Final Report: SAT Decisions Scraper & Search Application with RAG

9900-T10A-CHOCOLATE

Miguel Ilagan (z5117944@ad.unsw.edu.au, z5117944, Data Engineer)

Guan Xin Miao (z5442846@ad.unsw.edu.au, z5442846, Front end Developer)

Will Ren (z5429870@ad.unsw.edu.au, z5429870, Machine Learning Engineer)

Zequn Wu (z5485493@ad.unsw.edu.au, z5485493, Product Owner)

Zhengxin Zeng (z5457832@ad.unsw.edu.au, z5457832, Data Architect)

Qinan Ji (z5399710@ad.unsw.edu.au, z5399710, Machine Learning Engineer)

Haowei Lou (Tutor)

Basem Suleiman (Lecturer)

April 29, 2025

Table of Figures

Table of Tables

1. Summary

As per the project specifications, The key objective of this project is to develop a Python application that automatically scrapes, processes and indexes SAT (State Administrative Tribunal) decisions. The application should allow natural language queries using Retrieval-Augmented Generation (RAG) via AI Language Model to provide contextually relevant results, summaries, and insights. Additionally, the system must allow authorised users to manually upload new SAT decision documents, which will be processed, embedded, and immediately searchable. The primary goal is to enable efficient search and analysis of tribunal decisions using AI-driven insights (democratisation of law in WA).

This document details the business and technical considerations of the developed RAG application. The report contains research of existing technologies that perform similar tasks, review on the business case of the proposed application, system overview, technical approach, performance evaluation product testing, cost/performance analysis of different models, challenges and limitations and future improvements.

2. Introduction

The State Administrative Tribunal (SAT) has most of their legal decisions and orders publicly available in their database on their official eCourts portal. However, while the database contains most SAT decisions over the past 20 years, more recent SAT decisions take considerable time to be available in the database which may cause complications for judges and lawyers who want to access/reference these cases. Furthermore, due to the abundant amount of SAT decisions available in the database, searching, analysing and summarising relevant cases is a challenging task for judges, lawyers and the general public who want to access all necessary information quickly. Currently, SAT decisions can only be searched based on keywords and filtering cases based on input from the user which can make searching for relevant decisions a very time consuming task if the user has no general direction of what to look for. Furthermore, each case contains an abundant amount of information which may need to be quickly analysed for use in court. However, judges, lawyers and the general public generally do not have sufficient time to properly read and process all the detailed information from these lengthy documents.

The developed RAG application aims to make the process of searching, analysing and summarising relevant SAT decisions easier by allowing users to provide natural language queries to the application which will then output contextually relevant results, summaries and insights. Furthermore, the application allows users to manually upload SAT decision documents which can then be used by the

application. The intended users are for judges, lawyers and the general public who need quick search and analysis of relevant SAT decision cases for use in court.

3. Business Case

Current traditional legal case searching methods only support keyword search which is often not accurate and efficient. Furthermore, since users are typically unaware of technical jargon which can be important for legal cases, this can affect the quality of the output presented to the user. The application developed in this project utilises RAG AI to enhance the accuracy and efficiency of legal case search and analysis by retrieving semantically relevant documents using text embeddings. The application allows users to quickly collect information about relevant cases through natural language queries. This will greatly reduce the need for the user to perform manual search to find relevant cases which often takes a significant amount of time.

Furthermore, current search applications typically only support text as input which can limit the quality of the returned output. As such, the developed RAG application will accept different input types such as DOCX and PDF documents which typically contain more detailed information on the user's requirements. This will allow the RAG search to generate outputs which meet the needs of the user more closely.

The application has been developed to support the following use cases:

1. Interact with the AI SATChat bot
2. Build legal arguments
3. Build a citation graph to view the relationship between cases and relevant legislation.

Investigation of existing legal RAG technologies was performed to assess their benefits/limitations. This research guided the development of the application and provided insight on the application's market potential. A brief analysis of three existing systems (extracted from the initial proposal) is shown below:

1. **Westlaw Precision Australia from Thomson** [28]
    a. Problem it solves: Westlaw Precision Australia of Westlaw Precision Global is a legal research platform combining traditional search and AI assisted search with case law, statutes, and secondary legal resources. It solves the problem of inefficient search in tons of legal cases by enabling keyword, filtering and context aware retrieval. Their genAI system provides CoCounsel skills including Research, Review, Summarize and Draft with GPT-4.
    b. Strengths to consider in our project:

i. Keyword matching from Australia's most trusted legal database.
ii. Filtering for legal cases such as jurisdiction, year etc.
iii. Provide citation tracking such as links for users to click.

c. Weakness and Limitations to avoid or overcome:
i. Having to pay hundreds of dollars per month to access this service which may prevent the public from accessing it.
ii. The system has limited adaptability to SAT decisions in WA and the algorithms may not be fully suitable for SAT. Currently does not offer reasoning capability for legal analysis.
iii. Do not have the option to customise for SAT decisions and to evaluate the system performance.

2. **LexisNexis Australia** [29]

a. Problem it solves: LexisNexis Australia addressed the problem of efficient legal search by providing a wide range of databases and enabling professionals to analyze legal texts efficiently. Their AI system can Create full drafts, accelerate legal research, and surface meaningful insights using both internal knowledge bases and industry-leading LexisNexis resources.

b. Strengths to consider in our project:
i. Advanced search functionality including natural language query, filters and alerts which allow users to find relevant sources quickly.
ii. Offer extensive practical guidance and checklists, helping lawyers work efficiently.
iii. Offers an legal research platform called Lexis+ AI which has key features including conversational search, document drafting, summarisation, and analysis while offering linked citations.

c. Weakness and Limitations to avoid or overcome
i. Lexis+ AI is a paid subscription service which is quite costly, making it difficult for individual users, small firms, and the general public to access.
ii. Currently does not offer reasoning capability for legal analysis.
iii. Lexis+ AI is optimized for court case law, statutes, and general legal research. It does not specifically cater to SAT rulings.

3. **RAGFlow** [30]

a. Problem it solves: RAGFlow is an open-source RAG platform designed to improve search and summarisation in various domains. Users can download the system and build their own knowledge base and RAG system with LLMs.

b. Strengths to consider in our project:

i. Open-source and can be extended. Users can build their own RAG system based on it with some customization.
ii. Supports different embedding and LLM models
iii. Can be applied locally in a short time.

c. Weakness and Limitations to avoid or overcome

i. Lacks legal domain optimisation since it is designed for general domain knowledge.
ii. Still need technical people to install and configure, and time to fine-tune all the strategies and techniques..
iii. No comprehensive option to evaluate system performance. The practical evaluation needs to be tailored for the specific use case.

4. Installation Manual

4.1. Download & Setup the project

Clone/Extract the Project

If from zip file:

unzip capstone-project-25t1-9900-t10a-chocolate-1-for-docker.zip

cd capstone-project-25t1-9900-t10a-chocolate-1-for-docker

If from GitHub:

git clone

<https://github.com/unsw-cse-comp99-3900/capstone-project-25t1-9900-t10a-chocolate-1.git>

cd capstone-project-25t1-9900-t10a-chocolate-1

IMPORTANT: if clone from Github, you also need to download the satdata.dump file from https://drive.google.com/file/d/1bq5CdAGMFfHIyH3Z6Xf6nzertUWE2t5Q/view?usp=drive_link and place it in the project root directory. This file contains all the legal case data required for the application to function properly.

4.2. Run the Project with Docker Compose

```
# Ensure Docker Desktop is running first
# If Docker Desktop is not running, open it and wait until it's fully running
open -a Docker     # For macOS
# or start Docker Desktop from your applications/start menu on Windows/Linux

# Start all services
docker-compose up -d

# View logs (optional)
docker-compose logs -f

# Stop services when done
docker-compose down
```

Figure 1. Docker Compose [31]

> **NOTE ABOUT FIRST RUN:** On the first startup, the backend will download language model files (approx. 450MB) needed for text embeddings. This initial download may take several minutes depending on your internet connection. During this time, the backend container may appear to be restarting or inactive in Docker Desktop. This is normal - please be patient and allow **5-10 minutes** for the download to complete and backend to be fully initialized. Subsequent startups will be much faster as these models are cached in a persistent volume.

> **IMPORTANT:** After starting the services, wait until the backend API is fully initialized before using the frontend. You can verify the backend is ready by visiting http://localhost:8000 in your browser - you should see the message: `{"message":"Welcome to the SAT Legal Decisions API. Visit /docs for API documentation."}`.

Figure 2. Note about First Run

The application can be accessed at:

- Frontend: http://localhost:3000
- Backend API: http://localhost:8000
- API Documentation: http://localhost:8000/docs
- Neo4j Citation Visualizer: http://localhost:5001/visualizer

4.3. Login with Credentials (For Testing)

Username: will.ren@student.unsw.edu.au

Password: 123456

4.4. Notes When Testing

When testing the Build Arguments feature:

- For faster processing, toggle to single call mode in the interface
- Choose Claude as the selected model for faster response
- A sample case query file Client_Case_Sample_v1.docx is available in the root directory which can be uploaded for testing

5. System Overview

The overall system architecture is split into 3 different parts as shown in the Figure 1 below. Description of each component of the application is provided in the following sections.

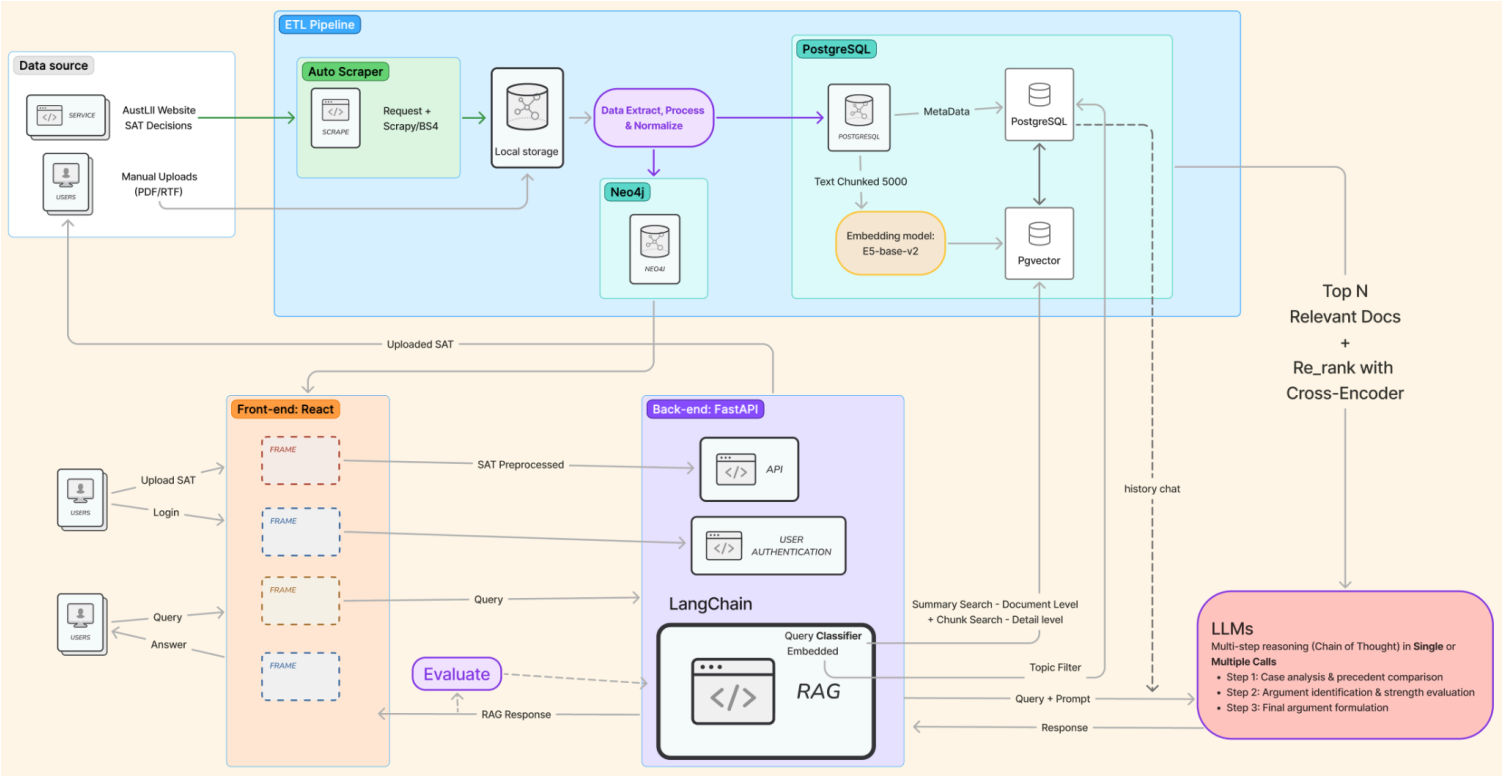

Figure 3. System Architecture

5.1. Data Layer

This layer includes an Automated Web Scraper and a Manual Document Upload Module. The web scraper extracts SAT decision data from the online AustLII database. The extracted text is then parsed, cleaned and processed. AI models are applied to generate summaries and convert the text into vector embeddings.

Each document is categorised into structured fields (e.g., citation number, legislations, etc.) and stored in a local PostgreSQL relational database. The embeddings are stored using pgvector, enabling efficient vector-based retrieval. This structured and vectorised data powers the RAG application, allowing for fast nearest-neighbour searches to identify contextually relevant decisions based on the provided user query.

We also store relationships between cases and laws in Neo4j Aura, a graph database that provides user-friendly visualizations.

5.2. AI and RAG Integration Layer

In this layer, the application processes user input and generates AI-driven results based on SAT decision data extracted and stored in the data layer. The user's query is first converted into a vector embedding using the same embedding model applied to documents in the data layer. The system then retrieves the most relevant SAT documents by measuring vector similarity and reranking the results, and filtered based on a predefined similarity threshold, which are used as input for the adopted large language models (LLMs) [32]. The LLMs generate contextually relevant summaries and build legal arguments which are presented to the user.

The following APIs are integrated to enable communication between the backend and frontend:

1. Chat API: Accepts natural language queries, performs embedding generation and vector search, and returns results with AI-generated summaries for frontend display.
2. Build Arguments API: Handles document uploads, performs embedding generation and vector search, and returns results with Chain-of-Thought reasoning from LLMs
3. User Management & Authentication: Manages user sessions, roles, and access rights to different functionalities.

5.3. Frontend Layer

This layer handles the web aspect of the application, providing an interface for users to interact with the system. The frontend layer supports the following features.

1. Chat + Arguments Interface: A web-based UI for users to enter natural language queries.
2. Results Display: Presents decision details, metadata, and AI-generated summaries and reasoning analysis.
3. Citation Graph: Users can click a button to view the citation of all related cases returned from RAG. Users can also check the citation graph of the cases that they are interested in which will help them to quickly have a visual intuition for the case, legislation and sections relationship.
4. Document uploading: A button to upload new documents and view the content before sending this message.
5. Responsive Design: Ensures the application is accessible and size dynamic across devices.
6. History Chat: a panel specifically designed for storing historical chat.

