TAYLOR'S UNIVERSITY

Wisdom • Integrity • Excellence

Project Report -WBL TRACK

YESCRIBE RAG-POWERED CP CHATBOT

SEE THOO JUN LOK

0354125

SCHOOL OF COMPUTER SCIENCE

BACHELOR OF COMPUTER SCIENCE (HONS)

APRIL 2025 [PART 1]

SUPERVISORS:

BRUCE CHIN YUAN ZHEN & BRIAN LEE TOH CHU

Acknowledgements

I would first like to express my sincere gratitude to **Taylor's University** for introducing the Work-Based Learning (WBL) track, which enabled me to gain practical industrial exposure. It has been instrumental in allowing me to apply knowledge acquired in classrooms, to real-world scenarios.

My deepest appreciation also goes to **YTL Communications Sdn. Bhd.** for taking me in as their intern for the WBL programme and offering me the opportunity to work on this project within the company. I am eternally indebted to my **industrial supervisors** within the company, **Bruce Chin Yuan Zhen** and **Brian Lee Toh Chu**, for their constant guidance, valuable feedback, and encouragement throughout the duration of this project. Their mentorship has been invaluable in enhancing both my technical and professional development.

I would also like to give credit to my **academic supervisor, Dr. Tee Wee Jing**, for providing continuous advice and constructive feedback, which greatly contributed to the successful completion of this project.

Special thanks are also given to my fellow **colleague interns**, **Chow Suet Ling** and **Cheah Zixu**, for their teamwork, collaboration, and support during this journey. At the same time, I am also equally thankful to the other colleagues and staff members at YTL Communications who provided assistance, shared their knowledge and created a supportive working environment that allowed me to thrive and grow.

Abstract

This report presents the development of YEScribe, a Retrieval-Augmented Generation (RAG) platform designed to streamline the ingestion, indexing, and querying of Concept Papers (CPs). The project was undertaken during my industrial training at YTL Communications Sdn Bhd as part of the academic requirements at Taylor's University for the Work-Based Learning (WBL) Track. The system integrates 5 document processing pipelines to extract and structure critical elements such as approval records, flowcharts, embedded files, tabular data, and Section 1 summaries. These components then chunk and embed the extracted content, turning them into vector embeddings which are inserted into a vector database. This enables semantic search, reranking, and context-aware natural language responses via a chatbot interface for the contents inside a CP. Firstly, a web-based upload portal was implemented in order to manage CP uploads, automate validation of uploaded CPs, and track processing of past and currently uploaded CPs. Afterwards, a Streamlit-based chatbot interface is also implemented which allows users to view approval statuses for all uploaded CPs and most importantly, allow users to query anything that is in a particular CP. The outcome of the solution reduces the need for manual document review, improves retrieval accuracy, and provides version-specific insights for decision-making. This project not only enhanced my technical proficiency in artificial intelligence, natural language processing, full-stack system design, and DevOps but also contributed to a scalable framework that can be extended for broader enterprise document management in the future.

Glossary

Abbreviations	Full Form	Definition
CP	Concept Paper	Business and technical documents used in YTL Communications (Yes), detailing new projects or features in existing applications
RAG	Retrieval-Augmented Generation	A technique in GenAI that leverages retrieval components to fetch external data sources beyond the data trained on LLMs to provide context to LLM to provide contextually accurate responses
UI	User Interface	A way for users to interact with applications in a user-friendly manner
LLM	Large Language Model	Type of artificial intelligence that is part of deep learning and neural networks that is able to generate output, such as text, images, and audio based on trained data
API	Application Programming Interface	A method for applications to utilise services or get data from external applications
ETL	Extract, Transform, Load	Pulling data from external sources, then making the data in a standardised way to be stored and used afterwards when needed

QA (i)	Quality Assurance	A team in software development that performs testing for software before being released to the public
QA (ii)	Question-Answer	Answering user questions
BA	Business Analyst	A person who translates organisational goals into detailed business requirements, tasks include requirements gathering and documentation
GenAI	Generative Artificial Intelligence	A subset in Artificial Intelligence that involves creating new data based on trained data
JSON	JavaScript Object Notation	Standardised file format commonly used in web technologies, such as HTTP and backend frameworks for forwarding requests
POC	Proof of Concept	Experiment or pilot project to determine the feasibility of a solution to an organisation or entity

Table of Contents

Chapter 1: Introduction

1.1. Project Proposal

1.1.1. Project Title

The name **YEScribe** was conceived to reflect both the organisation's identity and core functionality of the system. The prefix **"YES"** aligns with the company's name and branding, which incorporates the acronym into the names of its applications, such as MyYes, YOS, and to ensure consistency across internal platforms. The suffix **"scribe"** is derived from the verb "describe," which is used to represent the core purpose of the system in allowing users to describe and summarise the contents that are found inside of Concept Papers (CPs).

1.1.2. Introduction and Topic Analysis

YTL Communications, operating under the YES brand, is one of the 5 leading telecommunications providers in Malaysia offering high-speed internet, mobile services, and enterprise solutions. It is also one of the pioneering telecommunication providers as it was the first telco to launch 5G in Malaysia in December 2021 (Yes, 2025). They are also rather nascent and an underdog in the industry as well, only entering the telco market in 2010 and competing head on with other major players in the industry, particularly CelcomDigi, Maxis and U-Mobile.

As part of its ongoing innovation and digital transformation efforts to increase its market share over the telecommunication industry, the company constantly develops and deploys various new marketing initiatives or technical features for its applications and products. In order to manage these new ventures, internal systems are developed and utilised to support project planning, implementation, and quality assurance. Among these internal processes are Concept Papers (CPs), which serve as a crucial documentation format for outlining business requirements, technical specifications, and process flows for new initiatives.

Concept Papers are detailed project documents that capture detailed information such as business objectives, technical architecture, and workflow diagrams for new projects and concepts derived from research and the creation of Proof of Concepts (PoCs). These documents are used by multiple teams within the organisation as a single source of truth for project scope and

requirements, these include but not limited to: Quality Assurance (QA), backend development, frontend development, UI/UX designers, and enterprise teams.

However, in the current workflow, questions or clarifications related to CPs are typically directed to Business Analysts (BAs), who are responsible for interpreting the documents and providing the required answers to stakeholders. While effective for ensuring accuracy, this process is highly manual, dependent on individual expertise, and time-consuming, particularly when dealing with large volumes of CPs or complex version changes.

The reliance on manual document interpretation has resulted in challenges such as delays in information retrieval, inconsistent interpretations across different teams, and reduced agility in decision-making at the management level. The need for a more efficient, scalable, and user-friendly method to access the content of CPs has become increasingly apparent.

YEScribe was developed to address these challenges by leveraging generative AI (GenAI) technologies, specifically Retrieval-Augmented Generation (RAG), vector databases, and advanced large language models, which are required to enable conversational querying and summarisation of CP content.

With YEScribe, users can interact directly with a chatbot to obtain context-aware answers, compare changes between different CP versions, and automated interpretation of process flows without requiring manual intervention from BAs or manual read-up and understanding of the entire CP, allowing all stakeholders to have consistent, immediate access to the most relevant project information.

1.1.3. Problem Objectives

As previously mentioned, the main problem identified across all CPs lies in the lack of an efficient and consistent method for accessing and understanding the underlying contents. These documents are rich in technical, business, and procedural details; however, extracting relevant information at the moment requires manual reading, interpretation, and cross-referencing, tasks predominantly handled by Business Analysts (BAs) currently. As a result, it introduces bottlenecks, delays, and the risk of inconsistent interpretations across affected teams. In addition,

the way CP contents may sometimes even be authored with varying formats, terminologies, and levels of detail, which can further complicate the retrieval and comprehension process.

The objectives in examining this problem are as follows:

1. Understand the underlying contents of CPs to ensure that all stakeholders can access and interpret the information accurately without relying on BAs for unclear sections.
2. Identify inefficiencies in the current CP query and interpretation workflow, particularly in how information is requested and delivered between teams such as QA, backend, and enterprise solutions.
3. Determine how frequent CPs are updated and how these changes impact downstream processes, decision-making, and project timelines.
4. Examine how contents are filled into these documents to identify inconsistencies in structure, terminology, and formatting which might hinder the processing and retrieval process.
5. Recognise the opportunity to leverage AI technologies, such as Retrieval-Augmented Generation (RAG), vector databases, and language models to automate and streamline CP access and interpretation to be used intuitively by internal teams

1.1.4. Project Objectives

The objective of the YEScribe project is to design, develop, and deploy an AI-powered chatbot system that enables efficient, accurate, and secure retrieval of Concept Paper (CP) information within YTL Communications. It leverages Retrieval-Augmented Generation (RAG), custom-created ETL pipelines, vector databases, and advanced language models to provide answers that are rich in context with the contents of CPs. The system aims at reducing manual dependencies, accelerate information access, and ensure consistent interpretation of CP content across all teams.

The specific objectives of the project are as follows:

1. **Implement a RAG-based document retrieval system** that is capable of processing and indexing CPs, enabling semantic search and context-aware question answering.
2. **Integrate conversational memory** into the chatbot to maintain context across multi-turn interactions, allowing users to ask follow-up questions naturally.
3. **Enable CP version comparison** and change tracking to highlight differences in technical and business requirements over time.
4. **Provide multi-format content interpretation**, such as generated text-based summaries, embedded file extraction, flowchart analysis and summarisation, and tabular data extraction from CPs using custom ETL pipelines.
5. **Reduce reliance on Business Analysts (BAs)** by providing immediate, AI-generated answers to CP-related queries.
6. **Monitor and evaluate chatbot performance** through metrics and evaluation tools such as LangSmith, enabling continuous improvement, scalability, and auditability.
7. **Ensure scalability and security** by deploying YEScribe within a containerized environment (Docker) that is hosted on in-house infrastructure.
8. **Facilitate integration** with future internal systems or platforms to expand the chatbot's capabilities and coverage beyond CPs.

1.1.5. Assumptions

The development and deployment of YEScribe are based on several underlying assumptions that define the boundaries and working conditions of the project. These assumptions ensure that the system can be designed, implemented, and maintained effectively within the given constraints.

These key assumptions are as follows:

1. **Availability of complete and up-to-date Concept Papers (CPs)**: All relevant CP with all versions (latest included) will be accessible and stored in standardised formats (e.g., DOCX, PDF) that is suitable for automated processing.
2. **Consistency in document structure**: CPs will generally follow a recognizable structure as variations will inevitably exist, thus allowing for effective semantic chunking and indexing.
3. **Access to in-house infrastructure**: YEScribe will be deployed on secure, company-managed servers, ensuring compliance with internal data governance policies.
4. **Adequate computational resources**: The infrastructure will have sufficient processing power, memory, and storage to run RAG pipelines, vector databases, and containerized services efficiently.
5. **Stakeholder collaboration**: Business Analysts (BAs), QA, backend, and enterprise teams will provide input during the development and testing phases to ensure the chatbot meets practical usage needs.
6. **Stable integration environment**: Required APIs, frameworks (e.g., LangChain, FastAPI, Streamlit), and third-party dependencies will be available and remain compatible throughout the project lifecycle.
7. **Ongoing maintenance support**: A dedicated technical team will be available post-deployment to monitor performance, address mentioned issues, and implement enhancements.
8. **End-user training**: Users will receive basic training or guidance, such as creation of user manuals on interacting with YEScribe to maximize the system's effectiveness.

1.1.6. Project Summary

YEScribe is an AI-powered chatbot system designed to enhance the retrieval, understanding, and comparison of Concept Papers (CPs) within YTL Communications. CPs are critical documents used across multiple teams, including Quality Assurance (QA), backend development, and enterprise solutions to define technical requirements, business objectives, and process workflows. Currently, accessing and interpreting CP content relies heavily on Business Analysts (BAs), creating inefficiencies, bottlenecks, and risks of inconsistent interpretations.

The project leverages **Retrieval-Augmented Generation (RAG)**, custom ETL pipelines that are built from scratch with open-sourced and common libraries, vector databases, and advanced language models to allow users to query CPs conversationally, receive context-aware summaries, and track changes between document versions. The system incorporates conversational memory for multi-turn interactions, multi-format content interpretation (including text, tables, and flowcharts), and containerized deployment for scalability and security.

Developed under the assumption of standardized CP formats, accessible in-house infrastructure, and active stakeholder collaboration, YEScribe aims to reduce manual dependencies, accelerate decision-making, and ensure consistent access to accurate project information. The outcome will be a secure, scalable, and future-ready tool that integrates seamlessly with YTL's internal systems, enabling all stakeholders to interact with CP content efficiently and effectively.

1.1.7. Project Scope

The scope of YEScribe as a Proof of Concept (POC) focuses on demonstrating the feasibility, functionality, and value of applying Retrieval-Augmented Generation (RAG) and generative AI technologies to Concept Paper (CP) interpretation within the company. It aims to validate the system's core capabilities in a controlled environment, using a limited number of concept papers and infrastructure, before potential scaling into a full production system to be used by internal teams.

In Scope (POC Phase)

1. **Document Ingestion and Indexing**: Processing a representative sample of CPs in supported formats (DOCX, PDF) and indexing them into a vector database for semantic search.
2. **Custom ETL Pipeline**: Implementation of an AI-powered retrieval system to support context-aware answers based on extracted CP content.
3. **Conversational Query Interface**: Development of a chatbot with conversational memory capabilities to handle multi-turn interactions and follow-up questions.
4. **Version Comparison and Change Tracking**: Demonstrating the ability to highlight differences between CP versions within the selected number of CPs.
5. **Multi-Format Content Interpretation**: Extraction and summarisation of text, tables, and flowcharts within CPs for POC evaluation.
6. **Performance Monitoring and Evaluation**: Initial integration with LangSmith for performance tracking during testing.
7. **Containerized Deployment**: Containerising the project using Docker for deployment on in-house test infrastructure, testing its performance before further improvements.
8. **User Demonstration and Feedback**: Conducting product demonstrations and user acceptance testing for select stakeholders to gather feedback for future refinement.

Out of Scope (POC Phase)

1. Retrieval or processing of documents other than CPs in the initial phase, such as Business Requirement Documents (BRDs) and Change Request (CR) documents
2. Integration with enterprise-wide document management systems, such as Atlassian Jira and Microsoft Teams
3. Production-grade security hardening, redundancy, and failover mechanisms, such as AI Guardrails for the prevention of prompt injection.
4. Advanced analytics or predictive modelling beyond summarisation and retrieval.
5. Automatic generation or editing of CP content.
6. Implementation of Mixture of Experts (MoEs) for enhancing answer quality

1.1.8. Project Milestones and Deliverables

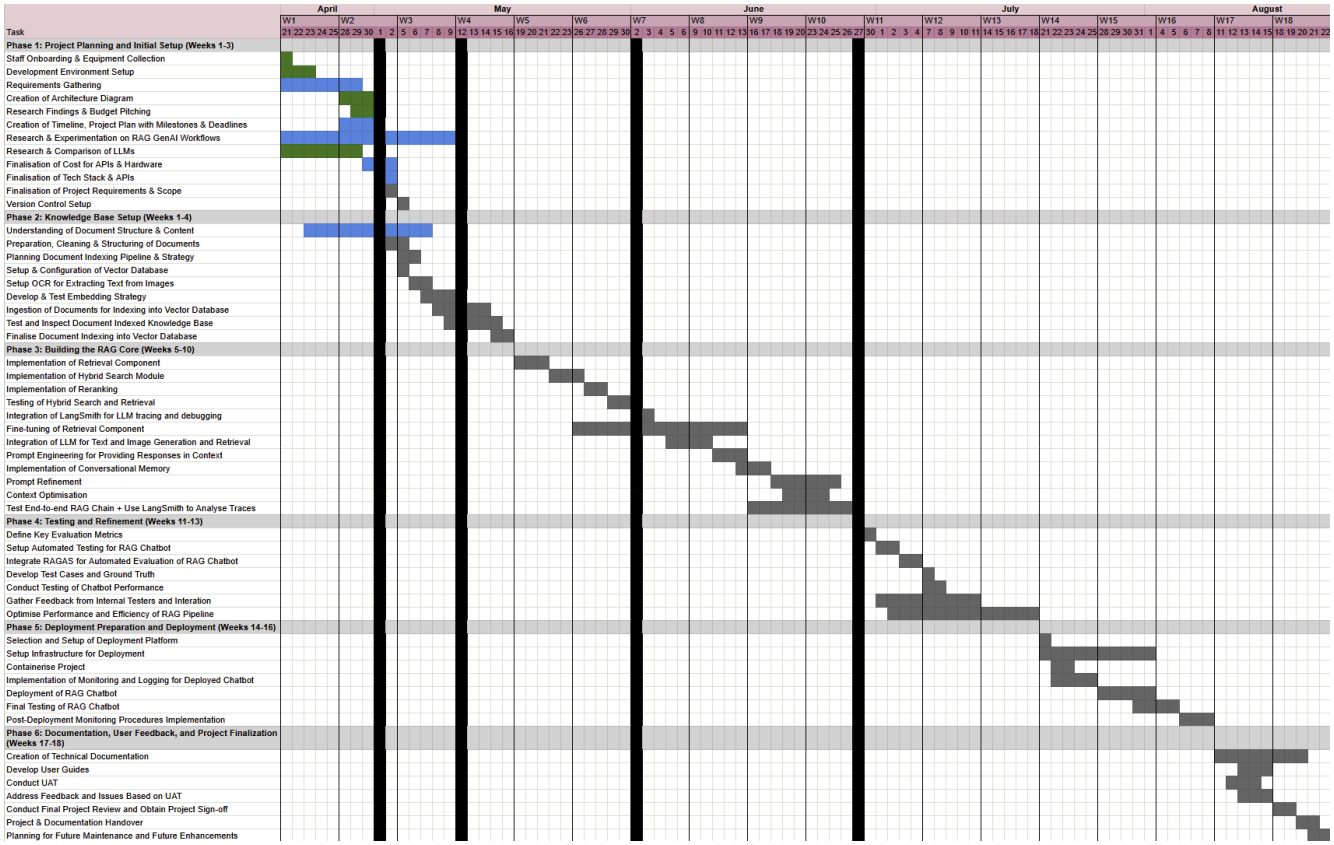


Figure 0: Initial Project Timeline

The project milestones outline the planned phases of the YEScribe Proof of Concept (POC) development, ensuring a structured and measurable approach to achieving its objectives. Each milestone is associated with specific deliverables that demonstrate progress, validate functionality, and provide tangible outputs for stakeholder review. Given the POC nature of this project, certain workstreams will run in parallel to accelerate development while maintaining quality through continuous testing and iteration. Testing will be integrated at each stage to validate both functional requirements (e.g., retrieval accuracy, ingestion completeness) and non-functional requirements (e.g., responsiveness, usability), with refinements made before progressing to subsequent milestones.

Table 1: Project Milestones and Deliverables

Milestone	Timeline	Deliverables
1. Project Initiation & Requirement Gathering	Week 1-2	• Confirm project scope and objectives • Identify key stakeholders and select key CPs to be tested • Document functional and non-functional requirements to be focused on in the development of the project
2. System Design & Architecture	Week 3	• Creation of high-level architecture diagram • Selection of technology stack (LangChain, ChromaDB, FastAPI, Streamlit, Docker) • Data ingestion and indexing workflow design
3. ETL Pipeline Development (Parallel)	Week 3-9	• Document ingestion pipelines for DOCX/PDF CPs • Semantic chunking and metadata tagging • Indexed select CPs into vector database • Unit testing for ingestion accuracy and metadata completeness
4. RAG Core Development (Parallel)	Week 6-9	• Development of retrieval pipeline with reranking • Integration of LLM for context-aware answers • Implementation of Conversational memory • Implementation of answer streaming from LLM • Enhancement of results and cost reduction measures • **Testing for retrieval relevance, responsiveness, and quality of answers**
5. Frontend Chatbot Development	Week 10-12	• Streamlit-based chatbot UI • Conversational interface linked to backend • Basic styling and usability features • **UI/UX testing for responsiveness and user experience**

6. API Endpoint Creation & Backend Integration	Week 13-14	• REST and WebSocket endpoints for chatbot • Backend connection to ETL and RAG pipelines • **Integration testing to ensure data flow and performance**
7. Containerisation, Deployment & Project Documentation	Week 15–16	• Dockerized application • Deployment on secure in-house infrastructure • **README.md file for developer onboarding** • **Deployment validation testing**
8. User Acceptance Testing (UAT), Final Documentation & Handover	Week 17–18	• UAT sessions with selected stakeholders • Feedback and evaluation report • Final POC report (capabilities, limitations, recommendations) • Handover of source code repository • Knowledge transfer for future maintainers • Creation of technical documentation and user manual

1.1.9. Executive Summary

YEScribe is a Proof of Concept (POC) AI-powered chatbot developed for YTL Communications to enhance the retrieval, interpretation, and comparison of Concept Papers (CPs). CPs are critical documents used across multiple departments, including Quality Assurance (QA), backend development, and enterprise solutions, to define technical requirements, business objectives, and process workflows.

