transactions, readiness, document parser.
`backend/app/similarity/`	Duplicate, previously-addressed and related-item services behind abstract contracts.
`backend/app/ai/`	Embedding provider, lexical and semantic retrievers, RRF fusion, chunk indexing, and an AI router (a subset of this package supports the future explanation phase; Section 1.5.2).
`backend/app/jobs/`	Durable job queue and worker process.
`backend/app/models/, schemas/`	SQLAlchemy ORM models and Pydantic contracts.
`backend/alembic/versions/`	Twelve versioned schema migrations.
`backend/scripts/`	CSV ingestion, migration runner, re-indexing, search benchmark, threshold evaluation.
`backend/tests/`	44 pytest modules (224 tests).

Path	Contents
`frontend/src/app/`	Next.js pages ((dashboard), (auth)) and BFF route handlers (api/).
`frontend/src/components/, hooks/, lib/`	UI components, data hooks, BFF proxy and authentication helpers.
`frontend/tests/`	Node source guards and Playwright visual tests.
`docker-compose*.yml, docs/, diagrams/`	Deployment configuration, specifications and UML sources.

5.2. Backend Application Framework

backend/app/main.py builds the FastAPI application. It configures structured JSON logging, adds CORS and a request-observability middleware, and registers 17 routers. A lifespan handler runs at start-up: it seeds the default roles and permissions, maps legacy role names onto them, and archives records of ended sessions.

Dependency injection through FastAPI's Depends is the main tool for applying the SOLID principles (Section 3.5). Routers never construct infrastructure themselves. For example, the submission endpoint receives a DuplicateChecker, a PreviouslyAddressedChecker, a RelatedItemLinker, an EmbeddingGenerator and a SimilarityResultFormatter, all declared as abstract types and supplied by app/similarity/dependencies.py. Tests replace these with deterministic fakes through app.dependency_overrides, and a different embedding model or vector store can be swapped in by changing one provider function. Figure 10 shows the workflow, authorisation and reporting services built on the same pattern.

Database access uses SQLAlchemy 2.0 sessions scoped to each request. Configuration is loaded once from environment variables into a cached Settings object. It includes the embedding model name, the embedding dimension (768), the RRF constant (60), the retrieval depths (20 lexical, 20 semantic, 5 final), the chunk size (1,800 characters with a 240-character overlap), the similarity thresholds (0.95, 0.99 and 0.72), timeouts, and the database pool and statement-timeout settings.

5.3. Authentication and Authorisation

Figure 5.1 shows the authentication flows. The login endpoint normalises the employee ID to upper case, loads the user, and applies the lockout rule: each failed password increments failed_login_attempts, and the fifth failure sets locked_at. On success it signs a JWT whose claims include the user's permission codes and a random jti, and it stores a user_sessions row.

The Next.js login route stores the token in the HttpOnly naz_token cookie. The client code never stores the token.

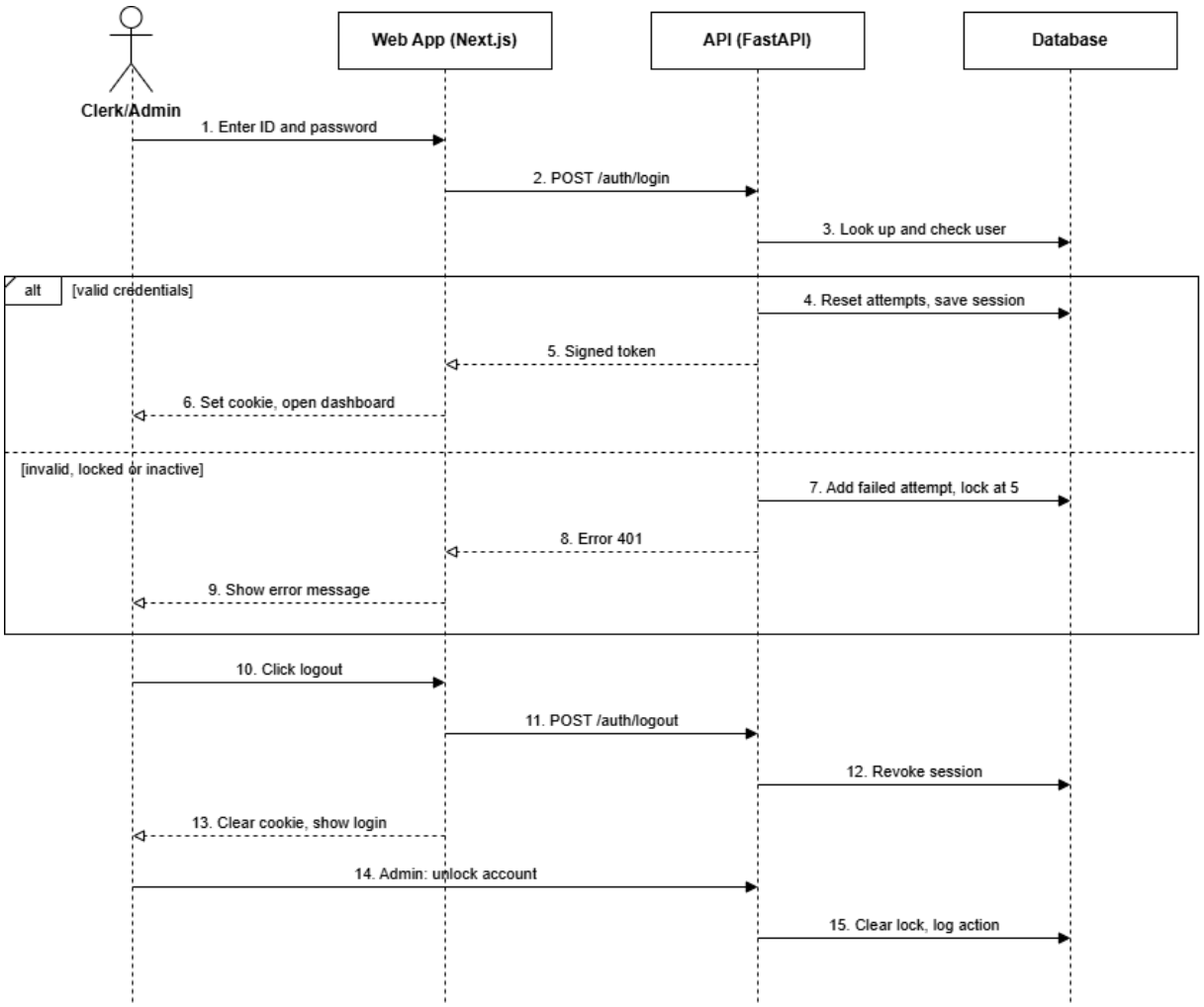


Figure 5.1: Sequence diagram: login, protected navigation, API calls, logout and unlock

Every protected endpoint depends on get_current_user, which rejects a request unless the token's session row is present, not revoked and not expired, and the account is Active. Permission checks are a one-line dependency factory:

```Python
def require_permission(permission_code: str):
    def dependency(user: User = Depends(get_current_user)) → User:
        if not user.has_permission(permission_code):
            raise HTTPException(status_code=403, detail="Insufficient permissions")
        return user
    return dependency

@router.post("/{user_id}/unlock")
```

```
def unlock_user(user_id: UUID, request: Request, db: Session = Depends(get_db),
                administrator: User = Depends(require_permission("manage_users"))):
'''
```

User.has_permission resolves permissions through the user's roles (users → user_roles → roles → role_permissions → permissions), so reassigning a permission in the database changes behaviour immediately. Row visibility is centralised in record_visibility.py and applied inside the queries themselves: drafts are visible only to their owner or to holders of review_submission, and archived rows require view_archive (listing and detail) or search_archive (search).

5.4. Submission and Review Workflow

5.4.1. Creating a Submission

Figure 5.2 shows POST /submissions. Pydantic first validates the payload and enforces type-specific rules: a question needs a Ministry and an answer type, and a motion must not have an answer type. The router then checks the item-type permission, claims the optional idempotency key, and confirms that the target session is Active or Upcoming. It runs the duplicate and previously-addressed checks (Section 5.5). If a possible duplicate is found and the request does not carry confirm_duplicate, the transaction is rolled back and HTTP 409 is returned with the ranked matches, the threshold and the threshold version. Figure 5.3 shows the submission screen on which a clerk enters these details.

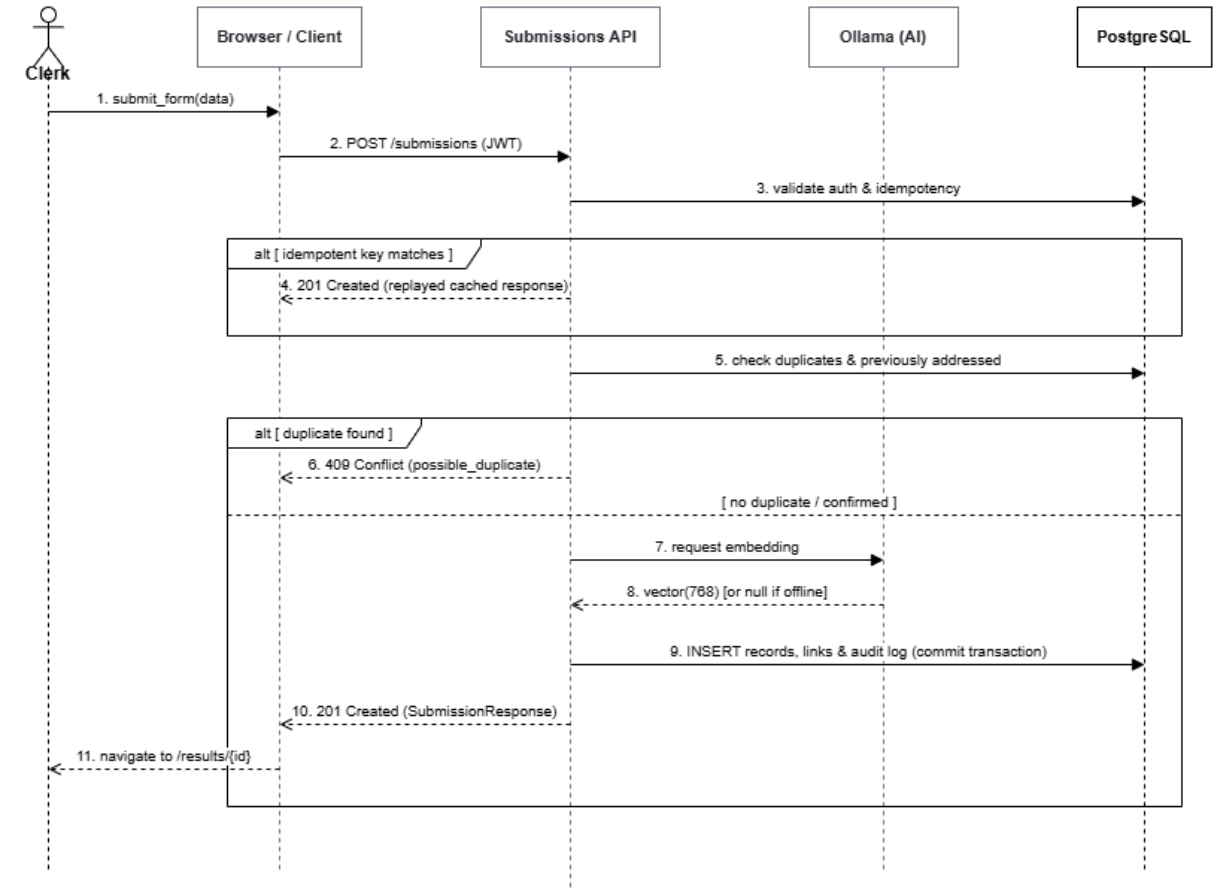


Figure 5.2: Sequence diagram: creating a question or notice of motion

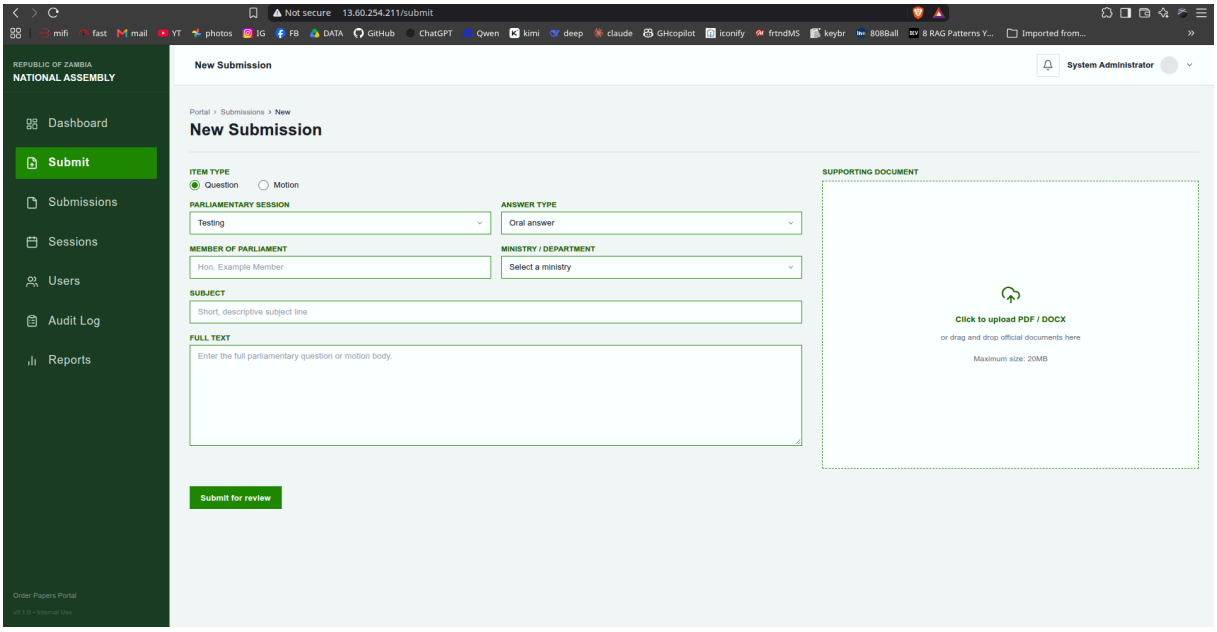


Figure 5.3: submission screen

Otherwise the record is embedded (if Ollama is reachable), created as Submitted, and moved to Under Review through the state service. An embed_record job and its outbox event are

written *in the same database transaction* as the record, which is the transactional outbox pattern [54]. Either the record and its indexing request are both committed, or neither is, so a record can never be left permanently unindexed. The router then links near-identical addressed items, writes the audit entry, stores the idempotent response and commits.

5.4.2. Lifecycle Enforcement

Every status change in the codebase goes through one function:

```
'''Python
def transition_submission(submission, target_status: SubmissionStatus) → None:
    current_status = SubmissionStatus(submission.status)
    target_status = SubmissionStatus(target_status)
    if target_status not in ALLOWED_TRANSITIONS[current_status]:
        raise InvalidStatusTransition(
            f"Cannot transition submission from {current_status.value} "
            f"to {target_status.value}")
    submission.status = target_status.value
'''
```

Workflow actions are declared as a table that maps each action to its permission and target state ("Approve": ("approve_motion", APPROVED), "Reject": ("reject_submission", REJECTED), "Request Changes": ("request_changes", DRAFT)). The router locks the record row (SELECT ... FOR UPDATE) before transitioning it, so two clerks acting at the same moment produce exactly one valid transition. A dedicated concurrency test checks this (Section 6.3). Only an authorized user can edit a returned Draft (PATCH /submissions/{id}), such as a clerk updating details on behalf of an MP, and a changed subject or text is re-embedded and re-queued. Resubmission moves the record from Draft to Submitted to Under Review.

5.4.3. Scheduling, Order Paper, Responses and Archiving

Scheduling (schedule_item) requires the sitting date to fall within the record's session and the record to be Approved. It then moves the record to Scheduled. GET

/order-papers/{sitting_date} selects the Scheduled records for that date, orders them by creation time and UUID so the output is deterministic, and returns a document titled "NATIONAL ASSEMBLY OF ZAMBIA ORDER PAPER" with a QUESTIONS section and a NOTICES OF MOTION section.

Recording a response (record_response) is handled by ResponseRecordingService. It verifies that the record is a Scheduled question with no existing response, inserts the question_responses row and moves the record to Answered through an injected StatusTransitioner, all within one transaction (FR-040 and FR-041). archive_ended_session_records() moves every non-archived record of a Closed or ended session to Archived. It is idempotent and runs at API start-up.

5.5. Duplicate and Previously-Addressed Detection

Detection is a two-stage algorithm implemented by DuplicateDetectionService and PreviouslyAddressedService (Figure C.3 in Appendix C and Figure 5.4). Figure 5.5 shows duplicate detection as it appears on the front end.