6. Technical Design

6.1. Frontend

This section outlines the technical approach to frontend development for the RAG application. Developed using React.js, the frontend serves as the user interface, enabling users to interact with the application through an integrated chat system, a legal argument builder and citation graph view. Additionally, the application allows users to upload documents, where the contained text is then extracted and automatically used as input for the chat system. The frontend communicates with the backend via several API endpoints to display AI-generated responses based on user input.

6.1.1. Project Summary

The frontend is developed as a web-based AI assistant designed to provide legal support to judges, lawyers, and the general public by leveraging insights from SAT decisions. It is developed using React.js and styled with custom CSS. Additional libraries are integrated for routing, PDF processing and DOCX rendering. The technologies used and core functionality of the application are summarised in the table below.

Item	**Description**
Project Type	Web-based AI assistant for legal support
Technology Stack	React.js (Frontend), Custom CSS, react-router-dom, pdfjs-dist, docx-preview
Core Functionality	Chat system, legal argument builder, citation graph view, document parsing, user authentication

Table 1. Project Summary

6.1.2. Key Functional Components

The frontend consists of various components, each serving a specific purpose within the application. These components are identified by their filenames, with their corresponding purposes and key features outlined in the table below.

Component	**Purpose**	**Key Features**

signUp.js	Handles user registration	Username, password, confirm password inputs, placeholder logic
loginPage.js	Provides login UI with local state	Username/password input, placeholder auth logic, redirect to ChatPage
forgotPassword.js	Reset password	Reset password
ChatPage.js	Main chat platform for interacting with the AI assistant	Chat history, topic dropdown, document upload (PDF/DOCX), markdown parsing, progressive display
CitationGraphPage.js	To view citation relationship in a graph	Search case, law and section. Add nodes to the graph. Toggle label etc.

Table 2. Key Functional Components

6.1.3. AI Communication Strategy

The communication between the frontend and backend for various tasks is outlined below.

Task Type	**API Endpoint**	**Behavior**
Chat	POST /api/v1/chat	Sends chat message, receives markdown-formatted AI reply
Build Arguments	POST /api/v1/build-arguments	Accepts case content/topic, returns insights, arguments, related cases
Citation Graph	Get /api/v1/visualizer	Access the citation visualizer interface

Table 3. AI Communication Strategy

6.1.4. Document Parsing Strategy

The frontend supports text extraction from uploaded PDF and DOCX files. The libraries used for parsing these documents are summarised in the table below.

File Type	**Library Used**	**Process**

PDF	pdfjs-dist	Loads pages and extracts all text content
DOCX	docx-preview	Renders document in hidden container, extracts visible text

Table 4. Document Parsing

6.1.5. User Interface and Styling

The frontend is styled to ensure that the application provides a comfortable and intuitive experience for users. The key design elements of the frontend application are outlined in the table below.

Area	**Description**
Layout	Flexbox-based structure with fixed header/sidebar and scrollable content areas
Theme Colors	Green and beige color palette (#435340, #CEDEBD, #065F46)
Chat Bubbles	Separate styling for user and bot messages, supports markdown
Sidebar Navigation	Includes history, task filters, and delete buttons
Message Input	Includes file upload, clear button, and send button with icon feedback

Table 5. User Interface and Styling

6.2. Data Layer

This section outlines the technical approach for data extraction, preprocessing, and storage in the application. The process begins by scraping SAT decision data from the AustLII database using Scrapy. The extracted information is then summarised using OpenAI's GPT-4o-mini model, and selected fields are embedded using the Hugging face E5-Base-V2 sentence-transformer model. The resultant data is stored in a JSON format and in a local PostgreSQL relational database with pgvector enabled for vector similarity search. FastAPI endpoints are used to query the database and retrieve relevant data for the backend.

6.2.1. Data Layer - Acknowledgements

The implementation of the data pipeline closely follows publicly available tutorials:

1. Scrapy Course – Python Web Scraping for Beginners - freeCodeCamp
2. How to build a FastAPI app with PostgreSQL - Eric Roby

Additional references are cited within the codebase.

6.2.2. Data Layer - Requirements

The key technologies used are:

1. Scrapy
2. PostgreSQL
3. pgvector

To install all required python libraries/frameworks:
pip install scrapy psycopg2 sentence-transformers langchain langchain-community openai fuzzywuzzy fastapi sqlalchemy psycopg2-binary pgvector uvicorn

6.2.3. Data Layer - Extraction and storage method

The data extraction process is handled in satspider.py, which defines the structure of each scraped document (e.g., citation number, legislation, etc.).
Scraped items are then processed through a series of pipelines defined in pipelines.py and enabled via settings.py:

1. SatscraperPipeline - Cleans and formats raw scraped text data (e.g., removes special characters, formatting specific fields)
2. TopicMappingPipeline - Maps each SAT decision to a case_topic based on its case_act, using mappings derived from the official SAT eCourts portal.

3. SavingToPostgresPipeline - Generates case summaries using OpenAI, produces text embeddings for selected fields, and saves the structured data into the PostgreSQL database.

FastAPI endpoints are implemented in main.py and database.py for querying the stored data.

6.2.4. Data Layer - Schema overview

The local relational PostgreSQL database stores structured SAT decision information as per the field structure of each case document. The database supports fast vector-based semantic search using pgvector. Below is a data schema overview of the tables used in the application.

satdata Table:

Field	**Data Type**	**Description**
id	SERIAL PRIMARY KEY	Unique identifier for each SAT decision document in the PostgreSQL database.
case_url	TEXT UNIQUE	Unique address/location of SAT decision document webpage.
case_title	TEXT	Case title.
citation_number	TEXT	Citation number of SAT decision document.
case_year	TEXT	Year of the case.
case_act	TEXT	Act associated with the SAT decision.
case_topic	TEXT	Topic associated with SAT decision derived from case_act field.
member	TEXT	Named members involved in the case.
heard_date	DATE	Date of hearing of the case.
delivery_date	DATE	Date of delivery of the case.
file_no	TEXT	File number of the case.
case_between	TEXT	Named parties involved in the case e.g. Applicant and Respondent

Field	Data Type	Description
catchwords	TEXT	Summary of key legal issues, statutes and principles raised in the decision.
legislations	TEXT	Legislations associated with the case.
result	TEXT	Result of case.
category	TEXT	Category of case.
representation	TEXT	Legal representation details of parties involved in the case, including names of Counsel and Solicitors representing the applicant and respondent.
referred_cases	TEXT	List of past SAT decisions referred to or cited.
reasons	TEXT	Reasons for decision of the tribunal.
reasons_summary	TEXT	Summary of reasons field
reasons_summary_embedding	vector(768)	Text embedding vector (768-dimension) of the reasons_summary, used for semantic search and retrieval.

Table 6. Satdata Schema

A database index is created on the case_topic column of the satdata table. This significantly speeds up searching and filtering on case_topic.

reasons_chunks Table:

Field	Data Type	Description
id	SERIAL PRIMARY KEY	Unique identifier for each chunk of the reasons text (from the satdata table) of a SAT decision document in the PostgreSQL database.
case_id	INTEGER REFERENCES satdata(id)	Foreign key which references primary key (id) of the satdata table. Enforces referential integrity with ON DELETE CASCADE.

Field	Data Type	Description
	ON DELETE CASCADE	
case_topic	TEXT	Topic associated with the SAT decision.
chunk_index	INTEGER	Position Index to indicate order of chunk within the full reasons text.
chunk_text	TEXT	The actual chunk of text extracted from the reasons field of the satdata table.
chunk_embedding	vector(768)	Text embedding vector (768-dimension) of the reasons_summary, used for semantic search and retrieval.

Table 7. reasons_chunks Schema

The Entity-Relationship Diagram (ERD) below visualises the structure of the relational database.

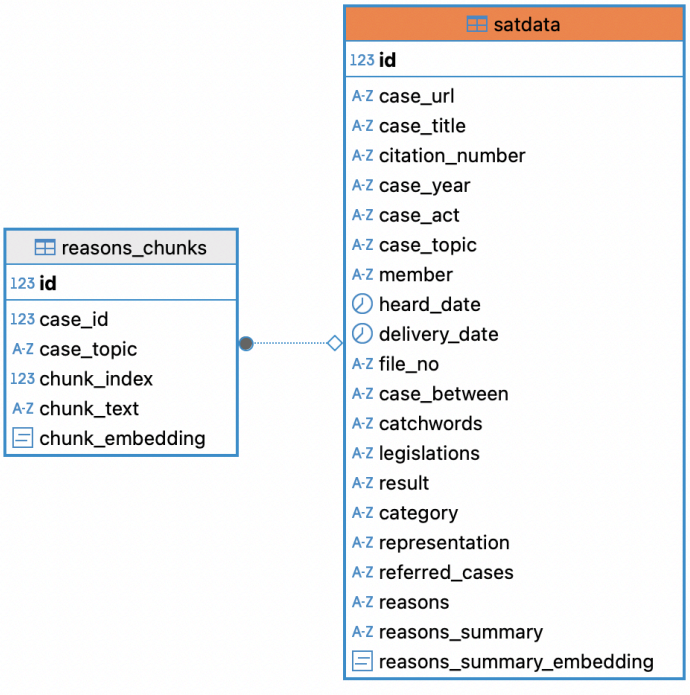

Figure 4. Entity-Relationship Diagram

6.2.5. Data Layer - Summary generation

The reasons field, which contains the reasons for the decision of the tribunal, is summarised to provide concise and readily available insights for each SAT decision. Summaries are generated using

OpenAI's GPT-4o-mini model. A prompt is fed into the model to extract key points from the reasons text and is shown below.

```
prompt = f"""
    Please summarise the following State Administrative Tribunal (SAT) decision cases following the below format:

    1. Introduction - A brief overview of the case and what it concerns.
    2. Background - Context about the parties involved and the nature of the dispute.
    3. Relevant Acts and Legislation - Highlight the referenced acts and legislation.
    4. Key Arguments - Outline the main arguments presented by involved parties.
    5. Key Insights - Summarise the reasons for the tribunal and the key findings.
    6. Outcome - Which parties won and lost the case. Detail why the respective parties won/lost.

    Reasons for the tribunal:
    {text}
    """
```

The resulting summary is then stored into the reasons_summary field in the satdata table and then used to create vector embeddings. The resulting vector embeddings are then stored in reasons_summary_embedding field in the satdata table.

6.2.6. Data Layer - Embeddings

Embedding models encode semantic information of text data [10] into compact vector format that can be used by the RAG application for semantic search and retrieval, and generate reasoning. Vector embeddings are generated using the text stored in the reasons_summary field in the satdata table as described in the previous section. Additionally, embeddings are also generated from chunked text of the reasons field in the satdata table for each SAT decision case. Chunks are generated using 'content-aware' Recursive chunking which iteratively splits the input text based on defined separators [14]. LangChain's RecursiveCharacterTextSplitter is used to chunk text with a chunk size of 5000 and chunk overlap of 500 based on performed experimentation on optimal chunk size. Embeddings are generated for each chunk using the Hugging face E5-Base-V2 model developed by Liang Wang, Nan Yang, Xiaolong Huang, Binxing Jiao, Linjun Yang, Daxin Jiang, Rangan Majumder, and Furu Wei [11]. The resulting vector of each chunk is stored in the reasons_chunks table in the chunk_embedding field. More details on the E5-Base-V2 model are available in the corresponding literature "*Text Embeddings by Weakly-Supervised Contrastive Pre-training*" [1].

6.2.7. Data Layer - Neo4j AuraDB

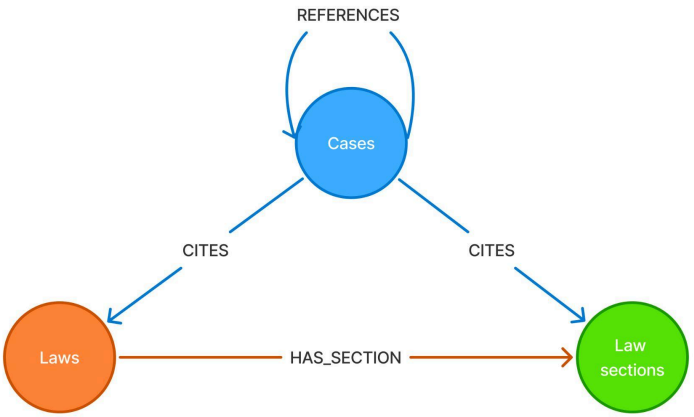

Figure 5. Neo4j Design

Two main components of a graph database are detailed below:

1. **Nodes design**

   Nodes represent objects or entities, abstracted from the real world. In the context of case law, we can identify three main types of nodes:

   a. Case: Represents individual legal cases.
   b. Law: Represents legal statutes or acts.
   c. Section: Represents specific provisions or sections within a law.

2. **Relationships design**

   Relationships provide connections between two nodes. We define three types of relationships:

   a. REFERENCES: Indicates that a case references other cases.
   b. CITES: Indicates that a case cites a specific law or a specific section of a law.
   c. HAS_SECTION: Indicates that a law contains specific sections (e.g., s 77(2)).

The benefits of using a graph database are detailed below:

1. **Visualisation:**

   Graph databases offer intuitive visual representations of how cases, laws, and sections interconnect, making it easier to explore and understand complex legal relationships.

2. **Relationship Insights:**

   By emphasising connections, graph databases help in uncovering patterns such as which cases reference the same law or how frequently certain sections are cited, insights that can be more difficult to spot in traditional relational databases.

The application of the graph database is described below:
The structure of case law, which is inherently interconnected, is naturally modeled as a graph, reducing the complexity of storing and querying legal references.

These advantages allow us to better capture the interconnected nature of legal data, offering deeper insights and more user-friendly visualisations than traditional database systems.

6.3. Backend

This section outlines the technical approach for the backend component of the application, which provides APIs for users to interact with the system, retrieve similar legal cases, build legal arguments and visualise citation networks. The RAG pipeline, illustrated below, serves as the core functionality of the application. It retrieves relevant legal cases and includes them as context into the LLM prompts.

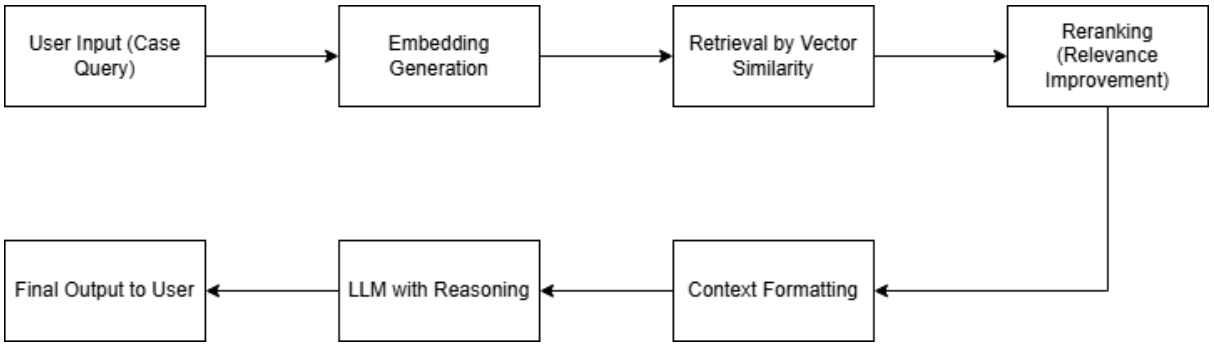

Figure 6. RAG Pipeline

The following sections detail the individual processes that make up this layer.