At present, accessing CP content relies heavily on Business Analysts (BAs) to manually locate and interpret information, resulting in bottlenecks, inefficiencies, and inconsistent interpretations. YEScribe addresses this challenge by leveraging **Retrieval-Augmented Generation (RAG)**, ETL pipelines, vector databases, and cloud-sourced Large Language Models (LLMs) to provide users with context-aware answers through a conversational interface. The system supports multi-format interpretation, including text, tables, and flowcharts, and enables version comparison to track changes over time.

The POC scope focuses on validating the feasibility and value of the solution through the ingestion of a selected number of Concept Papers (CPs), ensuring alignment with functional requirements such as retrieval accuracy and non-functional requirements such as responsiveness and usability. Development is carried out in parallel workstreams, including ETL pipeline creation, RAG core development, and frontend interface implementation with testing integrated at each stage to enable iterative refinement.

Upon successful completion, YEScribe will provide a scalable, secure foundation for AI-assisted CP management, reducing reliance on manual processes, improving decision-making speed, and ensuring consistent access to critical project information across the organisation.

1.2. Delegation of Tasks

1.2.1. Work Breakdown Structure

The Work Breakdown Structure (WBS) for this project is used to ensure that all deliverables are clearly defined as well as facilitating effective planning and tasks monitoring. The table below illustrates the tasks which are broken down into key phases and deliverables as well as its planned duration of work (in terms of man days).

Table 2: Work Breakdown Structure Table

Phase / ID	Task	Subtasks / Deliverables	Planned Duration
1.0 Project Management & Planning	1.1 Project Initiation	• Define POC scope and objectives • Problem identification • Identify key stakeholders • Creating requirements document	Week 1–2
	1.2 Scheduling & Coordination	• Create project timeline and milestones • Weekly progress meetings	Week 2, Ongoing
2.0 System Design & Architecture	2.1 Architecture Design	• Creation of high-level architecture diagram • Defining data flow and integration points	Week 3
	2.2 Technology Selection	• Select frameworks, libraries, and deployment tools for development	Week 3
3.0 Backend Development	3.1 ETL Pipeline	• Creation of document parsing pipelines (DOCX/PDF) • Semantic chunking and metadata tagging • Vector database indexing	Week 3–9

	3.2 RAG Core	• Creation of retrieval pipeline with reranking • LLM integration for QA • Implementation of conversational memory • Testing quality of LLM generation	Week 6–9
	3.3 API Endpoint Development	• Creation of REST and WebSocket endpoints • Backend-to-frontend integration • Integration testing	Week 13–14
4.0 Frontend Development	4.1 Chatbot Interface	• UI design and layout (Streamlit) • Connection to backend services • UI/UX testing	Week 10–12
5.0 Deployment & Documentation	5.1 Deployment	• Scaffold FastAPI endpoints • Creation of Dockerfile • Containerisation of project • Deploy on in-house infrastructure	Week 15–16
	5.2 Project Documentation	• Creation of README.md • Creation of technical documentation & user manual	Week 16–18
	5.3 UAT & Handover	• User acceptance testing • Feedback collection and report • Final POC report and source code handover	Week 18–19
6.0 Quality Assurance	(Ongoing across phases)	• Functional and non-functional testing • Iterative improvements based on test results	Continuous

1.2.2. Activity List

The Activity List outlines all the tasks required to complete the YEScribe Proof of Concept (POC). Derived from the Work Breakdown Structure (WBS) these activities are sequenced to reflect the planned development timeline, taking into account for the parallel workstreams and continuous testing and iteration.

Table 3: Table of Activity List for Project

No	Activity Description	Phase / WBS Reference	Planned Duration
1	Define POC scope and objectives	Project Initiation (1.1)	Week 1
2	Identify stakeholders and CP dataset	Project Initiation (1.1)	Week 1–2
3	Document functional and non-functional requirements	Project Initiation (1.1)	Week 1–2
4	Create project timeline and milestones	Scheduling & Coordination (1.2)	Week 2
5	Conduct weekly coordination meetings	Scheduling & Coordination (1.2)	Ongoing
6	Create high-level architecture diagram	Architecture Design (2.1)	Week 3
7	Define data flow and integration points	Architecture Design (2.1)	Week 3
8	Select frameworks, libraries, and deployment tools	Technology Selection (2.2)	Week 3
9	Develop document parsing scripts for DOCX/PDF	ETL Pipeline (3.1)	Week 3–4
10	Implement semantic chunking and metadata tagging	ETL Pipeline (3.1)	Week 4–6
11	Index CP sample set into vector database	ETL Pipeline (3.1)	Week 6–9
12	Conduct unit testing for ingestion accuracy	ETL Pipeline (3.1)	Week 6–9
13	Implement retrieval pipeline with reranking	RAG Core (3.2)	Week 6–7
14	Integrate LLM for context-aware answers	RAG Core (3.2)	Week 7–8

15	Add conversational memory	RAG Core (3.2)	Week 8–9
16	Test retrieval relevance and answer quality	RAG Core (3.2)	Week 8–9
17	Develop REST endpoints for chatbot backend	API Endpoint Development (3.3)	Week 13
18	Develop WebSocket endpoints for chatbot backend	API Endpoint Development (3.3)	Week 13
19	Conduct backend–frontend integration testing	API Endpoint Development (3.3)	Week 14
20	Design chatbot UI in Streamlit	Chatbot Interface (4.1)	Week 10–14
21	Connect UI to backend APIs	Chatbot Interface (4.1)	Week 11–14
22	Perform UI/UX testing	Chatbot Interface (4.1)	Week 12-14
23	Dockerise application	Containerisation (5.1)	Week 15-16
25	Create README.md for developer onboarding	Project Documentation (5.2)	Week 16
24	Deploy on in-house infrastructure	Containerisation (5.1)	Week 17-18
26	Prepare technical documentation	Project Documentation (5.2)	Week 16–18
29	Deliver final report and source code repository	UAT & Handover (5.3)	Week 18
30	Provide user guide	UAT & Handover (5.3)	Week 18
27	Conduct UAT sessions with stakeholders	UAT & Handover (5.3)	Week 18-19
28	Collect feedback and prepare evaluation report	UAT & Handover (5.3)	Week 18-19
31	Perform functional and non-functional testing at each stage	Quality Assurance (6.0)	Ongoing
32	Implement refinements based on testing results	Quality Assurance (6.0)	Ongoing

1.2.3. Workload Distribution

In the YEScribe Proof of Concept (POC), responsibilities are distributed across a small team with overlapping roles, given the limited resources typical of POC development. The table below explains in detail the roles and responsibilities to the members involved in the project.

Table 4: Workload Allocation Table

Team Member	Role	Responsibilities
Bruce Chin Yuan Zhen (Industrial Supervisor)	Project Manager	● Facilitate code review sessions ● Provide feedback on implementation ● Help gather key stakeholders for meetings
Me (See Thoo Jun Lok)	● Requirements Gatherer ● Full-Stack Developer ● AI Engineer ● Partial DevOps ● Technical Writer	● Gather functional and non-functional requirements ● Design and develop ETL pipeline for document ingestion and indexing ● Implement RAG core, including retrieval pipeline, reranking, LLM integration, and conversational memory ● Develop Streamlit-based frontend chatbot interface ● Containerise application in Docker ● Prepare README.md and technical documentation ● Conduct functional testing during development phases ● Conduct User Acceptance Testing (UAT)
Chow Suet Ling (Co-Intern)	UI/UX Designer	● Provide UI/UX design concepts for chatbot interface enhancements ● Propose usability improvements for potential production deployment
Pritish (Tech Lead)	Infrastructure Deployment	● Deploy Dockerised application to in-house infrastructure ● Manage server configuration and environment variables

		● Oversee production readiness for infrastructure
Stakeholders & Test Participants	UAT Testers	● Participate in User Acceptance Testing (UAT) ● Provide feedback on system usability, accuracy of answers, and responsiveness ● Suggest improvements for future phases

Chapter 2: Literature Review

2.1. Problems

2.1.1. Problem Statement

At YTL Communications, Concept Papers (CPs) play a crucial role as both business and technical documentation, supporting a wide range of upcoming, concurrent projects, both interrelated and independent. Given their importance, CPs are frequently updated according to stakeholder feedback and project requirements. However, the current process for accessing and interpreting Concept Papers (CPs) within YTL Communications presents multiple challenges that affect efficiency, accuracy, and scalability. CPs contain critical business, technical, and process-related information used by various teams, including but not limited to: Quality Assurance (QA), backend development, enterprise solutions, and project management to align project requirements and execution. However, the way these documents are authored, stored, and queried creates significant barriers.

Identified Problem Areas:

1. **Manual Retrieval and Interpretation**: Accessing the relevant information from CPs require manual reading, cross-referencing, and interpretation. This process is extremely time-consuming and easily prone to human error. In addition, this is further complicated by the fact that process flows within CPs can be extremely comprehensive and intricate, extending to multiple flows and outcomes. The large size and complexity of these diagrams make it difficult to locate specific steps or understand the full process workflow.

2. **Dependency on Business Analysts (BAs)**: Stakeholders, primarily the backend, quality assurance (QA), and developers frequently rely on BAs to answer CP-related queries. As a result, it creates bottlenecks and delays, especially when BAs are unavailable or overburdened with other tasks, such as creating new CPs.

3. **Lack of Version Tracking and Accessibility to Latest CPs**: CPs are updated periodically and have to undergo extensive approval from business owners, namely from

head of departments and directors at YTL Communications. However, there is currently no way for stakeholders to determine whether the CP that they have at hand is the latest version and will also have to ask BAs if the version that they have is the latest version. Moreover, stakeholders often have to manually compare older and newer versions to identify changes in flows when no explicit changes are mentioned in the revised sections.

4. **Scalability Concerns**: As the number of CPs grows, manual query of CPs becomes increasingly unsustainable, thus slowing decision-making processes and killing productivity across teams.

5. **Untapped Potential for AI Integration**: There is currently an opportunity to leverage AI technologies such as Retrieval-Augmented Generation (RAG) which can help to automate and enhance CP retrieval, interpretation, and summarisation. But such solutions currently do not exist due to the time and knowledge constraints across multiple teams.

2.2. Similar Developed Systems

To determine the feasibility and relevance of the YEScribe Proof of Concept (POC), a review of existing systems and solutions with similar objectives was conducted. The focus was on AI-powered document retrieval and conversational interfaces capable of interpreting structured and unstructured content, as well as systems that implement Retrieval-Augmented Generation (RAG) for domain-specific knowledge bases. A SWOT analysis table is constructed to provide an in-depth and summarised view of the similar systems' strengths, weaknesses, opportunities for improvement over the proposed RAG chatbot for YTL Communications, and the threats it brings that make them unfeasible.

Table 5: SWOT Analysis of Similarly Implemented Systems

System	Strengths	Weaknesses	Opportunities	Threats
ChatGPT with Custom Knowledge Base (Lin, 2024)	• Advanced natural language understanding and conversational abilities • Multi-domain knowledge with flexible API integration • Large ecosystem and frequent updates	• Requires external API calls (data privacy concerns) • Recurring API costs • Not optimised for CP-specific multi-format interpretation • No support for custom implementation	• Customise with domain-specific embeddings • Integrate with internal RAG pipelines Possibility of fine-tuning	• Vendor dependency and possible price changes • Risk of vendor lock-in

Microsoft SharePoint + Azure Cognitive Search (Pondhouse Data, 2024)	• Enterprise-ready integration • Secure document storage and access controls • Supports keyword and semantic search • No user interaction required	• Interpretation of document elements, such as embedded documents or images (process flows/flowcharts) • Manual version tracking • Extra services required for RAG, overcomplicating workflows	• Extend with Azure Cognitive Services • Integrate with Microsoft Teams workflows	• Licensing costs • Complexity in domain-specific adaptation
Haystack Framework (deepset-ai, 2022)	• Open-source and highly customisable, decreasing development and integration times • Supports RAG, multi-modal search, and local deployment • No vendor lock-in	• Requires development of custom pipelines • No native CP-specific features	• Fully tailor to organisational workflows • Leverages strong availability of support from open-source community	• Ongoing internal maintenance burden
Onyx (Danswer) (onyx-dot-app, 2025)	• Open-source and optimised for AI deployment with in-house infrastructure	• Smaller community compared to Haystack	• Ideal for secure deployment of AI-powered solutions	• Slower feature updates due to smaller dev base • Higher initial setup complexity

	• Strong orchestration for LLMs and RAG pipelines	• Requires in-house expertise for setup and maintenance	• Potential to fully customise for CP workflows	

Summary of Findings

Existing systems demonstrate the potential and viability of AI-powered document retrieval solutions but lack required features for building a chatbot with the context of CPs, these include: semantic multi-document extraction, automated version tracking, and in-house secure deployment. YEScribe differentiates itself by combining these elements into a single, domain-specific solution that is tailored to extract and understand CPs, allowing users to query and perform specific tasks catered to different teams.

2.3. Technologies

2.3.1. Technologies for Implementation

The development of YEScribe's Proof of Concept (POC) requires a carefully chosen set of technologies enabling secure, efficient, and intelligent retrieval of raw content from Concept Papers (CPs). As CPs contain sensitive business and technical information of unreleased features, the implementation stack must ensure data confidentiality while simultaneously maintaining high performance, responsiveness, and accuracy.

The technologies to be identified must address several core requirements:

1. **Data Security and In-House Processing**: All document storage and processing must occur within YTL Communications' infrastructure to protect sensitive CP content from being exposed to external systems.
2. **Semantic Retrieval and Contextual Understanding**: Beyond simple keyword search and plain text extraction, YEScribe requires semantic search capabilities using techniques such as vector embeddings and reranking models to return the most relevant content based on user queries.
3. **Conversational Querying**: Users are able to interact in natural language and receive contextually accurate, and easy to understand answers, powered by a connected large language model (LLM) while returning the source of its information in its answers for users to cross verify.
4. **Ease of Documentation**: The chosen technologies should be well-documented and support clear, structured documentation which will be used in aiding the understanding of the system's architecture, workflows, and codebase for current and future maintainers.
5. **Maintainability for Future Developers**: The stack should align with YTL Communications' existing development practices, ensuring that future maintainers can easily extend or modify the system without significant rework or having to relearn.
6. **Rapid Prototyping and Iteration**: As a POC, the technology stack must enable quick development cycles and easy testing of different components.

7. **Scalability and Future Extensibility**: While the initial focus is on the retrieval of CP documents, the chosen stack should be flexible enough to scale to more document types and support additional workflows.

The identification process involved:

1. Defining the **functional** and **non-functional** requirements of the POC.
2. Researching and shortlisting **technology options** for each functional layer including backend services, frontend interface and framework, ETL pipeline, AI models, RAG architecture, vector database, and deployment tools.
3. Evaluating these options against **benchmarks** of existing and similar implementations to determine the suitability for an in-house, RAG deployment.
4. Selecting the most appropriate tools based on **security, performance, maintainability, documentation quality, and integration compatibility**.

The following section (**2.3.2 Study on Technologies for Implementation**) details the specific tools and frameworks selected for YEScribe, along with the rationale behind each choice.

2.3.2. Study on Technologies for Implementation

A detailed evaluation of each shortlisted technology was conducted to ensure suitability for the YEScribe Proof of Concept (POC). The study considered performance, integration feasibility, data security, scalability, and alignment with the project's functional and non-functional requirements.

Programming Language (Python): Python was selected for its simplicity, ease of development, and wide variety of mature libraries for large language model (LLM) and Retrieval-Augmented Generation (RAG) tasks. It integrates seamlessly with **FastAPI** which is built on Python for building asynchronous endpoints, enabling support for parallel task execution (Ahmed, 2024). This design choice aims to reduce latency in delivering responses, which is essential for a smooth conversational experience. It also hastens the development process and allows for quick iteration. In addition, it is also used in the **Streamlit** framework for building simple yet powerful, production-ready frontend interfaces which help to further accelerate development while narrowing the development to be focused on improving chatbot performance instead (GeeksforGeeks, 2020).

Vector Database (Chroma): Chroma offers an open-source, in-house vector storage solution that supports semantic search while maintaining full control over CP data (Awan, 2023). This aligns with YTL Communications' preference for secure, internal processing without reliance on external cloud-hosted storage.

Embeddings Model (all-MiniLM-L6-v2 - Development), (Qwen Embedding 0.6B - Planned Production): For local development, **all-MiniLM-L6-v2** was chosen for its lightweight nature, fast processing, and low hardware requirements, enabling rapid iteration (Hugging Face, n.d.). For production deployment, **Qwen Embedding 0.6B** was identified as the preferred model for its superior semantic retrieval accuracy in domain-specific contexts with minimal hardware requirements as its bigger 8B model topping the **Massive Text Embedding Benchmark (MTEB)** which measures semantic performance with subsequent models of the same family achieving relatively similar results (as of July 2025) (Qwen Team, 2025). The selection of embedding models was guided by benchmark results from the **Massive Text Embedding Benchmark (MTEB)**, ensuring the choice was data-driven and performance-validated.

Reranker Model (BGE-Reranker-V2-M3): The reranker is another foundational component in a typical RAG architecture that aims at improving retrieval precision by reordering retrieved results based on semantic relevance, reducing irrelevant matches. Model selection was informed by **MTEB evaluations**, which compared reranker performance across diverse retrieval tasks (Beijing Academy of AI, 2023).

LLM (Google Gemini 2.5 Flash): Google Gemini 2.5 Flash was selected for its balance of speed, accuracy, and cost-efficiency, costing only RM1.29 and RM10.75 per million tokens for input and output respectively as compared to its Pro variant, which costs RM5.38 and RM40.30 per million tokens for input and output respectively (Google, 2025). Decision for its selection is solidified based on **LM Arena** evaluations for generation and content understanding, ensuring the chosen model performs well in both conversational and summarisation tasks. In addition, Gemini offers:

- **Seamless integration with LangChain** for RAG pipeline orchestration.
- **Traceability via LangSmith**, enabling logging, debugging, and optimisation.
- **Future extensibility with LangGraph** for advanced multi-turn conversational flows.

Gemini 2.5 Pro will instead be used for content analysis as it excels in understanding various document structures, ranking **first** in the **Intelligent Document Processing Leaderboard** for document analysis (Mandal et al., 2025).

Containerisation (Docker): Docker enables consistent deployment across environments by encapsulating dependencies and configurations into portable containers. This ensures reproducible builds and facilitates secure in-house deployment (Bhuyan, 2024). Upon further discussion, the deployment environment is in Linux while the development platform is in Windows where both have different system architectures and syntaxes.

Version Control (Git): Git was chosen as it is the standard version control system used across YTL Communications' existing projects. This ensures compatibility with established workflows and allows future maintainers to easily manage and extend the repository. Git's branching and rollback capabilities also support iterative POC development.

API Testing (Postman): Postman was used for testing and validating API endpoints throughout development. It allowed structured test cases, quick iteration during debugging, and verification of both REST and WebSocket responses before integrating them into the frontend (GeeksforGeeks, 2025).

2.3.3. Finalised Technology Stack

Based on the evaluation of potential solutions in Section 2.3.2, the following technology stack has been finalised into a table for the YEScribe Proof of Concept (POC).

Table 6: Finalised Technology Stack for Development

Layer	Technology	Purpose
Programming Language	Python	Backend and frontend (Streamlit) logic, AI pipeline orchestration, RAG workflows
Vector Database	ChromaDB	Local semantic search index
Embeddings Model (Dev)	all-MiniLM-L6-v2	Extremely lightweight vector representation of CP content during development, allowing for fast results
Embeddings Model (Prod)	Qwen Embedding 0.6B	High-accuracy, low-cost semantic vector representation in production
Reranker Model	BGE-Reranker-V2-M3	Reorders retrieved results by semantic relevance
LLM	Google Gemini 2.5 Flash/Pro	Conversational response generation and content understanding
Containerisation	Docker	Consistent, portable deployment environment
Version Control	Git	Codebase version tracking and collaboration
API Testing	Postman	API endpoint validation and debugging

2.4. Framework

2.4.1. Frameworks for Development

The selection of frameworks for YEScribe's Proof of Concept (POC) follows the same principles outlined for technology selection in Section 2.3. The chosen frameworks must support rapid prototyping, seamless integration between system components, and the ability to operate entirely within YTL Communications' internal infrastructure.