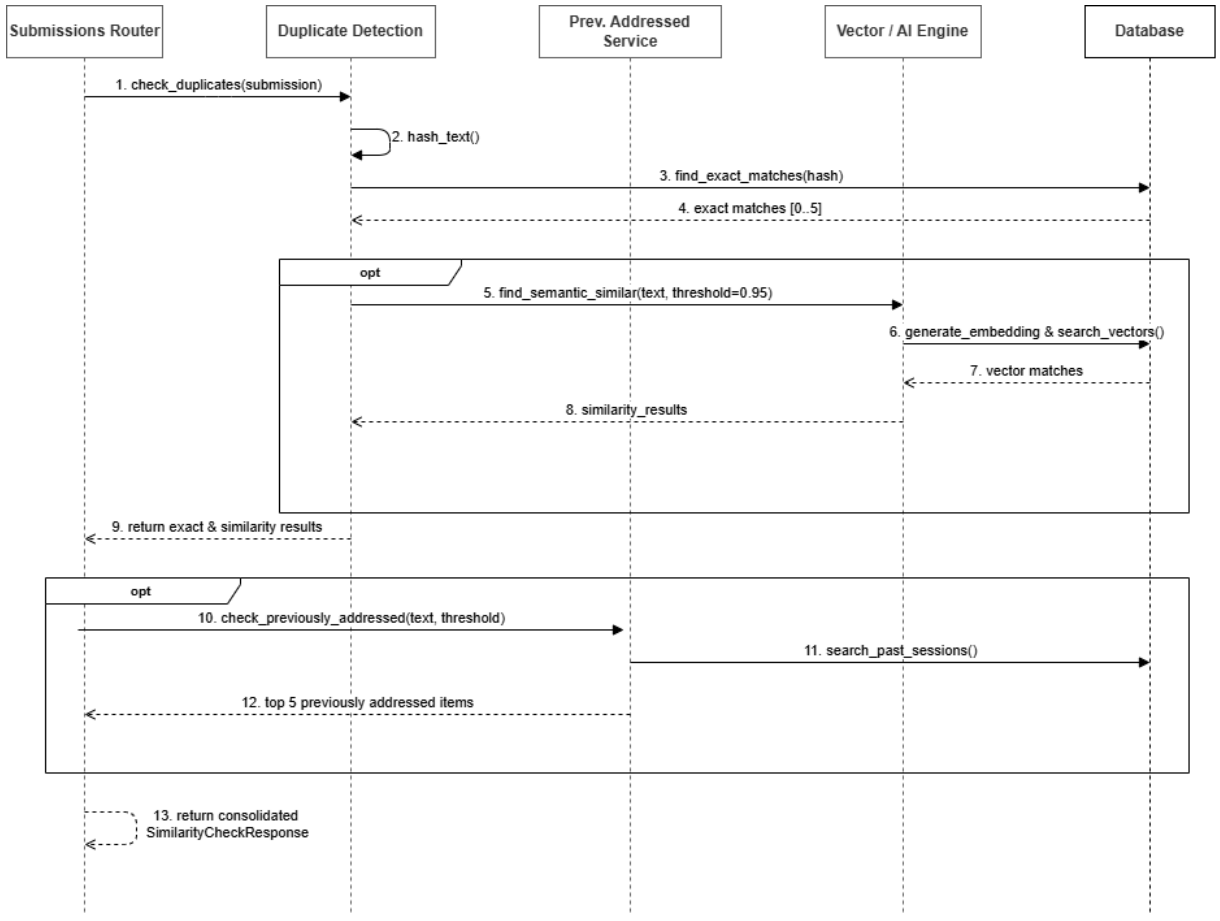


Figure 5.4: Sequence diagram: duplicate and previously-addressed detection

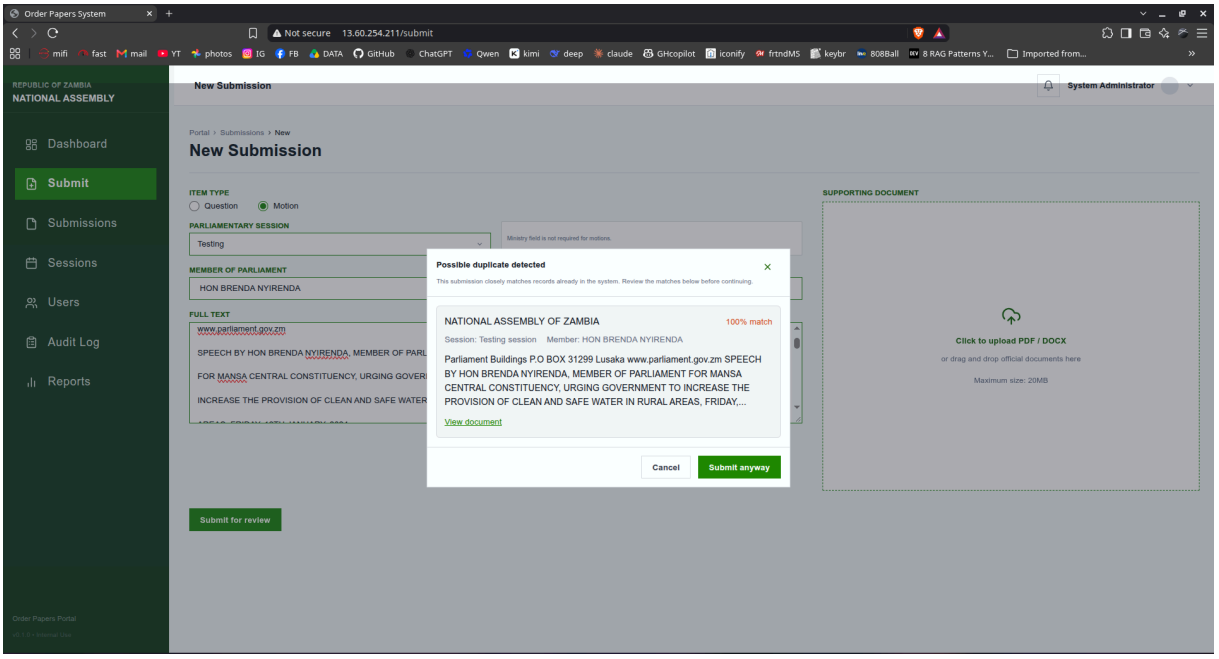


Figure 5.5: Duplicate detection on the front end

Stage 1: exact matching. The item type, subject and full text are each normalised with Unicode NFKD decomposition, lower-casing, replacement of non-alphanumeric characters with spaces and whitespace collapsing. They are joined with a separator and hashed with SHA-256. A SQLAlchemy before_insert/before_update hook stores this normalized_hash on every record and increments version when the content changes. An exact duplicate is therefore found with an indexed equality lookup, and it is not affected by capitalisation, punctuation or spacing. Such matches score 1.0.

Stage 2: semantic matching. If fewer than five exact matches are found, the subject and text are embedded and PgVectorSimilarityBackend runs a cosine-distance query:

```
'''Python
        distance = ParliamentaryRecord.embedding.cosine_distance(query_embedding)
        query = (select(ParliamentaryRecord.id, distance.label("cosine_distance"))
                 .where(ParliamentaryRecord.embedding.is_not(None))
                 .where(distance ⩽ 1 - threshold))
        ...
        score = max(0.0, min(1.0, 1.0 - float(row.cosine_distance)))
  '''
```

Records whose cosine similarity is at least DUPLICATE_SIMILARITY_THRESHOLD (0.95) are added as semantic matches. Exact matches take precedence, and the combined list is sorted by score and truncated to five. The result carries the threshold and its version (2026-08-05-v1), so every decision can be traced to the policy in force at the time.

Previously addressed. PreviouslyAddressedService searches only records whose status is Answered or Discussed, with the lower threshold of 0.72. It over-samples by a factor of three and then keeps only records of the same type from sessions that started *before* the current one, which prevents records of the current session from being reported as "previously addressed".

Automatic linking. RelatedItemLinkService stores a directional link in related_item_links only when a previously-addressed match reaches 0.99, and it skips links that already exist, so the operation is idempotent. A lower-confidence match is shown to the clerk but never linked automatically.

Graceful degradation. If the embedding service raises an error, the backend logs a warning and returns no semantic matches, so exact matching still protects the submission. The thresholds are configuration values, and scripts/evaluate_similarity_thresholds.py can derive them from clerks' historical decisions: it splits the decisions in time order, 80% for training and 20% for validation, and picks the lowest threshold that meets a target precision (0.95 for duplicates and 0.98 for automatic links).

5.6. Hybrid Retrieval

Search, related-record lookup and submission candidates all use AISimilarityService, which fuses a lexical retriever and a semantic retriever (Figure C.2 in Appendix C and Figure 5.6). This retrieval pipeline, described in full below, is the complete extent of this project's delivered artificial intelligence contribution; it produces ranked results only and does not generate natural-language text.

5.6.1. Lexical Retrieval

PostgresLexicalRetriever runs entirely in PostgreSQL. The weighted search_vector is a generated column defined as:

```
'''SQL
setweight(to_tsvector('english', coalesce(subject, '')), 'A') ||
setweight(to_tsvector('english', coalesce(full_text, '')), 'B') ||
setweight(to_tsvector('english', coalesce(member, '')), 'C') ||
```

```
setweight(to_tsvector('english', coalesce(ministry, '')), 'C')
```

A query is parsed with websearch_to_tsquery('english', q), matched with @@ against the GIN index, and ranked with ts_rank_cd. Visibility rules and the session, type, status, date, member and Ministry filters are applied *before* ranking and pagination, so page boundaries are stable and hidden rows never affect ranks.

5.6.2. Chunking and Semantic Indexing

Long records are split by chunk_record() into overlapping chunks of at most 1,800 characters with about 240 characters of overlap. Chunks break on word boundaries, and each chunk is prefixed with Subject: ... so that every passage carries the record's topic. ChunkIndexingService.index_record() compares the desired chunks, identified by chunk index and SHA-256 content hash, with the chunks already stored for the *same model name, model digest, dimension and preprocessing version*. It deletes stale chunks, embeds only the missing ones in a single batch request, and checks that every vector has 768 dimensions. Re-indexing is therefore incremental and idempotent, and upgrading the embedding model never mixes vectors from different model versions.

5.6.3. Semantic Retrieval

OllamaEmbeddingProvider calls Ollama's batch /api/embed endpoint. It retries up to twice with exponential back-off, and after repeated failures it opens a circuit breaker for 30 seconds, so a failing model server does not slow every request down. PgVectorSemanticRetriever embeds the query, keeps only chunks that match the active model digest and preprocessing version, applies the same visibility filters, and uses DISTINCT ON (record_id) to keep each record's best-matching chunk before ranking by cosine distance. Each result includes the matching excerpt, which the interface shows as evidence.

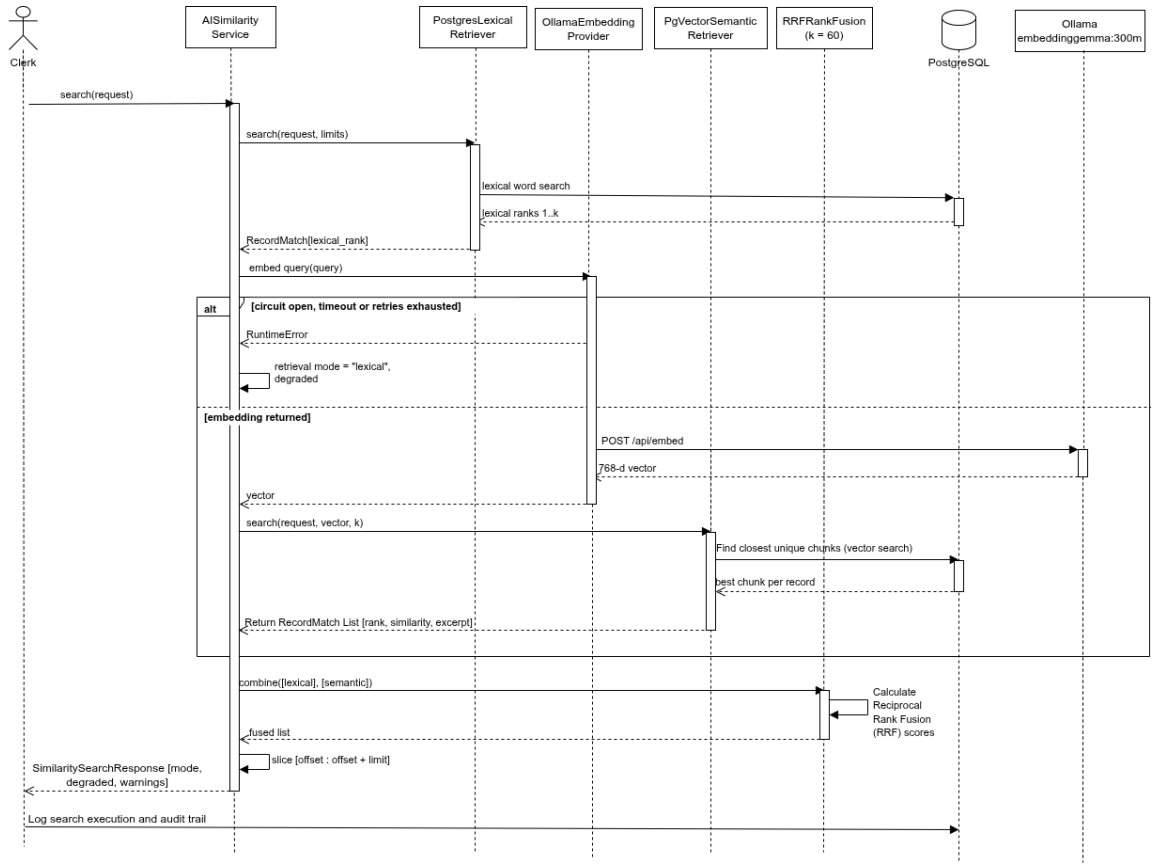


Figure 5.6: Sequence diagram: hybrid retrieval with lexical fallback

5.6.4. Reciprocal Rank Fusion

RRFRankFusion combines the two ranked lists [10]. For a record *d* with rank r_i(d) in list *i*, its score is

score(d) = Σ_i 1 / (k + r_i(d)), with k = 60,

and the score is divided by the maximum possible value (rank 1 in every list) so that it lies between 0 and 1. The fused list keeps both source ranks, the cosine similarity and the lexical score for display. If semantic retrieval fails, the service returns lexical results only, with retrieval_mode = "lexical", degraded = true and a warning, so search never fails because the embedding service is down. POST /search uses hybrid mode by default, and mode = "keyword" selects lexical-only ranking with exact total counts.

5.7. Background Processing and Operational Readiness

The worker (python -m app.jobs.worker) implements the job life cycle in Figure 5.7. enqueue_job() inserts with ON CONFLICT (deduplication_key) DO NOTHING, so enqueueing the same logical job twice is harmless. claim_jobs() selects due jobs with FOR UPDATE SKIP LOCKED, which lets several workers run safely, and gives each a

five-minute lease. A failed job is retried after min(2^n, 3600) seconds and is moved to dead_letter after five attempts. requeue_dead_letter_job() requires an operator-supplied reason and keeps the recovery history in the job payload. Stale embedding jobs, whose content hash no longer matches the record, complete without overwriting newer vectors. The job types this report's evaluated features use are embed_record, reindex_record, index_record_chunks and archive_records; a further generate_explanation job type is reserved in the queue for the future explanation phase (Section 7.6) and is not otherwise discussed here.

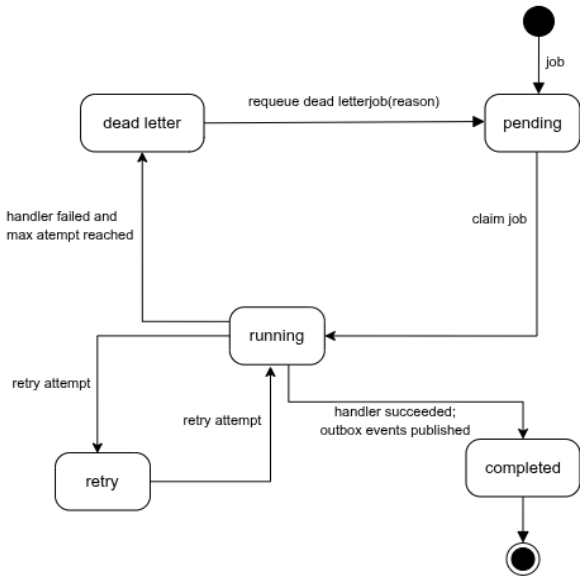


Figure 5.7: State machine of a durable background job

The worker records a heartbeat on each poll. /livez and /health report only that the process is alive. /readyz returns HTTP 503 unless the database is reachable, the schema is at the Alembic head, Ollama is reachable with the required embedding model loaded, and no job is dead-lettered. Docker Compose uses this to gate frontend startup. A request middleware assigns or propagates an X-Request-ID, and every request is logged as structured JSON with its method, route, status and duration.

5.8. Document and Data Ingestion

POST /submissions/upload pre-fills the submission form from a PDF, DOCX or TXT file. Because uploaded files are untrusted, the parser applies several layers of defence:

- The file is streamed to a temporary file with a 10 MB limit.
- The file extension, declared MIME type and magic bytes must all agree.

- DOCX archives are checked against limits on entry count (500), uncompressed size (50 MB) and compression ratio (200:1), which guards against archive bombs.
- PDFs are limited to 200 pages and extracted text to 2,000,000 characters.
- Parsing runs in a separate process with memory (RLIMIT_AS) and CPU-time (RLIMIT_CPU) limits and a 15-second timeout.

The first meaningful line becomes the subject and the rest becomes the full text.

Bulk historical data is loaded with scripts/ingest_csv.py, which maps session codes to sessions and normalises blank or legacy statuses to Archived. scripts/reindex_records.py and POST /ai/reindex enqueue chunk indexing for existing records in batches, and GET /ai/index/coverage reports the share of eligible records indexed under the active model.

5.9. Frontend Implementation

The frontend uses the Next.js App Router with two route groups: (auth) for the login page and (dashboard) for the protected area. Its pages are the dashboard, Submit, Search, Record detail (results/[id]), Reports, Audit, Sessions, Users and Notifications. The dashboard layout is a server component that validates the session against /auth/me before rendering. The sidebar shows only the destinations the user's permissions allow.

All data access goes through same-origin route handlers built on backendProxy.js, which attaches the bearer token from the cookie, returns HTTP 502 if the API is unreachable, and turns FastAPI validation errors into readable messages. Forms use React Hook Form with Zod schemas that mirror the backend rules. The record detail page brings together the record, the ranked similar records with their retrieval evidence (lexical rank, semantic rank, cosine similarity), the similarity decision form, the permission-aware workflow actions, draft editing and scheduling.

The interface was first built with Tailwind CSS and then migrated to plain CSS (globals.css). This removed a build dependency, and a source test prevents Tailwind from being reintroduced.

5.10. Deployment and Operations

`docker-compose.yml` defines five services: db `(pgvector/pgvector:pg16)`, migrate (a one-shot Alembic upgrade), backend (Uvicorn), worker and frontend. Health checks gate the start-up order. The API and worker reach Ollama on the host through

host.docker.internal:11434, and Ollama is bound to the Docker bridge rather than the public network. For production, docker-compose.ec2.yml requires every secret through environment variables (`DATABASE_URL`,`JWT_SECRET`,`FRONTEND_ORIGIN`, `OLLAMA_BASE_URL`), removes all published ports except the frontend on 127.0.0.1:3000, enables secure cookies, and uses immutable image tags. The runbook (`docs/operations/ec2-deployment.md`) covers TLS termination, running migrations before traffic, rollback, daily encrypted pg_dump backups copied off the host, and smoke checks against `/readyz`.