6.3.1. Backend - Retrieval Process

The retrieval process fetches relevant SAT documents based on a user's submitted query. The query is converted into a vector embedding using the same model described in Section 6.2.6 (E5-Base-V2), and compared against document embeddings stored in the local PostgreSQL database (Pgvector). The system ensures that all user queries follow the required format as detailed in the documentation for the adopted E5-Base-V2 model [11] which requires "query" to be added as a prefix. Similarity between the query embedding and document embeddings is computed by querying the database using the <-> operator for L2 vector distance to retrieve an initial set of candidate documents. A candidate multiplier mechanism is applied to retrieve more documents (2 times the required amount) before reranking to improve output quality. The retrieved documents are then reranked using cross-encoder ms-marco-MiniLM-L6-v2 model from SentenceTransformers to improve the relevance of results. Longer contexts are handled using Longformer reranking. Contextually relevant chunks from the reranked documents are selected based on a defined similarity threshold and progressed to the next stage[33].

An effective similarity threshold of 0.25 is defined in rag/generation.py which was empirically determined to ensure that the system retrieves sufficient documents in case strict matching is not

possible. Four documents are passed to the LLM as it provides sufficient contextual information for the LLM to generate meaningful responses while maintaining token usage.

As per research contained within the initial proposal, the reranking process changes the order of output results based on relevance to the search query. The system supports retrieval at both the document and chunk levels, with reranking as an option. Retrieval is implemented in rag/retrieval.py. IVFFlat indexing on the embedding columns (reasons_summary_embedding for full documents and chunk_embedding for document chunks) in the PostgreSQL database (Pgvector) is enabled for efficient nearest neighbour vector search. Additionally, case topic based filtering is supported to enable refined retrieval of case documents.

A general overview of the reranking process is shown in the Figure below. Relevance scores are computed using the cross-encoder model for each query and candidate document pair. The documents are then sorted in descending order based on their score. More details on reranking is available on sbert.net [19].

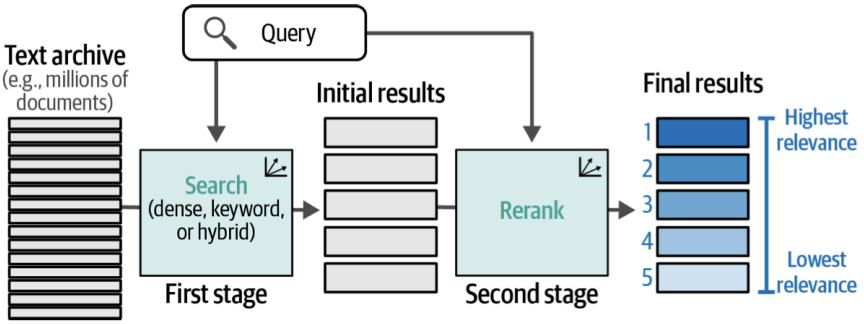

Figure 7. Reranking Process

6.3.2. Backend - Context Formatting

Documents returned from the retrieval process are structured into a formatted context containing case title, citation, URL, and extracted key points, which is then provided as input to the LLM. Each context entry explicitly labels whether it is a full document or a chunk. An example of the formatted context is shown below:

SIMILAR CASES:
1. [DOCUMENT] CASE A v. CASE B (2023)
   URL: https://...
   Key points: [Summary of relevant points]
   Similarity: 0.82

2. [CHUNK] CASE C v. CASE D (2022)
URL: https://...
Key points: [Summary of relevant points]
Similarity: 0.75

The system has a fallback mechanism that dynamically reduces the vector similarity threshold in the event that the primary threshold is not met to ensure that documents are still retrieved.

6.3.3. Backend - Optimised Reasoning Steps

The system supports a multi-call approach, where the selected LLM model is called multiple times to progressively generate reasoning. This method emulates a chain-of-reasoning process [34], progressively building legal arguments across multiple stages. This multi-step process aims to enhance the quality of legal reasoning by emulating a lawyer's workflow: first understanding the problem, then identifying potential arguments, and finally constructing structured arguments based on their analysis. Legal arguments are developed through the following three-step process, with each step using the outputs from the previous stage to refine the generated reasoning:

1. LLM analyses and compares the user's case with the retrieved similar cases.
2. Potential legal arguments are generated and assessed based on strength and relevance.
3. LLM formulates final structured arguments, including supporting cases and reasoning.

Technical Implementation:

1. Each reasoning step uses explicit prompt templates customized for the stage objective.
2. Token usage is monitored and optimized at each step using count_tokens() to stay within LLM context limits.
3. Intermediate step outputs are structured and persisted across calls.

6.3.4. Backend - Single-Call alternative

The system provides an alternative setting that enables a single-call approach [35], where the selected LLM model is called only once. The single-call reasoning approach uses a comprehensive prompt to follow the same reasoning process outlined in the multi-call approach. Outputs generated using this method may provide less detailed analysis, but significantly improve the latency of the response. This setting is offered to users as an option, depending on their individual requirements.

Technical Implementation:

1. Specialized prompt engineering embeds multiple reasoning stages into one structured prompt.
2. Output parsing ensures format consistency.

3. This method significantly improves latency while slightly trading off depth of analysis compared to the multi-call process.

6.3.5. Backend - Fallback Mechanisms

The system has a sophisticated fallback mechanism [36] to handle various edge cases:

1. Alternative vector similarity thresholds are supported in the retrieval process in the event that the primary threshold is not met.
2. Alternative LLM providers are available in the event that the primary model fails.
3. Multiple output parsing strategies are implemented to handle variations in LLM output formatting
4. Robust error handling and retry logic are built into each LLM provider client

6.3.6. Backend - LLMs

The system integrates multiple LLMs across different features to optimize performance, reasoning quality, reliability, and cost-effectiveness.

Feature-Specific Optimization

1. Chat: OpenAI's gpt-4o model [37] is used for general conversational interactions, prioritizing speed and responsiveness
2. Build Arguments: Users may select between Claude 3.7 Sonnet (Anthropic provider) [38] or DeepSeek R1 [39] to generate structured legal arguments with deeper reasoning capabilities.

Capability Differences

1. gpt-4o: General-purpose model with strong responsiveness, ideal for interactive chat.
2. Claude 3.7 Sonnet: Excels at nuanced, structured legal reasoning.
3. DeepSeek R1: Specialized in multi-step reasoning, suited for complex legal tasks.

Reliability and Fallback: Multiple providers (OpenAI, Anthropic, DeepSeek) ensure redundancy against API failures and allow dynamic fallback if a primary model fails.

Cost Optimization: Different models have different cost-performance tradeoffs; the system selects simpler models for lightweight tasks and reserves powerful, higher-cost models for complex reasoning.

Multiple LLMs are integrated across different application features to optimise response quality and user experience. Selection of the models primarily considers performance, which is critical for legal applications which require a high degree of accuracy. In summary, gpt-4o is chosen for "Chat" due to its low latency and responsiveness, suitable for interactive, general-purpose conversations. Claude 3.7 Sonnet and DeepSeek R1 are used for "Build Arguments" because of their stronger multi-step reasoning abilities, better suited to the demands of legal analysis by professionals such as lawyers and judges. Further detail on model analysis is detailed in Section 7.

6.3.7. Backend - APIs

The backend provides a secure RESTful API architecture built with FastAPI [40], enabling interaction between the frontend and core backend services, including retrieval, LLM generation, and document management.

1. Chat API
    a. Supports real-time RAG-enhanced chat with conversational context tracking.
    b. Maintains conversation history for coherent multi-turn interactions.
    c. Allows both authenticated and anonymous sessions.
2. Build Arguments API
    a. Generates structured legal arguments using multi-step or single-call reasoning.
    b. Retrieves and cites similar cases to support argument development.
    c. Supports saving argument sessions to conversation history.
3. Citation Graph API
    a. Users can click a button to view the citation of all related cases returned from RAG.
    b. Users can also check the citation graph of the cases that they are interested in which will help them to quickly have a visual intuition for the case, legislation and sections relationship.
4. User Management API
    a. Handles user registration, authentication, and session control.
    b. Provides JWT-based access token.

6.3.8. Backend - Structure of backend

The backend adopts a modular and maintainable architecture, separating concerns across different functional components to support scalability, code reuse, and future extensibility. The system is built using FastAPI and organized around clear module boundaries [41].

1. Application Core (app/)

    API Layer: Defines RESTful endpoints for search, chat, argument building, uploads, and user management.

    Database Layer: Manages SQLAlchemy ORM models (e.g., Case, CaseChunk, User) and PostgreSQL connection with Pgvector extension for efficient vector storage.

    Utilities: Common helper functions for database operations, authentication, and configuration management.

2. RAG Modules (rag/)

    Embeddings: Generates vector embeddings using E5-Base-V2 with custom query prefixing.

Retrieval: Performs vector similarity search and cross-encoder reranking (ms-marco-MiniLM-L6-v2) to retrieve relevant documents or chunks.

Generation: Implements multi-step and single-call reasoning workflows for legal argument construction.

LLM Providers: Abstracts integrations with multiple LLMs (OpenAI, Anthropic, DeepSeek) supporting synchronous and streaming generation.

3. Configuration (app/config.py)

Centralizes environment variables, model settings, API keys, and prompt templates.

Supports flexible switching between different LLM providers and retrieval thresholds.

4. Testing Modules (tests/)

Contains unit tests and integration tests to validate API functionality, retrieval correctness, and LLM generation workflows.

7. Design Justifications

This section provides justification for the chosen design based on cost-performance analysis of alternative technologies considered for this project. This section also showcases the evolution of the design from proposal based on project requirements and the consequent iteration of the design.

7.1. Frontend - Cost comparison

Technologies considered for frontend development are compared Table 8 below. For this project, React, Vue, and Angular were evaluated.

Aspect	**React**	**Vue**	**Angular**
License	MIT (Free)	MIT (Free)	MIT (Free)
Learning Curve [20]	Moderate	Low	High
Flexibility [21]	High	High	Moderate
Developer Availability [20]	Very High	Moderate	High
Average Developer Salary (USD/year) [22]	~$94K	~$91K	~$106K
Features [21]	Component-Based, Virtual DOM, JSX	Approachability, Flexibility, Reactivity	Full-Featured Framework, MVC Architecture, Two-Way Data Binding
Size [21]	31.8KB (core)	23KB (core)	143KB (core)

Table 8. Frontend Cost Comparison

React is flexible and scalable, with a strong developer community and it is a good balance between learning curve and project growth. It's a good choice for applications that need to expand over time as it has the highest developer availability.

Vue is easier to learn and lighter, making it ideal for small to medium-sized projects. Fast development and lower cost projects are more suitable.

Angular has the most built-in features but also the moderate learning curve and the biggest size. It's more suitable for large and complex enterprise projects where structure and full functionality are important.

For this project, React is selected due to its balance on flexibility, scalability, and strong foundation for future growth. Its component system makes it easier to build and manage complex pages, such as AI chatting, built arguments , and graph data visualisation (for judges, lawyers, and clients). Additionally, React's large ecosystem supports quick integration of features like real-time updates, which is important for a fast and reliable legal application.

7.2. Data layer - Cost comparison

The technologies considered for the data extraction and storage process are analysed and compared with respect to cost and performance. The technologies are associated with processes involving data extraction, storage, embedding and summarisation that occur in the data layer.

7.2.1. Data extraction - Cost comparison

A comparison was conducted between available tools for extracting structured data from web pages. For this project, Scrapy was selected due to its high-performance, asynchronous design and built-in support for scalable web crawling. Scrapy is particularly well-suited for applications like RAG, which require the extraction of large volumes of structured data across multiple web pages. An evaluation of alternative tools, including BeautifulSoup, was informed by research and tool comparisons such as those provided by ZenRows [2].

Scrapy offers a high-level framework designed specifically for web scraping and crawling. Its asynchronous architecture allows it to handle multiple web requests concurrently, significantly improving scraping speed and efficiency for large-scale tasks. Additionally, Scrapy supports the creation of custom data pipelines and can export data in multiple formats including CSV, JSON, and XML, or directly insert it into a database. Through these custom pipelines, data can be processed and stored. The modular design of these pipelines allows for the data to be easily integrated with the

adopted PostgreSQL database. Furthermore, Scrapy is open-source and can be executed locally. Hence, its operational cost is limited to the compute resources available to the developer.

In contrast, BeautifulSoup is a more lightweight tool that excels at parsing HTML and XML content but lacks native support for concurrent scraping. It is generally more appropriate for small-scale or one-off scraping tasks. It also requires third-party libraries to support features like data export, and does not provide the same level of pipeline customisation or scalability as Scrapy.

7.2.2. Database - Cost comparison

7.2.2.1. Database Management Systems

There are several viable options for storing the extracted data. For this project, PostgreSQL was selected after evaluating its capabilities against alternative databases [3, 4]. An article by *altexsoft* compares 12 most commonly used relational and non-relational Database Management Systems (DBMS) based on its advantages, disadvantages and use cases [4]. For the purpose of this report, comparison between PostgreSQL and MongoDB is documented.

PostgreSQL is an open-source relational database management system that provides strong support for storing and querying structured data using SQL. The SAT decision data from AustLII includes structured metadata, making PostgreSQL an ideal fit due to its ability to define schemas, enforce data integrity, and support complex queries.

For RAG applications, PostgreSQL supports vector-based similarity search through extensions such as Pgvector, allowing efficient storage and querying of vector embeddings. This makes it particularly suitable for AI-driven applications that rely on semantic search. Additional benefits of PostgreSQL include ACID compliance (ensuring data consistency and reliability), scalability and ability to run locally.

By contrast, MongoDB is a popular non-relational database that stores data in flexible, JSON-like documents and supports indexing and fast search. However, MongoDB is well-suited for unstructured or rapidly evolving data schemas using the MongoDB Query Language (MQL), which poses a steeper learning curve for our team which is more familiar with SQL.

Hence, PostgreSQL is the more appropriate choice for this project due to the structured nature of the data and the need for relational integrity and integrated vector-based querying.

7.2.2.2. Vector Database

The vector database stores all generated text embeddings and is responsible for efficient semantic search and retrieval which power the RAG application. A comparison between different vector databases is provided by Superlinked [5] which provides detail on the supported features and applications for each database. Furthermore, discussion is provided by MyScale [6] and Tejashri R.P [7] on vector databases such as FAISS and Pinecone.

FAISS supports fast retrieval of high-dimensional data using highly optimized algorithms for exact and approximate nearest neighbor search. It is designed for speed and memory and can handle large volumes of data. However, it requires greater technical expertise and lacks user-friendly integration with traditional databases.

Pinecone is a cloud-native vector database designed for natural language processing (NLP) applications. It is scalable, easy to integrate, and performs well for vector similarity search. However, as a third party service, it introduces architectural complexity and additional operational costs.

For this project, Pgvector was selected due to its native integration with PostgreSQL. It supports efficient vector operations directly within the existing database, eliminating the need to manage a separate vector database. This greatly simplifies the architecture and enables integration between embedding models and relational queries which is important for effective RAG implementation.