The frameworks must:

1. Enable **high-performance and scalable backend services** with asynchronous request handling for responsiveness.
2. Provide **rapid, frontend prototyping** to allow iterative testing and immediate stakeholder feedback, while putting more emphasis on improving RAG QA chain performance.
3. Offer **robust, yes scalable orchestration capabilities** for Retrieval-Augmented Generation (RAG) pipelines.
4. Integrate with **observability and debugging tools** which will be used to facilitate tracing, QA chain monitoring, and model fine-tuning.
5. Be **flexible and maintainable** to allow future enhancements, including scaling beyond CP-related workflows.

2.4.2. Study on Frameworks for Development

A selection of 3 frameworks were selected for the following layers in the project (Backend, Frontend & LLM Orchestration), these frameworks include:

- **FastAPI**: Selected for backend API development due to its asynchronous capabilities, high performance, and easy integration with AI pipelines as compared to other Python-based backend frameworks, such as Flask and Django (Lakshitha, 2025). Moreover, it supports both REST and WebSocket endpoints for chatbot interaction and allows straightforward containerisation.

- **Streamlit**: Used for developing the chatbot UI quickly and integrating it directly with the proposed FastAPI-powered backend. This makes it suitable for POC-phase testing and demonstration without requiring heavy front-end development effort, shifting the focus to developing and improving RAG chains and document extraction quality (GeeksforGeeks, 2020).

- **LangChain**: Serves as the orchestration framework for chaining together the proposed retrieval, reranking, and LLM generation components as well as extracted data (The Educative Team, 2025), simplifying development and deployment to production. In addition, LangChain also integrates with LangSmith for observability purposes, being able to perform tasks, such as tracing LLM calls, debugging and evaluating performance of QA chains, such as latency, token costs, etc (Langchain, n.d.). Lastly, LangChain also supports LangGraph, being able to integrate agentic capabilities and memory easily to YEScribe (LangChain, n.d.).

2.4.3. Finalised Frameworks for Development

Based on the evaluation of potential frameworks to be used for YEScribe in Section 2.4.2, the following frameworks used have been finalised for the YEScribe Proof of Concept (POC).

Table 7: Finalised Frameworks for Development

Layer	Framework	Purpose
Backend	FastAPI	Build high-performance, asynchronous REST and WebSocket APIs
Frontend	Streamlit	Develop chatbot UI and integrate with backend logic
RAG/LLM Orchestration	LangChain	Chain together retrieval, reranking, and LLM generation steps, integrates with LangSmith and LangGraph for observability and memory and agentic capabilities respectively

Chapter 3: System Design and Analysis

3.1. Frontend

3.1.1. UI Design and Assets

3.1.1.1 Design Inspiration & Scope

The UI takes heavy inspiration from popular conversational apps, such as **OpenAI ChatGPT**, **Google Gemini**, and **Anthropic Claude** of which prioritise a clean chat canvas, a sticky input bar, transparent sourcing, and a compact sidebar for in-context actions, such as selection of specific CPs and approval statuses.

This project's emphasis is instead focused on **improving RAG quality, approvals transparency, and ETL reliability**. The Streamlit-based UI is kept minimally to enable quick iteration and to validate backend behavior end-to-end. It also allows for easier testing as complicated interactions may stray away from the focus of the project.

3.1.1.2 Design Goals

This subsection concludes the guiding objectives for the frontend by enabling rapid iteration while validating the RAG, approvals, and ETL back end end-to-end.

- **Clarity:** A simple, legible chat layout and consistent spacing.
- **Traceability:** Every generated response links back the documents and content retrieved from the CPs.
- **Task separation:** Two dedicated interfaces for uploading the CPs (CP Upload Portal) and for querying the contents of the CP through a user-friendly chat interface (CP RAG Chatbot)
- **Safety:** The ability to reset and create a new chat in case the context is not what users want as well as deletion of uploaded CPs
- **Quick iteration:** Minimal focus on frontend enhancements while focusing on rapid iteration and development and enhancement of RAG backbone based on gathered user feedback from group meetings and discussions.

3.1.1.5 UI Overview

For this project, the frontend is splitted into **two dedicated interfaces** to reduce cognitive load and keep controls context-specific: an **Upload Portal** for ingestion/maintenance (upload, auto-upload, history, delete/recover with validation and an audit trail) and a **Chatbot** for inquiry (ask, reset, scope by CP/version, view approvals, show sources).

3.1.1.5.1 CP Upload Portal

A focused workspace for onboarding and maintaining Concept Papers (CPs).

- **Functions:** Upload .docx CP files, enable auto-upload from SharePoint/specified directory, view history, upload status display, and perform delete/recover operations.
- **Design intent:** Operational clarity and safety, with validation feedback and audit logs.

3.1.1.5.2 RAG CP Chatbot

A conversational workspace to query indexed CP content.

- **Functions:** Ask questions, reset sessions, scope retrieval by CP/version, view approvals, and inspect sources (text + images/tables via web URLs).
- **Design intent:** Transparent, verifiable answers with minimal UI friction.

3.1.1.4 Custom Components Specific to This Problem

To address the specifications and observed pain points, the frontend introduces a set of lightweight, purpose-built components, which are directly injected into the two UIs proposed for this project, these components include:

- **CP Selection Accordions (Chatbot UI)**: Searchable CP list and version selection to narrow results on known CPs
- **Approval Status Accordion**: Overall status (Approved / Pending) and per-approver lines (name, role, decision, timestamp, remark) for key stakeholders to view the completion/verification status of a particular CP.
- **Upload Controls**: Single/batch **CP** ingestion with inline validation feedback.
- **Auto-Upload Button/Toggle**: Enable a backend watcher to check an external storage (e.g., a predefined SharePoint directory) and ingest new files automatically.
- **CP History Accordion**: View what was uploaded and when; prevents BAs from re-uploading the same CP, and serves as an audit trail.

- **Delete / Recover (Guarded)**: Remove/recover a CP's indexed data and embeddings (for error recovery and hygiene).

3.1.1.6 Key Interaction Flows (Design Level)

This section guides the end-to-end user journeys across the two designed UIs (**Upload Portal and Chatbot**), which focuses on what the user triggers, what the system does (validation, pipeline enqueue, retrieval + rerank), and what feedback is shown (status indicators, History updates, Show Sources).

1. Upload CP → Validate (type/corruption/duplicate) → Send to ETL pipelines → Show upload status → Update History.
2. Enable Auto-Upload → Watcher monitors SharePoint → New CPs ingested automatically to Chroma vector database → History reflects events.
3. Delete/Recover → Typed confirmation → Deletion action executed → History updated.
4. Ask Chatbot → (optionally) Filter CP/version → Response generation with show sources.
5. Review Approvals → Open Approval Status to see overall and per-approver status.

3.1.1.8 Acceptance Criteria (Design-Level)

This section defines what must be visibly supported in the UI without prescribing implementation, allowing reviewers to verify whether the frontend meets the project's design intent (clarity, traceability, quick iteration).

- Upload Portal clearly supports **upload**, **auto-upload**, **history**, and **guarded delete/recover**.
- Chatbot clearly supports **chat**, **reset**, **CP/version scoping**, **approvals view**, and **show sources**.
- UI remains minimal to enable **quick iteration**; advanced visual polish is intentionally left out of scope due to limited time and resources, focus towards RAG development prioritised.

3.1.2. Finalised UI Design and Assets

3.1.2.1 CP Upload Portal UI

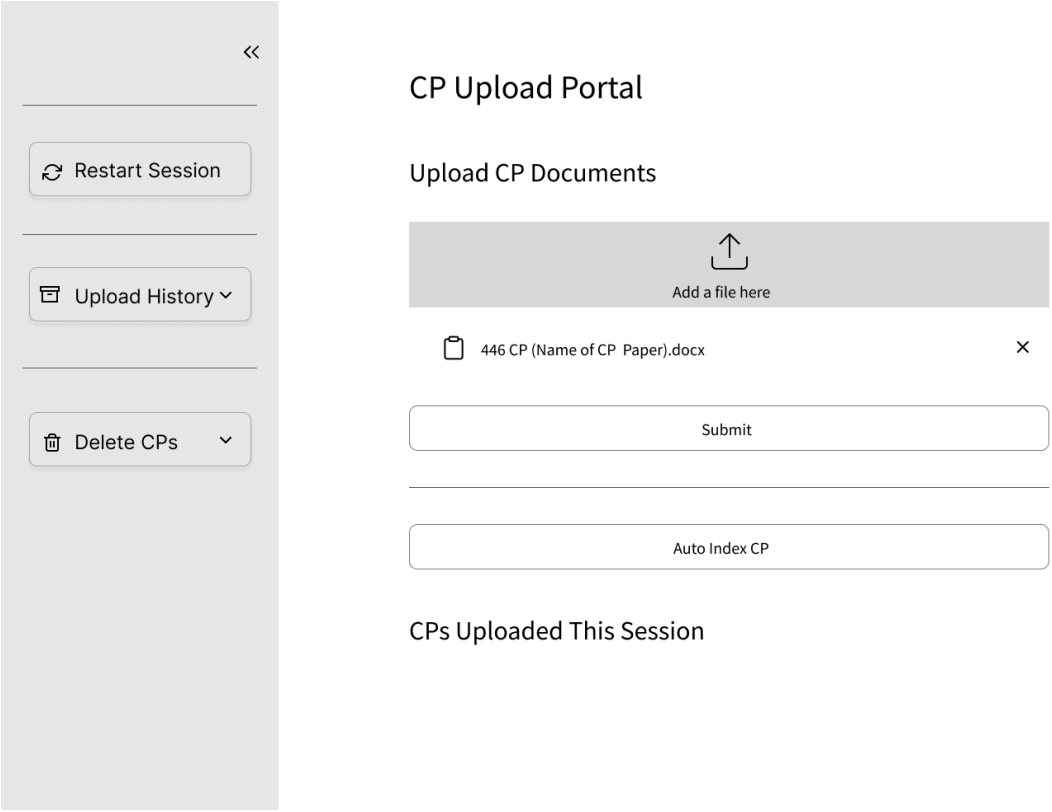


Figure 1: CP Upload Portal – Initial Upload Screen

User uploads a Concept Paper (CP) .docx file with options to submit or auto-index, alongside session controls for restarting, viewing upload history, or deleting CPs.

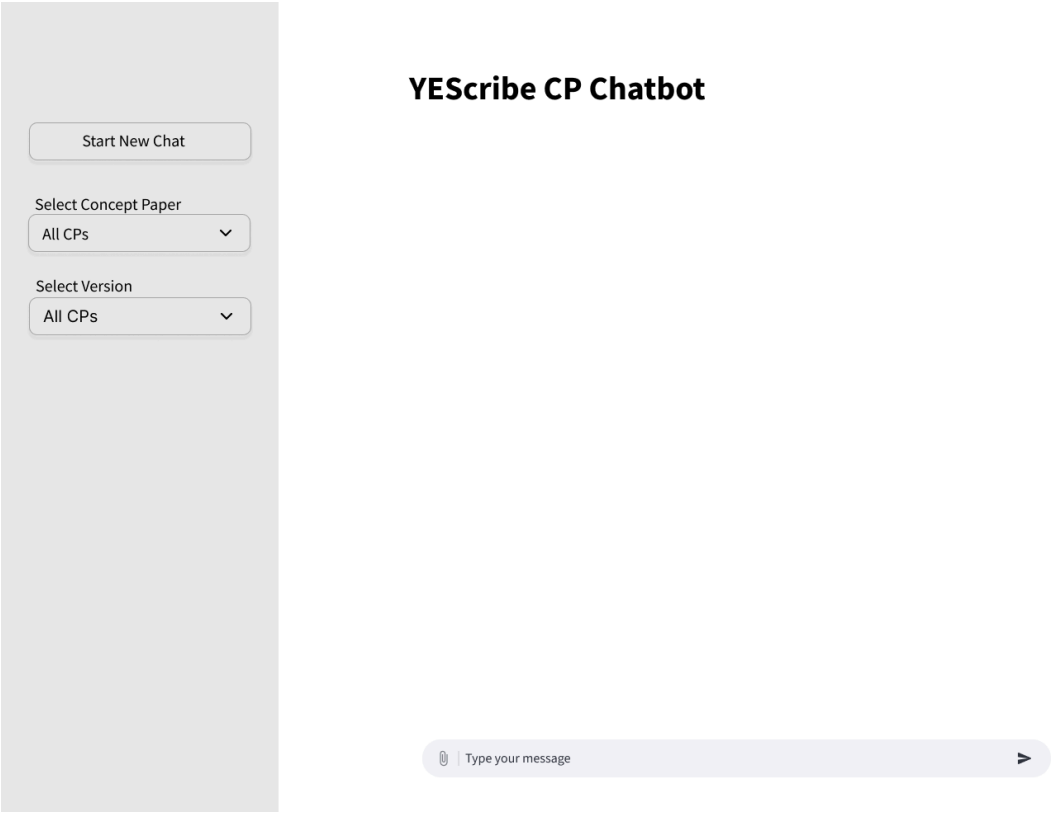


Figure 2: YEScribe CP Chatbot – Idle State

Chat interface where users can start a new chat, select a Concept Paper and version, and interact with the chatbot for queries.

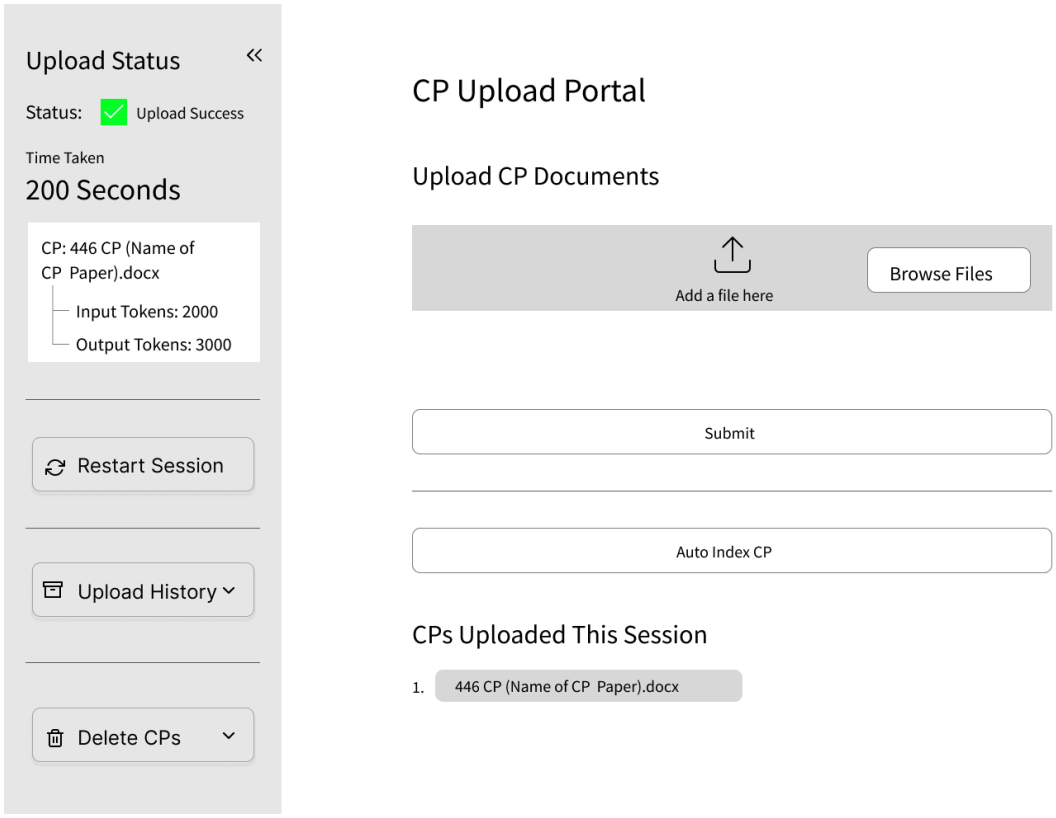


Figure 3: CP Upload Portal – Upload Success

After uploading, the status panel shows “Upload Success”, indicating successful indexing of a CP with processing time and token usage (input/output) shown for more information about the indexing process. Uploaded CPs are listed in the current session afterwards.

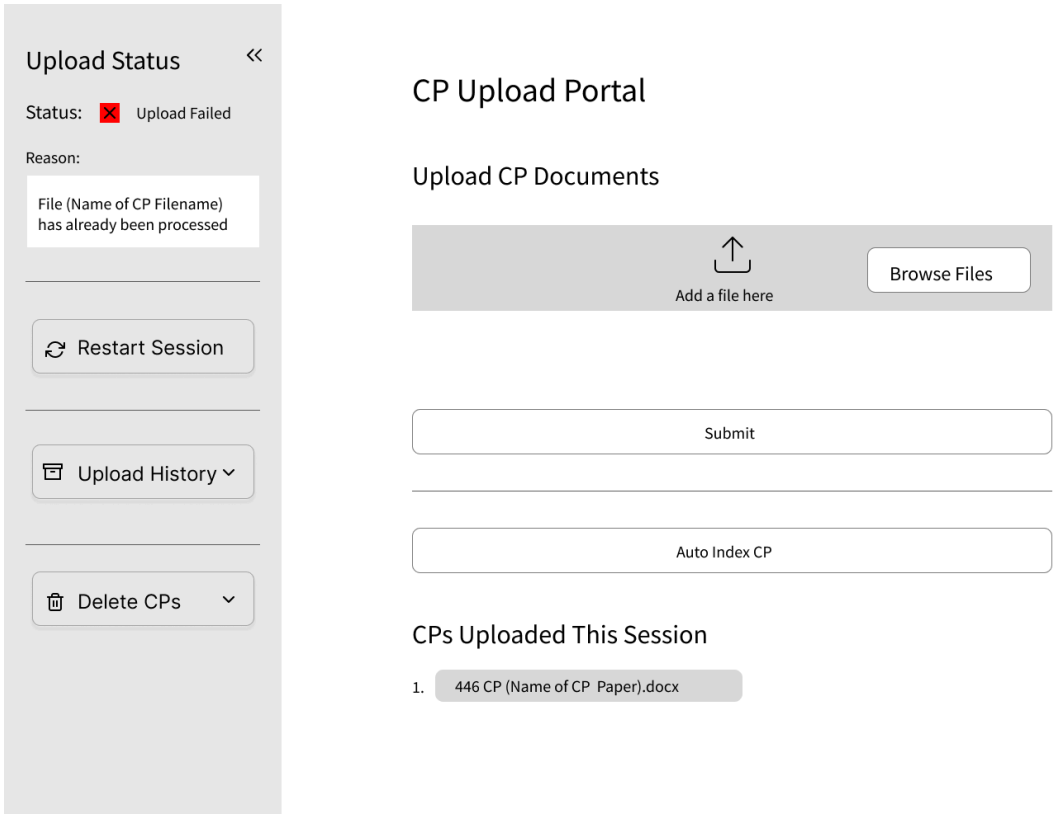


Figure 4: CP Upload Portal – Upload Failed

Upload attempt fails due to several reasons, such as duplicate detection (“File has already been processed”) (like the one shown in this example), invalid file types, or corrupted CP file detected. The status panel displays the specific reason for failure, with session controls still available.

3.1.2.2 RAG CP Chatbot UI

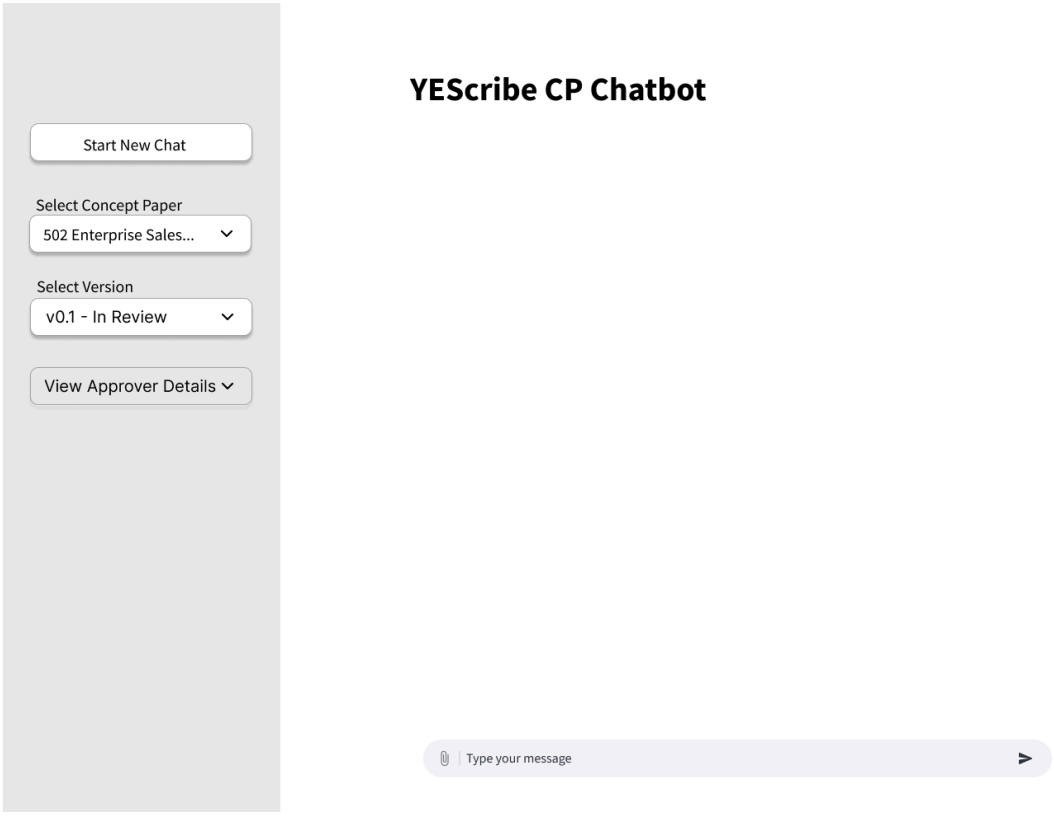


Figure 4: YEScribe CP Chatbot – CP Version Selection

User selects a specific Concept Paper and corresponding version (e.g., v0.1 – In Review) for scoped queries. Option to view approver details is available. The status of the CP is also shown to indicate whether a CP has been approved by all business owner or not.