5.11. Design Evolution

Several capabilities were built, evaluated and then deliberately retired. They are recorded here because they show how the design decisions were reached and because they can be reinstated.

Provider-agnostic email notifications (FR-031). On 24 July 2026 the team implemented status-change email notifications in four TDD steps.

- An `EmailProvider` abstract base class with one asynchronous send_email(to, subject, body, html_body) method.
- An SMTP implementation with TLS and timeouts.
- A factory that picks the provider from the `EMAIL_PROVIDER` setting through a registry of builder functions, so vendor-specific code stays inside app/email.
- A `NotificationService` that sends emails after scheduling, workflow and resubmission changes in background tasks, isolated from transaction rollback and provider failures.

The open-closed design meant a new provider (for example a government mail relay or an SMS gateway) could be added by registering a builder, without touching business code. The optional users.email column added by migration is still in the schema. In the deployment-readiness iteration of August the email package, the notification service and their tests were removed. The recovered design is proposed for reinstatement in Section 8.3 once NAZ provides a mail relay.

5.12. Implementation Challenges and Solutions

Table 5.2 summarises the main technical challenges and how they were resolved, with the commits that record each fix.

Table 5.2: Implementation challenges and solutions

Challenge	Solution
In-process BM25 loaded and ranked every candidate in Python, which does not scale to a multi-session archive.	Lexical retrieval moved into PostgreSQL with a generated weighted tsvector, GIN index and ts_rank_cd; the old module was removed.
The first embedding design used 384-dimensional vectors, which did not match the chosen EmbeddingGemma model.	Migration to vector(768) with model tracking; later chunk-level vectors tagged with model digest and preprocessing version, with dimension checks on every vector.
Mixing vectors from different model versions would silently corrupt similarity scores.	Semantic queries filter chunks by exact model digest, dimension and preprocessing version; index coverage is reported per model.
Containers could not reach Ollama on the host, and exposing it publicly would be unsafe.	Ollama is bound to the Docker bridge; containers use host.docker.internal; readiness requires the embedding model.
Runtime DDL at start-up competed with the SQL schema file and drifted between environments.	Alembic made the single schema authority, run once under an advisory lock; readiness verifies the head revision.
Concurrent clerk actions and client retries could double-apply transitions.	Row locks, transactional flushes that map integrity errors to HTTP 409, and idempotency keys protected by a PostgreSQL advisory lock.
Rendering the current date on the server caused React hydration mismatches, and visual snapshots changed from day to day.	Client-only date rendering; a fixed browser clock in Playwright tests.
Hard-coded role names blocked NAZ from reconfiguring access.	Permission-based RBAC stored as data, with legacy names mapped at start-up.
AI indexing had to be completed even if Ollama was unavailable when a record was saved.	Transactional outbox and durable worker with retries and dead-lettering.

6. Testing

This chapter describes how the system was verified. It covers the testing strategy (Section 6.1), the test environment (Section 6.2), the backend suite (Section 6.3), representative test cases (Section 6.4), frontend and browser tests (Section 6.5) and non-functional tests (Section 6.6). It closes with a summary of results (Section 6.7) and the known gaps in testing (Section 6.8). The evaluation of the system against its objectives follows in Chapter 7.

6.1. Testing Strategy

The test suite is layered as follows.

- **Unit and contract tests** check pure logic and abstract contracts in isolation. Examples are RRF scoring, text normalisation, chunking and the lifecycle graph. External services are replaced with fakes, for example a fake HTTP client standing in for Ollama.
- **API integration tests** drive the real FastAPI application through TestClient against a real PostgreSQL 16 database with pgvector. The database was deliberately not mocked, because much of the system's behaviour lives in SQL: full-text ranking, vector distance, CHECK constraints, row locks and SKIP LOCKED claiming.
- **Migration tests** upgrade empty and partly migrated databases with Alembic and check that no records are lost.
- **Frontend source guards** are fast Node tests that check architectural rules in the frontend source, for example that no Tailwind classes are used, that authorisation checks permissions rather than role names, and that the BFF routes exist.
- **End-to-end and visual tests** use Playwright to drive Chrome, log in, visit every page at fixed viewport sizes and compare full-page screenshots with approved baselines [50].
- **Performance and evaluation scripts** measure retrieval latency and quality (Chapter 7).

6.2. Test Environment

The backend tests are configured in backend/tests/conftest.py. The file points the application at a separate database, naz_order_papers_test, on the local PostgreSQL service, enables the vector extension, and rebuilds the schema before every test, so tests are independent and can run in any order. A client fixture overrides the request database session so that tests and API calls share one session. Authentication in tests uses real JWTs and real user_sessions rows. Roles and permissions are created per test, which also exercises the configurable RBAC.

The suite was run with pytest 8.3.5 on the development workstation, using Python 3.14 in the local virtual environment, whereas the container image uses Python 3.11. The command was:

```python
PYTHONPATH=backend .venv/bin/pytest -q -rA --durations=10 backend/tests
```

6.3. Backend Test Suite

The backend suite contains 224 tests in 44 modules. Table 6.1 groups them by functional area, matching this report's evaluated scope. A small number of these tests (in test_grounded_explanations.py and part of test_ai_endpoints.py) exercise the future-phase generative explanation prototype introduced in Section 7.6 rather than any feature described in Chapter 5; they are counted in the totals below for an honest suite count, but that prototype is not otherwise described or evaluated in this report.

Table 6.1: Backend test suite by area (224 tests, all passing)

Area	Test modules	Tests	What is verified
Authentication and access control	test_authentication, test_account_lockout, test_permissions, test_report_permissions, test_archive_access	16	Generic credential errors, lockout on the fifth failure, permission-guarded unlock, role inheritance, custom roles without code change, archive permissions.
Submission and validation	test_question_submissions, test_motion_submissions, test_submission_validation, test_submission_ownership, test_document_upload	49	Oral/written questions, motions, every required field, whitespace-only rejection, ownership and draft privacy, offline-embedding submission, idempotency, and 20 upload-safety cases.
Workflow and lifecycle	test_review_queue, test_workflow_reviews, test_submission_status, test_scheduling, test_responses,	40	Queue routing, action permissions, the exact state set and transitions, concurrency, scheduling rules,

Area	Test modules	Tests	What is verified
	test_auto_archiving, test_order_papers		atomic response recording, archival, Order Paper grouping and order.
Duplicate and similarity	test_duplicate_detection, test_previously_addressed, test_related_items, test_similarity_contract, test_similarity_normalization, test_similarity_results_display, test_pgvector_similarity_backend, test_pgvector_migration	29	Exact and near-duplicate detection, the 409 gate and override, earlier-session rules, automatic linking thresholds, hash normalisation, time-separated threshold evaluation, pgvector ordering and offline behaviour, and the session management endpoints.
Search and hybrid retrieval	test_keyword_search, test_postgres_lexical_search, test_record_filters, test_ai_hybrid_retrieval, test_ai_hybrid_search_end_to_end, test_ai_degraded_search, test_chunk_indexing, test_search_benchmark, part of test_ai_endpoints	37	PostgreSQL ranking and filters before pagination, stable pagination, RRF properties, hybrid fusion, chunk collapsing, lexical fallback, incremental and version-safe indexing, benchmark safety, AI-search response shape, and AI-suggestion review recording and its idempotency.
Embeddings and AI documentation	test_ai_embeddings, test_ai_documentation	10	768-dimension checks, batching, circuit breaker, and that the project's AI documentation matches the implemented routes and settings.
Reliability and operations	test_background_jobs, test_health, test_migration_authority, test_observability, test_csv_ingestion, test_session_report, test_notifications	34	Outbox atomicity, single claim across workers, lease expiry, back-off and dead-lettering, readiness failures, migration locking, request IDs, log redaction, CSV status normalisation, session reports, activity feed.
Future-phase prototype (not evaluated in this report)	test_grounded_explanations, part of test_ai_endpoints	9	Safety properties of the unevaluated generative explanation prototype (Section 7.6): permission and archive-visibility checks, evidence sanitisation, mandatory review, and

Area	Test modules	Tests	What is verified
			asynchronous job polling.

6.4. Representative Test Cases

Table 6.2 presents representative test cases, drawn directly from the suite, for the hybrid retrieval, duplicate-detection and workflow features this report evaluates.

Table 6.2: Representative test cases and results

ID	Test (module)	Input / action	Expected result	Result
TC-01	test_invalid_password_returns_same_generic_credentials_error (authentication)	Valid employee ID, wrong password	401 with "Invalid employee ID or password." (same as unknown ID)	Pass
TC-02	test_locked_account_rejects_correct_password (account_lockout)	Five wrong passwords, then the correct one	Account locked; correct password still rejected	Pass
TC-03	test_custom_role_and_permission_require_no_business_logic_change (permissions)	New role granted a permission through data only	Access granted without code change	Pass
TC-04	test_missing_required_field_returns_clear_validation_error[Question-ministry] (submission_validation)	Question without Ministry	422 with a readable field message	Pass
TC-05	test_create_returns_409_for_possible_duplicate_before_saving (duplicate_detection)	Submission matching an existing record	409 possible_duplicate; nothing persisted	Pass
TC-06	test_duplicate_confirmation_override_allows_submission (duplicate_detection)	Same submission with confirm_duplicate = true	201 Created	Pass
TC-07	test_unanswered_or_current_session_items_are_not_previously_addressed (previously_addressed)	Similar items that are unanswered or in the current session	Not reported as previously addressed	Pass
TC-08	test_concurrent_workflow_actions_produce_one_valid_transition (workflow_reviews)	Two threads approve the same record simultaneously	Outcomes [201, 409]; one decision row	Pass

ID	Test (module)	Input / action	Expected result	Result
TC-09	test_only_approved_item_can_be_scheduled (scheduling)	Schedule an Under Review item	409; status unchanged	Pass
TC-10	test_response_and_status_transition_are_atomic_on_forced_failure (responses)	Force a failure during the status change	No response row and no status change	Pass
TC-11	test_hybrid_search_degrades_to_lexical_when_embedding_provider_fails (ai_hybrid_search_end_to_end)	Embedding provider raises an error	Lexical results with degraded = true and a warning	Pass
TC-12	test_stale_model_vectors_are_excluded_and_chunk_hits_collapse (chunk_indexing)	Chunks from an old model digest present	Old vectors ignored; one hit per record	Pass
TC-13	test_hybrid_search_record_matching_both_modalities_ranks_first (ai_hybrid_search_end_to_end)	A record matches both the lexical and the semantic branch	It is fused to the top rank by RRF	Pass
TC-14	test_ollama_embedding_provider_opens_circuit_after_bounded_retries (ai_embeddings)	The embedding endpoint fails repeatedly	Retries are bounded; the circuit opens for 30 s	Pass
TC-15	test_two_workers_cannot_claim_the_same_job (background_jobs)	Two workers claim concurrently	Each job claimed exactly once	Pass
TC-16	test_dead_letter_jobs_fail_readiness_and_are_visible (health)	A job in dead_letter	/readyz returns 503 with the count	Pass
TC-17	test_reject_docx_archive_bomb (document_upload)	DOCX with an extreme compression ratio	Rejected before extraction	Pass
TC-18	test_migration_lock_rejects_a_concurrent_runner (migration_authority)	Second migrator while the lock is held	MigrationAlreadyRunning raised	Pass

The concurrency test (TC-08) shows how database behaviour was tested directly rather than mock.

The degradation test (TC-11) confirms the fail-safe design of Section 3.5: when the embedding provider is unavailable, hybrid search still returns lexical results rather than failing the request outright, and the response is marked so the caller knows retrieval was degraded.

6.5. Frontend and Browser Tests

Source guards. The Node test runner (npm run test:source) runs 21 tests in 19 files. They check that:

- Tailwind has been fully removed;
- the frontend never authorises by role name;
- the frontend does not access the database directly and no retired search prototype remains;
- submission types are derived from permissions;
- question fields and payloads are mapped correctly;
- validation errors are formatted readably;
- the BFF routes exist for the review queue, workflow actions, ownership tracking, draft recovery, scheduling, the Order Paper and document upload;
- the seven original status filter values are present;
- the date, member and Ministry filters are forwarded;
- the top bar renders without hydration errors and offers notifications and logout;
- Visual tests use a fixed clock.

All 21 tests passed in 0.98 s.

End-to-end and visual tests. The Playwright suite (frontend/tests/e2e/visual-baseline.spec.js) runs in Chrome with the en-ZM locale and the Africa/Lusaka time zone, one worker, and zero tolerance for pixel differences. It contains 14 scenarios:

- login at desktop and mobile sizes;
- seven authenticated pages (dashboard, submit, search, reports, sessions, users, notifications);
- record detail;
- audit, with dynamic rows masked;
- the mobile dashboard;
- three behavioural checks: a keyword that appears only in a record's full text is found and ranked, search filters stay collapsed until requested and can be reset, and the top bar opens notifications and logout.

The last recorded run, on 12 August 2026, passed against 12 approved baseline images. The interface layout was changed on 16 August 2026 (commit 16d5563) and has further uncommitted style changes, so the visual baselines must be reviewed and regenerated before the suite is used as a regression gate again (Section 6.8).

6.6. Non-Functional Testing

- **Security.** Authentication, lockout, permission and visibility tests (Section 6.3), and upload-safety tests (20 cases, including spoofed MIME types, encrypted PDFs, excessive page counts and archive bombs).
- **Reliability.** Outbox atomicity on rollback, single job claims across workers, lease-expiry recovery, exponential back-off, dead-lettering and manual recovery, and readiness failure when the database, schema, Ollama or queue is unhealthy.
- **Maintainability.** Migration-authority tests confirm that Alembic is the only schema manager, that the application runs no DDL at start-up, and that a database at revision 0006 upgrades without data loss. A documentation test checks that the project's AI documentation matches the implemented routes and settings.
- **Performance.** test_search_benchmark checks the benchmark's percentile calculation and its refusal to run on a non-isolated database. The full 190-record benchmark is reported in Section 7.4.

6.7. Summary of Results

Table 6.3 summarises the verification results obtained for this report. Figure 6.1 shows the same counts as a bar chart.

Table 6.3: Summary of verification results

Suite	Command	Tests	Result	Duration
Backend integration and unit tests	pytest -q backend/tests	224	224 passed, 0 failed (5 deprecation warnings)	199.5 s
Frontend source guards	npm run test:source	21	21 passed, 0 failed	0.98 s
Browser end-to-end and visual	npm run test:e2e	14 scenarios	Passed (last recorded run, 12 Aug 2026)	—
Lexical search benchmark (190 records)	python -m scripts.benchmark_search	40 timed queries	Passed (p95 266.5 ms against a 3,000 ms budget)	—

Figure 6.1: Verification results by test suite

The five warnings come from framework deprecations, mainly the Starlette test client's use of httpx. Log lines marked ERROR during the run came from tests that deliberately caused failures (for example a 384-dimension vector returned to a 768-dimension provider, and a job driven to dead_letter). They show that the error handling was exercised, not that tests failed.

6.8. Testing Limitations

The limitations of the testing are discussed in Section 7.6.

7. Results and Evaluation

7.1. Functional Completeness Against Requirements

Section 4.3 listed 23 functional requirements drawn from NAZ's prioritised backlog. Of these, 22 are implemented and verified by the automated suite in Chapter 6; FR-031 (email notification) was implemented, verified and then deliberately retired in favour of the in-application activity feed, for the reasons given in Section 5.11. One gap remains at the interface layer: FR-012 (duplicate blocking) is enforced by the API, but the Submit page does not yet display the returned matches or expose the confirm_duplicate override, so a genuine duplicate can currently only be resubmitted through the API directly. This is listed as an open item in Section 8.2.

Specific objective 2 (Section 1.3.2), an end-to-end workflow secured by permission-based access control and a full audit trail, is met: every one of the nine lifecycle states, all 15 permissions and the audit log are exercised by the 224 backend tests (Section 6.3), including the concurrency test that proves two simultaneous workflow actions produce exactly one valid transition (TC-08).

7.2. Retrieval Effectiveness

7.2.1. Method

To answer research question 3 (Section 1.3.3), the team ran a controlled experiment against the live system components: PostgresLexicalRetriever, PgVectorSemanticRetriever and AISimilarityService (hybrid), using the real embeddinggemma:300m model through Ollama and a real PostgreSQL 16 database with pgvector. The 24-record synthetic corpus (Appendix E) was inserted and chunk-indexed inside one transaction, which was rolled back at the end of the run, so the experiment left no trace in any persistent database. For each of the 18

evaluation queries, a paraphrase of exactly one target record that deliberately avoids the target's distinctive nouns (Section 3.9), the team recorded the rank of the target record in the top 10 results returned by each retrieval mode. Recall@k and Mean Reciprocal Rank (MRR@10) were computed as defined in Section 2.7 [29].

7.2.2. Results

Table 7.1: Retrieval effectiveness by mode, 18 paraphrased queries against the 24-record corpus

Mode	Recall@1	Recall@3	Recall@5	MRR@10	Queries with no hit	Latency (median / max, ms)
Lexical (PostgreSQL full-text)	0.000	0.000	0.000	0.000	18 / 18	3.0 / 8.9
Semantic (pgvector, embeddinggemma)	1.000	1.000	1.000	1.000	0 / 18	349.3 / 440.3
Hybrid (RRF fusion)	1.000	1.000	1.000	1.000	0 / 18	370.5 / 455.2

The result is stark by design: because every query was deliberately written to avoid the target record's key nouns, lexical search matched none of the 18 targets in the top 10 at all, which is exactly the vocabulary-mismatch failure mode described in Section 2.3 [18]. Semantic retrieval ranked the correct target first for every one of the 18 queries, and hybrid retrieval, which fuses semantic with an empty lexical contribution through RRF, preserved that result: fusing with a retriever that found nothing did not demote any correct answer. This experiment intentionally isolates the paraphrase case; it says nothing about queries that share exact terminology with a target, where Section 2.3 and the system's own design (Section 5.6.1) expect lexical retrieval to be fast and precise. The corpus is also small and deliberately curated, so a recall of 1.0 is an upper bound achievable on a clean, well-separated set of 24 records, not a claim about production-scale accuracy on the real NAZ archive, which will contain many more near-topic records and noisier language. Figure 7.1 shows the same results as a bar chart.