7.2.3. Embeddings - Cost comparison

Multiple text embedding models were evaluated based on the '*Massive Text Embedding Benchmark*' (MTEB) Leaderboard, hosted by Hugging Face, which compares over 100 text embedding models across a diverse range of embedding tasks [8, 9, 10]. For this project, the E5-Base-V2 model is selected due to its ease of integration, high quality embeddings and zero operational costs when executed locally. It is Borda ranked 56th on the MTEB leaderboard, indicating strong overall performance across multiple tasks.

As per documentation, the prefix "query:" and "passage:" is required to be added to the input text to improve the quality of the embedding, as per training methodology of the E5-Base-V2 model [11]. For the purpose of the RAG application, "passage" is added as a prefix to the text for generating vector embeddings to better encode text data for semantic search and summary generation. An example input from SAT decision case '*[2025] WASAT 28*' is formatted for embedding and is shown below:

"*passage: In October 2019 the applicants, Richard Barnard and Louise Barnard entered into a contract with the respondent, Barrier Reef Pools South West Pty Ltd for it to supply and install a fibreglass pool at their property in Bunbury and the pool was installed in December 2019.*"

While other widely used embedding models such as Sentence-BERT (e.g., all-MiniLM-L6-v2), OpenAI embeddings (e.g., text-embedding-ada-002), and Cohere embeddings (e.g., embed-english-v3.0) produce strong results, they require payment to utilise the models. In contrast, E5-Base-V2 offers strong performance at zero operational costs when executed locally which is suitable for the RAG application.

A comparison between select Sentence-BERT, OpenAI, Cohere embedding models and E5-Base-V2 is extracted from the MTEB Leaderboard [9] and shown in the following table.

Rank (Borda) *	Model *	Memory Usage (MB) *	Number of Parameters *	Embedding Dimensions *	Max Tokens *	Mean (Task) *	Mean (TaskType) *	Reranking *	Retrieval *	STS *
10	Cohere-embed-multilingual-v3.0	Unknown	Unknown	1024	Unknown	61.1	53.31	64.07	59.16	74.8
40	Cohere-embed-english-v3.0	Unknown	Unknown	1024	Unknown	48.19	41.85	50.68	45.41	64.72
56	e5-base-v2	418	109M	768	512	46.03	39.41	47.5	40.49	60.73
113	all-MiniLM-L6-v2	87	22M	384	256	41.43	35.18	40.28	32.51	56.08
158	text-embedding-ada-002	Unknown	Unknown	1536	8191	-	-	-	-	-

Table 9. MTEB Leaderboard Extract

*** Each of the metrics measured in the table above are defined on the official MTEB Leaderboard website [9].**

7.2.4. Cloud graph database - Cost comparison

Below is a table showing rough monthly cost ranges for a single-region production graph database instance across small, medium, and large I/O workloads. These numbers are estimates based on publicly available pricing info, example instance sizes/tiers, and assumed concurrency. Actual costs will vary.

I/O Tier (Est. Monthly Operations)	**Neo4j Aura (Fully Managed)Approx. Monthly [23]**	**Amazon Neptune (AWS) Approx. Monthly [24]**	**Azure Cosmos DB (Gremlin API) Approx. Monthly [25]**	**TigerGraph Cloud(Managed) Approx. Monthly**
Small (up to ~5M ops/month) (<10 QPS)	$80 – $130 (Entry-tier Aura "Pro")	$90 – $150 (db.r5.large + minimal I/O)	$30 – $100 (few hundred RU/s + minimal storage)	$90 – $180 (starter instance + minimal usage)
Medium (5M–50M ops/month) (10–100 QPS)	$120 – $600 (Larger Aura Pro tier)	$150 – $400 (db.r5.xlarge + moderate I/O)	$100 – $400 (moderate RU/s provisioning + storage)	$240 – $750 (mid-tier cluster)
Large (50M+ ops/month) (>100 QPS)	$600+ (Aura Enterprise tiers)	$400 – $1,200+ (larger instances + high I/O)	$400 – $1,250+ (high RU/s provisioning)	$750+ (enterprise-tier cluster)

Table 10. Cloud Graph Database

7.2.4.1. Cost explanation of Cloud graph database

Below is a concise description of the **Small / Medium / Large** usage tiers for popular graph databases in the Sydney (ap-southeast-2) region, including official pricing links for verification.

Usage Tiers Overview

1. Small
   - Up to ~5M operations/month (reads + writes)
   - Low concurrency (generally <10 QPS)
   - Typically a single small instance or low Request Units (RU/s)
2. Medium
   - 5M–50M operations/month
   - Concurrency 10–100 QPS
   - Moderate instance size or RU provisioning
3. Large
   - 50M+ operations/month
   - Concurrency >100 QPS
   - Larger/multiple instances or high RU provisioning

Provider-Specific Examples (Sydney Region)

Neo4j Aura

- Small: 1GB AuraDB Professional tier (e.g., ~\$80–\$130/month in Sydney)
- Medium: 4–8GB AuraDB Professional (\$200–\$600+)
- Large: 16GB+ AuraDB Business Critical (can exceed \$600–\$1,000/month)
- Official Link: Neo4j Aura [23]

Amazon Neptune

- Small: Single db.r5.large (\$0.42/hr) with minimal I/O, often ~\$90–\$150/month if usage is light or part-time
- Medium: db.r5.xlarge or multiple replicas; \$150–\$400+ depending on I/O and full-time usage
- Large: Larger instances (e.g., db.r5.4xlarge) or multi-replica setups can exceed \$1,000/month under heavy read/write
- Official Link: Amazon Neptune [24]

Azure Cosmos DB (Gremlin API)

- Small: ~400–1,000 RU/s single-region, storage a few GB, ~\$30–\$100/month
- Medium: 2,000–10,000 RU/s, potentially \$100–\$400/month
- Large: Tens of thousands of RU/s or multi-region replication can surpass \$400–\$1,000+ monthly
- Official Link: Azure Cosmos DB [25]

TigerGraph Cloud

- Small: Small instance (e.g., TG-00 / TG-0) possibly part-time. Overall ~\$90–\$180/month if you don't run 24/7
- Medium: Mid-tier (TG-1 / TG-2) in Sydney, \$240–\$750+ if run continuously
- Large: Enterprise configurations (TG-4 and above) or high availability can exceed \$750–\$1,000/month
- Official Link: TigerGraph Savanna (Cloud) Pricing [26]

These "Small / Medium / Large" groupings are illustrative. Actual cost depends on:

- How many hours per day your instance(s) run
- The volume of read/write operations (or RU/s)
- Storage size and replication needs

7.2.4.2. Why Choose Neo4j for Phase 1 (Small Project)

- Ease of Implementation
    - Neo4j Aura offers a fully managed, user-friendly setup. You can spin up a database quickly without needing deep DevOps knowledge or cluster management skills.
- Lower Cost at Small Scale
    - Neo4j Aura's entry-tier ("Aura Professional") is competitively priced at $80–$130/month in the Australia region for a low-volume workload. This tier often covers early-stage projects comfortably.
- Rich Graph Features & Mature Ecosystem
    - Neo4j's Cypher query language is widely adopted and relatively straightforward for developers who have some SQL background.
    - Large community support and extensive tooling (e.g., Neo4j Bloom) make onboarding simpler.

7.2.5. Summary generation - Cost comparison

Multiple large language models (LLMs) were evaluated based on the 'LLM Leaderboard' on llm-stats.com which provides analysis and comparison of AI models across benchmarks, pricing and capabilities [13]. In particular, the performance of the models were evaluated based on its *MMLU* and *DROP* benchmarks which influenced the selection of the AI model used for the summarisation component of the RAG application. The *MMLU (Massive Multitask Language Understanding)* benchmark measures the performance of a model across 57 subjects from STEM, humanities, social science subjects and specialised areas such as law and ethics [15]. The DROP (Discrete Reasoning Over Paragraphs) benchmark measures the performance of a model's reading comprehension and reasoning capabilities [13, 16]. These benchmarks closely align with the goals of this RAG application, which requires the application to understand and generate reasoning for complex legal cases.

Selection of the model was influenced by the API keys provided by the client for LLaMA, Qwen, OpenAI, and DeepSeek models. For this project, Open AI's GPT-4o-mini model is selected due to its strong performance with respect to both *MMLU* and *DROP* benchmarks (82.00% and 79.70% respectively), and cost effectiveness at $0.15 per million tokens. Compared to alternative models, the GPT-4o-mini model provides a strong balance between cost and performance which make it an attractive option for the RAG application.

Furthermore, since summarisation occurs in the data layer, summaries for legal cases are generated once and then stored in the local PostgreSQL database. This eliminates costs associated with repeated

summary generation which makes the GPT-4o-mini model a suitable option for the relatively large SAT dataset.

The following table provides a comparison of LLaMA, Qwen, OpenAI, and DeepSeek models based on benchmark performance and cost, as provided by llm-stats.com [13]:

Model *	License *	Parameters (B) *	Context *	Input $/M *	Output $/M *	MMLU *	DROP *
GPT-4	Proprietary	-	32,768	$30.00	$60.00	86.40%	80.90%
o1	Proprietary	-	200,000	$15.00	$60.00	91.80%	-
o1-preview	Proprietary	-	128,000	$15.00	$60.00	90.80%	-
GPT-4 Turbo	Proprietary	-	128,000	$10.00	$30.00	86.50%	86.00%
o1-mini	Proprietary	-	128,000	$3.00	$12.00	85.20%	-
GPT-4o	Proprietary	-	128,000	$2.50	$10.00	88.00%	-
o3-mini	Proprietary	-	200,000	$1.10	$4.40	86.90%	-
Llama 3.1 405B Instruct	Open	405	128,000	$0.90	$0.90	87.30%	84.80%
DeepSeek-R1	Open	671	131,072	$0.55	$2.19	90.80%	92.20%
GPT-3.5 Turbo	Proprietary	-	16,385	$0.50	$1.50	69.80%	70.20%
Llama 3.2 90B Instruct	Open	90	128,000	$0.35	$0.40	86.00%	-
Qwen2.5 72B Instruct	Open	72.7	131,072	$0.35	$0.40	-	-

Model *	License *	Parameters (B) *	Context *	Input $/M *	Output $/M *	MMLU *	DROP *
Qwen2.5 7B Instruct	Open	7.6	131,072	$0.30	$0.30	-	-
DeepSeek-V3	Open	671	131,072	$0.27	$1.10	88.50%	91.60%
Llama 3.1 70B Instruct	Open	70	128,000	$0.20	$0.20	83.60%	79.60%
Llama 3.3 70B Instruct	Open	70	128,000	$0.20	$0.20	86.00%	-
GPT-4o mini	Proprietary	-	128,000	$0.15	$0.60	82.00%	79.70%
QwQ-32B-Preview	Open	32.5	32,768	$0.15	$0.20	-	-
DeepSeek-V2.5	Open	236	8,192	$0.14	$0.28	80.40%	-
Qwen2.5-Coder 32B Instruct	Open	32	128,000	$0.09	$0.09	75.10%	-
Llama 3.2 11B Instruct	Open	10.6	128,000	$0.06	$0.06	73.00%	-
Llama 3.1 8B Instruct	Open	8	131,072	$0.03	$0.03	69.40%	59.50%
Llama 3.2 3B Instruct	Open	3.2	128,000	$0.01	$0.02	63.40%	-
GPT-4.5	Proprietary	-	128,000	-	-	90.00%	-
o1-pro	Proprietary	-	128,000	-	-	-	-
o3	Proprietary	-	128,000	-	-	-	-

Model *	License *	Parameters (B) *	Context *	Input $/M *	Output $/M *	MMLU *	DROP *
QvQ-72B-preview	Open	73.4	32,768	-	-	-	-
Qwen2.5 14B Instruct	Open	14.7	131,072	-	-	79.70%	-
Qwen2.5 32B Instruct	Open	32.5	131,072	-	-	83.30%	-
Qwen2.5-Coder 7B Instruct	Open	7	128,000	-	-	67.60%	-

Table 11. LLM-Stats Comparison Extract

*** MMLU - Knowledge and reasoning across science, math and humanities [13]**

*** DROP - Reading comprehension with reasoning over paragraphs [13]**

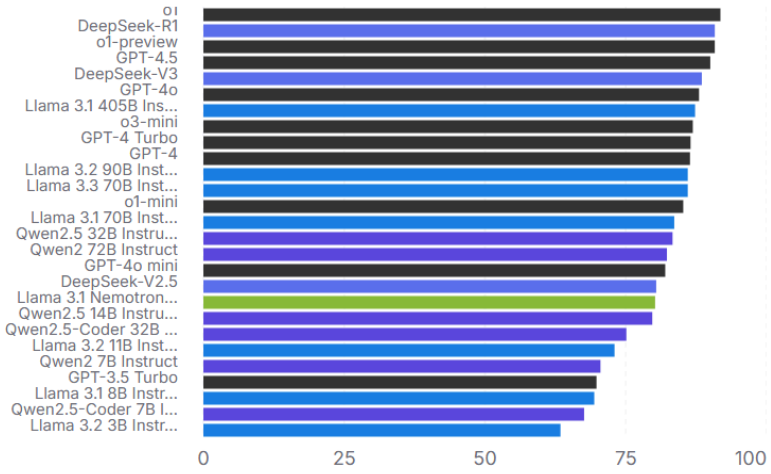

Figure 8. MMLU Leaderboard

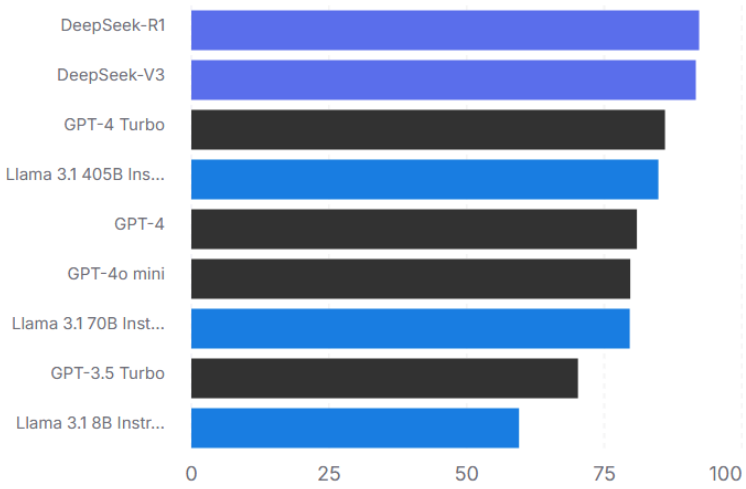

Figure 9. DROP Leaderboard

7.3. Backend cost comparison

This section extends the cost-performance analysis presented in Table 11 to specifically consider the backend component of the RAG application. As described in Section 6, multiple LLMs are integrated across different features. The adopted models are gpt-4o, Claude 3.7 Sonnet and DeepSeek R1, with cost and performance data extracted from llm-stats [13].