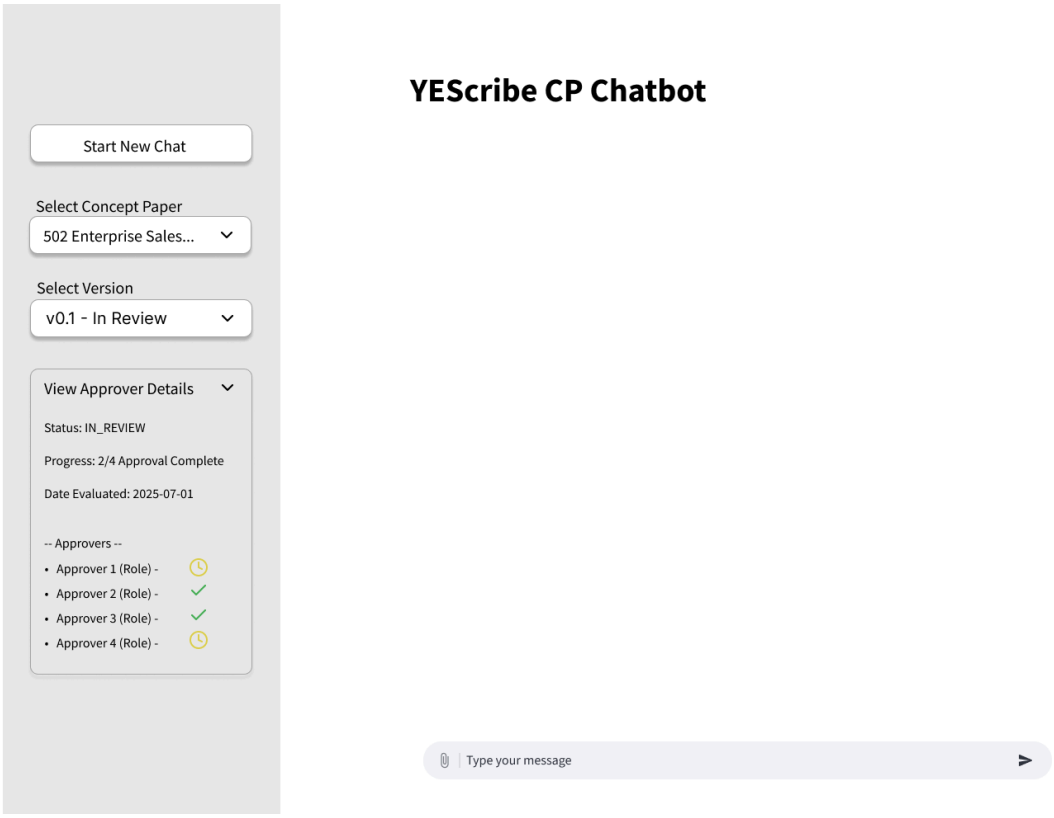


Figure 5: YEScribe CP Chatbot – Approver Details View

Approval status of the selected CP version is shown, indicating review progress, evaluation date, and individual approver statuses with clear indications: tick for approved, clock/hourglass for pending state.

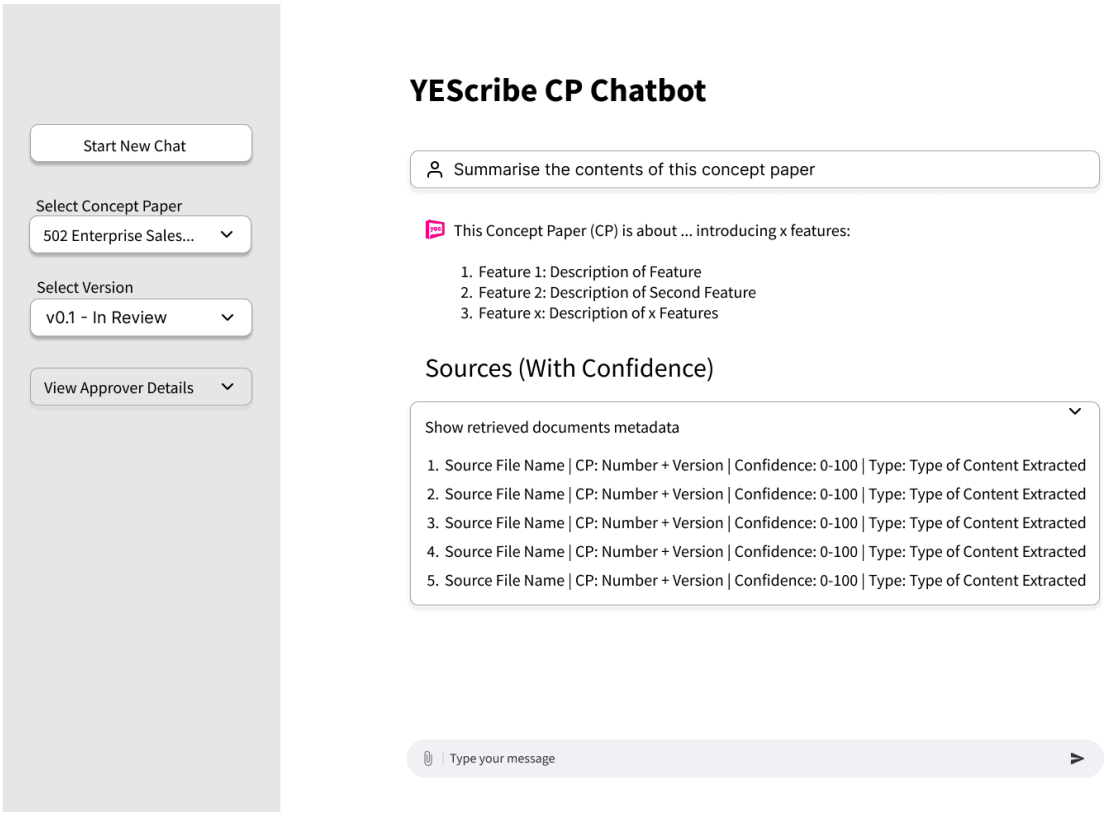


Figure 6: YEScribe CP Chatbot - Query Response with Sources

After sending a prompt to the chatbot, the chatbot will stream messages directly to the user after a short while (retrieval, reranking and prompt restructuring performing). After streaming, the content used to generate the appropriate response will be shown to the user via an accordion below the finalised response, clearly indicating the name of the source file, which CP and version it was retrieved from, the confidence score which indicates the semantic similarity of the content extracted to answer the user's query, and finally the type of content retrieved from the CP.

3.2. Backend

3.2.1. Entity-Relationship Diagram (ERD)

3.2.1.1 ERD For Concept Paper Indexing / ETL

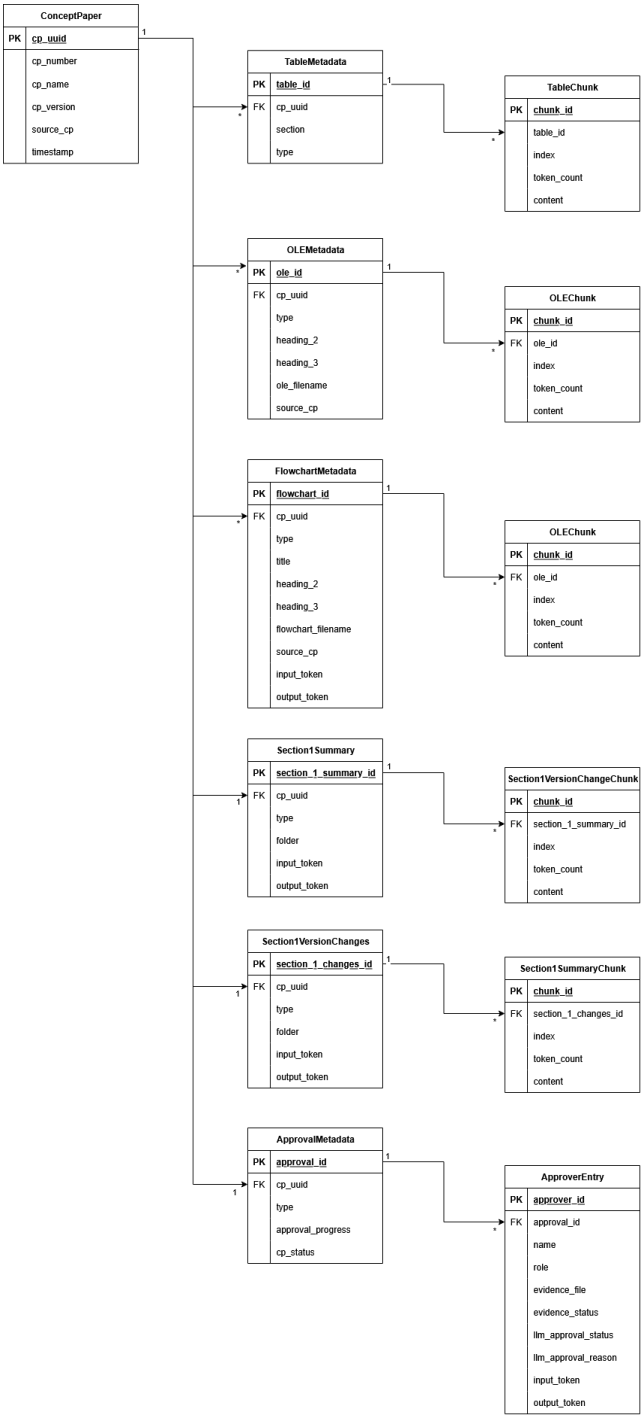


Figure 7: ERD For Concept Paper ETL Chunks to Chroma Vector DB

The ERD above illustrates how the CP Upload Portal backend logic manages the **ETL (Extract, Transform, Load) process for** Concept Papers (CPs) that have been uploaded. At the core is the **ConceptPaper** entity, which serves as the parent record for all uploaded CPs, it captures the CP's number, name, version, and upload location. Once uploaded, the CP is processed through 5 separate ETL pipelines, which include: **Tables, OLE Objects, Flowcharts, Section 1 Summaries, Section 1 Version Changes, and Approvals** with each having its own metadata entity. These metadata entities preserve contextual details such as section headers located near the extracted elements from the CP, name of the extracted content/files, approval status (exclusive for approval pipeline), and token usage (exclusive for approval, flowchart, and section 1 pages).

To support semantic retrieval and in reducing both the input token cost and improving response times, every pipeline output is further broken down into **chunk entities** via the implementation of a chunking strategy, where extracted text, images, or structured data are normalized into smaller fragments with token counts and chunk index (to indicate the number of chunks created from a particular file/extracted content. These chunks are then embedded and stored in **Chroma**, forming the knowledge base that powers retrieval-augmented generation (RAG) chatbot for CP documents.

This modular structure ensures that each CP version can be fully indexed, while also enabling fine-grained querying across tables, flowcharts, OLE documents, and approval records.

3.2.1.2 ERD For RAG Chatbot Session

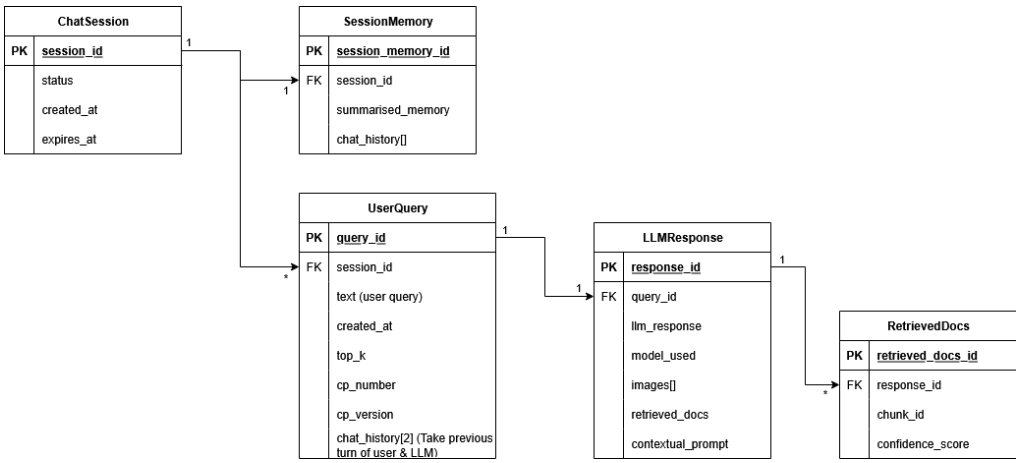


Figure 8: ERD For Rag Chatbot Session

The ERD for the actual RAG CP chatbot interface is used to illustrate how each conversation created by the user to interact with the chatbot is structured and stored to support both user experience and backend processing while maintaining security over the internals of the system.

Firstly, a **ChatSession** is created and is unique for every user to prevent other users from accessing the same chat while isolating memory pertaining to that session. The ChatSession entity also acts as the container for an entire interaction of multiple user queries (**UserQuery**). Each **UserQuery** records what the user asked and produces a corresponding **LLMResponse** from the LLM, unique to that UserQuery. Moreover, it stores the answer text and links it to the **RetrievedDocs** which are the chunks retrieved from the Chroma vector database used as evidence and reference for the response.

To maintain conversational flow, a **SessionMemory** entity keeps track of summarised dialogue history, ensuring responses remain relevant over multiple turns.

3.2.2. Proposed System Workflow

3.2.2.1 CP Upload Portal

The numbered flow below provides a detailed explanation of the proposed system workflow of the YEScribe CP Upload Portal for CP upload and indexing in the backend:

1. **User Uploads CP Files** via one of three modes:
    - **Single upload** (manual file selection).
    - **Batch upload** (multi-file selection, processed in sequence).
    - **Monitored folder** (auto-upload - system detects and enqueues new/updated files automatically.
2. **Validation** checks:
    - Ensure the uploaded CP is in.docx extension format.
    - Check for file corruption (valid ZIP structure for extracting images and embedded files (OLE)).
    - Reject temporary or partial CP (~$, ~, .tmp) due to cloud sync issues/internet connectivity.
    - Check against processing_log.csv to prevent processing the same CP twice.
3. **Pipeline Execution** (sequential, per file):
    - **Approval Pipeline:** Extract approval files (.msg).
    - **Table Pipeline**: Extract tables appearing before Section 1.
    - **OLE Pipeline**: Extract embedded files/objects.
    - **Flowchart Pipeline**: Extract and process flowcharts.
    - **Section 1 Summary Pipeline**: Generate Section 1 summary and version changes (if older version available).
4. **Embedding & Storage**:
    - Documents are **chunked and embedded** into **ChromaDB**.
    - Embeddings and metadata are stored in the specified directory.
5. **Logging & Archiving**:
    - Capture **token usage, processing time, and chunk counts**.
    - Move processed files to a **dated archive folder**.
    - Record archive details in processing_log.csv.

3.2.2.2 RAG CP Chatbot Workflow

The numbered flow below provides a detailed explanation of the proposed system workflow of the RAG QA chain and conversational memory-powered YEScribe RAG CP Chatbot Workflow in the backend:

1. **Application Launch & Session Initialization**
    - When the application starts, a unique **session id** is generated, which is used in tracking queries, responses, and conversation memory across multiple interactions between the user and chatbot.
2. **CP/Version Selection** *(Optional, for Narrowed Scope)*
    - The user may select a specific CP and version from the repository.
        i. If provided → retrieval is scoped strictly to that CP/version.
        ii. If omitted → retrieval searches across all CPs/versions.
    - This helps pass in to the chatbot as a filter when retrieving chunks of data from the Chroma vector database knowledge base for CPs
3. **User Query Input**
    - The user types a natural language question into the chatbot interface.
    - Example queries:
        i. *Without CP selected*: "What approvals were required for project submissions?" → global retrieval.
        ii. *With CP v1.2 selected*: "Summarise the changes in the newly modified enterprise sales flow." → scoped retrieval.
4. **Frontend Request Packaging**
    - The frontend sends a structured JSON payload to the backend with the following attributes:
        i. session_id
        ii. cp_number
        iii. cp_version
        iv. user_query
    - Additionally, the backend also retrieves any stored history (summarised history and recent 1 turn), to be passed to other components later on, such as the retriever

and LLM, providing conversational context and enabling multi-turn interactions, such as follow-up questions from users.

5. **Memory Summarisation (Long-Range Context)**
    - Older turns are summarised using ConversationSummaryBufferMemory.
    - This compresses earlier context while preserving continuity of goals, preferences, and CP focus, with the aim of reducing token usage, reducing latency for generation and keeping responses within the context window.
    - Example: Multiple approval-related queries → summarised as *"User comparing approval workflows across CP versions."*
6. **Dual-Context Query Reformulation (Summary + Last Turn + Prompt Template)**
    - A standalone query is built for retrieval by combining:
        i. User Query (new input).
        ii. Summarised History (before last turn) (long-range memory).
        iii. Full Previous Turn (verbatim last exchange).
        iv. Prompt Template (Retriever Layer) (instruction to rewrite into a fully self-contained query).
    - The reformulated query is:
        i. Self-contained (no unresolved "this" or "that").
        ii. Explicitly scoped to CP/version if provided.
        iii. Optimised for embedding and retrieval.
7. **Embedding & Retrieval**
    - The reformulated query is embedded into vector embeddings using a pre-defined embedding model.
    - ChromaDB is queried for top-k relevant chunks based on the vectorised, reformulated query. In addition, if selected, it also scopes to the defined filter of the selected CP and version, else it will traverse the entire vector database.
8. **Reranking**
    - Retrieved chunks are reranked using **BGE-Reranker-M3-V2**, ensuring that the most semantically relevant results are prioritised as well as providing a confidence score at the end afterwards.