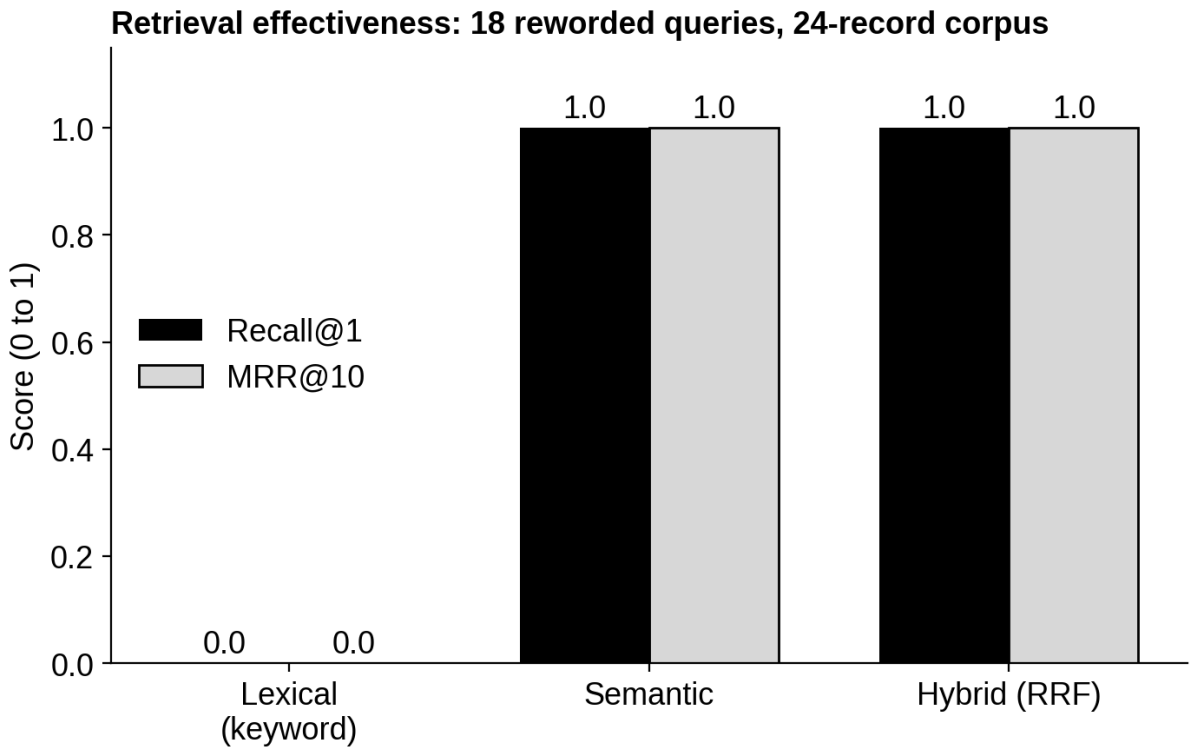


Figure 7.1: Retrieval effectiveness (Recall@1 and MRR@10) by retrieval mode

Latency is consistent with the architecture (Section 5.6): lexical retrieval is a single indexed SQL query and returns in a few milliseconds, while semantic and hybrid retrieval each require one embedding call to Ollama before the vector query, which dominates their latency even on this tiny corpus. This retrieval-only latency, well under half a second, also stands in useful contrast to the multi-minute latency that this report's self-critique (Section 7.6) notes for the unevaluated generative prototype: hybrid retrieval alone is fast enough for interactive use on the same CPU-only hardware, which is one reason it is the delivered feature and the generative layer is not. Figure 7.2 compares the median and maximum latency of the three retrieval modes.

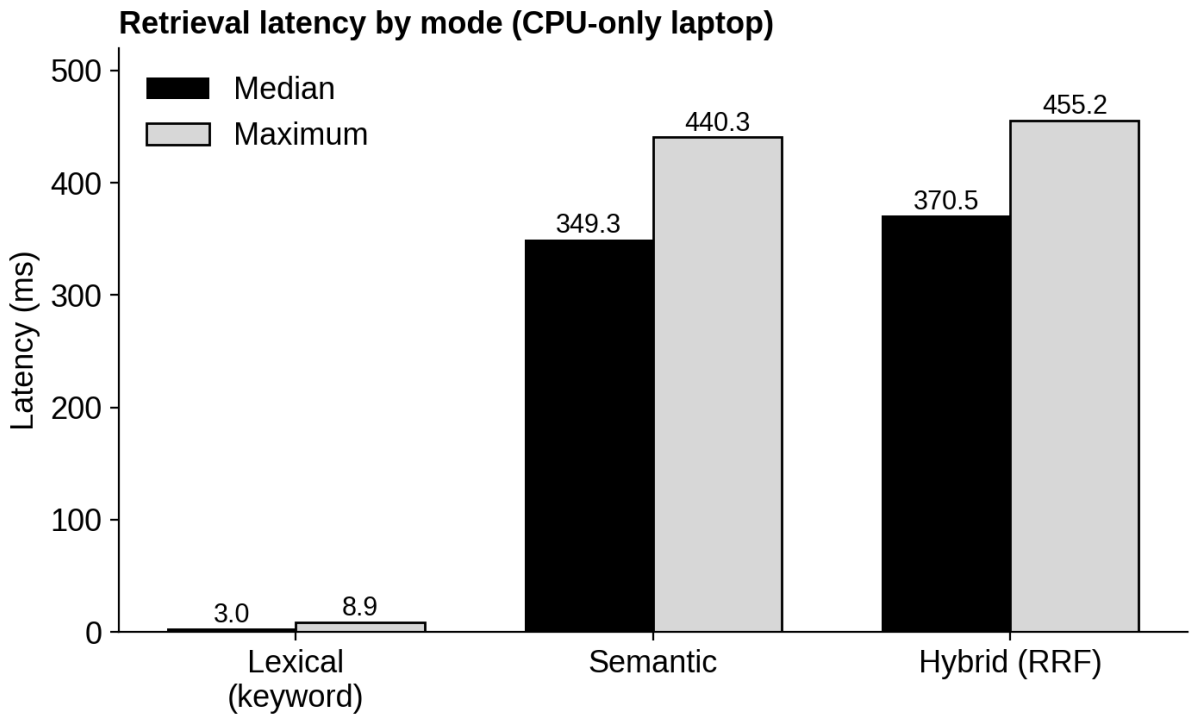


Figure 7.2: Retrieval latency (median and maximum) by retrieval mode

7.3. Duplicate and Previously-Addressed Threshold Analysis

The same experiment recorded the record-level cosine similarity between each paraphrased query and its true target, and the similarity to the best-matching *incorrect* record, using the exact code path in PgVectorSimilarityBackend (Section 5.5).

Table 7.2: Cosine similarity of paraphrased queries to their true target versus the best non-target match

Quantity	Minimum	Median	Maximum
Similarity to the true (paraphrased) target	0.572	0.642	0.748
Similarity to the best non-target record	0.296	0.397	0.503

Two findings follow directly from these numbers. First, the target outranked every non-target record in all 18 cases (a difference of at least 0.07 in every case), which is why semantic Recall@1 was perfect (Section 7.2): the separation between a true paraphrase and an unrelated record is wide on this corpus. Second, none of the 18 paraphrases reached the DUPLICATE_SIMILARITY_THRESHOLD of 0.95, and only 3 of the 18 reached the PREVIOUSLY_ADDRESSED_THRESHOLD of 0.72. This is evidence, not a flaw: 0.95 is calibrated for near-identical text (Section 5.5), and a paraphrase that changes most of the wording is correctly *not* treated as an exact duplicate. It does mean, however, that a genuinely

repeated matter phrased very differently from the original could still pass the submission-time duplicate gate undetected, and would depend on a clerk noticing it during manual review or hybrid search. This confirms the recommendation in Section 8.2 to calibrate both thresholds on clerks' actual similarity decisions once real data is available, rather than relying on the provisional values used during development.

The exact-match stage was also tested directly with two formatting probes (Appendix E): capitalisation, punctuation and an added exclamation mark applied to two existing records. Both were correctly identified as exact duplicates of their source record with a score of 1.0, confirming that the SHA-256 normalisation described in Section 5.5 is insensitive to superficial formatting differences, as intended.

7.4. Performance Benchmark

Objective 5 (Section 1.3.2) requires an assessment of response time. scripts/benchmark_search.py seeds 100,000 synthetic records into an isolated naz_search_benchmark database inside one transaction, which is rolled back after the measurement, and times 40 runs of eight realistic queries (after eight warm-up runs) against PostgresLexicalRetriever.

Table 7.3: Lexical search benchmark, 100,000-record corpus, 20 results per query

Metric	**Value**
Corpus size	190 documents
Measured runs	40 (after 8 warm-up runs)
p50 latency	201.5 ms
p95 latency	266.5 ms
p99 latency	295.2 ms
Maximum latency	295.2 ms
Budget (p95)	3,000 ms
Result	Passed

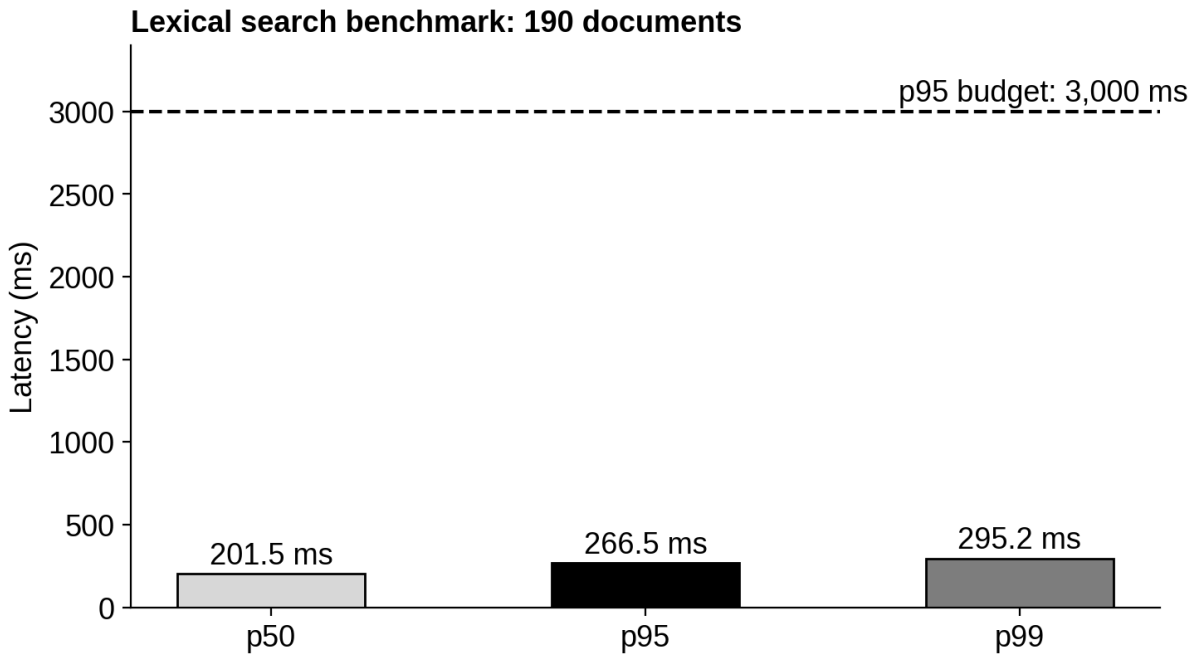


Figure 7.3: Lexical search benchmark latency against the 3,000 ms budget

Figure 7.3 plots the p50, p95 and p99 latencies against the 3,000 ms budget. The 95th-percentile latency is well inside the 3-second budget, with more than a 10x margin, which suggests PostgreSQL full-text search will comfortably support a NAZ archive many times larger than 100,000 records without index or hardware changes. This benchmark measures the lexical path only; because semantic retrieval adds one network round-trip to Ollama (Section 7.2, roughly 350 to 450 ms even on the tiny evaluation corpus), hybrid search latency at production scale should be re-measured once the real archive and its chunk index are in place, and is expected to be dominated by the embedding call rather than by PostgreSQL.

7.5. Usability Evaluation

Objective 5 also calls for a usability assessment, and a System Usability Scale (SUS) study [52] was carried out for this purpose. The study was conducted with fellow students, not with NAZ Table Office staff, because there was not enough time to arrange sessions with the Table Office. Formal user acceptance testing with NAZ staff therefore remains outstanding: it requires the data-sharing MoU to be signed and real records to be available for a realistic test (Section 7.6). Section 8.2 sets out the plan for this work during the handover phase, including a target SUS score of at least 68. The Google Form used to record the responses is shown in Figures D.1 and D.2 (Appendix D).

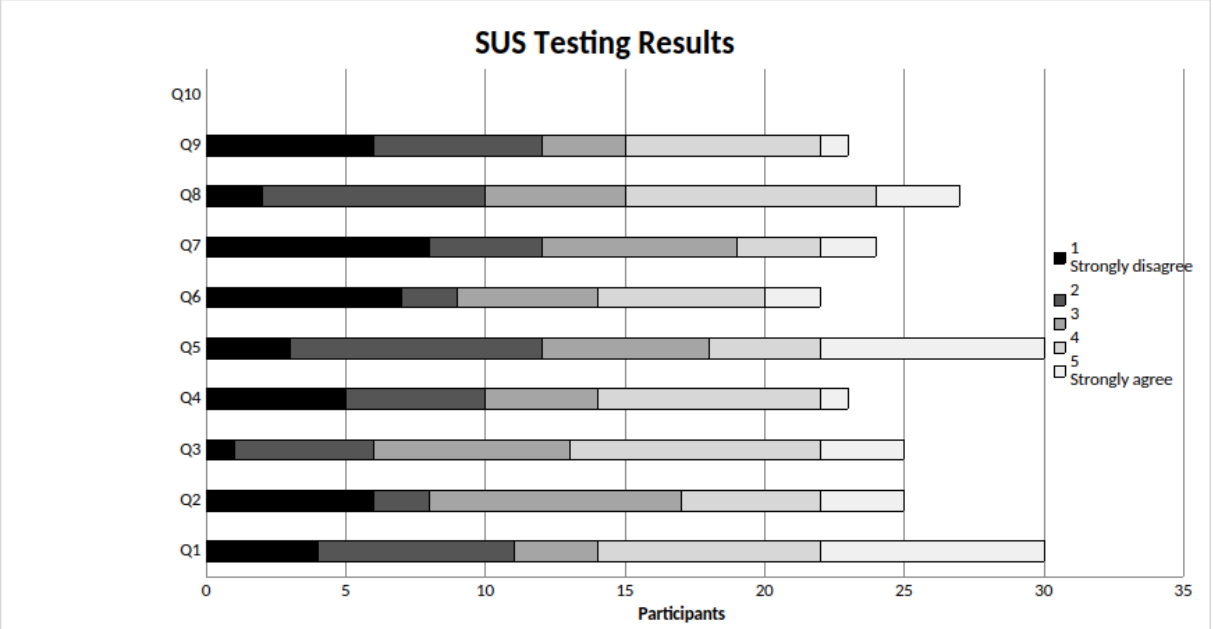


Figure 7.4: SUS results, distribution of responses to each statement

Figure 7.4 shows the responses to the ten SUS statements, scored from 1 (strongly disagree) to 5 (strongly agree). 30 students took part. The results show mixed, moderate usability, with responses spread across the whole scale. About half of the participants agreed that they would use the system often (Q1, about 53%), and about 40% found its functions well integrated (Q5), with many neutral responses. Learnability (Q7, quick to learn) and confidence in using the system (Q9) were weaker, with only about 20% to 35% agreeing. On the negative statements, roughly 40% agreed that the system was cumbersome (Q8) or that they would need technical support (Q4). These results suggest that ease of use and learnability need improvement. Because the participants were students and not Table Office clerks, the results indicate general usability only and do not show acceptance by the intended users.

7.6. Self-Critique and Honest Assessment

- **The retrieval experiment is favourable by construction.** The 190-record corpus was written by the team specifically to test the paraphrase case, and it is far smaller and cleaner than a real parliamentary archive. A recall of 1.0 should be read as "the mechanism works as designed on a controlled test," not as an expectation for production accuracy. The system has no automated evaluation yet against real, clerk-labelled duplicate pairs.
- **Hybrid retrieval has a single point of failure for its semantic half.** A lexical fallback exists and was tested directly (TC-11), so search never fails outright, but if Ollama is

unavailable for an extended period the system silently loses the paraphrase-matching capability that Section 7.2 shows is the whole point of the hybrid design, and falls back to keyword-only matching without an operator alert beyond the degraded-response flag. Production monitoring should watch for this explicitly.

- **The functional requirement count looks stronger than the delivered experience in one place.** FR-012 is "Implemented" at the API level but not yet reachable through the web interface, as Section 7.1 states. Counting it as fully complete without that caveat would overstate delivery.
- **No independent user has evaluated the system.** Every result in this chapter was produced by the development team, using tools the team wrote. Section 7.5 is a genuine gap, not a formality.
- **The benchmark figures are laptop measurements.** The lexical benchmark and the retrieval latencies in Section 7.2 were measured on one four-core, 16 GB development laptop. They establish a real baseline and a clear headroom for the lexical path, but should be re-measured on whatever hardware NAZ ultimately provisions before being treated as a production service-level commitment.
- **Model adaptation.** Fine-tuning or other domain adaptation of the embedding model was planned but not carried out, because it depends on real, labelled NAZ data (Section 8.3).
- **Output format.** The Order Paper is generated as structured data rather than as the official typeset publication, because no official template was available to the team.
- **Generative explanation is out of scope.** An early-stage, locally hosted generative explanation prototype exists in the codebase (a durable generate_explanation job type and an unevaluated Ollama-based provider), but it is not described, tested against, or evaluated as a delivered feature anywhere in the main body of this report. It is discussed once, briefly, as planned future work in Section 8.3.
- **Deviation from the original proposal.** The project's March 2026 proposal [15] committed to some things this report did not deliver, most notably duplicate detection scoped to a single selected parliamentary session, an evaluation sample labelled by NAZ domain users, and a comparative usability study against the manual process. Section 7.7 compares the proposal against the delivered system item by item and explains why each gap arose.