Model	License	Parameters (B)	Context	Input $/M	Output $/M	MMLU	DROP
Claude 3.7 Sonnet	Proprietary	-	200,000	$3.00	$15.00	-	-
Claude 3.5 Sonnet	Proprietary	-	200,000	$3.00	$15.00	90.4%	87.1%
DeepSeek-R1	Open	671	131,072	$0.55	$2.19	90.8%	92.2%
GPT-4o	Proprietary	-	128,000	$2.50	$10.00	88.0%	-

Table 12. Backend LLMs Cost

Since Claude 3.7 Sonnet is a relatively new and proprietary model, no benchmark results are currently available. Thus, comparison is made to its predecessor Claude 3.5 Sonnet, for this section. One main reason to choose Claude 3.7 is that it is a hybrid model with MoE architecture which can balance speed and reasoning performance well. So it is a good choice to start with. In addition, similar to the selection of models in the data layer, the backend LLMs are chosen for their strong balance between cost and performance, making them suitable for generating appropriate legal responses.

7.4. Design evolution

The design has evolved significantly from the initial proposal based on clarified project requirements. Evolution of the design for each component of the application is covered in the following sections.

7.4.1. Frontend - Evolution

This section highlights the evolution of the frontend component of the RAG application from sprint 1 to sprint 3 of the project.

7.4.1.1. Frontend - Sprint 1

For sprint 1, the login page presented in the demonstration is shown below. This page serves as the primary gateway through which users can access the legal RAG application. Users are required to input their username and password into the designated input fields. Additional features such as 'Sign Up' and 'Forgot Password' were displayed but not yet functional since they had not been developed at this stage in the project.

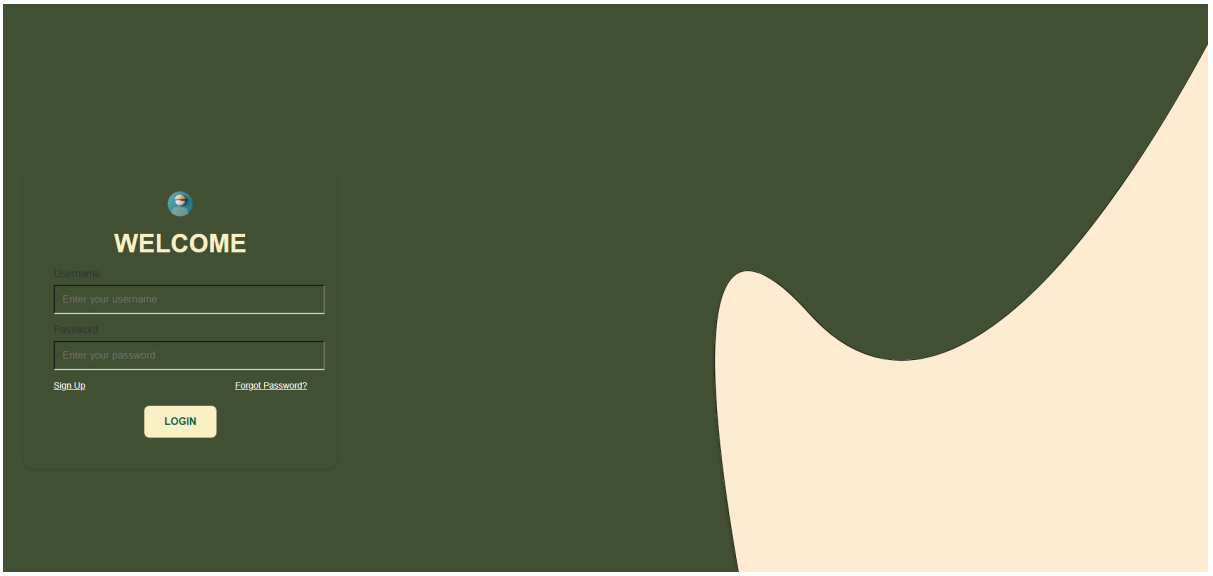

Figure 10. Login Page

Once the user logs in, they are directed to the *'Reasoning'* page (as shown in the image below), where they can interact with the AI SATChat bot through the chat box at the bottom of the page to generate legal reasoning based on the provided input. The top left of the page displays the different features of the application, namely *'Search'*, *'Reasoning'*, *'Statement'* and *'Document'* which allows the user to perform different legal tasks as specified by the client and project specifications. The middle section of the page displays the user's current conversation with the AI SATChat bot, allowing users to scroll

through the chat history. The left section of the page shows the user's chat history in the form of a list, and the right section of the page displays links to documents generated by the AI.

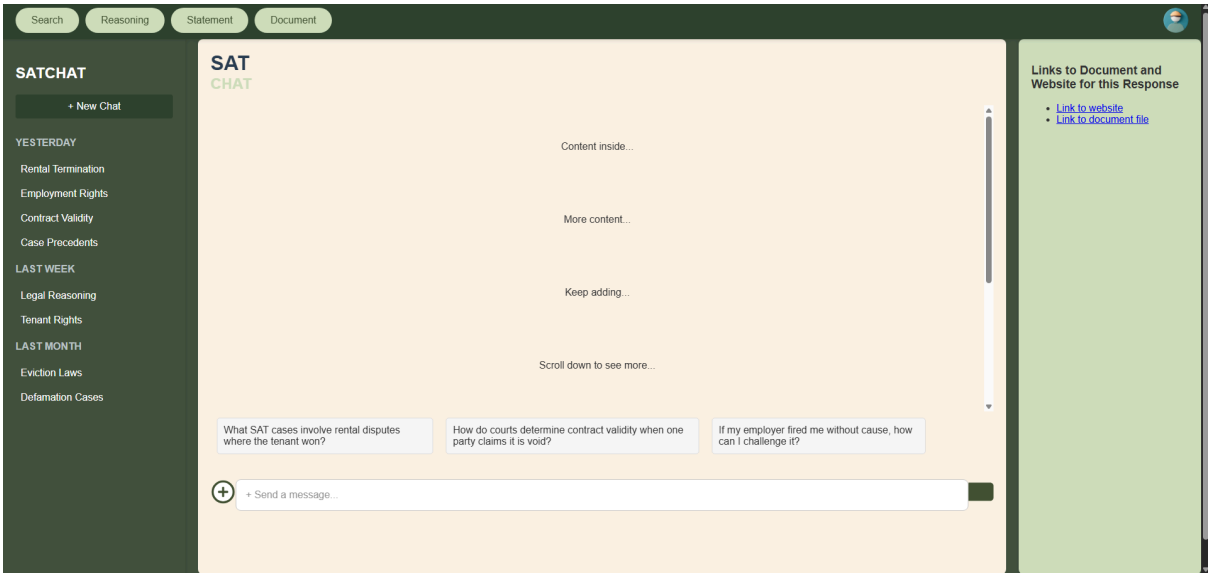

Figure 11. 'Reasoning' Page - Sprint 1

Similar to most AI chatting products on the market, the application at this stage supported local file uploads. By clicking the '+' button next to the chatbox, users can open a file browser to select and upload files as shown in the image below.

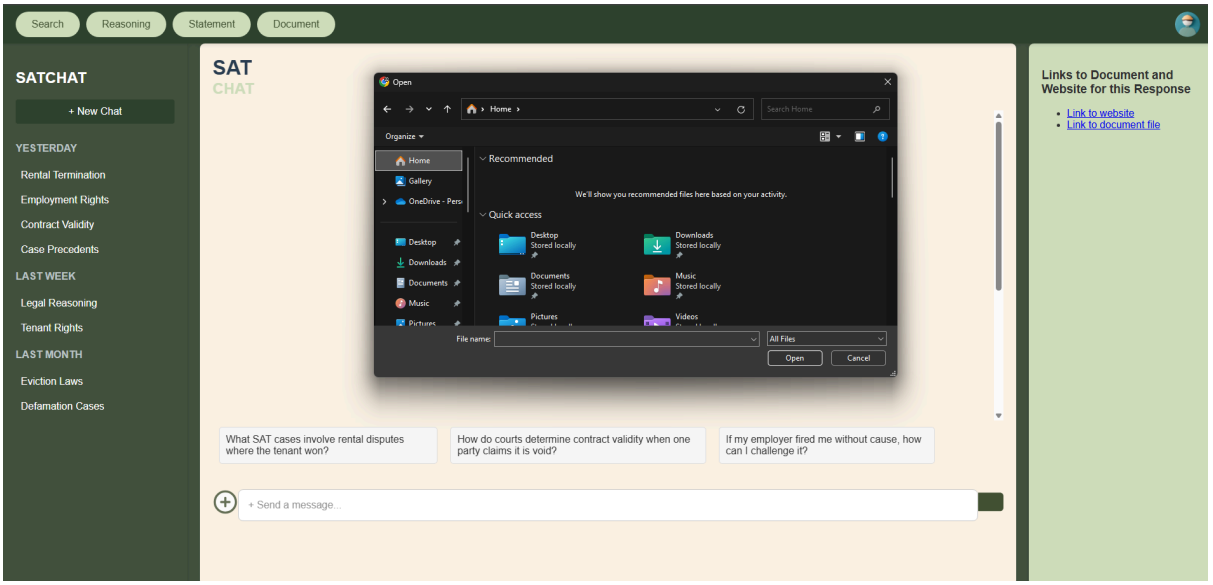

Figure 12. Upload Feature

At this stage in development, the *'Search'* page was developed to allow users to search for relevant SAT documents based on keyword search as shown in the image below. Users could also filter search results based on the Australian state through the provided check boxes.

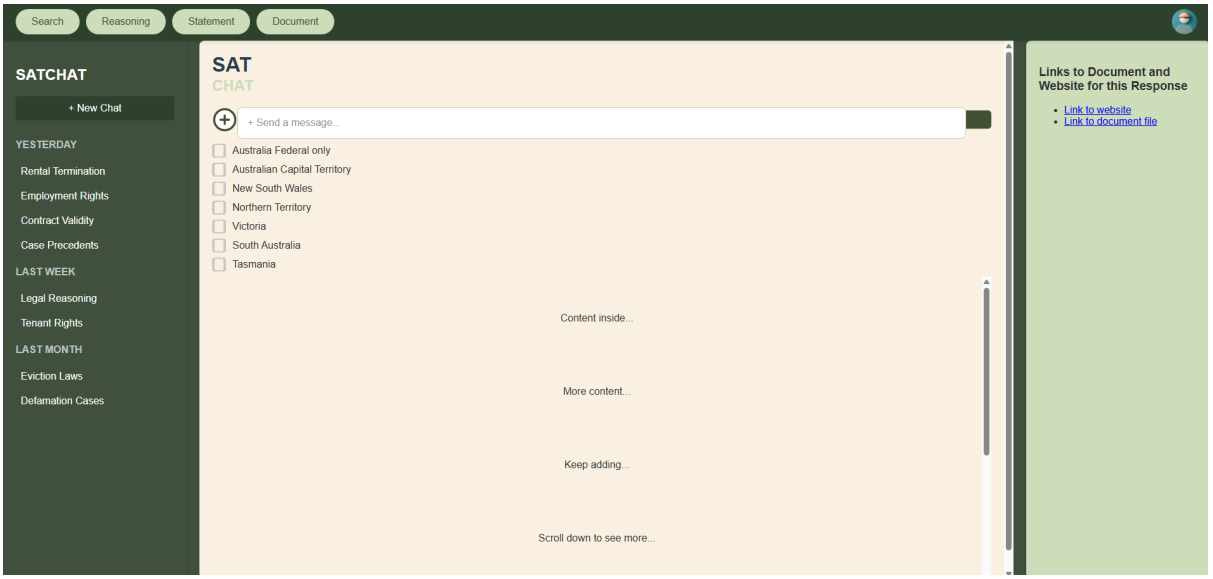

Figure 13. Filter Search

At this stage, the *'Statement'* and *'Document'* pages were not developed and only there as placeholders for future potential features. Furthermore, the frontend was not yet connected to the backend, so the features of the application were not yet functional through the web interface. As such, all items displayed during the presentation were hardcoded.

7.4.1.2. Frontend - Sprint 2

For this sprint, the right section of the page which displayed links to documents generated by the AI (from sprint 1) was removed since the backend already provides HTML context which includes hyperlinks to relevant documents. The document hyperlinks allow the user to access documents by simply clicking on it. Additional changes to the frontend include renaming the following pages:

1. *'Reasoning'* page to *'Chat'*
2. *'Search'* page to *'Build Arguments'*

This was done to present the features of the application more intuitively to the users.

On the *'Build Arguments'* page, the check boxes from sprint 1 were removed and changed into a drop-down bar. The options inside the drop-down bar allow the user to filter on 'Case topics' instead of Australian states from sprint 1. This is because SAT decision cases are from Western Australia, hence inclusion of other state law is irrelevant to this application. Consequently, users can better build arguments closer to their needs by filtering by 'Case topic' from the built in drop-down bar.

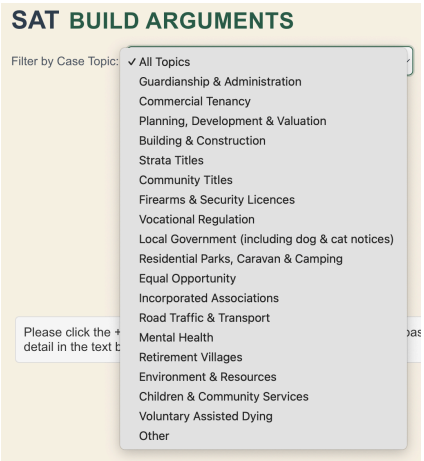

Figure 14. Case Topic Drop-Down

FastAPI is used to connect the backend to the frontend. In this version, users can provide input to the AI SATChat bot. Just like the example input below, the user submits a sample case by uploading a docx file. Once they click the send button, the website will generate a formatted output using the AI model the user selected.

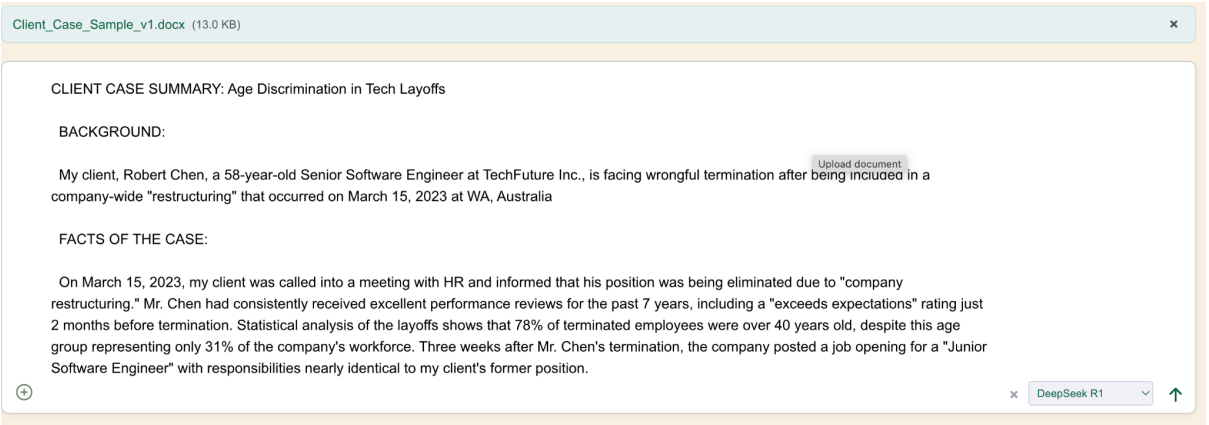

Figure 15. 'Build Argument' Question - Sprint 2

An example of the *'Build Arguments'* feature is shown below where the user wants to build an argument. The presented output is well formatted and structured under relevant headings. Key headings and text is also changed to a green colour to highlight important information to improve readability and user experience. The "Strength" shown in Key Insights and Key Arguments refers to how persuasive the insights and arguments are in answering the user's prompt.