9. **Context Enrichment**
    - Reranked results are enriched with previously inserted metadata into the created and stored chunks from the vector database, providing grounded evidence for the response; these include:
        i. Flowchart images (process visualisations).
        ii. Approval records (tables).
        iii. Section/page references.
10. **LLM Generation**
    - The final answer is generated using Geminis 2.5 Flash using a structured prompt , composed of the following elements:
        i. User Query - User's original query
        ii. Full Previous Turn - Verbatim last exchange for pronoun resolution.
        iii. Summarised History (before last turn) - Compressed long-range context.
        iv. Retrieved Context - Reranked document chunks with metadata retrieved from the vector database.
        v. Prompt Template - Fine-tuned instructions guiding answer style, structure, and grounding.
11. **Streaming Response Delivery**
    - The LLM output is streamed token-by-token to the frontend.
    - The UI displays:
        i. Incremental answer text.
        ii. Linked sources with confidence scores (after generation).
12. **Monitoring & Evaluation**
    - All traces (interactions) are logged in LangSmith for:
        i. Token usage.
        ii. Latency.
        iii. Traceability.
    - Continuous monitoring ensures system improvement and fixes deployed swiftly in case of system errors

3.2.3. UML Class Diagram

3.2.3.1 CP Upload Portal Architecture

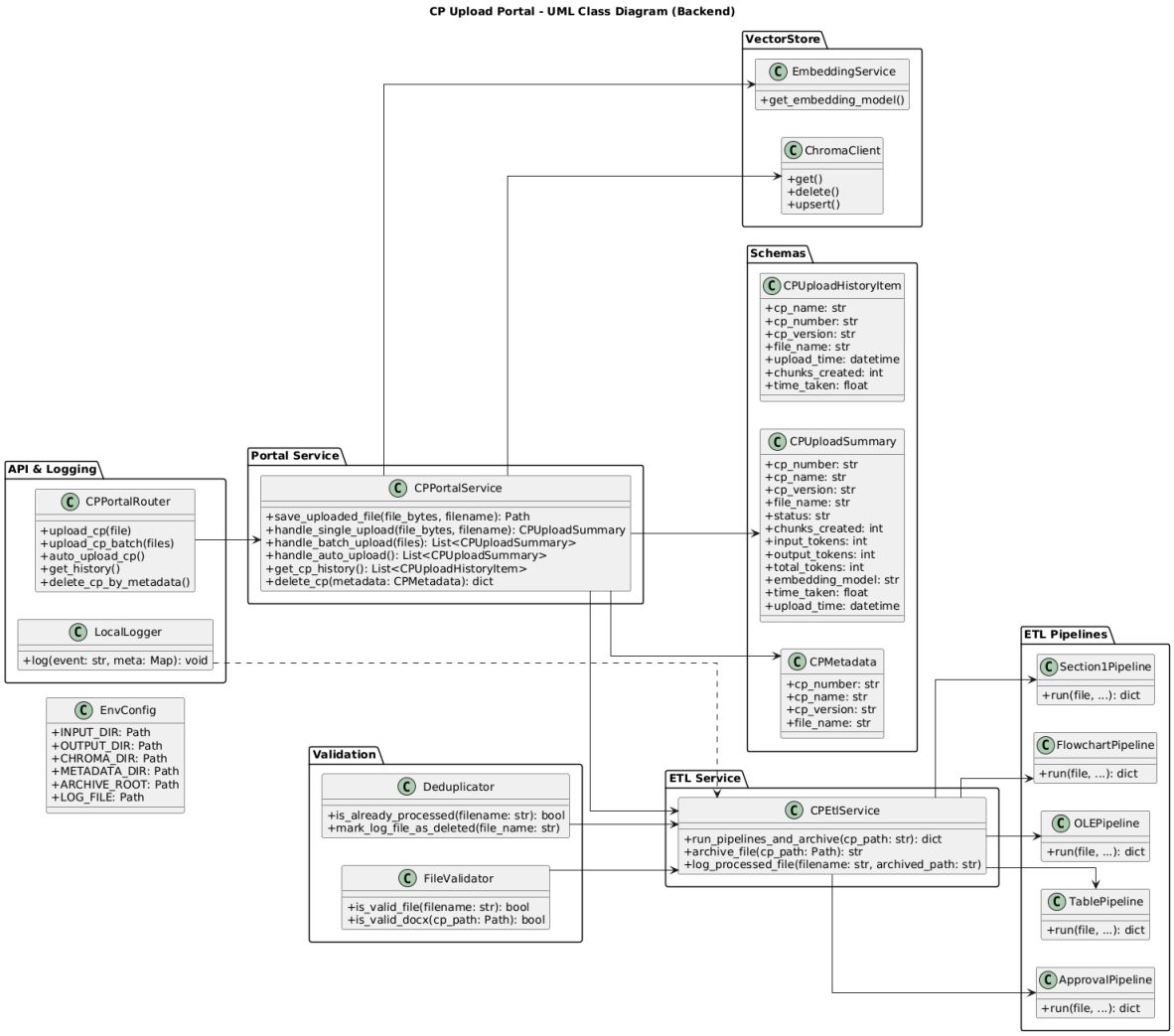


Figure 9: UML Class Diagram - CP Upload Portal

The class diagram for the CP Upload Portal above is organised into several but curated groups that reflect on the proposed system's architecture. Each group of classes addresses a distinct stage in the lifecycle for the indexing of CPs across multiple pipelines, from API entry and validation, to ETL pipeline execution, vector embedding storage, and schema management for maintaining structure and type safety of extracted data (chunks). This ensures that responsibilities are clearly defined, components are reusable, and the system remains scalable and maintainable for future implementations. The below section explains, in detail, each of the groups and grouped classes on its specific functionality and linkage between one another.

3.2.3.1 API & Logging

This group provides the external interface for the CP Upload Portal and ensures traceability through logging and configuration management.

- **CPPortalRouter:** The CPPortalRouter acts as the entry point for all external requests from the frontend upload interface. It defines all the related API endpoints for uploading CPs, retrieving history, and managing deletions. Instead of handling the logic itself, it delegates responsibilities to another class/file called CPPortalService. Other than that, it also interacts with LocalLogger to record events, helping to ensure that every request is properly tracked for accountability.
- **LocalLogger:** The LocalLogger is responsible for recording system events across the lifecycle of CP processing. It helps in logging CP uploads, file validation checks, pipeline executions, and archival results. It provides an independent trail for system activity, which helps to enable operational traceability and debugging and is mainly called by CPPortalRouter, CPPortalService, and CPEtlService.
- **EnvConfig:** The EnvConfig class centralises all environment-specific paths and configurations, ensuring security and consistency across the system by hiding the path from the code itself and providing a single source of truth for directories like input, output, archive, metadata, and Chroma. Other components such as CPPortalService, CPEtlService, and ChromaClient rely on it to access locally defined paths and API keys for key resources, such as LLM calls for content inference reliably.

3.2.3.2 Validation

This group ensures that only valid, unique, and process-ready files are admitted into the ETL workflow.

- **FileValidator:** The FileValidator verifies that uploaded files are only in .docx format, intact, and not temporary placeholders. It runs before indexing to be performed by the 5 ETL pipelines, helping the system by protecting downstream services from unnecessary failures caused by invalid or corrupted files.

- **Deduplicator:** The Deduplicator prevents redundant processing by checking against processing_log.csv or equivalent records. It flags already-processed files and records new entries after successful archival.

3.2.3.3 Portal Service

This group orchestrates the overall CP upload lifecycle, coordinating between validation, ETL, logging, and storage.

- **CPPortalService:** The CPPortalService is the main orchestrator of the upload portal. It manages uploads by first invoking FileValidator and Deduplicator, then handing off valid files to CPEtlService. Once processing is complete, it generates CPUploadSummary objects and updates CPUploadHistoryItem records. It also coordinates deletions by requesting the ChromaClient to remove vectors based on the specified CP tied to it.
- **CPEtlService:** The CPEtlService manages and chains all 5 defined ETL pipeline execution for each valid CP to be run in a specified sequence. It calls specialised pipelines (Approvals, Tables, OLE, Flowcharts, Section 1) to extract metadata and content, breaking them down into smaller chunks, then calls EmbeddingService to transform chunked outputs into vectors while finally storing them into the Chroma vector database through the ChromaClient class. In addition, it also measures token counts and logs performance metrics, ensuring complete oversight of the extraction process.

3.2.3.4 ETL Pipelines

These 5 classes specialise in extracting and transforming structured and unstructured data from Concept Papers, each focusing on a different type of content.

- **ApprovalPipeline:** The ApprovalPipeline extracts approval records, approver details, and decision statuses. Its outputs feed into downstream embeddings, making approval tracking searchable within the chatbot. Moreover, it also stores a local JSON file for showing approval status to be linked and handled in the frontend UI as well (linking with “Show Approval Status” section in the frontend chatbot UI).

- **TablePipeline:** The TablePipeline focuses on extracting specified, structured tables appearing before Section 1. It parses and normalises table data into chunks, retaining headers and page references for retrieval grounding based on user queries.

- **OLEPipeline:** The OLEPipeline processes embedded files and objects (such as Excel, PowerPoint, PDF, or email .msg files) that can be found in the CP. It extracts text or summaries and records metadata like filenames and nearby headings, ensuring that these objects can be retrieved as evidence.

- **FlowchartPipeline:** The FlowchartPipeline extracts flowcharts (process flows) while generating textual descriptions to support contextual retrieval. It links each generated description to flowchart filenames, associated headings as well as the path of where the extracted flowchart images are stored enabling multimodal context for chatbot responses.

- **Section1Pipeline:** The Section1Pipeline generates summaries of Section 1 and version changes (if older versions available). Its outputs are used to provide high-level overviews of the main body content of the CPs, helping retrieval systems quickly answer introductory or change-related queries.

3.2.3.5 **Vectorstore**

This group handles the transformation of extracted data into embeddings and their persistence in the Chroma vector database, including the metadata that is linked to each embedded chunk.

- **EmbeddingService:** The EmbeddingService converts extracted or generated text chunks into dense vector embeddings. It is defined in a way which allows future maintainers to switch the models without much effort based on its performance.
- **ChromaClient:** The ChromaClient manages the persistence of vectors and associated metadata in ChromaDB which serves as the knowledge base for the YEScribe chatbot, handling insertions during ingestion and deletions during CP removal in the CP Upload Portal UI. This separation allows the rest of the system to remain independent of database implementation details.

3.2.3.6 Schemas

These classes store structured records of metadata, summaries, and histories, which are hugely recommended for FastAPI when sending and receiving responses by ensuring uniformality of data as well as enabling traceability and analytics.

- **CPMetadata:** The CPMetadata provides a minimal record of each CP, capturing identifiers such as CP number, version, and file details. It anchors all extracted data to the original document, allowing for enhanced auditability and context for the chatbot.
- **CPUploadSummary:** The CPUploadSummary records the outcome of each upload, including status, chunk counts, and token usage. It provides a concise snapshot for front end users and BAs.
- **CPUploadHistoryItem:** The CPUploadHistoryItem maintains a long-term history of all uploads, including timestamps, versions, and archival detail, helping ensure that past actions are traceable and accessible for audits or future analysis while at the same time preventing BAs from uploading the same CP(s) to the vector database knowledge base.

3.2.3.2 RAG CP Chatbot Architecture

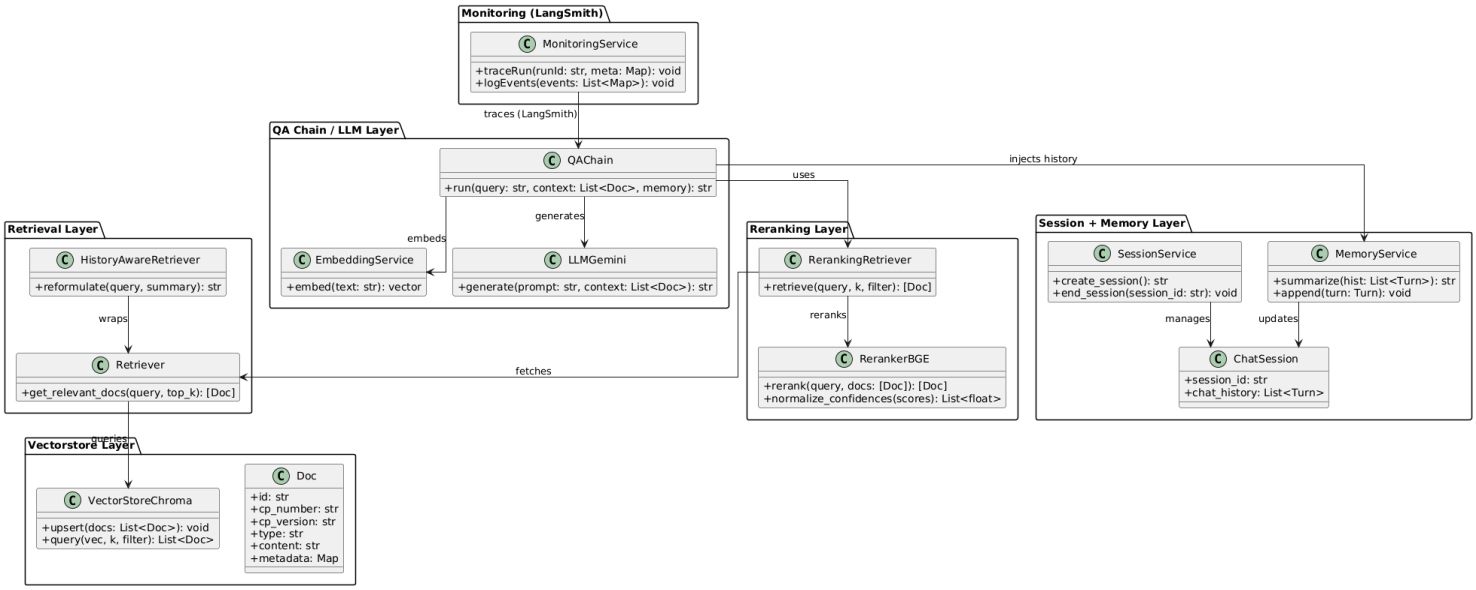


Figure 10: Class Diagram - RAG CP Chatbot Architecture

The Chatbot Class Diagram illustrates the modular layers of the Retrieval-Augmented Generation (RAG)-powered system. Each layer has a distinct role; retrieval fetches candidate documents based on user queries, reranking improves relevancy of retrieved documents passed to it, QA/LLM generates responses based on the restructured prompt based on original user prompt, context, fine-tuned prompt template and history, session and memory maintain conversational context, vectorstore ensures persistence, and monitoring tracks system behaviour. Together, these layers create a pipeline that transforms user queries into contextually accurate and explainable chatbot responses. The section below explains, in detail, each of the groups and their classes, focusing on their specific functionality and the linkages between one another.

3.2.3.2.1 Retrieval Layer

This layer manages query reformulation and document retrieval from the vectorstore.

- **HistoryAwareRetriever:** The HistoryAwareRetriever is responsible for reformulating user queries using summarised memory. It wraps the history around the base Retriever to ensure that conversational context (such as references to “this section” or “previous CP”) is clarified into a self-contained query.

- **Retriever:** The Retriever fetches relevant documents from the Chroma vectorstore based on embeddings. It exposes the method get_relevant_docs, which retrieves top-k documents matching the query vector. This class is the backbone of retrieval, and is wrapped by HistoryAwareRetriever for context-aware operation.

3.2.3.2.2 VectorStore Layer

This layer handles the storage and retrieval of document embeddings and metadata.

- **VectorStoreChroma:** The VectorStoreChroma manages persistence of document vectors and metadata in Chroma. It provides methods to upsert new documents (after embedding) and query existing vectors with optional filters such as CP number or version.

- **Doc:** The Doc class defines the schema for stored documents in the Chroma knowledge base. It includes identifiers (id, cp_number, cp_version), type, raw content, and metadata. This class ensures that retrieved results carry sufficient contextual information for grounding responses.

3.2.3.2.3 QA Chain / LLM Layer

This layer executes the RAG chain: embedding queries, generating answers, and integrating retrieved context.

- **QAChain:** The QAChain coordinates the overall question-answering process. It runs queries by passing reconstructed text, relevant retrieved context, and memory to the LLMGemini. It ensures that the generated output is grounded in retrieved knowledge.

- **LLMGemini:** The LLMGemini class represents the core large language model (Gemini). It generates final answers given a prompt, user query, and document context, ensuring output style, completeness, and grounding based on the finalised reformulated prompt.

- **EmbeddingService:** The EmbeddingService converts user queries into dense vector embeddings, which are sent to the retriever where contextual retrieval is performed in order to ultimately provide a contextually accurate response (explanation provided previously, under the QAChain and Retriever classes).

3.2.3.2.4 Reranking Layer

This layer improves the quality of retrieved results by applying semantic reranking models.

- **RerankingRetriever:** The RerankingRetriever is responsible for retrieving documents and then reranking them for higher contextual accuracy. It first fetches candidate documents from the retriever then delegates to the reranker to prioritise the most semantically relevant results based on the user's original query.

- **RerankerBGE:** The RerankerBGE implements the reranking algorithm using a BGE model. It reorders documents based on confidence scores, normalises scores, and passes the improved ranked list back to the RerankingRetriever.

3.2.3.2.5 Session + Memory Layer

This layer manages user sessions and conversational memory for multi-turn dialogue.

- **SessionService:** The SessionService manages the lifecycle of chat sessions. It creates new sessions, ends existing sessions, and tracks active session IDs. It ensures that each conversation has a unique context, linked to a collection of stored memory.
- **MemoryService:** The MemoryService maintains and updates conversational memory. It helps to summarise long histories for efficiency and appends new turns as they occur. It works hand in hand with SessionService to keep session memory consistent.
- **ChatSession:** The ChatSession stores session identifiers and a list of chat turns between users and the chatbot. It acts as the persistent record for ongoing conversations, updated continuously by MemoryService.

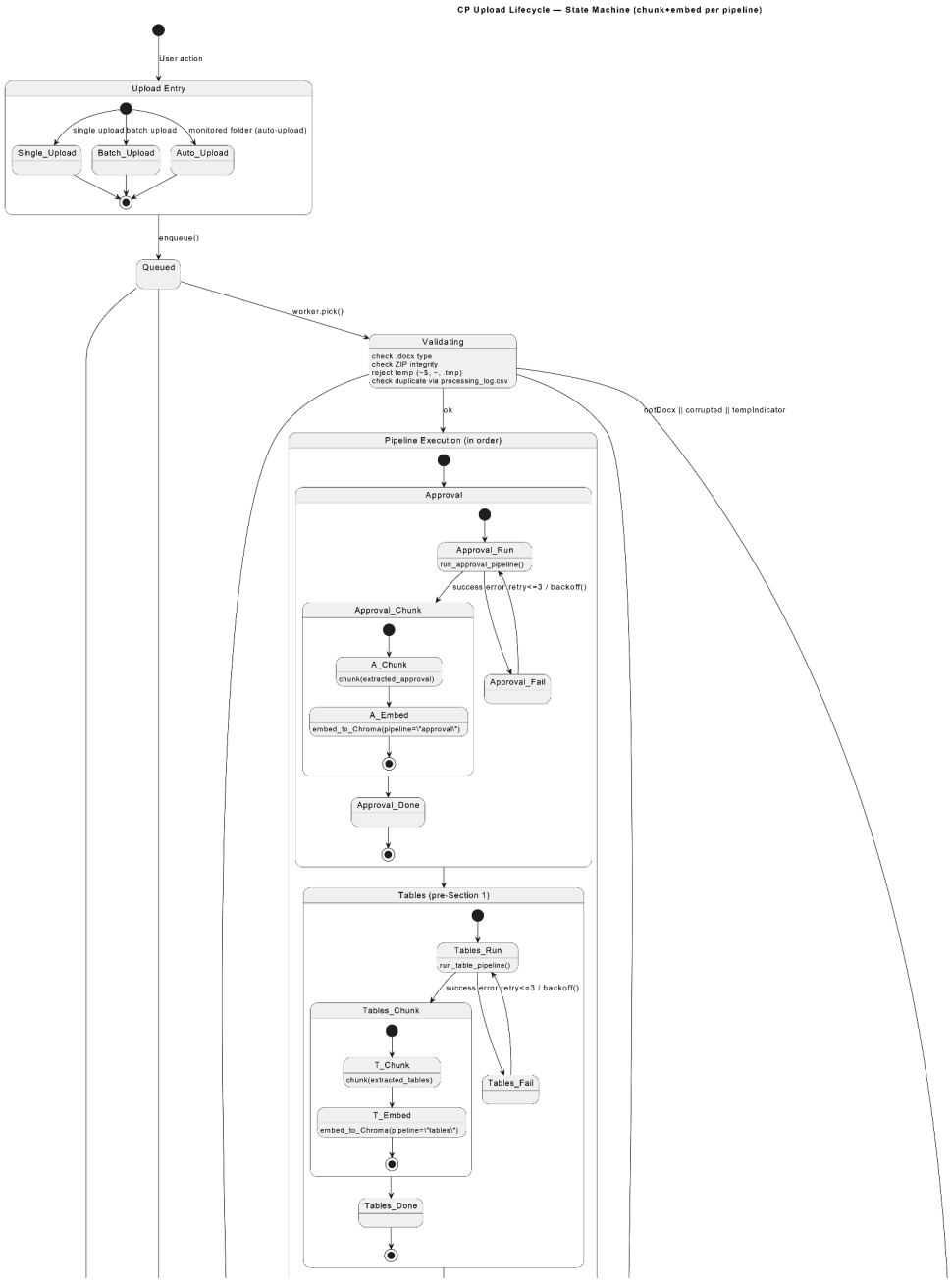
3.2.3.2.6 Monitoring Layer (LangSmith)

This layer integrates tracing and monitoring into the RAG workflow.

- **MonitoringService:** The MonitoringService connects to LangSmith for logging and tracing runs. It records execution events, traces queries of the QA chain, and captures metadata such as latency, usage tokens, and errors.