7.7. Comparison with the Original Project Proposal

Sections 7.1 to 7.6 evaluate the system against the requirements and objectives set out in Chapter 1 and Chapter 4, which were refined through the project's engagement with NAZ (Section 3.4). Before that engagement, the project began from a formal proposal submitted to the Department of Computing and Informatics on 20 March 2026 [15]. Because the proposal was written before the client consultations that shaped the final requirements, some of its planned deliverables changed in the course of the project, some were legitimately superseded, and a few were never carried out. This section records that comparison directly, rather than leaving it implicit, because an honest reading of the proposal turned up commitments that the rest of this report does not otherwise call out.

Table 7.4: The delivered system compared against the March 2026 project proposal

Proposal commitment	**Delivered**	**Assessment**
Aim: determine whether a question or motion "has previously been asked within a selected National Assembly session" [15]	Duplicate detection (Section 5.5) checks the entire record history regardless of session; SimilarityCheckRequest.session_id is accepted by the API but is not passed to DuplicateDetectionService.check(), only to the separate previously-addressed check, which deliberately looks at *earlier* sessions rather than one selected session	Not implemented as proposed. The system checks more broadly than specified, not less, but it does not offer the session-scoped check the aim describes
Objective 2: "a data mining and text similarity model capable of identifying exact matches and semantically related questions and motions"	Exact-hash and cosine-similarity duplicate detection, plus hybrid lexical/semantic search (Sections 5.5, 5.6)	Met
Objective 3: a web-based system for searching and reviewing matches	Full submission, review, search and scheduling workflow (Chapter 5)	Met, and exceeded: the proposal describes retrieval only, while the delivered system also implements the surrounding admission workflow (Sections 4.9, 5.4)
Objective 4 / Expected Outcome: "an evaluation report covering system accuracy, response time and usability"	Accuracy and response time are measured with real experiments (Sections 7.2, 7.4); usability is not evaluated	Partially implemented (Section 7.5)
Methodology stage 2: "Historical Order Papers, relevant debate/proceedings	No historical or debate records were supplied or ingested (Section 7.6)	Not implemented, for the reason already given (the

Proposal commitment	Delivered	Assessment
records and related submission records supplied by the National Assembly will be collected, cleaned and normalised"		data-sharing MoU was still pending)
Risk mitigation for data delay: "begin with publicly available parliamentary records while waiting for internal datasets" [15], referring to Order Papers such as the one the proposal itself cites [3]	Development and evaluation used entirely team-authored synthetic records (Section 3.7, Appendix E); NAZ's own published Order Papers were not ingested as an interim dataset	Not implemented. This was the proposal's own suggested stopgap for exactly the data-delay risk that materialised, and it was not carried out
Testing stage: "evaluate the quality of the matching engine using a labelled sample provided or validated by domain users from the National Assembly. Appropriate metrics may include precision, recall and top-k relevance"	The retrieval evaluation (Section 7.2) computes Recall@k and MRR, but on a team-authored corpus with team-assigned ground truth, not a sample labelled or validated by NAZ clerks	**Partially implemented.** The metrics the proposal asked for were computed; the domain-validated dataset it asked for was not available
Testing stage: usability testing to show "reviewers can reach decisions faster and with less effort than under a purely manual search process"	No comparative time-and-motion or usability study was conducted (Section 7.5)	**Not implemented**
Tools: "React typescript, Next js" [15]	Next.js 15 with React 19, written in plain JavaScript (JSX); no TypeScript is used anywhere in the frontend	**Implemented differently.** A deliberate but unrecorded deviation from the proposed tooling
Hardware: "National Assembly's iMacs" [15]	The system is designed and documented for Linux-based deployment through Docker Compose (Section 4.12, Appendix D), not for macOS/iMac hardware	**Implemented differently**
Expected outcome: "technical documentation, system design documentation and a user manual"	This report, the UML diagram set (diagrams/) and Appendix B	Met

Two points put this comparison in context rather than simply listing it as shortfall. First, the proposal's own architecture diagram and literature review describe a retrieval system only: keyword and semantic search over historical records, with a human reviewer making the final

call. It says nothing about generative AI or a locally hosted language model producing natural-language text, which matches exactly what this report delivers and evaluates: hybrid retrieval, not generation. The generative explanation prototype discussed as future work in Section 8.3 goes beyond what the proposal committed to; it was explored during the project but is deliberately not claimed as a delivered contribution here, which is also why the comparative usability study and the domain-labelled evaluation set were not reached before this report was due. Second, several of the "not implemented" items in Table 7.4 are not independent problems; they share one root cause: the data-sharing MoU with NAZ was still awaiting signature at the time of writing (Section 7.6), which blocked real records, blocked domain-labelled evaluation data, and blocked realistic usability testing all at once. Section 8.2 turns each remaining item in this table into a specific handover action, including one not previously listed in this report: restoring session-scoped duplicate checking as a configurable option, so NAZ can choose between the proposal's original narrower behaviour and the broader cross-session check now implemented, once clerks have confirmed which matches their procedure.

8. Conclusion and Recommendations

8.1. Conclusion

This project set out to design, implement and evaluate a secure web-based system that digitises the lifecycle of parliamentary questions and notices of motion at the National Assembly of Zambia and uses locally hosted hybrid retrieval to help clerks detect duplicate and previously addressed matters (Section 1.3.1). The NAZ Order Papers System meets that objective at the level of a working, tested and deployable artefact. The research questions are answered as follows.

RQ1: workflows, rules and requirements. Consultations with the NAZ ICT Directorate and Table Office, together with analysis of the Standing Orders and published Order Papers, produced a prioritised requirement set (FR-001 to FR-041), ten non-functional requirements and a model of four actors and 15 permissions (Chapter 4). The analysis showed that the central difficulty is a retrieval problem: finding the earlier business that a new item resembles.

RQ2: a valid, authorised and auditable lifecycle. The lifecycle is enforced by a single state machine with nine states, backed by database constraints, row locks, idempotency keys and a transactional audit log. The 224 backend tests, including concurrency and atomicity tests, show that invalid transitions, unauthorised actions and double-applied requests are rejected (Chapter 6).

RQ3: the value of hybrid retrieval. Weighted PostgreSQL full-text search fused with EmbeddingGemma chunk vectors through Reciprocal Rank Fusion found the correct earlier record for every one of 18 reworded queries that keyword search alone missed entirely, while keeping keyword precision for exact terms (Section 7.2). The same experiment showed that the provisional 0.95 duplicate threshold is suitable for near-copies but must be calibrated on real NAZ decisions before it is relied on for paraphrased duplicates (Section 7.3).

RQ4: groundwork for a future generative phase. The delivered hybrid retrieval pipeline already puts in place most of what a future generative explanation layer would need: retrieval results carry structured evidence (record IDs, matched excerpts, lexical and semantic ranks, cosine similarity) rather than opaque scores, so a future component could condition generation on exactly this evidence; the durable job queue and worker (Section 5.7) already support adding a new job type without changing their claiming, retry or dead-letter logic; and the model-digest and preprocessing-version tagging used for embeddings (Section 5.6.2) is a

pattern that a future generation model could reuse to avoid mixing incompatible model versions. An early, unevaluated prototype of the generation layer itself exists in the codebase (Section 7.6) and was useful for confirming this groundwork is sufficient, but it is deliberately not presented or evaluated as a delivered contribution of this report. Section 8.3 sets out what remains to complete and evaluate it.

The project also produced reusable engineering assets for NAZ:

- a layered architecture that keeps retrieval components behind abstract contracts, ready to accommodate a future generation stage;
- a durable, observable job pipeline;
- versioned migrations and a single-host deployment runbook;
- a complete set of UML diagrams;
- a large regression suite that documents the intended behaviour.

8.3. Future Work

8.3.1. Completing the Generative Explanation (RAG) Phase

This is the primary direction for the next phase of the project. The delivered system implements hybrid retrieval only: it ranks and returns existing records, and it generates no natural-language text. A retrieval-augmented generation (RAG) layer, in which a locally hosted language model would draft a short, evidence-grounded explanation of why a retrieved record is related to a new submission, was explored during this project as an early prototype (an `ai_inference_runs` audit table, a durable `generate_explanation` job type, and an Ollama-based provider using qwen3.5:4b) but is deliberately excluded from this report's delivered and evaluated scope (Section 7.6).

Exploratory, unevaluated measurements taken with that prototype during development are recorded here only to size the remaining engineering work, not as a claim about system quality: on the CPU-only development laptop, a 400-token output budget was too small for the model to complete a schema-valid response, and raising the budget to 1,200 tokens produced valid output but took 200 to 250 seconds per explanation, which is too slow for interactive use. Completing this phase would need, at minimum:

- GPU-accelerated inference, sized against an agreed latency budget for clerks;
- a properly tuned output budget and prompt, re-validated once real NAZ evidence text is available;

- the same kind of controlled evaluation this report applied to hybrid retrieval (Chapter 7), including adversarial testing of prompt-injection resistance and evidence-grounding, before any generated text is shown to a clerk; and
- an explicit decision, made with NAZ, about how a generated explanation would be presented so that it is clearly advisory and never mistaken for a procedural decision.

8.3.2. Other Future Work

- **Domain adaptation.** Once real data is available, fine-tune the embedding model contrastively on pairs labelled by clerks' similarity decisions [30]. Parameter-efficient methods such as LoRA or QLoRA [32], [33] would apply equally to adapting the future generation model. The model-digest and preprocessing-version tagging already in place allows an adapted embedding model to be introduced by re-indexing, without mixing vectors from different models.
- **Re-ranking.** Add a cross-encoder re-ranking stage after RRF fusion to improve precision at the very top of the ranked list, independently of any generative work.
- **Official publication.** Generate the Order Paper as a typeset PDF using NAZ's official template, and export records in Akoma Ntoso for interoperability and open data [16].
- **Notifications.** Reinstate the provider-agnostic email layer described in Section 5.11, which the repository history preserves, against NAZ's mail relay, and add SMS through the same EmailProvider-style abstraction.
- **Member-facing services.** While MPs currently interact indirectly through clerks, future work could explore extending direct access so MPs or their constituency offices can submit drafts directly with electronic signatures, and publishing answered questions to the public website.
- **Multilingual retrieval.** Evaluate retrieval for submissions that contain Zambian languages or mixed-language text.

9. References

[1]

Government of the Republic of Zambia, *Constitution of Zambia (Amendment) Act No. 2 of 2016*. Lusaka, Zambia: Government Printer, 2016.

[2]

National Assembly of Zambia, *Standing Orders, 2021*. Lusaka, Zambia: National Assembly of Zambia, 2021.

[3]

National Assembly of Zambia, "Order Papers," National Assembly of Zambia, Lusaka. [Online].Available:https://www.parliament.gov.zm/publications/order-paper-list. [Accessed: Sep. 16, 2026].

[4]

Inter-Parliamentary Union, *World e-Parliament Report 2020*. Geneva, Switzerland: Inter-Parliamentary Union, 2021.

[5]

Inter-Parliamentary Union, "Centre for Innovation in Parliament," Inter-Parliamentary Union,Geneva.[Online].Available: https://www.ipu.org/impact/democracy-and-strong-parliaments/centre-innovation-in-parliament. [Accessed: Sep. 16, 2026].

[6]

UK Parliament, "Written questions, answers and statements," UK Parliament, London. [Online]. Available: https://questions-statements.parliament.uk. [Accessed: Sep. 16, 2026].

[7]

United Nations Department of Economic and Social Affairs, "Bungeni: Parliamentary and Legislative Information System," Africa i-Parliaments Action Plan, 2010. [Online]. Available: https://www.bungeni.org. [Accessed: Sep. 16, 2026].

[8]

N. Reimers and I. Gurevych, "Sentence-BERT: Sentence embeddings using Siamese BERT-networks," in *Proc. EMNLP-IJCNLP*, Hong Kong, China, 2019, pp. 3982–3992.

[9]

N. Muennighoff, N. Tazi, L. Magne, and N. Reimers, "MTEB: Massive text embedding benchmark," in *Proc. 17th Conf. European Chapter of the Association for Computational Linguistics (EACL)*, Dubrovnik, Croatia, 2023, pp. 2014–2037.

[10]

G. V. Cormack, C. L. A. Clarke, and S. Büttcher, “Reciprocal rank fusion outperforms Condorcet and individual rank learning methods,” in *Proc. 32nd Int. ACM SIGIR Conf. Research and Development in Information Retrieval*, Boston, MA, USA, 2009, pp. 758–759.

[11]

S. Bruch, S. Gai, and A. Ingber, “An analysis of fusion functions for hybrid retrieval,” *ACM Trans. Information Systems*, vol. 42, no. 1, pp. 1–35, 2023.

[12]

Government of the Republic of Zambia, *The Data Protection Act No. 3 of 2021*. Lusaka, Zambia: Government Printer, 2021.

[13]

Ollama, “Ollama documentation: Embeddings and chat API,” Ollama. [Online]. Available: https://docs.ollama.com. [Accessed: Sep. 16, 2026].

[14]

H. S. Vera *et al.*, “EmbeddingGemma: Powerful and lightweight text representations,” arXiv:2509.20354, 2025.

[15]

M. Chavula, B. Mainza, and D. Zulu, “Order Papers (Question and Motion Submission): Final Year Project Proposal,” Dept. of Computing and Informatics, School of Natural and Applied Sciences, Univ. of Zambia, Lusaka, Zambia, unpublished, Mar. 20, 2026.

[16]

M. Palmirani and F. Vitali, “Akoma-Ntoso for legal documents,” in *Legislative XML for the Semantic Web*, G. Sartor, M. Palmirani, E. Francesconi, and M. A. Biasiotti, Eds. Dordrecht, The Netherlands: Springer, 2011, pp. 75–100.

[17]

Parliamentary Monitoring Group, “Parliamentary Monitoring Group,” Cape Town, South Africa. [Online]. Available: https://pmg.org.za. [Accessed: Sep. 16, 2026].

[19]

G. Salton and C. Buckley, “Term-weighting approaches in automatic text retrieval,” *Information Processing & Management*, vol. 24, no. 5, pp. 513–523, 1988.

[20]

S. Robertson and H. Zaragoza, “The probabilistic relevance framework: BM25 and beyond,” *Foundations and Trends in Information Retrieval*, vol. 3, no. 4, pp. 333–389, 2009.

[21]

PostgreSQL Global Development Group, “Chapter 12. Full Text Search,” *PostgreSQL 16 Documentation*. [Online]. Available: https://www.postgresql.org/docs/16/textsearch.html. [Accessed: Sep. 16, 2026].

[22]

A. Z. Broder, “On the resemblance and containment of documents,” in *Proc. Compression and Complexity of Sequences*, Salerno, Italy, 1997, pp. 21–29.

[23]

G. S. Manku, A. Jain, and A. Das Sarma, “Detecting near-duplicates for web crawling,” in *Proc. 16th Int. Conf. World Wide Web (WWW)*, Banff, Canada, 2007, pp. 141–150.

[24]

T. Mikolov, K. Chen, G. Corrado, and J. Dean, “Efficient estimation of word representations in vector space,” arXiv:1301.3781, 2013.

[25]

J. Devlin, M.-W. Chang, K. Lee, and K. Toutanova, “BERT: Pre-training of deep bidirectional transformers for language understanding,” in *Proc. NAACL-HLT*, Minneapolis, MN, USA, 2019, pp. 4171–4186.

[26]

V. Karpukhin *et al.*, “Dense passage retrieval for open-domain question answering,” in *Proc. EMNLP*, 2020, pp. 6769–6781.

[27]

Y. A. Malkov and D. A. Yashunin, “Efficient and robust approximate nearest neighbor search using hierarchical navigable small world graphs,” *IEEE Trans. Pattern Analysis and Machine Intelligence*, vol. 42, no. 4, pp. 824–836, Apr. 2020.

[28]

pgvector contributors, “pgvector: Open-source vector similarity search for Postgres,” GitHub repository. [Online]. Available: https://github.com/pgvector/pgvector. [Accessed: Sep. 16, 2026].

[29]

E. M. Voorhees, “The TREC-8 question answering track report,” in *Proc. 8th Text REtrieval Conf. (TREC-8)*, NIST Special Publication 500-246, Gaithersburg, MD, USA, 1999, pp. 77–82.

[30]

S. Gururangan *et al.*, “Don’t stop pretraining: Adapt language models to domains and tasks,” in *Proc. 58th Annu. Meeting of the Association for Computational Linguistics (ACL)*, 2020, pp. 8342–8360.

[31]

I. Chalkidis, M. Fergadiotis, P. Malakasiotis, N. Aletras, and I. Androutsopoulos, "LEGAL-BERT: The Muppets straight out of law school," in *Findings of the Association for Computational Linguistics: EMNLP 2020*, 2020, pp. 2898–2904.

[32]

E. J. Hu *et al.*, "LoRA: Low-rank adaptation of large language models," in *Proc. Int. Conf. Learning Representations (ICLR)*, 2022.

[33]

T. Dettmers, A. Pagnoni, A. Holtzman, and L. Zettlemoyer, "QLoRA: Efficient finetuning of quantized LLMs," in *Advances in Neural Information Processing Systems 36 (NeurIPS)*, 2023.

[34]

P. Lewis *et al.*, "Retrieval-augmented generation for knowledge-intensive NLP tasks," in *Advances in Neural Information Processing Systems 33 (NeurIPS)*, 2020, pp. 9459–9474.