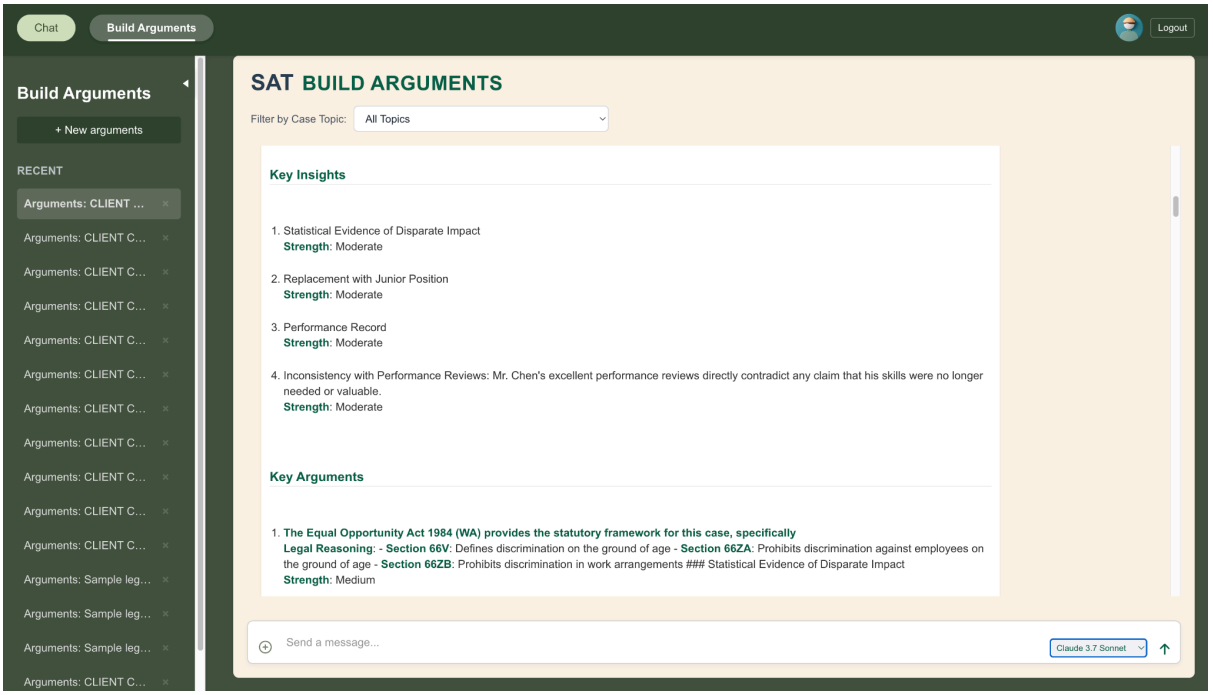

Figure 16. 'Build Argument' Page - Sprint 2

An example of the *'Chat'* feature is shown below where the user wants to see cases about disability discrimination in the workplace. The application then displays the 5 most relevant SAT cases to the user which can be scrolled through. The generated context is well formatted and the titles of the cases are clickable and underlined with the corresponding document API.

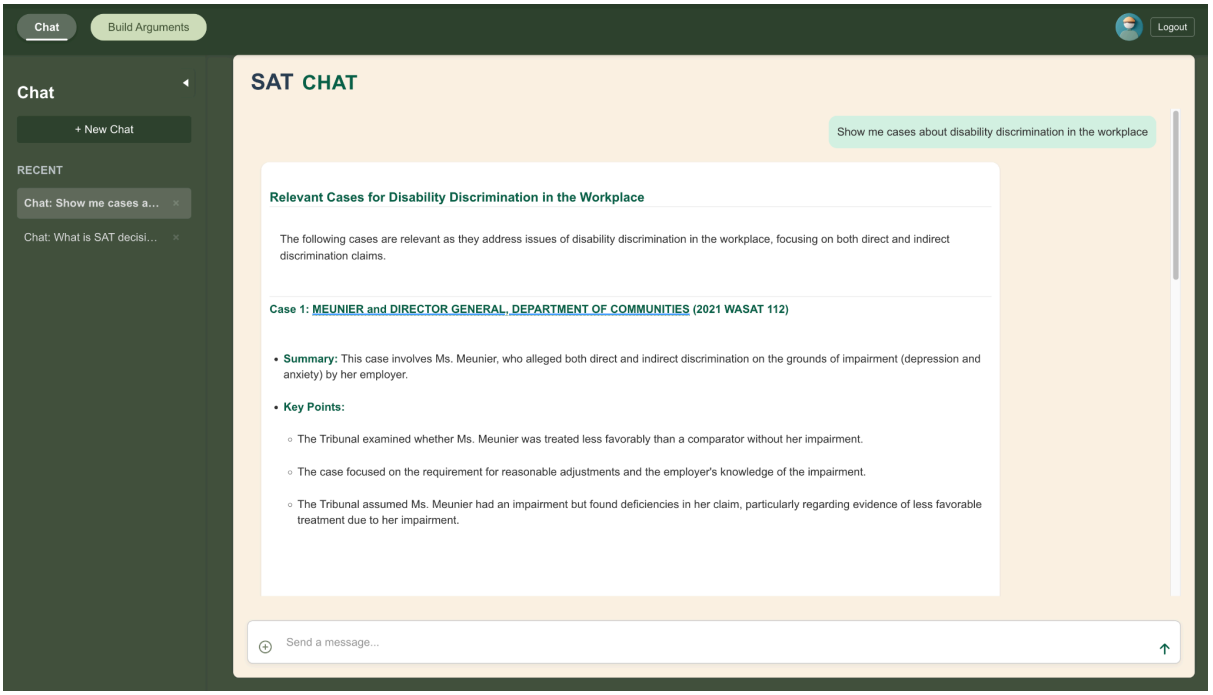

Figure 17. 'Chat' Feature - Sprint 2

Additionally, a new feature was implemented that enabled the application to automatically scrape text from uploaded documents. The scraped text would then be placed in the chat-box which could be used as part of the user's query.

7.4.1.3. Frontend - Sprint 3

For sprint 3, the application supports account creation. Users can sign up by entering their email and password into the respective input fields as displayed on the webpage (shown in the image below). The page also has a *'Back to Login'* link which directs the user back to the login page, and a *'Sign Up'* button to register the user. Upon successful user registration:

1. The front-end automatically navigates the user to the main *'Chat'* page
2. The user's credentials (email and password) are securely stored in the database. The user can login with these credentials for future sessions.

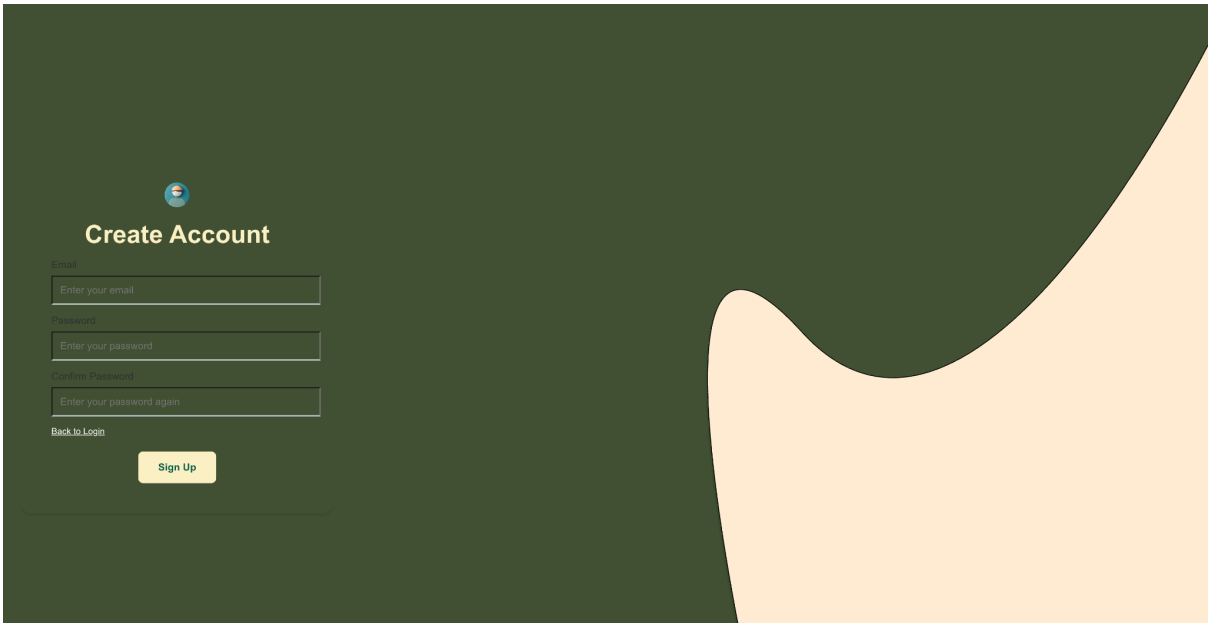

Figure 18. Sign Up Page - Sprint 3

Improvements were also made to the *'Build Argument'* response format. It now highlights identified legal issues more clearly and structures the response using legal reasoning components e.g '*Rule', 'Application',* and more as shown in the image below. This greatly improves readability and aligns with formal legal writing standards. Furthermore, the frontend now provides a button on the *'Build Arguments'* page in the chat box which allows users to select the AI model (Claude 3.7 Sonnet or DeepSeek R1) used to generate the output. Additionally, a button (next to the drop-down list to filter by case topic) has been included to allow users to use different prompting strategies (single call versus multi call) based on their individual needs.

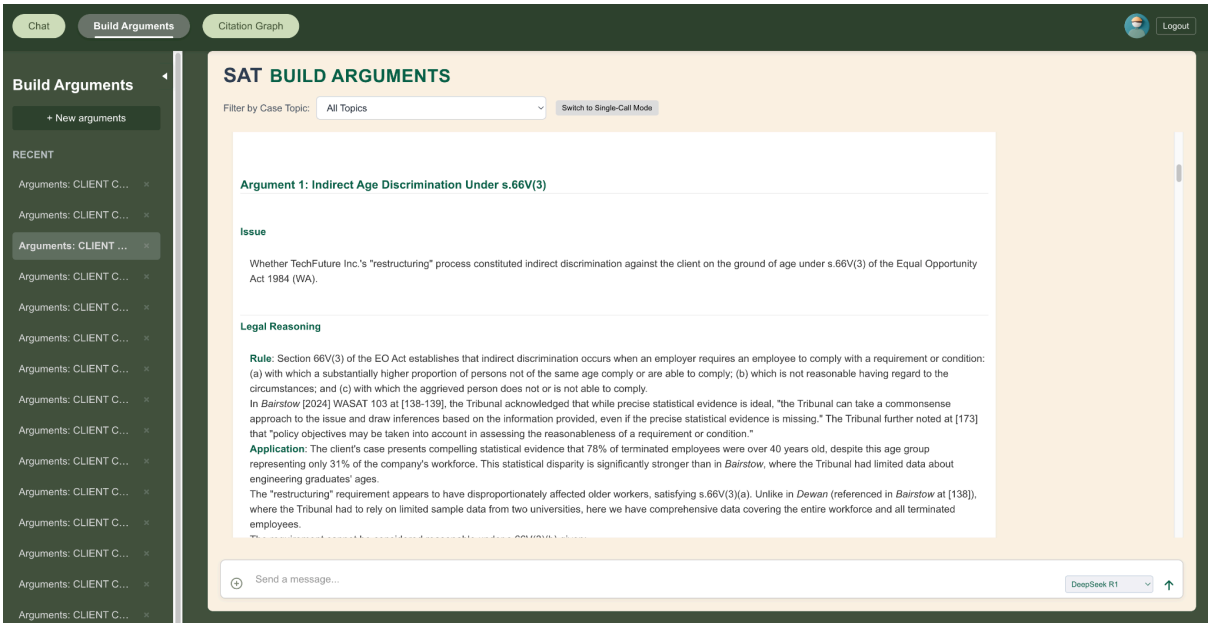

Figure 19. 'Build Arguments' Page - Sprint 3

A new *'Citation Graph'* page was created where users can visualise the relationships between SAT decision cases, law and specific sections of law in the form of a connected graph, particularly when related cases' relationship can be created in a graph via simple click. Users can input a case citation number (e.g., [2021] WASAT 129) into the search bar located at the top of the page. The frontend then displays a connected graph where:

1. Blue nodes represent SAT decision cases
2. Green nodes represent sections under specific legal provisions
3. Orange nodes represent the main legal principles

All nodes are interactive where:

1. Clicking a node opens a pop-up where users can view and edit its metadata
2. Each node also includes a direct link to the relevant case, legal source, or legal section

The graph below is dynamically generated using Neo4j via a simple click, with the underlying logic and integration developed by our team. So users will easily find the common citations for these related cases, and can even further expand to see what other cases also cited one law or section.

6. **Outcome**

The tribunal dismissed Mr. Al-Shabib's complaints of discrimination and victimization. The decision was based on the lack of evidence supporting his claims and the finding that the university's actions were in line with standard academic practices. Consequently, Mr. Al-Shabib lost the case, while Curtin University was found not liable for any discriminatory actions.

Similarity: 43.3%

Visualize All Related Cases in Citation Graph

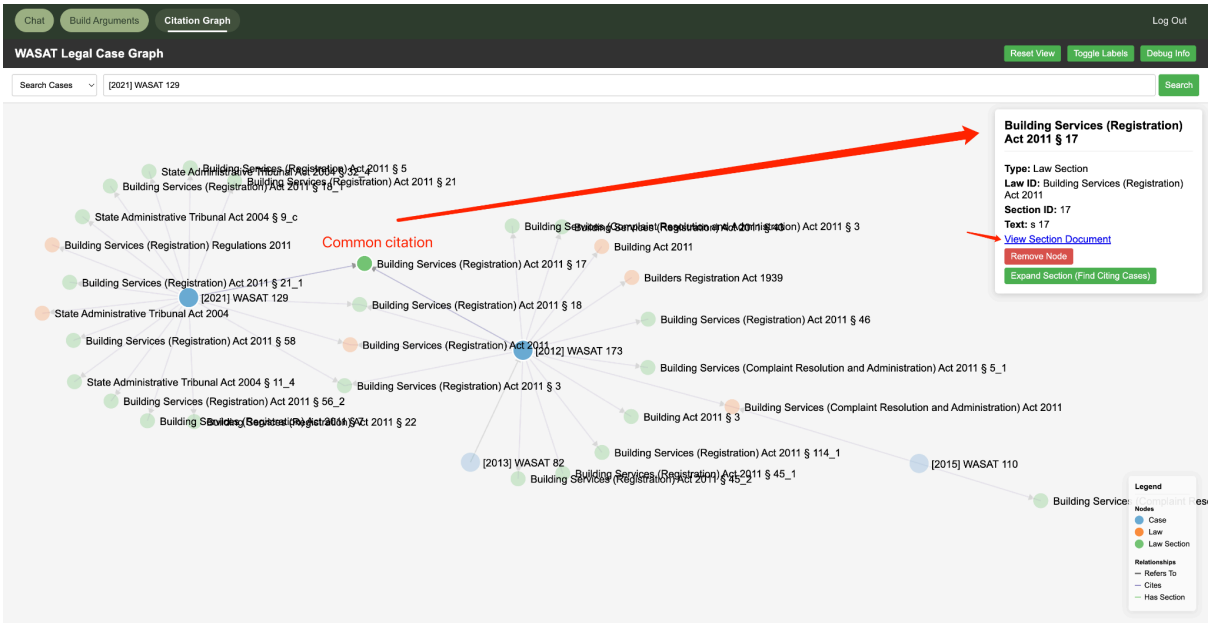

Figure 20. Citation Graph

7.4.2. Data Layer - Evolution

The design of the data layer has changed significantly from the initial proposal. The list below outlines these changes along with the reasoning behind them.