3.2.4. UML State Chart Diagram

3.2.4.1 CP Portal Upload State Chart Diagram



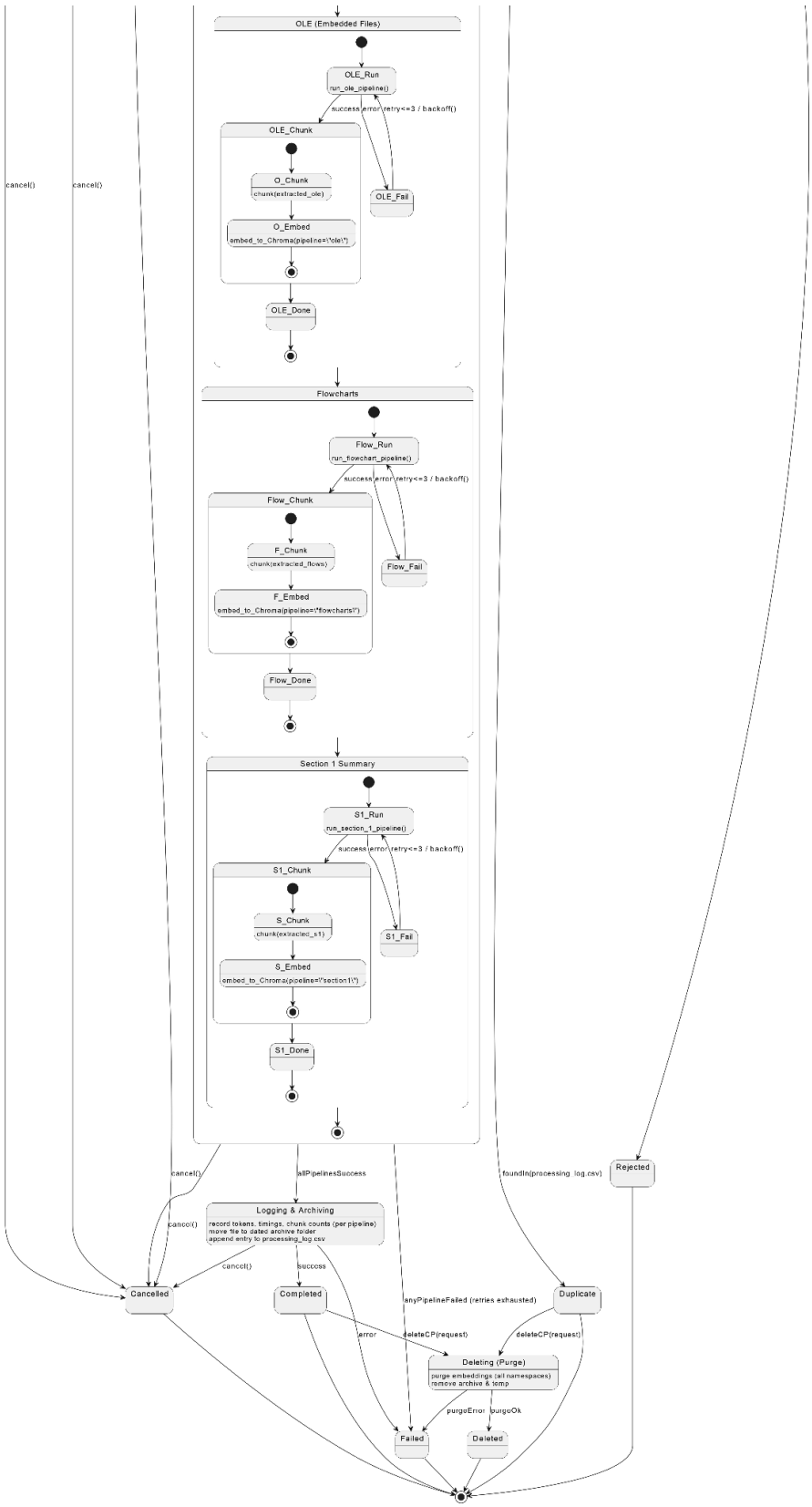
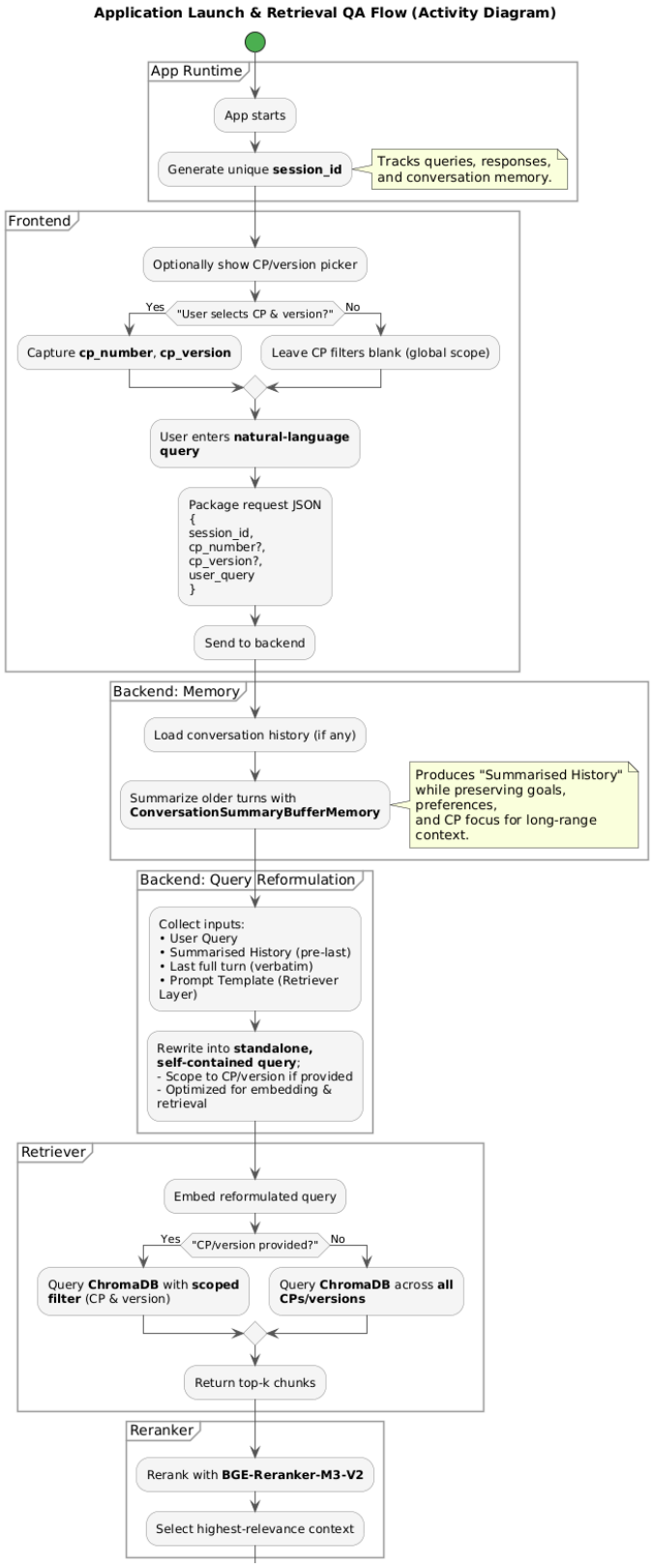


Figure 11: UML State Diagram - CP Portal Upload UI

3.2.4.2 Chatbot UML State Chart Diagram



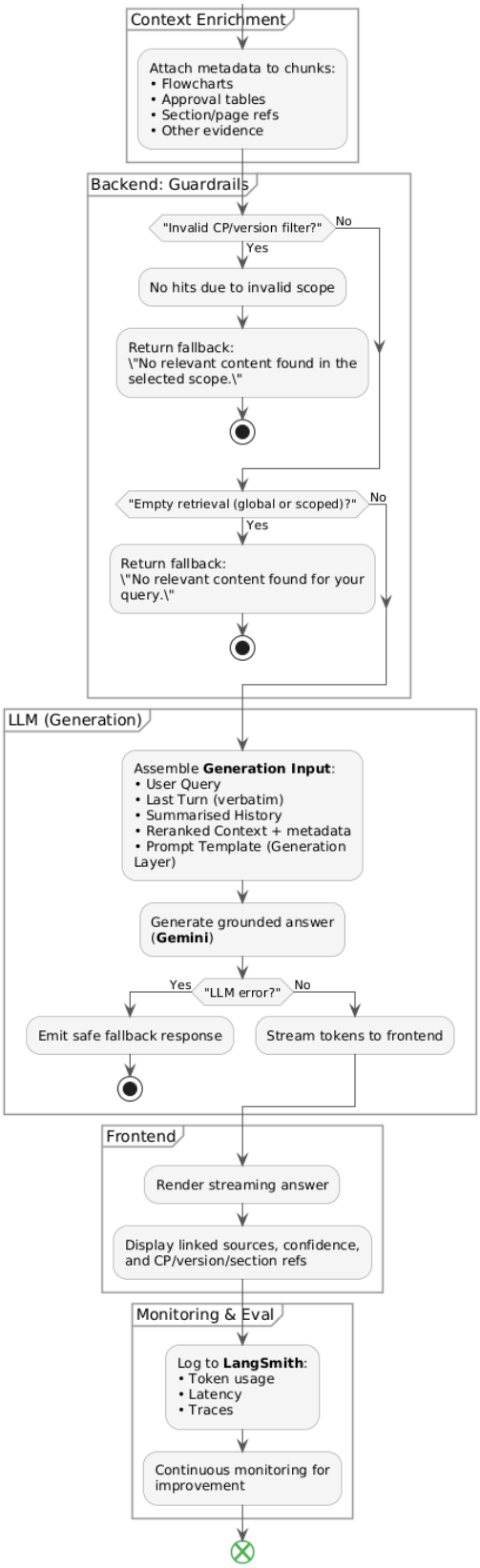


Figure 12: UML State Diagram - RAG CP Chatbot

3.2.5. Data Flow Diagram

3.2.5.1 CP Upload Portal

The Data Flow Diagrams (DFDs) for the CP Upload interface provide a structured view of how it interacts with backend services during the document ingestion process. They are illustrated at two levels of abstraction:

- **Level 0 DFD (Context Diagram):** The high-level view shows the CP Upload UI as a single process, connecting users (Business Analysts) to the backend. It captures essential inputs (CP files uploaded by users), key processes (validation and ETL execution), and outputs (indexed CP data, status updates, and archived files). At this stage, the focus is on showing the entities and connected data sources.

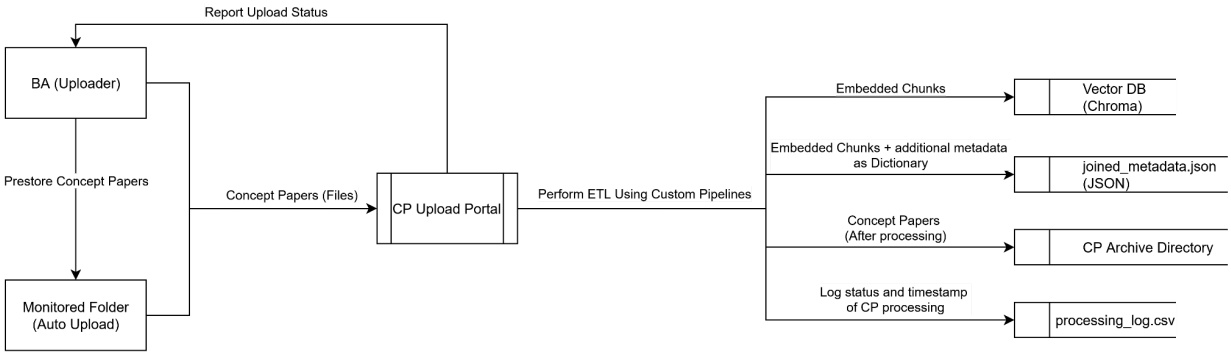


Figure 13: Level 0 DFD for CP Upload Portal

- **Level 1 DFD (Detailed Decomposition):** This refined view breaks down the single process from Level 0 into major sub-processes: validation, metadata extraction, sequential pipeline execution (Approvals, Tables, OLE, Flowcharts, Section 1), and logging/archiving. It highlights how data (CP files, metadata, extracted chunks, and token usage logs) flows between each sub-process and the storage systems (ChromaDB, metadata JSONs, and archive folders). This level exposes the internal workings of the upload process, clarifying how the system works with data at a lower level view.

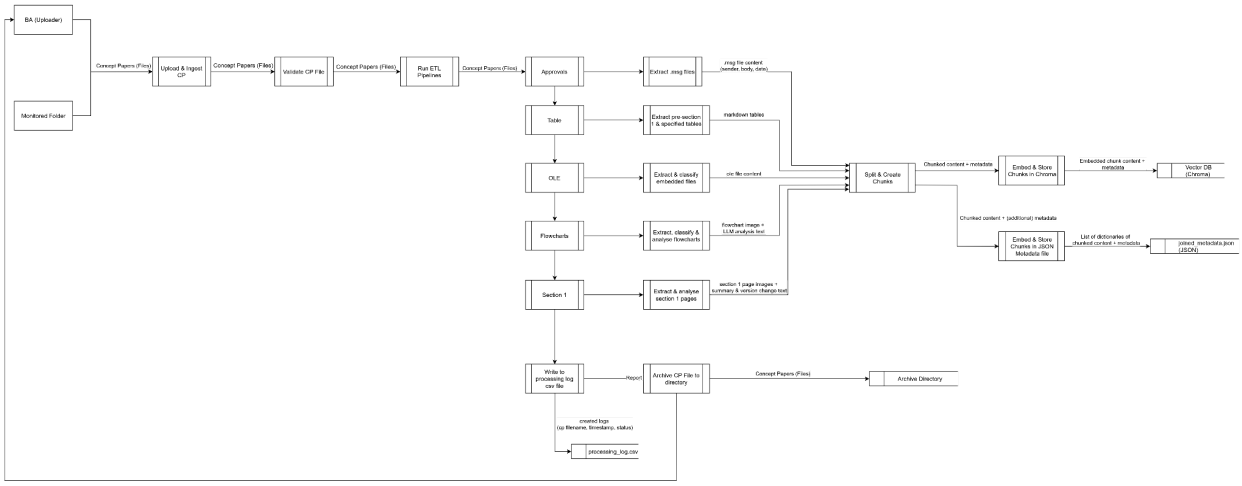


Figure 14: Level 1 DFD for CP Upload Portal

3.2.5.2 RAG CP Chatbot

This section provides an overview of the Data Flow Diagrams (DFDs) for the RAG Chatbot UI, illustrating how user queries are processed through the Retrieval-Augmented Generation (RAG) architecture. Similar to the previous DFDs for the CP upload interface, two levels of abstraction are used to explain the chatbot's data flow and internal workings.

- **Level 0 DFD (Context Diagram):** At the highest level, the diagram presents the chatbot UI as a single process linking the user, the vector database (Chroma), and chat history memory. A query from the user is then transformed into a retrieval request to the Chroma vector database knowledge base and enriched with conversation memory (if available) before producing a contextually-aware response. This diagram highlights the external interactions and boundaries of the system.

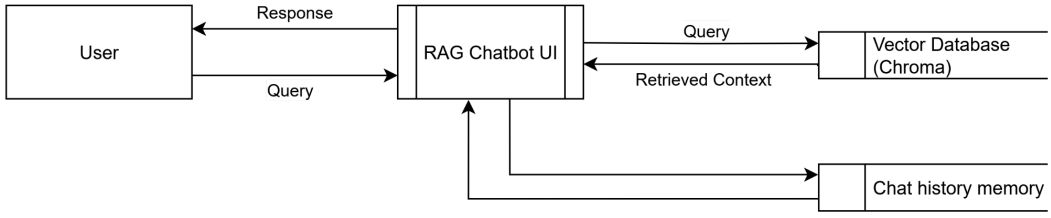


Figure 15: Level 0 DFD - CP RAG Chatbot UI

- **Level 1 DFD (Detailed Decomposition):** The detailed view expands the chatbot into its internal components and sub-processes. User queries are first vectorised by the predefined embedder, then passed to the Retriever to search for relevant documents according to the user's query from the Chroma knowledge base. Afterwards, these documents are passed and are refined by the Reranker, ensuring only the most contextually relevant results are forwarded to the LLM. The Gemini LLM then combines reranked documents with stored chat history memory to generate a context-aware answer, which is sent back to the user through the chatbot UI.

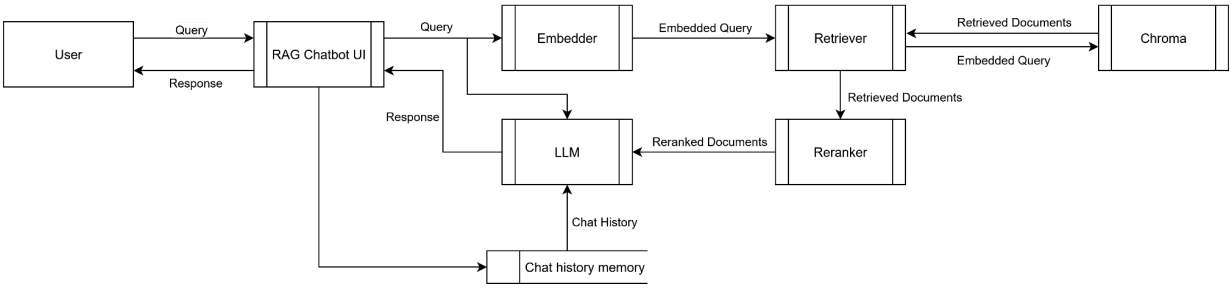


Figure 16: Level 1 DFD - CP RAG Chatbot UI

3.2.6. Sequence Diagram

The sequence diagrams for the CP Upload Portal and the RAG Chatbot UI illustrate the step-by-step interactions between the different components present in the system over time. For the CP Upload Portal, the diagram shows how uploaded Concept Papers are validated, processed through ETL pipelines, and afterwards archived with logs. For the RAG Chatbot UI, the diagram captures how a user query flows through embedding, retrieval, reranking, and generation before returning a context-aware response back to users.

3.2.6.1 CP Upload Portal

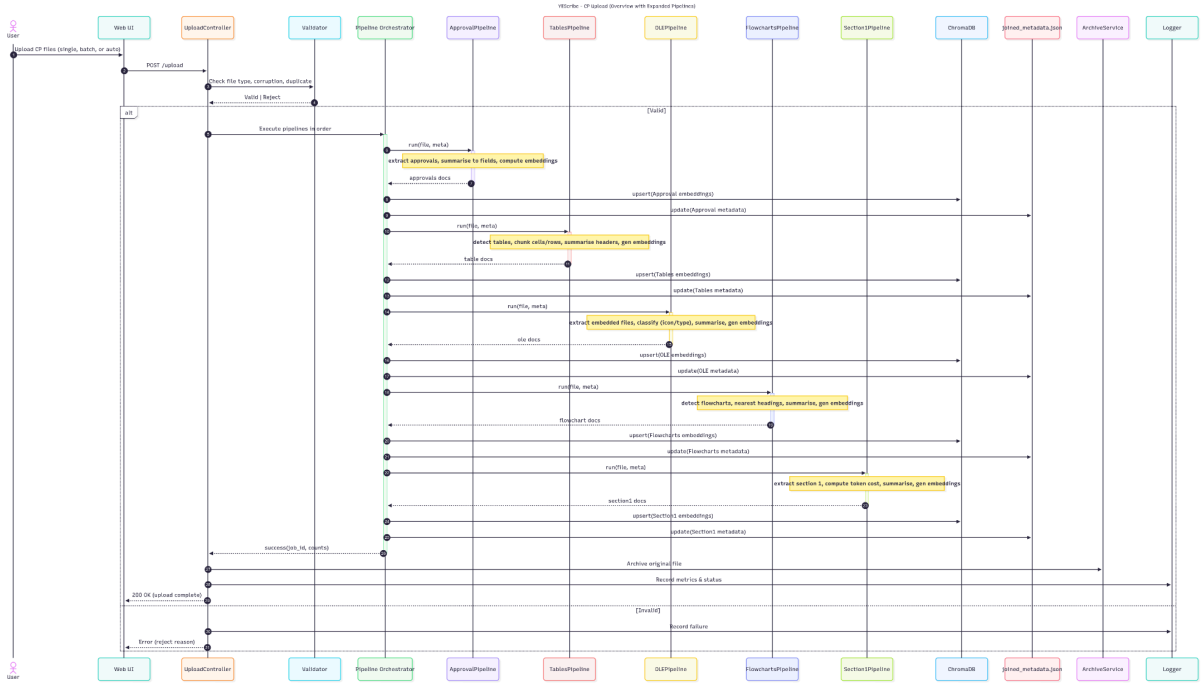


Figure 17: Sequence Diagram - CP Upload Portal UI

3.2.6.2 RAG CP Chatbot

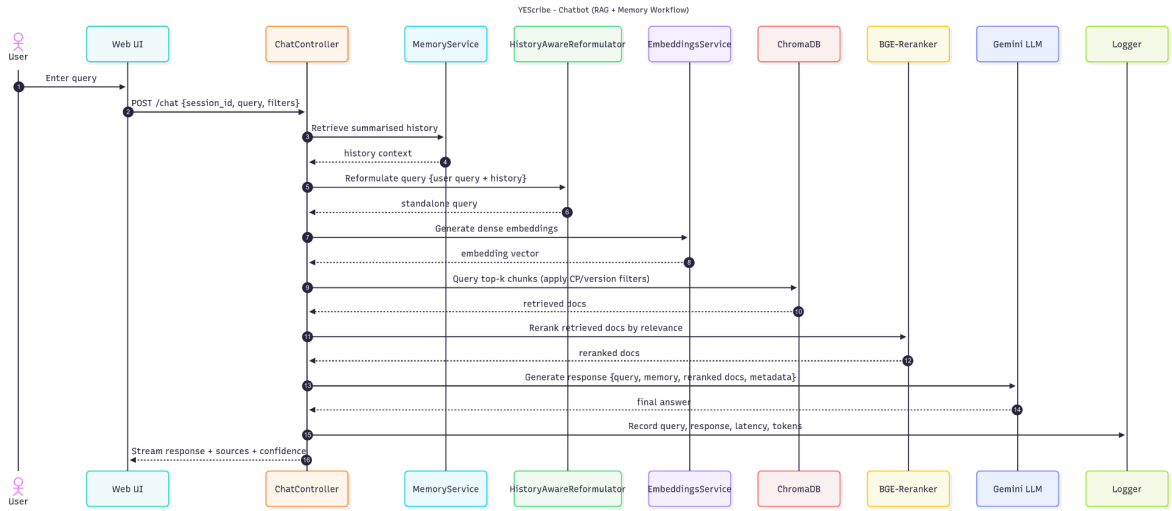


Figure 18: Sequence Diagram - RAG CP Chatbot

3.2.7. Initial Pseudocode/Flowchart

The pseudocode and flowcharts provide an initial starting point for the system's logic and execution flow, outlining the core steps of both the CP Upload Portal (validation, pipeline execution, archiving) and the RAG Chatbot UI (query embedding, retrieval, reranking, generation). It serves as a guide for developers to translate design into code, while also doubling as a way to show how the system processes data from input to output in an abstract and intuitive manner.