[35]

Y. Gao *et al.*, "Retrieval-augmented generation for large language models: A survey," arXiv:2312.10997, 2023.

[36]

Z. Ji *et al.*, "Survey of hallucination in natural language generation," *ACM Computing Surveys*, vol. 55, no. 12, Art. no. 248, 2023.

[37]

OWASP Foundation, "OWASP Top 10 for Large Language Model Applications," OWASP GenAI Security Project, 2025. [Online]. Available: https://genai.owasp.org/llm-top-10/. [Accessed: Sep. 16, 2026].

[38]

A. R. Hevner, S. T. March, J. Park, and S. Ram, "Design science in information systems research," *MIS Quarterly*, vol. 28, no. 1, pp. 75–105, Mar. 2004.

[39]

K. Beck *et al.*, "Manifesto for agile software development," 2001. [Online]. Available: https://agilemanifesto.org. [Accessed: Sep. 16, 2026].

[40]

K. Beck, *Test-Driven Development: By Example*. Boston, MA, USA: Addison-Wesley, 2003.

[41]

R. C. Martin, *Agile Software Development, Principles, Patterns, and Practices*. Upper Saddle River, NJ, USA: Prentice Hall, 2003.

[42]

M. Kleppmann, *Designing Data-Intensive Applications*. Sebastopol, CA, USA: O'Reilly Media, 2017.

[43]

Vercel, "Next.js documentation: App Router," Next.js. [Online]. Available: https://nextjs.org/docs. [Accessed: Sep. 16, 2026].

[44]

S. Ramírez, "FastAPI documentation," FastAPI. [Online]. Available: https://fastapi.tiangolo.com. [Accessed: Sep. 16, 2026].

[45]

SQLAlchemy authors, "SQLAlchemy 2.0 documentation," SQLAlchemy. [Online]. Available: https://docs.sqlalchemy.org/en/20/. [Accessed: Sep. 16, 2026].

[46]

SQLAlchemy authors, "Alembic documentation," Alembic. [Online]. Available: https://alembic.sqlalchemy.org. [Accessed: Sep. 16, 2026].

[47]

N. Provos and D. Mazières, "A future-adaptable password scheme," in *Proc. USENIX Annual Technical Conf., FREENIX Track*, Monterey, CA, USA, 1999, pp. 81–91.

[48]

M. Jones, J. Bradley, and N. Sakimura, "JSON Web Token (JWT)," Internet Engineering Task Force, RFC 7519, May 2015.

[49]

pytest-dev team, "pytest documentation," pytest. [Online]. Available: https://docs.pytest.org. [Accessed: Sep. 16, 2026].

[50]

Microsoft, "Playwright: Fast and reliable end-to-end testing for modern web apps," Playwright. [Online]. Available: https://playwright.dev. [Accessed: Sep. 16, 2026].

[51]

Docker Inc., "Docker Compose overview," Docker Docs. [Online]. Available: https://docs.docker.com/compose/. [Accessed: Sep. 16, 2026].

[52]

J. Brooke, "SUS: A 'quick and dirty' usability scale," in *Usability Evaluation in Industry*, P. W. Jordan, B. Thomas, B. A. Weerdmeester, and I. L. McClelland, Eds. London, U.K.: Taylor & Francis, 1996, pp. 189–194.

[53]

S. Newman, “Pattern: Backends for frontends,” samnewman.io, Nov. 2015. [Online]. Available: https://samnewman.io/patterns/architectural/bff/. [Accessed: Sep. 16, 2026].

[54]

C. Richardson, “Pattern: Transactional outbox,” microservices.io. [Online]. Available: https://microservices.io/patterns/data/transactional-outbox.html. [Accessed: Sep. 16, 2026].

[55]

OWASP Foundation, “OWASP Top 10: 2021,” OWASP Foundation, 2021. [Online]. Available: https://owasp.org/Top10/. [Accessed: Sep. 16, 2026].

[56]

P. A. Grassi *et al.*, “Digital identity guidelines: Authentication and lifecycle management,” National Institute of Standards and Technology, Gaithersburg, MD, USA, NIST Special Publication 800-63B, 2017.

APPENDICES

APPENDIX A: INDIVIDUAL CONTRIBUTION STATEMENT (GROUP PROJECTS)

Purpose of this Statement

This statement is required for students who completed their final year project as part of a group. Although the project may have been conceived, developed, and evaluated collaboratively, each student is expected to demonstrate their own understanding and contribution through an individual final report. The statement should be consistent with the content presented in the main report. Students should not claim the work of another group member as their own.

INDIVIDUAL CONTRIBUTION REPORT - Dalitso Zulu_2022026047

1. Project and Group Information

Project Title: NAZ Order Papers System: A Questions and Motions Management System with Hybrid AI-Assisted Duplicate and Similarity Detection

Student Name: Dalitso Zulu

Computer Number: 2022026047

Programme: Bachelor of Science in Computer Science

Institution: University of Zambia

Project Supervisor: Dr Lighton Phiri

Group Members:

1. Bruce Mainza — 2022038479
2. Dalitso Zulu — 2022026047
3. Maliseni Chavula — 2022056205

2. Overview of the Group Project

The project involved the development of a web-based Order Papers system for managing parliamentary questions and motions at the National Assembly of Zambia (NAZ). The main problem addressed by the project was the time-consuming manual process of checking

whether newly submitted parliamentary questions or motions had already been raised in previous parliamentary records.

The manual process can make it difficult for clerks to identify submissions that are similar in meaning but expressed using different words. This can result in duplicated work and delays in reviewing parliamentary submissions.

The proposed system provides a digital platform for managing questions and motions while supporting the identification of exact duplicates and semantically similar submissions. It combines traditional keyword-based retrieval with semantic search to help users find potentially related historical records.

The system was developed using Next.js for the frontend, FastAPI for the backend, and PostgreSQL with pgvector for data storage and vector similarity search. The search functionality incorporates BM25 lexical retrieval, semantic retrieval using EmbeddingGemma embeddings, and Reciprocal Rank Fusion (RRF) to combine the results from the two retrieval approaches.

The project was divided among three group members, with responsibilities covering backend development, frontend development, system integration, DevOps, database management, scheduling, and audit management.

3. My Role and Responsibilities

My primary responsibility in the group project was backend development, particularly the implementation of the Authentication and Authorisation, Submission Management, and Hybrid Search and Similarity Detection modules.

I worked alongside Bruce Mainza on the backend. While I concentrated mainly on the first three modules, Bruce also contributed to backend development and was responsible for DevOps activities and the implementation of the Scheduling and Order Paper Generation and Database and Audit Management modules. Maliseni Chavula focused mainly on frontend development.

My responsibilities included:

- Contributing to the implementation of secure user authentication and authorisation.
- Developing backend functionality for creating, validating, managing, and tracking parliamentary questions and motions.
- Implementing hybrid search functionality that combines BM25 lexical retrieval with semantic search.
- Supporting the identification of exact duplicate and semantically similar submissions.
- Integrating backend functionality with the frontend and database where required.
- Participating in system testing, debugging, and the resolution of implementation issues.
- Collaborating with other group members to ensure that the different system modules worked together.
- Contributing to project documentation and the explanation of the system's technical implementation.

4. Detailed Individual Contribution

4.1 Authentication and Authorization Module

I contributed to the development of the authentication and authorisation module, which manages user access to the system.

The module supports user authentication and helps ensure that system functionality is available according to the user's assigned role and permissions. My work involved backend authentication logic, password verification, session handling, and the protection of relevant system operations.

The implementation uses bcrypt for password hashing and verification. JSON Web Tokens (JWTs) are used for session authentication, with the tokens stored in HTTP-only cookies. This approach helps reduce the risk of client-side scripts accessing authentication tokens.

I also contributed to the backend logic required to protect system endpoints and support role-based access control. These features are important because the system handles official parliamentary submissions that should only be accessed or modified by authorised users.

4.2 Submission Management Module

I was responsible for contributing to the backend implementation of the submission management module.

The module allows users to manage parliamentary questions and motions through the system. It provides the foundation for recording new submissions and managing their information throughout the submission process.

My work focused on backend functionality for receiving submission data, validating information, and supporting the creation, editing, and tracking of questions and motions.

The module is important because it establishes a structured way of storing parliamentary submissions and makes them available for other system functions, particularly the hybrid search and similarity detection module.

I also contributed to resolving issues associated with the handling and processing of submission data and ensuring that the backend could support the frontend's submission-related operations.

4.3 Hybrid Search and Similarity Detection Module

One of my main contributions was the development of the hybrid search and similarity detection functionality.

The purpose of this module is to help clerks identify earlier parliamentary questions or motions that may be duplicates or similar in meaning to a newly submitted item. Unlike a system that relies only on exact keyword matches, the hybrid approach combines lexical retrieval and semantic retrieval.

The main techniques involved were:

BM25 lexical retrieval: BM25 is used to retrieve records based on the words and terms contained in a search query. It provides a ranking method that considers factors such as term frequency and document length.

Semantic retrieval: Semantic retrieval uses EmbeddingGemma to represent text as numerical vectors, or embeddings. These embeddings are stored in PostgreSQL using pgvector, allowing the system to retrieve records based on vector similarity. This helps identify submissions that may express similar ideas using different wording.

Reciprocal Rank Fusion (RRF): RRF combines the ranked results from lexical and semantic retrieval into a single ranking. This allows the system to benefit from both exact terminology matches and semantic similarities.

Exact duplicate detection: The system also uses text normalisation and SHA-256 hashing to help identify exact duplicates. Normalisation helps standardise text before its hash is calculated.

My contribution involved working on the backend search logic and supporting the integration of these techniques into the system. This module was particularly important because it addressed the main problem identified at the beginning of the project: the difficulty of finding previously submitted questions and motions when the wording differs.

The system presents potentially similar records to support human review. It should not be interpreted as automatically deciding that every similar submission is a duplicate; the final assessment depends on the context and review of the relevant parliamentary record.

5. Individual Deliverables

The main deliverables associated with my contribution were:

1. **Authentication and Authorisation Backend:** Backend functionality supporting authentication, password verification, session handling, and access control.

2. **Submission Management Backend:** Functionality supporting the handling, validation, and management of parliamentary questions and motions.

3. **Hybrid Search and Similarity Detection:** Backend search functionality combining BM25, semantic retrieval using EmbeddingGemma and pgvector, and Reciprocal Rank Fusion.

4. **Exact Duplicate Detection Support:** Text normalisation and SHA-256 hashing functionality used to help identify exact duplicate submissions.

5. **Backend Integration and Testing Support:** Contributions to integrating the modules with other system components, testing their behaviour, and resolving implementation issues.

6. **Technical Documentation:** Explanations of the implemented modules, technologies, search methods, and their roles within the overall system.

These deliverables contributed to the system's ability to manage submissions securely and retrieve potentially relevant historical parliamentary records.

6. Contribution to the Main System Output

My contribution supported the development of the core backend features required by the Order Papers system.

The authentication and authorisation module provides a foundation for controlling access to the system. The submission management module provides the functionality for recording and managing parliamentary questions and motions. The hybrid search and similarity detection module addresses the central research and implementation challenge by helping users discover earlier records that may be related to new submissions.

These modules work together. Submissions entered into the system can be stored and made available for retrieval, while the search functionality can help users compare new submissions with relevant existing records.

Bruce Mainza's work on scheduling, Order Paper generation, database and audit management, together with his DevOps responsibilities, supported the wider system infrastructure and the management of parliamentary submissions. Maliseni Chavula's frontend contribution provided the user-facing interface through which users interact with the system.

The combined contributions of the group produced a system intended to improve the organisation, retrieval, and review of parliamentary questions and motions.

7. Individual Implementation and Technical Work

My main technical work was concentrated on the backend and the search functionality.

7.1 Backend Development

I worked with FastAPI and Python to implement backend functionality. This involved developing and connecting application logic to the system's data storage and frontend requirements.

7.2 Authentication and Security

I worked with bcrypt for password hashing and verification and JWTs for session authentication. HTTP-only cookies were used to store authentication tokens. These mechanisms contribute to protecting user accounts and controlling access to system functionality.

7.3 Submission Processing

I contributed to backend operations for handling parliamentary questions and motions. This included processing submission data and supporting validation and management operations.

7.4 Lexical Search Using BM25

I worked on the hybrid search functionality that uses BM25 for lexical retrieval. BM25 ranks records according to their relevance to the words in a query. This approach is useful when a user searches using specific terms or phrases that also appear in historical submissions.

7.5 Semantic Search Using EmbeddingGemma and pgvector

I contributed to the semantic retrieval functionality that uses EmbeddingGemma to generate text embeddings. The embeddings are stored and searched using PostgreSQL with pgvector.

This approach helps retrieve records that may be similar in meaning even when they do not contain exactly the same words as the query.

7.6 Combining Search Results Using RRF

Reciprocal Rank Fusion combines the rankings produced by lexical and semantic retrieval. Instead of depending entirely on either keyword matching or vector similarity, the system uses both sources to produce a combined ranking of potentially relevant records.

7.7 Exact Duplicate Detection

The system also uses normalised text and SHA-256 hashing to help identify exact duplicate content. This complements the hybrid search approach, which is intended to identify records that may be related even when their wording is different.

7.8 Collaboration and Integration

Although my main focus was on Modules 1, 2, and 3, I collaborated with the other group members where integration between the backend, database, and frontend was required. This helped ensure that my modules could operate as part of the complete application.

8. Individual Testing and Evaluation Contribution

I contributed to the testing and evaluation activities associated with the system, particularly the backend functionality and search features.

Testing was important for checking whether the implemented modules behaved as expected and whether the search functionality could retrieve relevant historical records.

The project included automated backend tests and frontend source tests. The reported results were 224 passing backend automated tests and 21 passing frontend source tests.

The project also evaluated lexical search performance using 100,000 synthetic records. The reported 95th-percentile latency for lexical search was 266.5 milliseconds, compared with a 3,000-millisecond performance budget.

A controlled experiment involving 18 reworded queries also compared hybrid retrieval with keyword-only search. In that experiment, hybrid retrieval found the correct earlier record in all cases where keyword-only search missed it.

These results provide preliminary evidence that combining lexical and semantic retrieval can be useful for finding reworded parliamentary submissions. However, the results must be interpreted carefully because the experiment was limited in size and used synthetic data rather than the National Assembly's complete historical archive.

The project had not yet completed a formal user acceptance or usability study with National Assembly clerks. Similarity thresholds would also need to be calibrated using real records and judgements from domain experts.

My contribution to testing involved supporting the verification of backend behaviour, investigating implementation issues, and helping ensure that the relevant features worked as intended.

9. Collaboration with Other Group Members

The project required collaboration among all three group members because the frontend, backend, search functionality, database operations, scheduling, and deployment-related activities were connected.

9.1 Collaboration with Bruce Mainza

Bruce Mainza and I both contributed to backend development. My primary focus was Authentication and Authorisation, Submission Management, and Hybrid Search and Similarity Detection.

Bruce was also responsible for DevOps activities and the implementation of the Scheduling and Order Paper Generation and Database and Audit Management modules. His work supported the wider operational requirements of the application, including the management of scheduled submissions, preparation of Order Papers, database-related functionality, and audit activities.

We worked within the same overall backend project and collaborated where integration between our responsibilities was necessary.

9.2 Collaboration with Maliseni Chavula

Maliseni Chavula focused mainly on frontend development. His work involved the user-facing parts of the system through which users interact with the application's functionality.

My backend work supported the frontend by providing the application logic and operations required for authentication, submission management, and search. Coordination between the frontend and backend was necessary to ensure that users could access and use the implemented features.

9.3 Team Collaboration

As a group, we contributed to the same overall system and coordinated our work to support integration between the different modules. This collaboration helped us combine our individual areas of responsibility into one application.

The division of responsibilities allowed each member to focus on particular parts of the system while still contributing to the shared project objectives.

10. Summary of My Contribution

My main contribution to the NAZ Order Papers system was backend development, with a particular focus on Authentication and Authorisation, Submission Management, and Hybrid Search and Similarity Detection.

I contributed to implementing secure authentication mechanisms using bcrypt and JWTs, developing backend functionality for parliamentary submissions, and supporting a search approach that combines BM25 lexical retrieval with semantic retrieval using EmbeddingGemma and pgvector. I also contributed to the use of Reciprocal Rank Fusion to combine search rankings and text normalisation with SHA-256 hashing to help identify exact duplicates.

My work addressed important parts of the system's main objective: to provide a structured platform for managing parliamentary questions and motions and to assist users in identifying potentially related historical records.

The project also strengthened my practical experience in backend development, API implementation, authentication, database integration, information retrieval, semantic search, testing, and collaborative software engineering.

11. Estimated Individual Contribution

The table below provides a starting point for discussing the distribution of work within the group. The percentages are illustrative estimates only and must be reviewed and adjusted by the group to reflect the actual work completed by each member.

Project Activity	My Estimated Contribution	Estimated Contribution by Other Members
Problem Analysis	30%	70%
Literature Review	20%	80%
Requirements Analysis	30%	70%
System and Research Design	30%	70%
Implementation and Development	35%	65%
Testing and Evaluation	30%	70%
Documentation	30%	70%
Backend Integration and Related Activities	35%	65%

These estimates should not be treated as verified measurements of individual effort. They are intended to be reviewed with the other group members and the supervisor before submission. The group should ensure that the final percentages accurately reflect the actual distribution of responsibilities and work.

12. Statement of Authorship and Contribution

I, Dalitso Zulu, declare that this report describes my individual contributions to the development of the NAZ Order Papers system as accurately as possible.

My primary responsibilities were backend development for Authentication and Authorisation, Submission Management, and Hybrid Search and Similarity Detection. I also participated in integration, testing, and documentation activities associated with the project.

I acknowledge that this was a group project and that the overall system was developed through the combined contributions of Bruce Mainza, Maliseni Chavula, and myself. Bruce Mainza contributed to backend development, DevOps, Scheduling and Order Paper Generation, and Database and Audit Management, while Maliseni Chavula focused mainly on frontend development.

I understand the importance of representing individual contributions honestly and distinguishing my own work from the work completed by other members of the group.