1. In the proposed system architecture, SAT decision data was to be scraped and stored into an S3 data lake before being inserted into a local PostgreSQL database. In the final implementation, the scraped data is inserted directly into the PostgreSQL database. This change was made to reduce complexity of the design.
2. Initially, the WASAT RSS feed available on the online AustLII database was to be used to detect any updates to the SAT decision cases which would then trigger updates to the local PostgreSQL database. However, due to significant investment in developing a robust web scraper and focus on developing key components of the data pipeline such as summary generation and text embeddings, this feature was not a priority due to time limitations.
3. The initial proposal listed the following fields for extraction and insertion into the local database: date/time of extraction, case URL, date of case, case title, involved parties, all subsections under section "*REASONS FOR DECISION OF THE TRIBUNAL"*. In the final design, additional fields were extracted to enable more extensive experimentation with the data.
4. From the initial proposal, the extracted data would be stored in JSON format, which would then be used to populate the local PostgreSQL database. However, further investigation into Scrapy revealed that its built-in pipelines allow for direct insertion into PostgreSQL. This discovery resulted in reduced design complexity and simpler implementation.
5. Initially, no research was conducted into generating embeddings and summary generation due to limited technical expertise.Text embeddings are important for the RAG application as they capture the semantic meaning of the text enabling more accurate search and retrieval. Additionally, summary generation significantly reduces computation time during user queries by condensing the content. In the final design, these are implemented and detailed within this report.

More detail on the progress of the data layer is outlined in the following sections which detail the evolution of the data layer from sprint 1 to sprint 3 of the project.

7.4.2.1. Data Layer - Sprint 1

For sprint 1, data is extracted from the AustLII database using Python framework *'Scrapy'*. SAT decision documents generally follow the same structure which means that text data under relevant sections in the document such as *'case title'*, *'citation number*' etc. (as shown below) could be extracted by locating the corresponding HTML element in the document.

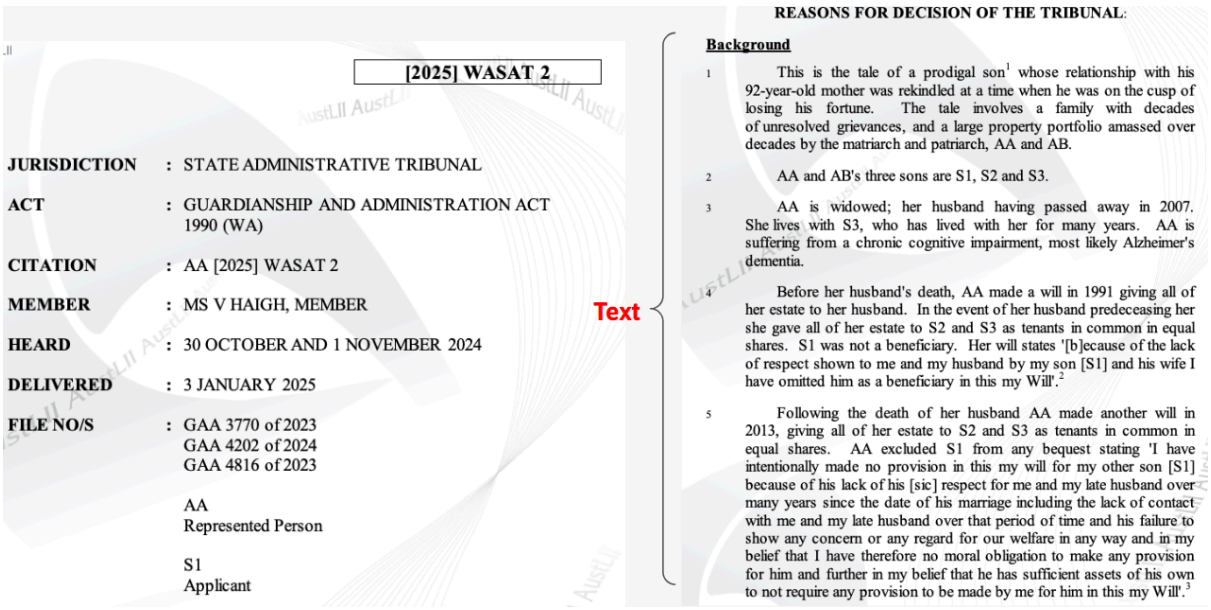

Figure 21. Typical SAT Case Format

At this stage of development, the web scraper only supported data extraction from a single year of SAT decision cases since extracting the data takes a significant amount of time. Additionally, a basic pipeline called SatscraperPipeline was created to perform basic preprocessing of the data (simple formatting of specific fields) which would then be automatically output into JSON format.

In a SAT decision case, there is a Section called *"REASONS FOR DECISION OF THE TRIBUNAL"* as shown in the image above. This section consists of many subsections such as "Background", "Introduction" and many more. It was initially planned as part of our proposal to extract each of these subsections. However, these subsections are not consistent among all SAT decision cases (for example some cases would not have "Background" subsection and certain subsections were unique to the individual case). Since there are over 4000 SAT decision cases, it is not feasible to account for every variation for each document. Thus, it was decided to extract all text under *"REASONS FOR DECISION OF THE TRIBUNAL"* to then be used to generate a summary containing key insights that could be used for the backend.

7.4.2.2. Data Layer - Sprint 2

For sprint 2, the web scraper supported scraping for multiple years of SAT decision data. However, since the scraper from sprint 1 was only based on the HTML structure for the 2025 data, it was discovered that the HTML structure for older data had slight variations which resulted in missing fields or inconsistent data for many cases. Modifications to the scraper were made in an attempt to address these variations in HTML structure. But since there are over 4000 decision cases it is not feasible to account for every single variation which would need to be manually observed. Hence, further improvement for this aspect of the scraper was not pursued.

More detailed preprocessing of the extracted data was implemented in SatscraperPipeline to remove special characters and format specific fields so that they could be properly utilised by the backend. It was noticed that certain fields of the data had abnormal *'\n'* and *'\t'* characters that occurred randomly within the text which interrupted structure and flow of sentences and paragraphs. Since these can affect how the AI model interprets the data and generates reasoning, all *'\n'* and *'\t'* characters were removed from the text. Data was also formatted so that they all followed the same structure. For example, all case citation numbers were formatted as *"[YEAR] WASAT CASE_NUMBER"*.

Furthermore, the following pipelines were also created to handle certain tasks:

1. TopicMappingPipeline - Maps each SAT case to its corresponding *'case_topic'* based on its *'case_act'*. This was created so that users could filter by case topic to meet their needs more closely. A database index was created on *'case_topic'* of the satdata table to significantly speed up searching and filtering.
2. SavingToPostgresPipeline - In this pipeline, summary for *"REASONS FOR DECISION OF THE TRIBUNAL"* (stored in the reasons field) and text embeddings are generated for all SAT decision cases. All of the prepared data is then inserted into the PostgreSQL database.

7.4.2.3. Data Layer - Sprint 3

Since we now have structured and clean data, this sprint focuses on delving deeper to gain more insights. We introduced a new feature using Neo4j and integrated it into our front end.

Data Modeling

We abstracted the case law system into three types of nodes (Cases, Laws, and Sections) and three types of relationships (REFERENCES, CITES, and HAS_SECTION). More detailed information is available in Section 6.2.7 (Data Layer – Neo4j AuraDB).

Why Neo4j?

We chose Neo4j due to its easy deployment and the availability of a free tier suitable for quick development. A more detailed cost comparison is provided in Section 7.2.4 (Cloud Graph Database – Cost Comparison).

Restful API for quick setting up

Because the free tier does not offer a dedicated API, we used our account credentials to create a makeshift API for querying. This setup is sufficient for small-scale tests and rapid implementation, and we plan to enhance it in future iterations.

7.4.3. Backend - Evolution

Overall, the backend did not have a big difference from the project brief and our proposal. Its core purpose is to build a RAG component to retrieve similar cases from our SAT knowledge database created by our team, and generate summary and insights. Below is how we iterated our backend from sprint 1 to 3.

7.4.3.1. Backend - Sprint 1

Following a common RAG architecture, we designed our proposal while considering client requirements and our use case. For Sprint 1, since all components were not connected together and data was not yet stored into a local database, development of the backend was separate. This meant that text was extracted manually by choosing the past 3 years data and used for testing of the code. We firstly implemented a retrieval feature with FAISS data which stored the past 3 years data. Mainly, we experimented the chunking method with different chunk sizes, and opted for chunk size - 5000 characters which performed better than other chunk sizes. Initial retrieval results turned out high similarity for cases returned given our queries.

7.4.3.2. Backend - Sprint 2

During sprint 2, while we are still building and finalizing our database, we are comfortable about uncertainty and ambiguity. We pivoted to use Postgresql + Pgvector database which combines the strengths of both relational database and vector database. We experimented with our RAG component based on an unfinalized database. We firstly designed and implemented our backend code structure with a clear and modularized method which includes app and rag. For app subfolders, there were api, db and services. Backend structure can be found below. So we just need to create a file or a new service under each sub_folder.

```
backend/
├── app/
│   ├── __init__.py           # FastAPI application setup
│   ├── config.py             # Configuration and environment variables
│   ├── db/
│   │   ├── __init__.py       # Database exports
│   │   ├── database.py       # SQLAlchemy setup
│   │   └── models.py         # Database models (User, Case)
│   ├── api/
│   │   ├── __init__.py
│   │   ├── routes/
│   │   │   ├── __init__.py   # Route exports
│   │   │   ├── chat.py       # Chat endpoint
│   │   │   ├── arguments.py  # Build arguments endpoint
│   │   │   ├── citation_graph.py # Citation graph endpoints
│   │   │   ├── users.py      # User authentication endpoints
│   │   │   └── case_chunks.py # Case chunks processing endpoints
│   │   └── schemas/
│   │       ├── __init__.py   # Schema exports
│   │       ├── chat.py       # Chat request/response models
│   │       └── arguments.py  # Argument request/response models
│   └── services/
│       ├── __init__.py       # Service exports
│       ├── chat_service.py   # Chat business logic
│       └── arguments_service.py # Arguments business logic
├── rag/
│   ├── __init__.py           # RAG component exports
│   ├── embeddings.py         # Vector embeddings generation
│   ├── retrieval.py          # Document retrieval from vector store
│   ├── generation.py         # Text generation with retrieved context
│   └── models.py             # Model loading and management
```

Figure 22. Backend File Structure

Based on this backend structure, we built our two main features which are chat service and build_arguments service. And demonstrated the initial result for these 2 features on demo B, although the feature or database was not finalized. That is agile development we have learnt in lectures and workshops.

7.4.3.3. Backend - Sprint 3

During sprint 3, we firstly optimized current features and fixed bugs for the 2 existing features. We implemented a new IRAC prompt method that is suitable for legal doman which tells LLM to analyze and create answers in IRAC format.

Also, we created a single call mode compared to 3 calls in sprint 2. So we instruct LLMs to think in steps but within a single call. This method is suitable for reasoning models with strong reasoning capability. This single call mode will reduce latency while still having good performance.
What's more, after discussion and suggestion from tutor, we integrated a new feature called citation graph which visualized citation relationships for case decisions, laws and law sections. This feature will create a citation graph for related cases in LLM response by a simple click. We believe this really helps users to quickly be familiar with all these complex relationships and help them do legal research.
Evaluation is a critical part in LLM application. We implemented our own evaluation in two methods. One empirical way is asking feedback from students studying law. The other method is by quantifying different performance metrics like cost, latency, token usage for the LLM services.

8. User-Driven Evaluation of Solution

We performed a series of user-driven evaluations to validate the design choices and measure the legal reasoning capabilities of our RAG system. The evaluation focused on chunk retrieval strategies, prompt tuning for legal reasoning, field-specific evaluation criteria, and output quality benchmarking across different LLMs.

8.1. Chunk testing

We compared retrieval performance using smaller (1000 characters) and larger (5000 characters) document chunks. Testing showed that while smaller chunks retrieved faster (~7s), they often produced incomplete or fragmented answers, missing key legal references. Larger chunks (~12s retrieval) preserved richer legal context, better case citations, and more complete reasoning structures. Although retrieval time increased slightly, the improvement in output quality justified the trade-off. Given the importance of completeness in legal reasoning, we selected larger chunks for final deployment, accepting a small latency increase to ensure higher legal precision.

8.2. Prompt Tuning: Aligning with Legal Reasoning Structures (IRAC/MIRAT)

To ensure that the RAG system outputs professional and legally structured responses, we aligned the prompt design with standard legal reasoning frameworks — IRAC (Issue, Rule, Application, Conclusion) and MIRAT (Material facts, Issues, Rules, Arguments each way, Tentative conclusion) [27].

These frameworks are commonly used by lawyers to break down legal problems into clear sections:

1. **Issue** - Identify (and articulate) the legal issue
2. **Rule** - State the law on the legal issue
3. **Apply** - The law to the facts of the problem scenario
4. **Conclude** - Give a conclusion where possible

This structure was adopted to ensure that the RAG application follows a more professional and logical reasoning process. Additionally, we implemented ten prompt tuning strategies based on UNSW Law School's guidance for legal writing, aimed at encouraging the LLM to focus on applying real legal rules and case law rather than relying on general opinions.

This tuning helped produce outputs that are more structured, legally accurate, and easier to evaluate for future benchmarking, reinforcing the principle that clearer input leads to better LLM outputs.

8.3. Evaluation based on legal field

Given the variation in legal reasoning across different fields, we tailored evaluation criteria specific to the domain under analysis. For contract law, the outputs were assessed based on the following core steps, aligned with legal practice standards:

1. **Identification of Relevant Terms**: Accurately recognizing the specific contract terms applicable to the dispute.
2. **Assessment of Breach**: Evaluating whether the identified terms were breached based on the facts.
3. **Determination of Termination Rights**: Assessing whether the breach entitles a party to terminate the contract according to legal principles.

In addition to the evaluation process described above, the assessment also considered several common legal issues in contract law, including whether a valid contract was formed, the specific terms and obligations agreed upon by the parties, whether the parties performed their contractual obligations, whether any breach justified termination or rescission of the contract, and what remedies were available for breach [27].