3.2.7.1 CP Upload Portal

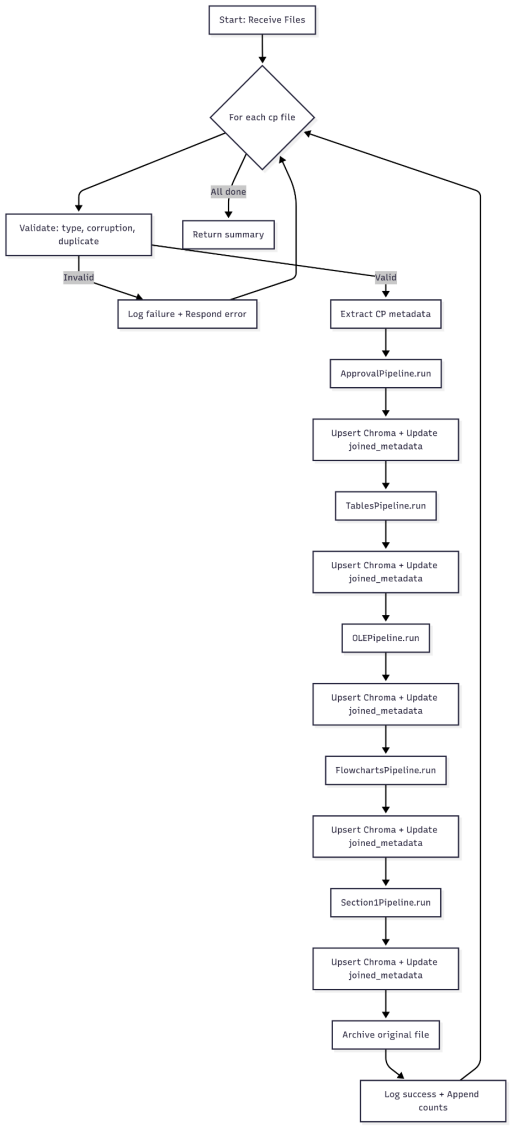


Figure 19: Proposed Flowchart Flow for CP Upload Portal

3.2.7.2 RAG CP Chatbot

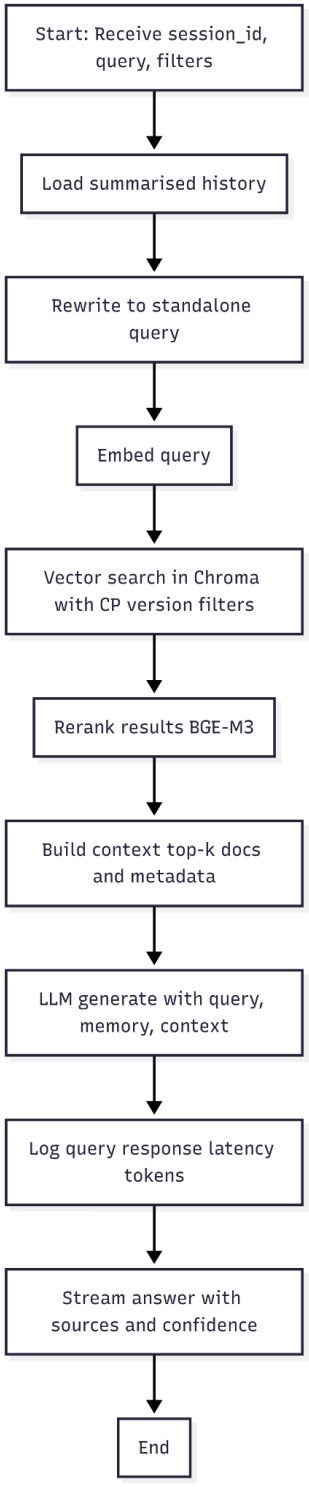


Figure 20: Proposed Flowchart Flow for RAG Chatbot UI

Chapter 4: System Development (Execution)

4.1. Frontend

4.1.1. Prototype Design Into Code

4.1.1.1 CP Upload Portal UI

As noted in the Frontend Design, the CP Upload Portal was built using Streamlit. Streamlit provides a set of ready-made components that are directly mapped to the prototype design, which reduces the need for custom widget development and accelerated implementation. The main page was kept simple and focused on file selection and submission, while the sidebar was used as a dedicated control hub for displaying status, results, upload history, and deletion options. The table below explains the specific components used for the defined use case and design planning in the frontend UI development for the CP Upload Portal:

Table 8: Table of Components Used - CP Upload Portal UI Streamlit

Component	Extended Name	Description
st.toggle	Toggle Switch	Mode Switch (Single vs. Batch). Switches the uploader between accepting a single file or multiple files in the same interface.
st.file_uploader	File Uploader	File Intake for .docx. Accepts one or more Concept Paper files.
st.button	Button	Primary Actions (Submit, Auto Index, Delete, Reset). Provides explicit triggers for user actions such as submitting files, starting auto-indexing, deleting CPs, and resetting the session. Buttons are only enabled when prerequisites are met (e.g., Submit is disabled until a file is chosen).

st.sidebar	Sidebar	Control & Results Hub. Serves as the dedicated region for secondary controls and outputs, including progress, results, history, deletion, and reset.
st.spinner	Spinner	Displays a visual loading indicator during long-running tasks such as uploads and indexing, preventing duplicate clicks and reassures users that the system is active. Once the backend responds, the spinner is replaced by the actual results.
st.metric	Metric Display	Lightweight Performance Telemetry. Shows elapsed time for each operation, giving users quick insight into system performance.
st.expander	Expander	Collapsible Utilities (History, Deletion). Groups secondary features like upload history and deletion into collapsible panels which helps keep the main upload UI free from distractions.
st.markdown / st.write / st.info / st.error / st.success	Text & Feedback Renderers	Feedback Rendering. Used to render upload results and feedback from the backend. Rich text formatting makes summaries of chunks, tokens, embedding models, and errors easier to read and interpret as compared to raw JSON.
st.session_state	Session State	Transient State Across Reruns. Maintains session continuity by storing uploaded files, results, and flags even when the script reruns while preventing loss of data.
st.rerun	Rerun Trigger	Hard Refresh After Destructive Actions. Re-executes the app immediately after reset or delete actions, clearing stale data and ensuring the sidebar and session list reflect the actual backend state.
st.subheader	Subheader	Session List Header. Used in the main pane to clearly label the section showing files uploaded in the current session.

4.1.1.2 RAG CP Chatbot UI

As with the CP Upload UI, the Chat UI is also implemented using **Streamlit**, mapping the previously created prototype directly onto built-in components directly available for use for chat,

filtering, and feedback. The **main page** is reserved for user-chat interactions while the **sidebar** contains session actions, CP/version scoping, approvals info, and retrieval tuning.

Table 9: Table of Streamlit Components Used - Detailed Rationale & Behavior

Component	Extended Name	Explanation
st.chat_message	Chat Message Renderer	Conversation Turns. Renders assistant and user messages (with assistant avatar) in a chat-style format.
st.chat_input	Chat Input Bar	Prompt Entry. Provides a single input bar for user prompts. It does not call the backend directly; instead, it sets state (e.g., pending query and awaiting response) which then triggers the streaming workflow.
st.empty	Empty Container	Streaming Container. Acts as a placeholder for incremental tokens during WebSocket streaming; the placeholder is repeatedly updated to simulate "typing" until the final response is complete.
st.markdown	Markdown Renderer	Displays rich text for messages, headings (e.g., "Sources"), and formatted metadata.
st.sidebar	Sidebar	Controls & Context. Shows session actions (Start New Chat), CP/version selectors, approvals info, and advanced settings. This helps to keep the center focused on the chat interface when hidden.
st.selectbox	Select Box	CP & Version Selection. Allows users to scope retrieval to a specific Concept Paper and version. Changing selection resets chat context to prevent cross-CP leakage.
st.expander	Expander	Collapsible Details. Used for "Approver Details" and "Sources (with Confidence)" to keep secondary information available without cluttering the main view.
st.button	Button	Session Actions. Triggers "Start New Chat," which resets local state and requests a new server session.

st.spinner	Spinner	In-Progress Feedback. Wraps session/network operations (e.g., streaming) to signal activity and prevent duplicate submissions while the model responds.
st.session_state	Session State	Chat State. Persists session_id, chat history, retrieved docs, flags (awaiting/streaming), and selected CP/version. Critical because Streamlit re-executes on interaction.
st.rerun	Rerun Trigger	UI Refresh. Forces a clean re-render after streaming or on new session which ensures the final assistant message and sources display consistently.

4.1.2. Web Portal API Integration

4.1.2.1 CP Upload Portal

Once the Streamlit components were set up, they were integrated with the **FastAPI backend** under the /cp-portal route. Each component either sent structured inputs to the backend (files, identifiers) or displayed structured outputs (JSON summaries, history, status messages).

Table 10: API Integration - CP Upload Porta

Section	Component(s)	API Linkage	Data Flow & Behavior
Main Page	st.toggle + st.file_uploader	N/A	Allows selection of one or multiple .docx files. Stores buffers + metadata in memory/session. No backend call is made here.
	st.button("Submit")	POST /cp-portal/upload (single) or POST /cp-portal/upload-batch (batch)	Sends files previously collected by st.file_uploader as multipart/form-data. Backend responds with JSON (file name, chunks created, token usage, processing time). Sidebar renders results.
	st.subheader("Files Uploaded This Session") + st.session_state	N/A	Displays session-scoped list of uploaded filenames.
Sidebar	Status & Timing (st.spinner + st.metric)	Wraps /upload, /upload-batch, /auto-upload	Displays elapsed time and progress until backend responds.
	Results Panel (st.markdown)	Consumes output from /upload and /upload-batch	Renders embedding model used, number of chunks, token counts, and timings.

	Auto-Index (st.button("Auto Index Folder"))	POST /cp-portal/auto-upload	Backend ingests files from monitored folder and returns per-file summaries.
	History (st.expander(" Upload History"))	GET /cp-portal/history	Retrieves and displays past uploads with CP metadata (number, version, file name, timestamp).
	Deletion (st.expander(" Delete CP Documents") + st.multiselect + st.button)	DELETE /cp-portal/delete	Sends identifiers for single/multiple (cp_number, cp_name, cp_version, file_name). Backend confirms deletion in JSON, marking processing log file status for targeted CP as deleted; UI refreshed via st.rerun().
	Reset (st.button("Reset"))	N/A	Clears st.session_state and re-runs app to reset UI state.

4.1.2.2 RAG Chatbot UI

The Chat UI integrates **REST** for session management and approvals context, and a **WebSocket** for real-time, token-level streaming of answers and confidence metadata. The tables below describe how each UI element interacts with the backend.

Table 11: Session Lifecycle & Settings

UI Area	Component(s)	API Linkage	Data Flow & Behavior
Session create/init	st.session_state, (internal)	POST /chat/session/create, POST /chat/session/init	On first load, create a server session and optionally initialize it. The returned session_id is stored in st.session_state and used on all subsequent calls.
Start new chat	st.sidebar + st.button("Start New Chat")	POST /chat/session/remove (best-effort), then POST /chat/session/create, POST /chat/session/init	Ends the old session (if any), creates a new one, resets local state (chat_history, retrieved_docs, flags), and calls st.rerun() to start fresh.
Top-K tuning	st.sidebar.expander + st.slider("Top-K Reranked Docs")	N/A	Stores the selected value in st.session_state.top_k and includes it in the WebSocket payload for each query.

Table 12: CP Scoping & Approvals Context

UI Area	Component(s)	API Linkage	Data Flow & Behavior
CP picker	st.sidebar.selectbox("Select Concept Paper")	GET /approval/cp-tree	Fetches available CPs and versions. Selecting a CP populates the version selectbox; changing CP resets chat context to avoid mixing contexts.
Version picker	st.sidebar.selectbox("Select Version")	(from cp-tree), GET /approval/status?cp_number=&cp_version=	Selecting a version fetches approvals status markdown and shows it under the expander. Selection updates cp_number/cp_version sent with each chat request.
Approver details	st.sidebar.expander("Approver Details") + st.markdown	Consumes output of /approval/status	Displays the status markdown returned by the backend; expandable to reduce clutter.

Table 13: Chat Surface & Streaming Workflow

UI Area	Component(s)	API Linkage	Data Flow & Behavior
Render history	st.chat_message + st.markdown	N/A	Iterates over st.session_state.chat_history to display prior user/assistant turns with roles and assistant avatar.
Prompt entry	st.chat_input (on submit handler)	N/A (sets state only)	Captures the user's prompt, appends to chat_history, sets pending_user_query, and flips awaiting_response=True to trigger streaming.
Streaming connect	websockets.connect(API_WS_CHAT_URL)	**WS** ws://…/chat/generate	Opens a WebSocket with keepalive settings; sends a JSON payload containing session_id, query, chat_history (excluding the streaming placeholder), and optional cp_number/cp_version.
Stream tokens	st.empty + st.markdown (updated incrementally), st.spinner	**WS** messages: {"token": "…"}	For each token message, append to a growing string and render with a trailing cursor (e.g., ▌) to simulate typing.
Completion & sources	(same container), st.expander("Sources …")	**WS** message: {"done": true, "confidence_scores": [...]}	On done, finalize the message in chat_history, store confidence_scores into st.session_state.retrieved_docs, clear flags (awaiting_response=False, is_streaming=False), and st.rerun().
Show sources	Center pane st.markdown("Sources…") + st.expander	Consumes st.session_state.retrieved_docs	After streaming finishes, lists retrieved document metadata (file, CP, version, confidence, type) for transparency and verification.

4.2. Backend

4.2.1. Database Setup and Implementation

As previously mentioned, the project uses **ChromaDB** as the vector database to persist embeddings of chunked Concept Paper (CP) content. The Chroma instance is managed via LangChain's Chroma wrapper, which can be linked to LangSmith to trace document insertion, retrieval, and filtering, thus allowing developers to audit queries, verify retrieved chunks, and track embedding persistence at a fine-grained level. This adds an extra layer of **observability and auditability** during development and testing.

4.2.1.1 Local Persistence Setup

ChromaDB is configured in local mode, with the storage directory path defined through environmental variables (CHROMA_DIR=api/data/chroma). At runtime, this directory is passed to the persist_directory parameter when initializing the Chroma instance. This allows embeddings and their metadata to be written directly to disk, which guarantees persistence across sessions without requiring a managed database connection from external servers or infrastructure.

4.2.1.2 Document Insertion

During ETL pipeline execution, extracted chunks are wrapped into **LangChain Document objects** (LCDocument). Each document is assigned rich metadata which provides the retriever with more context during retrieval. Fields include: CP identifiers, source filenames, nearest section headings, and in the case of flowcharts (process flows), image_paths.

This metadata ensures that documents can be retrieved, reranked, and filtered later by CP number and version, which is selected when users narrow down on their query results to the CPs that they would like to know more about.

4.2.1.3 Dedicated Vectorstore Client

When calling the vector database from the **chatbot UI**, the project uses a dedicated vectorstore/chroma_client.py file which aims at separating the initialisation of the Chroma vector

database setup from the rest of the backend logic instead of having to call it in each module/component defined in the RAG chatbot architecture. This makes it such that whenever the chatbot UI needs to perform retrieval, it calls get_chroma_client() to return a persisted Chroma instance, consistently pointing to the same storage directory defined in .env.

4.2.1.3 Metadata JSONs for Auditability

Besides storing the data in Chroma, the system writes into 2 separate JSON files found in METADATA_DIR, with each having its own purpose in the following:

- **joined_metadata.json:** Complete record of all pipeline outputs (approvals, flowcharts, tables, OLEs, Section 1) that is nearly identical to the data indexed to the Chroma vector database, except that it also includes other metadata fields, such as token count and timestamps of when it was inserted into the vector database, fields that only pollute the retrieval but also proves crucial and can later be leveraged for auditing, regression testing, or data quality validation.
- **approval_metadata.json**: Complete approval records for all CPs and versions that are actively being used by the frontend chatbot sidebar, where users can quickly view the approval tree and per-approver status of a CP. This metadata is extracted by the Approval Pipeline, stored as JSON, and also aggregated and later indexed into Chroma for retrieval to allow users to directly query the approval status of the CP.

4.2.2. Web Portal Implementation

4.2.2.1 CP Upload Portal

4.2.2.1.1 CP Upload and Validation

The CP Upload Portal forms the first stage of the backend ingestion pipeline. Its purpose is to accept Concept Paper (CP) files through 2 main entry points (direct file upload or scanned directory), verify their validity, and prepare them for downstream processing. The backend is implemented in **FastAPI** with defined routes in cp_portal_routers.py, which forward incoming requests from the portal frontend to functions stored in cp_portal_service.py for validation and orchestration.

4.2.2.1.1.1 Upload Modes

The portal supports three different upload methods:

- **Single Upload (POST /cp-portal/upload)**: Handles one .docx file at a time.
- **Batch Upload (POST /cp-portal/upload-batch)**: Accepts multiple CP documents in a single request.
- **Auto Upload (POST /cp-portal/auto-upload)**: Monitors a pre-configured directory (via AUTO_UPLOAD_DIR) for new files and triggers ingestion automatically.

Each of these endpoints packages the uploaded file(s) into a request context containing the file path, metadata, and user input parameters before triggering the validation step.

4.2.2.1.1.2 Validation Workflow

Validation is a **critical safeguard** in the system, ensuring that only well-formed and non-duplicate CPs are admitted into the pipelines. The workflow is defined in cp_portal_service.py and consists of the following technical checks:

1. **File Type Verification**
    - The system enforces .docx MIME type validation.
    - Any other extensions (e.g., .pdf, .txt) are rejected immediately with an HTTP error response.
2. **Corruption and Integrity Check**
    - The .docx file, which internally follows a ZIP structure, is opened and inspected using low-level file I/O.
    - If the archive cannot be parsed (indicating corruption or incomplete upload), the system aborts processing.
3. **Temporary / Ephemeral File Rejection**
    - Temporary files created by Microsoft Word or OneDrive sync clients often begin with reserved prefixes (e.g., ~$) or extensions (.tmp).
    - These are automatically filtered and flagged as invalid inputs.
4. **Duplicate Prevention**
    - The system maintains a **ledger (processing_log.csv)** inside the archival directory.
    - For every new upload, a hash and metadata comparison (CP number, version, file name) is performed against the ledger.
    - If the file has already been processed, the upload request is rejected with a duplicate status.
5. **Metadata Extraction**
    - A utility function extract_cp_metadata_from_filename() parses the CP **name**, **number**, and **version** from the uploaded filename.
    - This extracted metadata is embedded into the request context so that subsequent pipelines can tag embeddings and metadata consistently.
    - Example: 522_CP_v0.5.docx → cp_number=522, cp_version=v0.5, cp_name="CP 522 LTE Migration".