INDIVIDUAL CONTRIBUTION REPORT - Bruce Mainza_2022038479

Project Title: NAZ Order Papers System

Student Name: Bruce Mainza

Computer Number: 2022038479

Supervisor: Dr Lighton Phiri

Group Size: 3 Students

Group Members: Dalitso Zulu (2022026047) Maliseni Chavula (2022056205)

Overall Group Project

The NAZ Order Papers System is a web-based system for the Table Office of the National Assembly of Zambia (NAZ). It supports the preparation and management of order papers, including motions and questions submitted for parliamentary business. The group's aim was to reduce manual effort and error in handling submissions. The system detects duplicate and previously addressed matters using embeddings and vector similarity (EmbeddingGemma, run locally via Ollama). It is built with Python/FastAPI and PostgreSQL, using a modular

design. It has role-based access for three user types: Clerk, Senior Clerk (the only role that can approve, reject and delete submissions), and Admin (manages user accounts and views the audit trail). The team followed Agile Scrum, TDD, and SOLID principles under the supervision of Dr Lighton Phiri.

Individual Role and Responsibilities

My primary role: Backend development
Main responsibilities: System architecture design, Testing and Deployment

Detailed Individual Contribution

I contributed primarily to the backend design and implementation of the NAZ Order Papers System, the AI-assisted submission checking, and the system's evaluation, deployment and documentation.

- **System design and architecture**. I worked on the backend architecture using Python/FastAPI and PostgreSQL. The design emphasised low coupling and modularity, following SOLID principles, so that the system could be handed over to and maintained by NAZ. This included an email abstraction layer with SMTP as the default, because NAZ's email provider was unknown at the time of development.
- **Role-based access control**. I implemented the permission model for the three Table Office roles. Clerks handle submissions. Senior Clerks can do everything a Clerk can and are the only role able to approve, reject and delete submissions. Admins only manage user accounts and view the audit trail.
- **Duplicate and previously addressed matter detection**. I implemented the embedding-based similarity check using EmbeddingGemma run locally through Ollama, with vector similarity to flag motions and questions that duplicate earlier submissions or matters already addressed.
- **Evaluation.** I designed the evaluation using a synthetic set of 100 fake motions and questions in PDF format, and analysed how well the system detected duplicates and handled the documents.
- **Deployment.** I set up the deployment of the system to a cloud server. I used AWS to deploy the system on an ubuntu EC2 instance.
- **Documentation**. I contributed to the project documentation. Drawing all the system diagrams and the Literature review and the Testing and evaluation.

Individual Deliverables

List the specific outputs or deliverables produced primarily by the student.

1. Backend services and role-based access control.
2. Duplicate detection and provider-agnostic email service.
3. Automated test suite and evaluation.
4. Deployment and project documentation.

Contribution to the Main System or Research Output

Backend architecture and core services. I designed and implemented the FastAPI and PostgreSQL backend on which the rest of the system is built. This included the database

schema, the API endpoints and the business logic for creating, reviewing and managing motions and questions. The backend follows SOLID principles and low coupling, so individual modules can be changed or replaced without affecting the rest of the system, which supports its handover to the National Assembly of Zambia.

Role-based access control. I implemented the permission model for the three Table Office roles. Clerks handle submissions, Senior Clerks inherit all Clerk permissions and are the only role able to approve, reject or delete submissions, and Admins manage user accounts and view the audit trail. This enforces a clear separation of duties within the system.

Duplicate and previously-addressed matter detection. I developed the component that generates embeddings with EmbeddingGemma, run locally through Ollama, and uses vector similarity to compare new submissions against earlier ones. It flags motions and questions that duplicate previous submissions or repeat matters already addressed, helping clerks avoid repeated parliamentary business.

Testing and evaluation. I wrote the automated test suite for the components I built, following strict test-driven development with each functionality committed separately. I also took part in evaluating the system using a synthetic set of 100 fake motions and questions in PDF format, assessing how well the system processed documents and detected duplicates.

Deployment. I set up and deployed the system on a cloud server, first on AWS EC2 and later on DigitalOcean after the AWS account was suspended, so that the system could be demonstrated and prepared for handover.

Jointly developed work. The requirements analysis, overall system design and project documentation were developed jointly with Dalitso Zulu and Maliseni Chavula under the guidance of Dr Lighton Phiri, and my components were integrated with the parts of the system they built.

Individual Implementation and Technical Work

Backend services and data layer. I implemented the backend in Python using FastAPI, with PostgreSQL as the database. I designed the database schema for motions, questions, users and the audit trail, and exposed the system's functionality through RESTful API endpoints. The code is organised into modular layers, with business logic kept separate from the API and data-access code. Following SOLID principles and low coupling, components depend on abstractions rather than concrete implementations, so they can be tested in isolation and replaced without affecting the rest of the system.

Authentication and role-based access control. I implemented user authentication and the permission checks that restrict each endpoint by role. A Clerk can create and manage submissions. A Senior Clerk inherits all Clerk permissions and is the only role allowed to approve, reject or delete submissions. An Admin can only manage user accounts and view the audit trail. I enforced these rules at the API level so that restrictions cannot be bypassed from the client, and actions are recorded in the audit trail.

Duplicate and previously-addressed matter detection. I implemented the semantic similarity component using the EmbeddingGemma model, served locally through Ollama. When a submission is received, its text is converted into an embedding vector and compared

against the embeddings of earlier submissions using vector similarity. Submissions whose similarity to an existing one is high are flagged as possible duplicates or as matters already addressed, so the Table Office can review them before they enter an order paper. Running the model locally keeps the parliamentary data within NAZ's own infrastructure.

Test-driven development. I developed the backend using strict test-driven development. Tests were written before the implementation of each feature, covering the API endpoints, the permission rules and the similarity detection logic. Each functionality was committed separately, giving a traceable history of the development.

Evaluation tooling. I prepared and used a synthetic set of 100 fake motions and questions in PDF format to evaluate the system. The set was used to check that documents were processed correctly and that duplicate and previously-addressed matters were detected as expected.

Deployment. I set up the server environment and deployed the backend, PostgreSQL database and the locally hosted embedding model on a cloud server, first on AWS EC2 and then on DigitalOcean after the AWS account was suspended.

Individual Testing and Evaluation Contribution

I tested the system at the backend, frontend, browser and non-functional levels, and ran all the tests myself. The backend suite has 224 pytest tests in 44 modules, all passing. They run through FastAPI's TestClient against a real PostgreSQL 16 database with pgvector, with the schema rebuilt before every test, real JWTs for authentication, and fakes for Ollama. They cover authentication and lockout, role permissions, submission validation, lifecycle enforcement, duplicate detection, audit logging and Alembic migrations.

I also wrote 21 frontend source-guard tests, 14 Playwright end-to-end and visual scenarios, and 20 upload-safety tests. The reliability tests covered job claiming, retries and dead-lettering.

For evaluation, I ran a controlled retrieval experiment with EmbeddingGemma through Ollama, using 18 paraphrased queries against a 24-record synthetic corpus. Lexical search found none of the targets, while semantic and hybrid retrieval achieved Recall@1 and MRR@10 of 1.0. I analysed cosine similarity scores to assess the duplicate and previously-addressed thresholds. I benchmarked lexical search at a p95 latency of 266.5 ms against a 3,000 ms budget. I also ran a System Usability Scale study with 30 students.

Collaboration with Other Group Members

I worked closely with Dalitso Zulu on the backend and with Maliseni Chavula on the frontend. We worked in Agile Scrum sprints, met regularly with Dr Lighton Phiri and the NAZ ICT team, and agreed requirements, architecture and API contracts together.

Dalitso built the authentication and the lexical retrieval. My duplicate detection, semantic retrieval and hybrid search depended on his lexical retrieval, and my endpoints used his authentication and permission checks.

Maliseni built the Next.js frontend, which consumed the backend API. I tested the integration, reported faults to the person responsible, and we fixed them together. My source-guard and Playwright tests were written against her interface and used by the group as a regression check.

The requirements analysis, overall design and the project report were joint work. I drew the system diagrams and wrote the literature review and the testing and evaluation chapter.

Contribution Summary

(1) Work primarily completed by me

- System architecture design
- Duplicate and previously-addressed detection, and semantic and hybrid retrieval using EmbeddingGemma, pgvector and Reciprocal Rank Fusion
- All testing: the 224-test backend suite, the frontend source guards, the Playwright end-to-end tests, and the upload-safety and reliability tests
- Evaluation: the retrieval experiment, the threshold analysis, the performance benchmark and the SUS study
- Deployment
- All system diagrams, the literature review, and the testing and evaluation chapter

(2) Work completed jointly with other group members

- Problem analysis, requirements analysis and overall design decisions, agreed with Dalitso Zulu and Maliseni Chavula and in consultation with Dr Lighton Phiri
- Development of the FastAPI and PostgreSQL backend, shared with Dalitso Zulu
- The project report, where each member wrote the sections on their own components

(3) Work primarily completed by other group members but used in my work

- Authentication, permission checks and lexical retrieval, built by Dalitso Zulu. My endpoints used the authentication and permission checks, and my hybrid search fused his lexical results with my semantic results.
- The Next.js frontend, built by Maliseni Chavula, which I tested through the source-guard and Playwright suites.

Estimated Contribution

Problem Analysis:	Individual 30% / Collaborative 70%
Literature Review:	Individual 90% / Collaborative 10%
Requirements Analysis:	Individual 30% / Collaborative 70%
System/Research Design:	Individual 60% / Collaborative 40%
Implementation/Development:	Individual 35% / Collaborative 65%
Testing and Evaluation:	Individual 90% / Collaborative 10%
Documentation:	Individual 50% / Collaborative 50%
Other (Deployment):	Individual 90% / Collaborative 10%

Statement of Authorship and Contribution

I confirm that the contribution described in this statement accurately represents the work that I personally undertook as part of the group project. I have identified collaborative work and have not knowingly claimed the work of another group member as my own.

INDIVIDUAL CONTRIBUTION REPORT - Maliseni Chavula_2022056205

Project and Group Information

Project Title: NAZ Order Papers System: A Questions and Motions Management System with Hybrid AI-Assisted Duplicate and Similarity Detection for the National Assembly of Zambia

Student Name: MALISENI CHAVULA

Computer Number: 2022056205

Programme: Bachelor of Science in Computer Science

Supervisor: Dr Lighton Phiri

Group Size: 3

Group Members: Maliseni Chavula (2022056205), Bruce Mainza (2022038479), Dalitso Zulu (2022026047)

Overall Group Project

Team CMZ built a web application for the National Assembly of Zambia (NAZ) that digitises the lifecycle of parliamentary questions and notices of motion. Clerks used to compare each new submission against earlier records by hand, which was slow, hard to audit and blind to repeated matters worded differently. The system covers submission, validation, clerk review, scheduling for a sitting and generation of the Order Paper. It adds hybrid duplicate and similarity detection that fuses lexical and semantic (pgvector with a local embedding model) retrieval, with a human reviewer making the final decision. It is built with a Next.js front end, a FastAPI back end, PostgreSQL 16 with pgvector and a background worker, and is packaged for deployment with Docker Compose.

Individual Role and Responsibilities

My primary role was UI/UX design and frontend development. I designed and implemented the user interface through which NAZ staff use the system. I assisted with the frontend side of authentication and authorisation, which builds on the backend module led by Dalitso Zulu. I also took part in group discussions on how to implement some parts of the system.

Detailed Individual Contribution

- *UI/UX design and implementation.* I designed and built the interface in Next.js (App Router) and React, covering the login page, dashboard, Submit, Search, Record detail, Reports, Audit, Sessions, Users and Notifications. This included layout, navigation, forms, responsive behaviour for desktop and mobile, and the display of search results with their retrieval evidence. Forms use React Hook Form with Zod validation that mirrors backend rules, and the sidebar shows only the destinations a user's permissions allow.
- *Authentication and authorisation (assisting).* Dalitso led the backend: bcrypt password hashing, JWT session tokens and permission checks. I assisted on the client side, working on the login flow, session handling in the frontend (the HttpOnly cookie, the same-origin Backend-for-Frontend route handlers and protected-route checks) and permission-aware navigation and actions.
- *Design decisions.* I contributed to decisions such as the BFF pattern, permission-based access control rather than role names, migrating the interface from Tailwind CSS to plain CSS.

Individual Deliverables

1. **UI/UX designs:** wireframes/mockups and the interface design for the system's main screens

2. **Frontend application:** the Next.js/React interface, including forms, navigation, responsive layouts and search and record views.

3. **Frontend authentication integration:** login, session handling and permission-aware navigation, built on Dalitso's backend.

4. **Design input:** contributions to implementation decisions across the system.

Contribution to the Main System or Research Output

The component I primarily developed is the user-facing interface, which is how clerks and other users reach every backend feature. Authentication and authorisation were developed jointly: Dalitso led the backend, and I worked on the frontend side. I did not lead the backend modules or the deployment setup, which were Dalitso's and Bruce's.

Individual Implementation and Technical Work

My technical work was in the presentation tier: App Router pages and route groups ((auth) and (dashboard)), reusable components, client-side validation, the same-origin route handlers that connect the interface to the FastAPI API, and the login and protected-route handling. I worked against the API contracts that Dalitso and Bruce built.

Individual Testing and Evaluation Contribution

I tested the interface during development and fixed layout, validation and usability problems. Frontend source-guard tests, Playwright visual tests, running and organising the System Usability Scale study with 30 students.] The study showed moderate usability and that learnability needs improvement, which the main report states. It was done with students, not NAZ staff.

Collaboration with Other Group Members

Dalitso Zulu: My interface depended on his backend for authentication, submission management and search. We coordinated the API requests and responses, and the login and session flow.

Bruce Mainza: I depended on his work on scheduling, Order Paper generation, database and audit management, and on his DevOps setup, which runs the system. My Audit, Reports and scheduling screens present his modules.

Both: We shared design decisions, integration and testing.

Contribution Summary

1. Primarily mine: UI/UX design and frontend implementation.
2. Done jointly: authentication and authorisation (backend by Dalitso, frontend by me), integration, testing, and the report.
3. Primarily by others, used in my work: backend authentication, submission management and hybrid search (Dalitso), and scheduling, Order Paper generation, database, audit and DevOps (Bruce).

Estimated Contribution

Activity	**Individual % | Collaborative %**
Problem Analysis	30 | 70
Literature Review	20 | 80
Requirements Analysis	30 | 70
System/Research Design	35 | 65
Implementation/Development	35 | 65
Testing and Evaluation	30 | 65
Documentation	20 | 80

Statement of Authorship and Contribution

I confirm that the contribution described in this statement accurately represents the work that I personally undertook as part of the group project. I have identified collaborative work and have not knowingly claimed the work of another group member as my own.

Student Name: Dalitso Zulu

Computer Number: 2022026047

Signature:

Student Name: Bruce Mainza

Computer Number: 2022038479

Signature:

Student Name: MALISENI CHAVULA

Computer Number: 2022056205

Signature:

Date: 23rd October, 2026.

Supervisor Verification

I confirm that, to the best of my knowledge, the contribution described above is consistent with the student's participation in the group project.

Supervisor Name: Dr. Lighton Phiri

Signature: _____

Date: _____

APPENDIX B: USER MANUAL / INSTALLATION GUIDE

NAZ Order Papers System — Installation & Usage Guide

Requirements

- Docker with Docker Compose

- Git

- Ollama (optional, for semantic search)

1. Clone the repository:

```bash

git clone https://github.com/brucemainza/NAZ-ORDER-PAPERS-REPO.git

cd NAZ-ORDER-PAPERS-REPO

```

2. Start the system:

```bash

docker compose up --build

```

3. Open the application:

- Frontend: http://localhost:3000

- API: http://localhost:8080

- API documentation: http://localhost:8080/docs

Login

Use one of the active demo accounts:

Table B.1: Demo login accounts for the installation guide

Employee ID	Password	Role
EMP-00	naz@2026	Adim
EMP-00	naz@2026	Admin
EMP-00	naz@2026	Senior CLerk
EMP-00	naz@2026	Clerk

Change demo credentials before using the system in production.

Basic Usage

1. Sign in with your employee ID and password.

2. Submit a question or notice of motion.

3. Track your submission status from the submissions page.

4. Clerks review, approve, reject, or request changes.

5. Approved items can be scheduled for a sitting date.

6. Generate and view the Order Paper for the scheduled sitting.

7. Use archive and search features to find previous records.

Optional Ollama Setup

The system uses Ollama for semantic search. Run Ollama on port `11434` and make it reachable from Docker. Do not expose the Ollama API to untrusted networks.

Stop the System

```bash
docker compose down
```

To remove containers and stored database volumes:

```bash
docker compose down -v
```

For backend tests:

```bash
PYTHONPATH=backend .venv/bin/pytest -q backend/tests
```

For frontend development:

```bash
cd frontend
npm install
npm run dev
```

For more details, see the main `README.md` and the documentation in `docs/`.