Evaluation of outputs focused on whether these legal elements were correctly identified, logically reasoned, and accurately applied to the given fact scenario. This ensured that the system's performance was judged not just by superficial textual matching, but by substantive legal reasoning quality appropriate to the specific field.

8.4. Backend - Evaluation Methods

Two independent methods were used for evaluation:

1. Manual Review: Assisted by law students, reviewing outputs against real-world legal standards and marking criteria.
2. LLM-Based Review: Using GPT-o3's reasoning capabilities to cross-verify argument soundness, particularly for application steps and reasoning chains.

8.5. Performance marking criteria

Outputs were evaluated against the following criteria:

1. Format (20%): IRAC structure
2. Accuracy (40%):
    a. Legal Argument (10%)
    b. Supporting Case Law (10%)
    c. Cited Statute / Rule (10%)

    d. Application (10%)
3. Reasoning (40%):
    a. Contract terms correctly identified (10%)
    b. Breach correctly assessed (15%)
    c. Termination rights well reasoned (15%)

8.6. Backend - Performance Metrics

The backend tracks fine-grained performance metrics for each generation task, including:

1. Token Usage: Input/output token counts per step and overall.
2. Execution Time: Per-step and total reasoning duration.
3. Step Breakdown: Time distribution across multi-step reasoning stages.

Token counting is implemented with:

1. Exact token measurement for OpenAI models.
2. Approximate estimation for Anthropic and DeepSeek based on whitespace tokenization.

This performance monitoring enables continuous optimization of system latency, cost, and output quality.

Will and kynna to add table

LLM	Performance - chatgpt-o3	Performance - law student	Time /seconds	Total Cost/$	Input token count	Input token Price / million token	Output token count	Output token Price / million token
Claude 3.7 - multi calls	81	8.5	103	0.40	109,447	3	4657	15
Deepseek R1 - single call	69	5.5	113	0.02	32,540	0.55	805	2.19

Table 13. LLM Performance Metrics

8.7. Backend - Evaluation Insights

We tested the Build Arguments function using a real-world style contract dispute scenario involving Bonizo Pty Ltd and Tappers Ltd. Outputs were generated by Claude 3.7 Sonnet and DeepSeek R1 models and evaluated based on structure, reasoning, and legal accuracy.

Key Findings:

- Claude 3.7 Sonnet produced consistently professional, detailed legal arguments with strong supporting cases and practical final advice. It delivered higher legal reasoning quality but incurred a higher cost (~$0.40 per run) due to multi-step processing and larger token counts.
- DeepSeek R1 was much cheaper (~$0.02 per run) and faster in single-call mode. However, its legal reasoning was less structured, with advice that was generally broader and less specific.
- Manual reviews by law students emphasized that case matching must prioritize the alignment of legal dispute points, not merely surface topic similarity, to ensure meaningful legal reasoning.

Overall, while DeepSeek offers significant cost advantages, Claude's outputs demonstrate higher reliability and professionalism, making the trade-off acceptable for scenarios where legal accuracy is critical.

9. Limitations

9.1. Data layer limitations

Within the data layer, the following limitations exist:

1. Data is extracted by identifying the HTML elements and tags containing the relevant information in each SAT decision document. While most web pages follow a consistent structure, there are minor variations in the HTML structure which result in some inconsistencies in the scraped text. This is particularly apparent in older SAT decision documents, where the HTML structure has slight discrepancies with more recent documents. Due to the large volume of documents available on the AustLII database, it is not feasible to account for all variations in HTML structure. As a result the extracted data may occasionally contain inconsistencies or missing fields.
2. Future iterations of the RAG application may want to utilise an alternate LLM to generate summaries of the reasons field to improve the quality of the generated output. In this case, all cases will need to be summarised again, which takes a considerable amount of time.
3. Real-time updates to the AustLII database are not captured. Hence, new cases and updated cases are not utilised by the RAG application unless the scraper is run again.
4. The application only scrapes data available on the AustLII database and does not extract and store data from alternative file types (e.g audio, pdf, docx). Allowing the application to store data from multiple file types could improve accessibility and response quality.
5. For Neo4j, we currently use our account credentials to package it as an API, which is sufficient for small-scale prototyping. We plan to implement a more robust solution later.

9.2. Frontend limitations

Within the frontend implementation, the following limitations exist:

1. A proper user management page has not been developed, limiting the ability to view, edit, or manage user accounts through the interface.
2. The login functionality does not currently distinguish between different user types (e.g., judge, lawyer), preventing role-based rendering and tailored frontend experiences.

Addressing these frontend limitations will be essential to improving system usability, security, and role-specific workflows.

9.3. Backend limitations

Within the backend implementation, below are the following limitations:

1. The *'Build Arguments'* feature is only available for lawyers, but not for other users such as judges and the general public. As a result, the LLM outputs legal reasoning based solely on a lawyer's needs. Different users may need varying levels of legal reasoning, which is not currently supported in the application. However, this could be easily addressed by tuning the LLM prompts for different user types.
2. The generated output displayed to the user does not present the reasoning process behind it, which could increase user confidence in the generated answer.
3. The quality of generated summaries of different AI models were not tested and evaluated due to time constraints.
4. Since the development team does not have a legal background, there was no expertise available to assess the quality of the outputs in a legal context. Furthermore, the application has not yet been tested by a legal professional, such as a lawyer.
5. Metadata from uploaded case documents is not stored in a database, which means that the RAG application cannot utilise this information for future user sessions.

10. Future Work

The developed RAG application can be further improved to more closely meet the needs and growing demands of the client. The following areas of future work are proposed:

1. Enable more reasoning model options in the backend and improve prompt tuning to meet different user requirements. This would support users with varying needs for response time and response quality across different levels of legal reasoning. Different LLMs outside of those available from the provided API keys can be explored by reviewing benchmarks on llm-stats.
2. Test and evaluate generated summaries of different AI models to ensure optimal responses. This would improve output quality and increase user confidence in the generated answers.
3. Extract data outside of SAT decision data to broaden the scope of the application. This would enable the application to be used across a wider range of legal contexts. Additional data is available from other legal databases available on AustLII and storing the information in corresponding data tables. Users would then be able to filter legal decisions based on their specific needs.
4. Metadata from uploaded case documents can be stored in the local PostgreSQL database so that it can be used for future user sessions. This would help improve the quality of generated responses, making them closely align with user needs. This can be done by scraping all the unstructured text data from the uploaded documents and inserting it into the database. The text data would then be converted into a vector embedding which can be retrieved by the backend.

5. Enable support for storing a wide range of file types (e.g audio, pdf, docx). This would enable users to utilise different forms of legal documentation to better guide the RAG model in generating appropriate responses. This could be done by implementing a variety of document loaders available for each respective file type and then extracting the information.
6. Implement a user management page to allow users to manage their account. This would allow users to have more control on their personal information and set user preferences based on their needs.
7. Generated outputs should display the thought process behind all legal reasoning responses. This would increase user confidence in the generated response. This could potentially be implemented by tuning the LLM prompt to include the reasoning process.
8. Further UI and backend improvements were needed to improve user experience, such as indicator of reasoning steps, stopping generating etc.

11. References

1. Wang, L., Yang, N., Huang, X., Jiao, B., Yang, L., Jiang, D., Majumder, R. and Wei, F., 2022. Text embeddings by weakly-supervised contrastive pre-training. *arXiv preprint arXiv:2212.03533*.
2. ZenRows 2024, *Scrapy vs BeautifulSoup: Which Is Better For You?*, accessed 10 March 2025, <https://www.zenrows.com/blog/scrapy-vs-beautifulsoup#similarities>.
3. Amazon Web Services n.d., *MongoDB vs PostgreSQL - Difference Between Databases*, Amazon Web Services Inc, accessed 10 March 2025,
4. Altexsoft (2023). *Database Management Systems (DBMS) Comparison: MySQL, Postgr*. [online] AltexSoft. Available at: https://www.altexsoft.com/blog/comparing-database-management-systems-mysql-postgresql-mssql-server-mongodb-elasticsearch-and-others/.
5. Superlinked.com. (2024). *Vector DB Comparison*. [online] Available at: https://superlinked.com/vector-db-comparison.
6. Myscale 2024, *Faiss vs. Pinecone: Ideal Vector Database Comparison*, MOQI Singapore Pte Ltd, accessed 10 March 2025, <https://myscale.com/blog/faiss-vs-pinecone-ideal-vector-database/>.
7. Pathak, TR 2024, '*Understanding Faiss and Pinecone Vector Databases: A Comprehensive Guide'*, Medium, weblog, 2 March, accessed 10 March 2025, <https://medium.com/@tejupathak/understanding-faiss-and-pinecone-vector-databases-a-comprehensive-guide-66502279077e>.
8. Pinecone.io. (2024). *Choosing an Embedding Model | Pinecone*. [online] Available at: https://www.pinecone.io/learn/series/rag/embedding-models-rundown/ [Accessed 27 Apr. 2025].
9. huggingface.co. (n.d.). *MTEB Leaderboard - a Hugging Face Space by mteb*. [online] Available at: https://huggingface.co/spaces/mteb/leaderboard.
10. huggingface.co. (n.d.). *MTEB: Massive Text Embedding Benchmark*. [online] Available at: https://huggingface.co/blog/mteb.
11. Huggingface.co. (2022). *intfloat/e5-base-v2 · Hugging Face*. [online] Available at: https://huggingface.co/intfloat/e5-base-v2.
12. Austlii.edu.au. (2025). *BARNARD and BARRIER REEF POOLS SOUTH WEST PTY LTD [2025] WASAT 28 (4 April 2025)*. [online] Available at: https://www.austlii.edu.au/cgi-bin/viewdoc/au/cases/wa/WASAT/2025/28.html.
13. Stats, L. (2025). *LLM Leaderboard 2025 - Compare LLMs*. [online] LLM Stats. Available at: https://llm-stats.com/.

14. Schwaber-Cohen, R. (2023). *Chunking Strategies for LLM Applications | Pinecone*. [online] www.pinecone.io. Available at: https://www.pinecone.io/learn/chunking-strategies/.
15. Creates, D. (2024). *Using client.chat.completions.create() - Doug Creates - Medium*. [online] Medium. Available at: https://medium.com/@Doug-Creates/nightmares-and-client-chat-completions-create-29ad0acbe16a [Accessed 27 Apr. 2025].
16. paperswithcode.com. (n.d.). *Papers with Code - MMLU Dataset*. [online] Available at: https://paperswithcode.com/dataset/mmlu.
17. Deepeval.com. (2025). *DROP | DeepEval - The Open-Source LLM Evaluation Framework*. [online] Available at: https://www.deepeval.com/docs/benchmarks-drop.
18. Huggingface.co. (2025). *cross-encoder/ms-marco-MiniLM-L6-v2 · Hugging Face*. [online] Available at: https://huggingface.co/cross-encoder/ms-marco-MiniLM-L6-v2.
19. Sbert.net. (2025). *Retrieve & Re-Rank — Sentence Transformers documentation*. [online] Available at: https://www.sbert.net/examples/sentence_transformer/applications/retrieve_rerank/README.html.
20. Stack Overflow. (n.d.). *Stack Overflow Developer Survey 2023*. [online] Available at: https://survey.stackoverflow.co/2023/#most-popular-technologies-webframe-learn.
21. GeeksforGeeks. (2022). *React VS Angular VS Vue - Which Framework is the Best?* [online] Available at: https://www.geeksforgeeks.org/react-vs-angular-vs-vue-which-framework-is-the-best/.
22. Glassdoor. (n.d.). *Company Salaries*. [online] Available at: https://www.glassdoor.com.au/Salaries/.
23. Neo4j Graph Data Platform. (n.d.). *Neo4j Pricing*. [online] Available at: https://neo4j.com/pricing/.
24. Amazon Web Services, Inc. (n.d.). *Fully Managed Graph Database - Amazon Neptune Pricing - Amazon Web Services*. [online] Available at: https://aws.amazon.com/neptune/pricing/.
25. azure.microsoft.com. (n.d.). *Pricing - Azure Cosmos DB | Microsoft Azure*. [online] Available at: https://azure.microsoft.com/en-us/pricing/details/cosmos-db/autoscale-provisioned/.
26. TigerGraph Documentation. (2025). *Pricing - TigerGraph Savanna*. [online] Available at: https://docs.tigergraph.com/savanna/main/overview/pricing.
27. Chan, W.C. (2020) *Australian contract law: Principles and cases*. Pyrmont, NSW: Thomson Lawbook.
28. Thomsonreuters.com.au. (2024). *Trusted AI legal solutions built by legal experts, for legal experts*. [online] Available at: https://www.thomsonreuters.com.au/en-au/legal/ai-solutions.html [Accessed 29 Apr. 2025].

29. Lexisnexis.com. (2025). *Lexis+ AI: Conversational Search Platform | LexisNexis®*. [online] Available at: https://www.lexisnexis.com/en-au/products/lexis-plus-ai?l=j01.
30. infiniflow (2025). *GitHub - infiniflow/ragflow: RAGFlow is an open-source RAG 68 (Retrieval-Augmented Generation) engine based on deep document understanding.* [online] GitHub. Available at: https://github.com/infiniflow/ragflow.
31. Docker (2019). *Docker Documentation*. [online] Docker Documentation. Available at: https://docs.docker.com/.
32. GitHub. (2025). *LangChain*. [online] Available at: https://github.com/orgs/langchain-ai/repositories?type=all&q=rag [Accessed 29 Apr. 2025].
33. Github.io. (2023). *A Practical Approach to Retrieval Augmented Generation Systems - 4 From Simple to Advanced RAG*. [online] Available at: https://mallahyari.github.io/rag-ebook/04_advanced_rag.html [Accessed 29 Apr. 2025].
34. Schwarcz, D., Manning, S., Barry, P.J., Cleveland, D.R., Prescott, J.J. and Rich, B. (2025). AI-Powered Lawyering: AI Reasoning Models, Retrieval Augmented Generation, and the Future of Legal Practice. doi:https://doi.org/10.2139/ssrn.5162111.
35. Chen, K.E. (2025). *Using DeepSeek R1 for RAG: Do's and Don'ts*. [online] SkyPilot Blog. Available at: https://blog.skypilot.co/deepseek-rag/.
36. Langchain.com. (2024). *Fallbacks | 🦜🔗 LangChain*. [online] Available at: https://python.langchain.com/v0.1/docs/guides/productionization/fallbacks/ [Accessed 29 Apr. 2025].
37. https://openai.com/index/hello-gpt-4o/
38. Anthropic (2025). *Claude 3.7 Sonnet and Claude Code*. [online] Anthropic.com. Available at: https://www.anthropic.com/news/claude-3-7-sonnet.
39. Deepseek.com. (2025). *DeepSeek-R1 Release | DeepSeek API Docs*. [online] Available at: https://api-docs.deepseek.com/news/news250120.
40. FastAPI (2023). *FastAPI*. [online] fastapi.tiangolo.com. Available at: https://fastapi.tiangolo.com/.
41. GitHub. (2025). *LangChain*. [online] Available at: https://github.com/orgs/langchain-ai/repositories?type=all&q=rag.