APPENDIX C: ADDITIONAL SYSTEM DESIGNS / DIAGRAMS

Figure C.1: Conceptual ERD Diagram

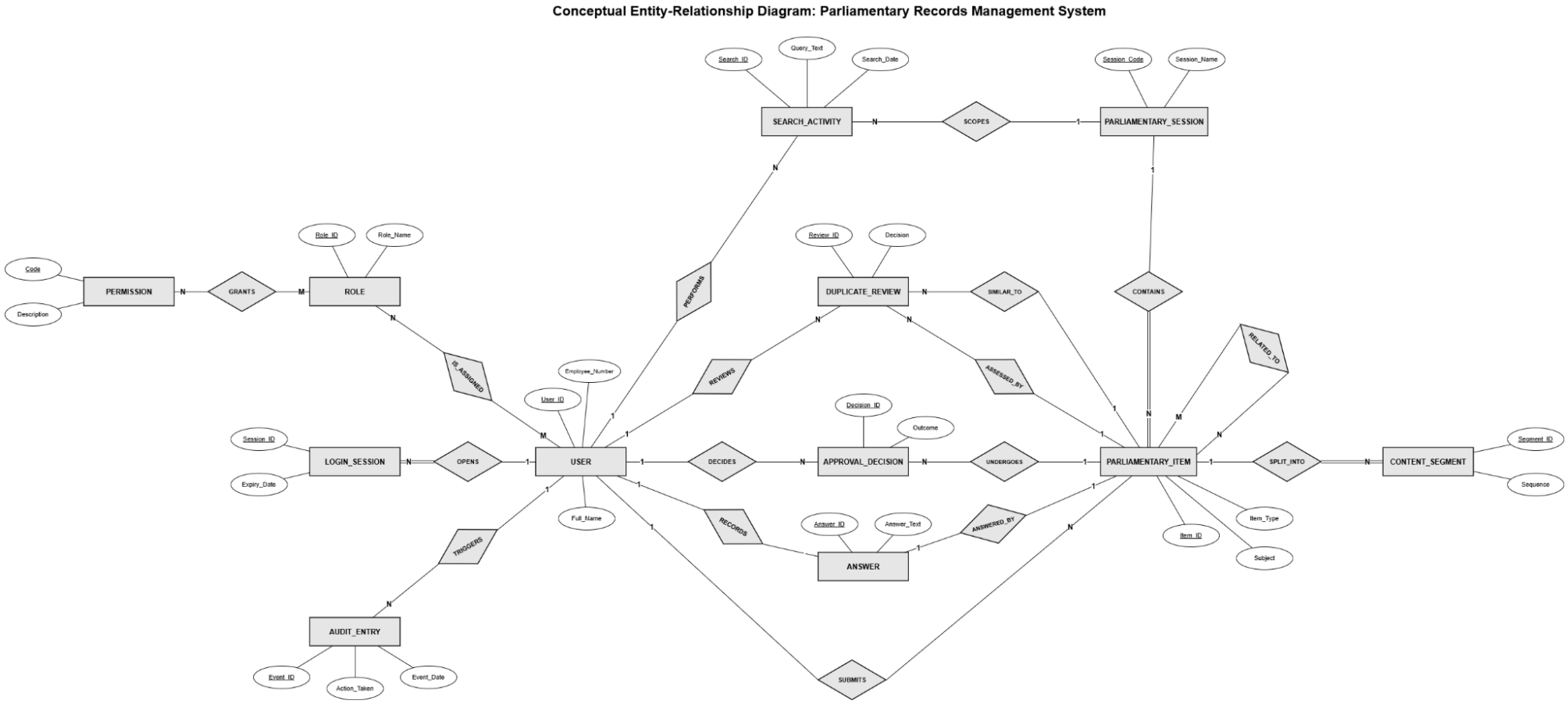


Figure C.2: Hybrid Retrieval Class Diagram

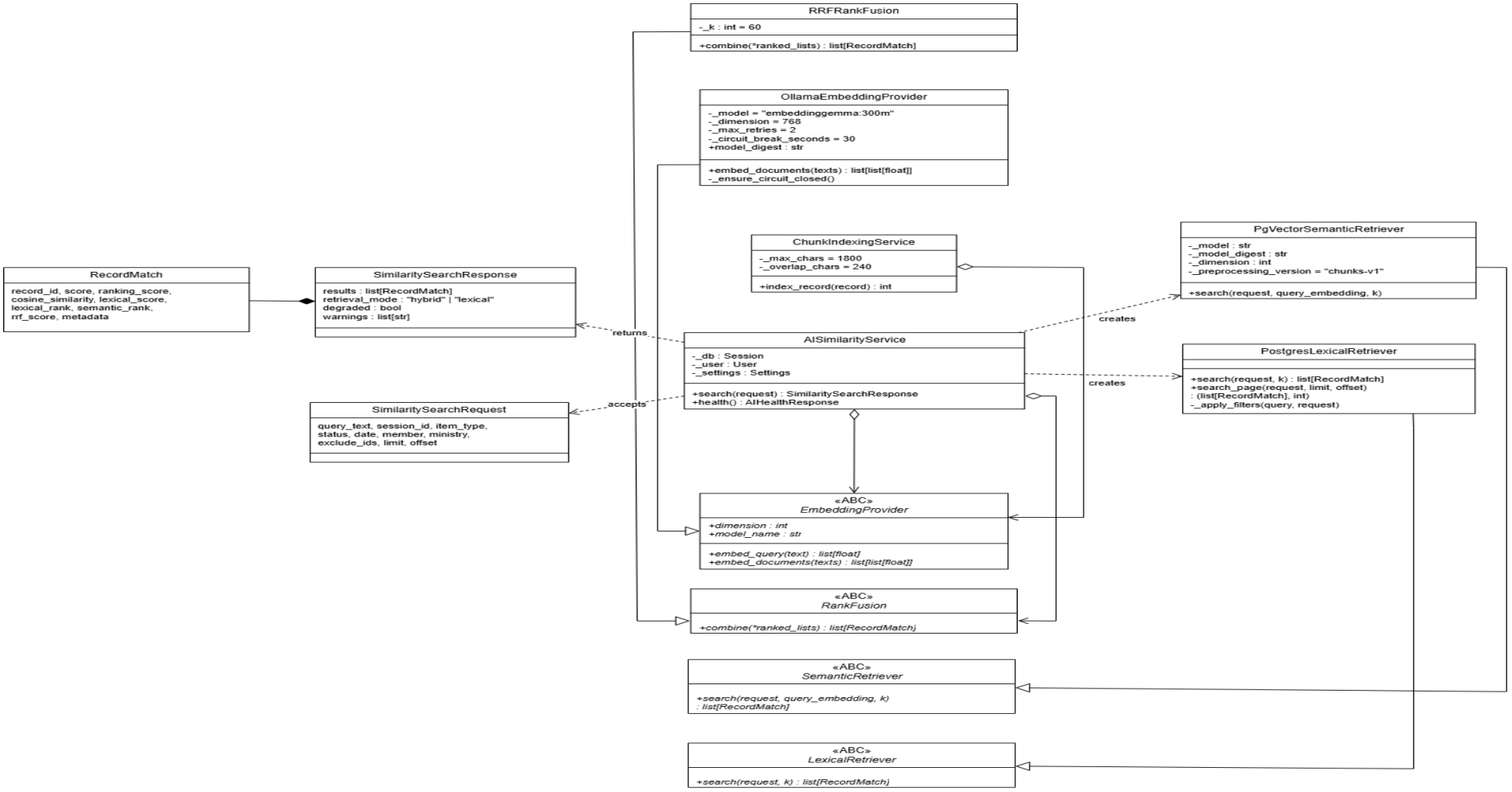


Figure C.3: Duplicate Package Class Diagram

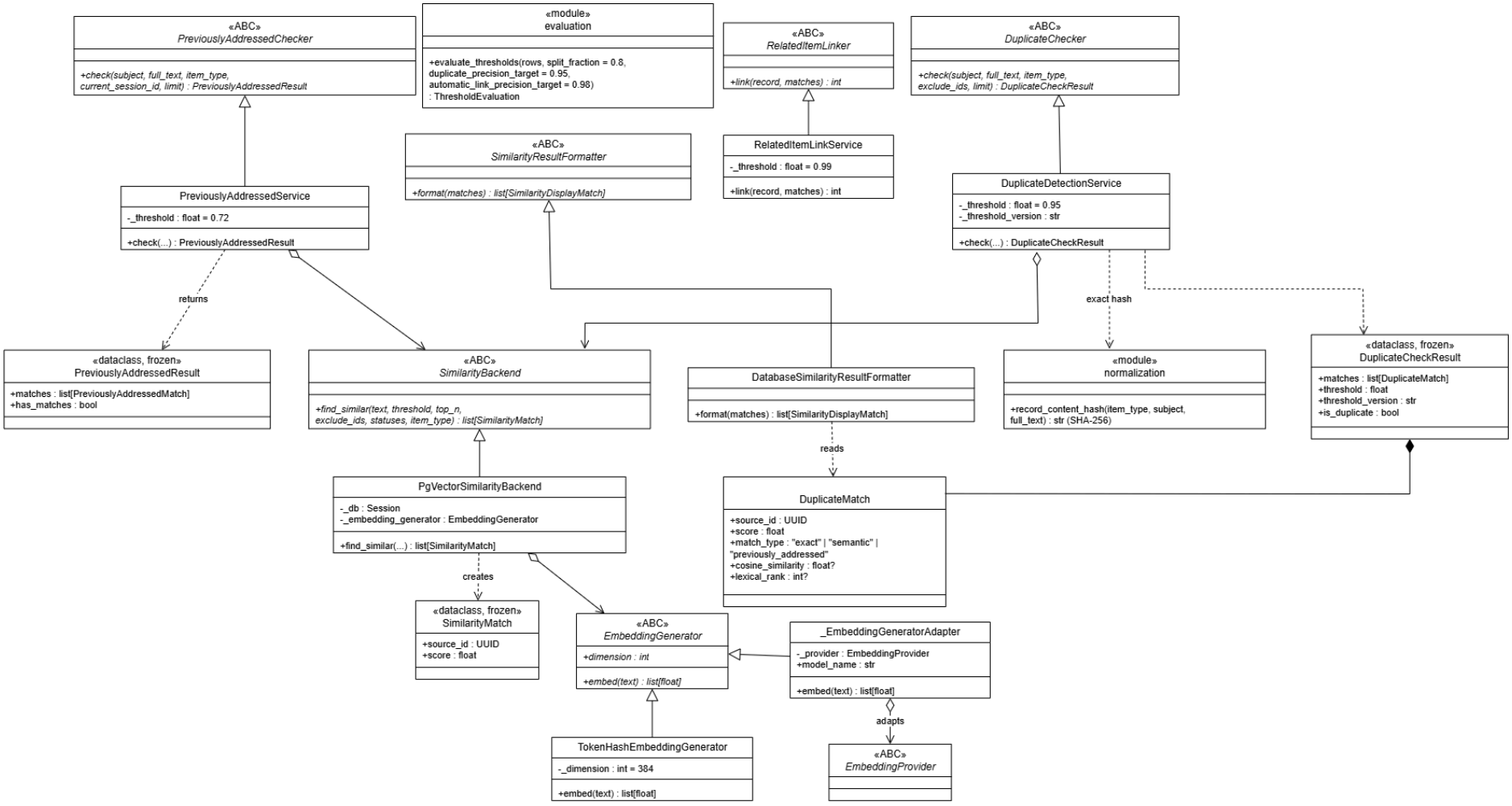
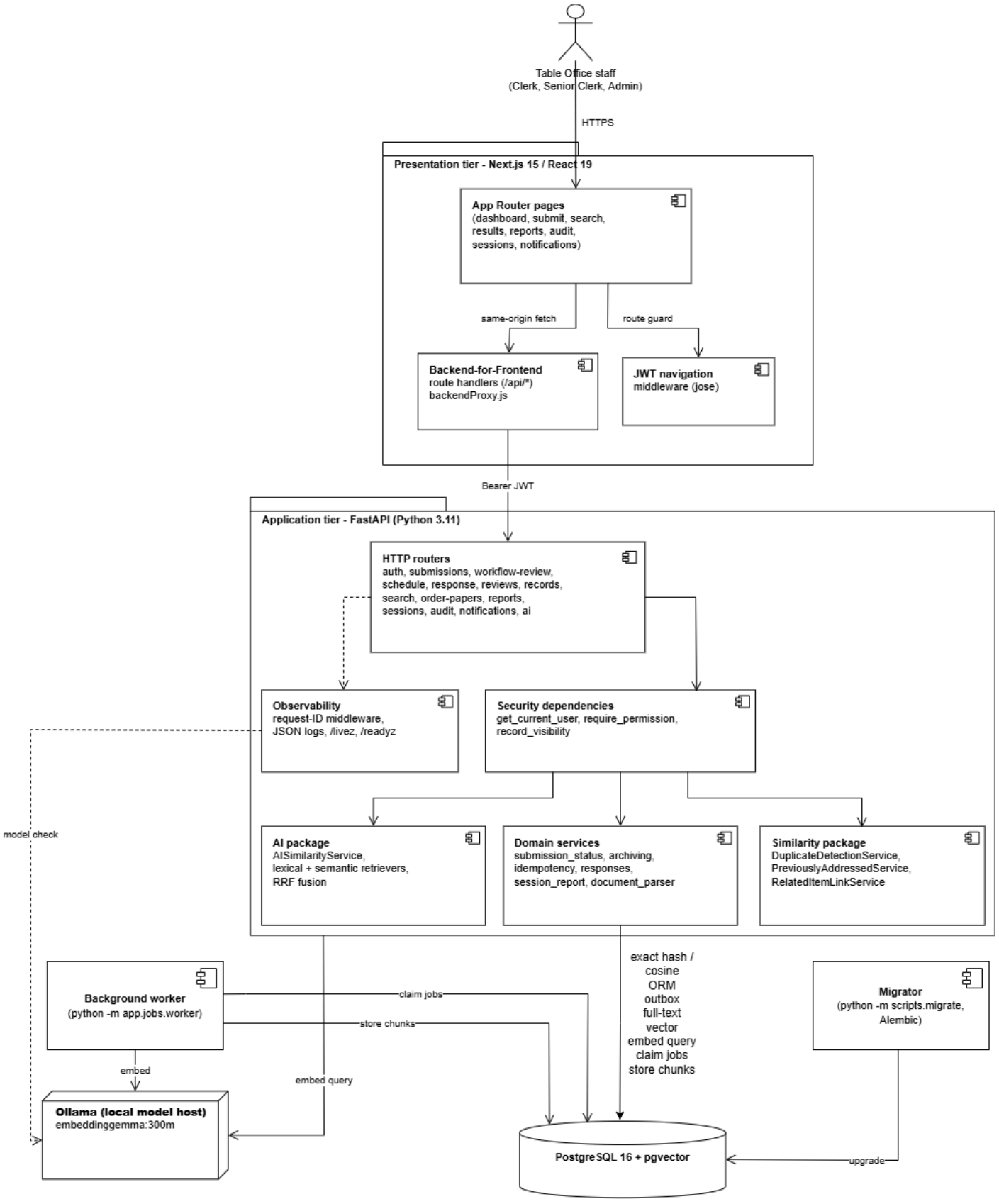


Figure C.4: Component architecture diagram

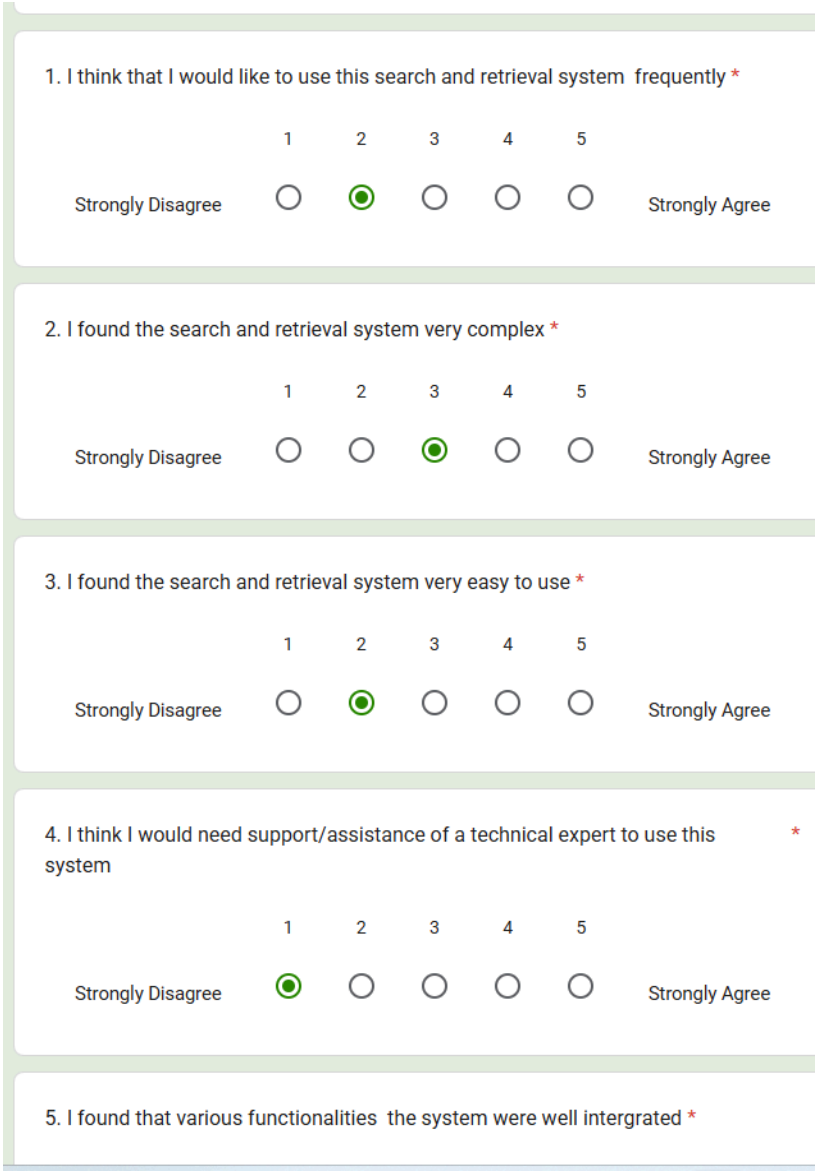


APPENDIX D: DATA COLLECTION INSTRUMENTS

A google form was used to record sus test results that were then calculated in a spread sheet. Below are some of the screenshots of the Google Form (Figure D.1 and Figure D.2).

Figure D.1: Google Form used to record SUS responses (first screenshot)

Figure D.2: Google Form used to record SUS responses (second screenshot)



7. I would imagine that most people would learn to use this system very quickly. *

	1	2	3	4	5	
Strongly Disagree	○	○	○	◉	○	Strongly Agree

8. I found the system very cumbersome to use. *

	1	2	3	4	5	
Strongly Disagree	○	◉	○	○	○	Strongly Agree

9. I felt very confident using the system. *

	1	2	3	4	5	
Strongly Disagree	○	◉	○	○	○	Strongly Agree

10. I needed to learn a lot of things before I could get going with this system. *

	1	2	3	4	5	
Strongly Disagree	○	○	○	◉	○	Strongly Agree

Submit Clear form