4.2.2.1.1.3 Error Handling and Response Structure

If a file fails validation, the API immediately returns a structured JSON response, e.g.:

Figure 21: Sample JSON response for failed uploads

On success, the validation layer passes the file to the ETL orchestrator (cp_etl_service.py) for further processing. The orchestrator then handles pipeline execution, logging, and archiving.

4.2.2.1.2 ETL Orchestration

Once a CP document passes validation, it is then passed to the **ETL orchestration service (cp_etl_service.py)**, which coordinates the sequential execution of 5 dedicated document ingestion pipelines. The orchestration ensures that every CP version is processed consistently, with results embedded into the vector database and metadata archives afterwards.

The run_pipelines_and_archive() function in cp_etl_service.py performs five major responsibilities:

1. **Pipeline Invocation**: Executes each pipeline in a predefined order.
2. **Chunking and Embedding**: Splits extracted content into semantic chunks and embeds them using the shared embedding model.
3. **Metadata Recording**: Appends extraction details (chunk IDs, CP metadata, token counts, section headings) into joined_metadata.json and other metadata files.
4. **Logging and Metrics**: Collects execution time, token usage, and chunk counts for audit and monitoring, then shows the status and results of the upload to the frontend UI
5. **Archiving**: Moves the processed CP into a dated archive folder, while writing a new entry into processing_log.csv to log the processed CP file

4.2.2.1.2.1 Pipeline Execution Flow

The pipelines are invoked sequentially to preserve data dependencies:

1. **Approval Pipeline (run_approval_pipeline())**
    - **Process:** Extracts approval tables and parses embedded .msg files for approval evidence.
    - **Outputs:** Approval metadata (approval_metadata.json) & approval chunks in Chroma.
2. **Tables Pipeline (run_table_pipeline())**
    - **Process:** Converts document into Markdown, captures specific pre-Section 1 tables, including **Title Table** and **CP Change History**.
    - **Outputs:** Chroma vectors tagged as type=table.
3. **OLE Pipeline (run_ole_pipeline())**
    - **Process:** Extracts embedded files (Excel, PPT, PDF) and their icons, classifies icons using Gemini, maps nearest headings, renames extracted files based on icon names generated using Gemini, converts content, then chunks and embeds.
    - **Outputs:** OLE document embeddings & enriched metadata (file name, heading2/heading3, page reference).
4. **Flowcharts Pipeline (run_flowchart_pipeline())**
    - **Process:** Identifies embedded diagrams/images, validates whether they represent process flow, then summarises them via Gemini invokes.
    - **Output:** Produces predefined, structured summaries, attaches image paths and identified as metadata, and embeds them into Chroma.
5. **Section 1 Pipeline (run_section_1_pipeline())**
    - **Process:** Converts the CP's front matter to PDF, extracts Section 1 pages, generates a Gemini summary, and compares with the previous CP version to index change logs if available.
    - **Outputs:** Summaries and version changes chunks embedded in Chroma.

4.2.2.1.2.2 Technical Characteristics of Orchestration

- **Deterministic Execution Order:** Pipelines are executed in a fixed order (Approvals → Tables → OLE → Flowcharts → Section 1), guaranteeing consistency across different CP versions while preventing missing dependencies.

- **Chunk and Metadata Enrichment**

  Each pipeline returns LangChain Document objects with rich metadata, including:

Figure 22: Sample Langchain Document JSON

- **Performance Metrics:** Token counts, execution times, and chunk totals are collected per pipeline using utility functions like summarize_token_usage(). These metrics are stored in both the sidebar feedback for users and the backend ledger for traceability.

- **Error Tolerance:** Each pipeline is wrapped with exception handling so that a failure in one stage (e.g., missing OLE files) does not prevent the subsequent pipelines from not being executed.

4.2.2.1.3 ETL Pipelines

The ETL orchestration consists of five pipelines, each targeting a distinct type of content from Concept Papers (CPs). These pipelines run sequentially after validation and ensure that all structured elements are extracted, semantically chunked, embedded into the vector database, and archived with full metadata.

4.2.2.1.3.1 Approval Pipeline

The Approval Pipeline extracts business approval metadata and supporting evidence embedded within a CP document. It is essential for tracking whether a CP version has been formally approved by the specified business approvers in the CP.

Flow of Approval Pipeline:

1. Convert .docx CP into Markdown for easier parsing.
2. Locate the *"CP Business Approver"* table.
3. Extract and parse embedded .msg email files from the CP.
4. Apply an LLM classifier to detect approval decisions from emails.
5. Aggregate results into overall CP approval status and progress.
6. Write metadata to approval_metadata.json and embed chunks into ChromaDB.

Output: Approval metadata and approval status embeddings stored in Chroma and approval_metadata.json.

4.2.2.1.3.2 Tables Pipeline

The Tables Pipeline targets key pre-Section 1 tables that contain critical metadata, primarily the Title Table and CP Change History tables

Flow of Tables Pipeline:

1. Convert CP to Markdown.
2. Clean and normalise table structures.
3. Identify the Title Table and CP Change History Table.
4. Transform table rows into retrievable chunks.
5. Embed chunks into Chroma with enriched metadata (CP number, version, table index, section heading).

Output: Chroma vectors representing tables, enabling queries like *"What changes were introduced in CP 601 v1.1?"*.

4.2.2.1.3.3 OLE Pipeline

The OLE (Object Linking and Embedding) Pipeline extracts and processes embedded files such as Excel, PDF, and PowerPoint documents. These documents often contain crucial data separate from the CP body content.

Flow of OLE Pipeline:

1. Detects embedded OLE objects and associated icons.
2. Filter out irrelevant icons (temporary or oversized).
3. Use Gemini to classify, match, and rename icons.
4. Convert embedded documents into text using tools such as PDFPlumber or LibreOffice.
5. Apply semantic chunking to document content.
6. Embed chunks into Chroma with metadata linking them to their nearest Heading 2/3.

Output: Embedded annexes stored as retrievable chunks, enabling queries like *"Show me the Excel data embedded in CP 520"*.

4.2.2.1.3.4 Flowcharts Pipeline

The Flowcharts Pipeline focuses on procedural knowledge represented in diagrams. Converting flowcharts into structured text makes them searchable and retrievable.

Flow of Flowcharts Pipeline:

1. Detects all embedded images in the CP.
2. Apply LLM-based classification to confirm whether an image is a flowchart.
3. Summarise confirmed flowcharts with a structured Gemini prompt, capturing decisions, roles, and outcomes.
4. Split summaries into semantic chunks.
5. Insert chunks into Chroma with metadata including CP details, headings and image path.

Output: Flowchart embeddings with multimodal references, enabling queries like *"Explain the process flow in CP 530"*.

4.2.2.1.3.5 Section 1 Pipeline

The Section 1 Pipeline extracts and summarises the first section of CPs, which details the main content of a CP, such as purpose, scope, and key parameters.

Flow of Section 1 Pipeline:

1. Convert .docx to PDF and locate Section 1 pages.
2. Rasterise the pages into images for parsing.
3. Generate summaries using Gemini's content-understanding model by passing rasterised pages as input.
4. Chunk and embed the summaries into Chroma.
5. Store plaintext summaries locally for audit purposes.
6. Compare with previous CP versions to generate change logs (if available).
7. Append metadata and change history to joined_metadata.json.

Output: Section summaries and change logs embedded in Chroma, enabling queries such as *"What is the scope of CP 522?"* or *"What changed between CP 522 v0.5 and v0.4?"*.

4.2.2.1.4 File Archiving

After a Concept Paper (CP) has been processed through the ETL pipelines, the system performs an archiving operation to ensure that the file and its metadata are safely stored for future reference. This process involves both the relocation of the original file and the updating of a predefined archival log.

1. **Archiving the Processed File**
    - The processed .docx file is moved from the input directory (INPUT_DIR) into a dated folder under the archive directory defined in the environment file (.env) (ARCHIVE_DIR), preventing reuploads.
    - Example: api/data/archived_files/2025-08-14/522_CP_v0.5.docx
2. **Updating the Archive Log**
    - A central CSV file, processing_log.csv, located in the same archive directory (api/data/archived_files/processing_log.csv), is updated with a new row for each processed CP.
    - This row includes metadata such as CP number, CP name, version, file name, processing status, number of chunks created, token usage, and processing time.
    - Once added, the CP is marked as **processed**, preventing CPs of the same name and version from being re-indexed in the user interface.
3. **Integration with API Endpoints**
    - The archive log (processing_log.csv) serves as the backing data source for two FastAPI endpoints implemented in cp_portal_routers.py:
        - **History Endpoint (GET /cp-portal/history)**: Retrieves records from processing_log.csv, then returns a sorted list of all processed CPs with their identifiers and timestamps.
        - **Deletion Endpoint (DELETE /cp-portal/delete)**: Takes in parameters such as cp_number and cp_version, removes the corresponding embeddings from ChromaDB based on the provided parameters, deletes processed outputs from OUTPUT_DIR, and updates the relevant row with the specified CP in processing_log.csv to mark its status as deleted.

4. **Status Tracking**
    - By introducing the status field in processing_log.csv (e.g. *success*, *deleted*), the system ensures that users are not allowed to accidentally re-index the same CP version.

4.2.2.1.5 History and Deletion Functions

The CP Upload Portal provides two crucial backend functions in supporting the deletion and viewing the history of previously uploaded CPs. Both functions are implemented as REST API endpoints in cp_portal_routers.py and operate on the archival log file processing_log.csv.

4.2.2.1.5.1 Upload History Function

The History function allows users to retrieve a complete record of all previously uploaded and processed CPs.

- **Endpoint:** GET /cp-portal/history
- **Handler Function:** get_cp_history() in cp_portal_service.py
- **Data Source:** processing_log.csv located in ARCHIVE_DIR
- **Process:**
    1. Reads all rows from processing_log.csv.
    2. Parses metadata such as CP number, version, file name, status, and processing timestamp.
    3. Sorts results in descending order based on the processing date.
    4. Returns the structured history to the frontend for display in the sidebar section.

Example Response:

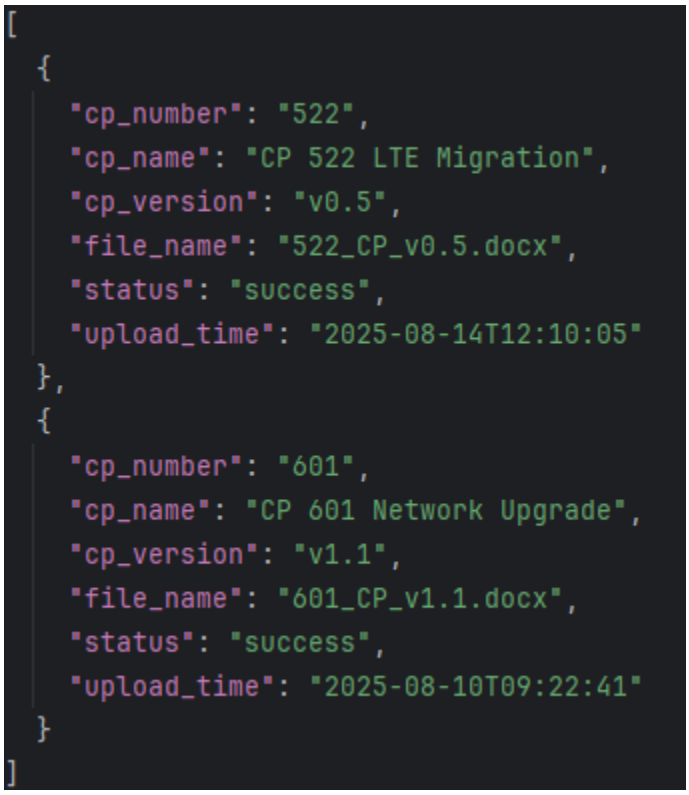


Figure 23: Sample Response for Uploaded CP Status (History)

4.2.2.1.5.2 CP Deletion Function

The Deletion function provides the ability to remove the associated chunks to the CP for targeted deletion from the Chroma vector database while maintaining a permanent record in the archive log.

- **Endpoint:** DELETE /cp-portal/delete
- **Handler Function:** delete_cp_by_metadata() in cp_portal_service.py
- **Process:**
    1. Accepts input parameters cp_number and cp_version.
    2. Deletes associated embeddings from ChromaDB using the chroma_client.delete() function.

3. Removes processed outputs from the local OUTPUT_DIR.

4. Updates the corresponding row in processing_log.csv by marking the status as "deleted".

Example Request:

DELETE /cp-portal/delete?cp_number=522&cp_version=v0.5

Example Response:

Figure 24: Sample Response after CP Deletion

4.2.2.2 RAG Chatbot UI

4.2.2.2.1 Session and Memory Management

In the YEScribe RAG chatbot, in order to enable multiple users to interact with the system simultaneously while maintaining conversational continuity, a session mechanism is required. Each session ties a conversation to one unique chat instance, allowing the chatbot to retain memory across multiple turns and ensuring that follow-up questions build on prior exchanges. This approach also enforces isolation between users so that queries and results remain bound to the correct session, preventing overlap or leakage across different chats.

4.2.2.2.1.1 Session Fundamentals

A session represents the lifecycle of a conversation between the user and the chatbot. Each session is uniquely identified by a **universally unique identifier (UUID)** generated by the backend during session creation. This identifier acts as the binding reference across all subsequent requests, ensuring that each user's conversation context is consistently maintained.

Sessions are necessary because, without them, every query would be treated as a standalone request, leading to a loss of conversational flow. By assigning a persistent UUID and linking it with a memory container, the system ensures that continuity is preserved until the session is reset or removed.

4.2.2.2.1.2 Session Initialisation

The initialisation process establishes the session link between the frontend and backend and provisions memory containers for conversational context. When the chatbot interface is first loaded, the frontend sends a request to the backend via POST /chat/session/create. Then, the backend generates a new UUID using Python's uuid library. The UUID is returned to the frontend and stored in st.session_state, where Streamlit will further handle the session afterwards in the frontend.

After creation, the frontend calls POST /chat/session/init with the session_id, which initialises a new collection to store the dialogue of interactions between the user and chatbot. It also

initialises a new ConversationalSummaryBufferMemory object tied to the session id passed in the API call, summarising older interactions in order to keep input token costs low as well as reducing the risk of going over the LLM context window. Then, the backend verifies or reinitialises the memory container for that session, ensuring it is ready to maintain summarised conversational state. At present, the backend memory only supports recent turns and a rolling summary.

The **full verbatim chat history** is handled exclusively on the **frontend** (st.session_state.chat_history). This design is intentional: chat history is treated as ephemeral, cleared on refresh or when starting a new chat. Persisting conversations would require a user account system and a backend database to securely link chat sessions to authenticated users. Due to additional complexity and time constraints, this was not implemented in the current project but may be considered for future enhancements.

4.2.2.2.1.3 Session Deletion

The chatbot also supports deletion of sessions to allow users to reset or start a new conversation. This is handled through the POST /chat/session/remove endpoint. When invoked, the backend clears the memory container and summarised conversation memory instance associated with the given session ID using the remove_session_memory(session_id) function.

On the frontend, this process is triggered when the user selects **"Start New Chat"** in the interface. The existing session is removed, a new session is created through the same /chat/session/create request, and the initialisation steps are repeated. This ensures that the user always begins with a clean state when starting a fresh conversation.

4.2.2.2.2 Model Warmup

To reduce the latency of the components of the RAG architecture, such as Large Language Models (LLMs), embedders, and rerankers, the YEScribe chatbot implements a **model warmup routine** combined with a **caching strategy**. Without these optimisations, the initial request after server startup would incur significant delays as the models are loaded into memory and executed for the first time, reducing the overall responsiveness of the application.

Warmup is executed as part of the FastAPI application lifespan. When the server starts, a background task (_do_warmup) is triggered to preload the retrieval and generation pipeline. This is achieved by performing a simple "dry run" query under a dummy session (session_id="__warmup__").

The warmup process ensures that the embedding model, retriever, reranker, and the Chroma knowledge base are initialised through get_embedding_vectorstore_retriever() before actual calls are made.In addition to warmup, the system employs **caching mechanisms** to sustain low latency during operation:

- **Embedder caching**: Frequently requested embeddings are cached to avoid recomputation, reducing overhead when similar or repeated queries occur.
- **Reranker caching**: Candidate document scores are cached for short intervals, preventing redundant rescoring when the same query-document pairs are evaluated.
- **Prompt caching for the LLM**: Repeated queries or summarisation prompts, are cached at the backend. This enables the model to return results instantly without recomputing full generations.

After completion of the warmup, a flag (app.state.warmed_up = True) is set so that subsequent requests bypass cold-start behaviour. If warmup fails, the system remains operational, but the first user query may still incur a one-time delay as components initialise on demand.

4.2.2.2.3 Showing CP Approvals

The backend exposes approval information for Concept Papers (CPs) by reading from a structured JSON file (approval_metadata.json) that contains the list of approvers and their approval statuses for each CP and version. This file is produced during earlier processing and serves as the sole record for approval metadata.

4.2.2.2.3.1 Backend Mechanism

Managed by the /approval/status endpoint, when an approval request is received after selecting the specific CP and version, the backend locates the corresponding entry in the approval metadata JSON. Each entry is indexed by CP number and version, and contains approver names, roles, and their respective decision outcomes. This method allows the system to serve approval data directly without reprocessing the source documents.

4.2.2.2.3.2 API Endpoints

Approval data is exposed to the client through two endpoints under the /approval namespace:

- **GET /approval/cp-tree**: Returns a JSON structure listing all CPs and their available versions, annotated with their overall approval status (e.g., *Approved*, *In Review*).
- **GET /approval/status**: After a CP and version are selected, return the detailed approval information for that entry, queried from the returned JSON file. The response is formatted in markdown, providing the approver list and their decision statuses in a structured table.

4.2.2.2.3.3 Data Returned

Both endpoints read directly from the JSON file to provide the specified approval data. The CP tree endpoint first returns a hierarchical JSON object with CP numbers, versions, and their status labels then the approval status endpoint provides a markdown string containing the approver list and their decisions for the selected CP and version based on the CP tree endpoint called.

4.2.2.2.4 CP Filtering

The YEScribe backend supports filtering of Concept Papers (CPs) during retrieval operations to allow queries to be restricted to a specific CP and version. This allows users to scope chatbot interactions to a single document rather than searching across the entire repository, improving contextual accuracy while reducing latency.

4.2.2.2.4.1 Backend Mechanism

When a filter request is applied, the backend attaches the specified cp_number and cp_version to the retrieval process. These parameters are included as metadata filters when querying the vector database, ensuring that only chunks belonging to the selected CP and version are returned.

4.2.2.2.4.2 API Integration

The CP filter is passed as part of the chat request payload to the backend. For example:

Figure 25: Sample request made in frontend chatbot by user query

On request receive, the backend validates the CP parameters and applies them as constraints during vectorstore retrieval. If no CP filter is provided, the system defaults to performing a global search across the entire Chroma knowledge base.

4.2.2.2.4.3 Data Returned

The retrieval results returned by the backend include only the chunks that match the CP filter applied. Metadata for each chunk (including CP number and version) is also included in the response to provide transparency and enable source citation.

4.2.2.2.5 RAG QA Chain

The Retrieval-Augmented Generation (RAG) QA chain is the central pipeline responsible for answering user queries in the YEScribe chatbot. It integrates several key GenAI techniques, such as retrieval, reranking, multimodal context handling, prompting, and conversational memory into a single orchestration.

The chain is implemented in the backend service chain_service.py through the function **get_chain()**, which builds the multimodal RAG pipeline per session request.

4.2.2.2.5.1 Historic Retrievers and Rerankers

The chain begins by wrapping the **base retriever** (Chroma vectorstore retriever) with a **reranking layer**. The reranker is initialised through get_reranker() and applied using the RerankingRetriever class.

- The retriever retrieves the top-k semantically similar Concept Paper (CP) chunks, with optional filters for **CP number** and **version**.
- The reranker then applies a cross-encoder model to score each retrieved chunk relative to the user's query, ensuring that the most relevant evidence is prioritised.
- Retrieved documents (docs) are returned with associated metadata, including any **image paths**, which can be resolved for multimodal context injection into the LLM.

This design balances **recall** (broad retrieval coverage) with **precision** (fine-grained reranking of the most useful chunks) for optimising contextual accuracy.

4.2.2.2.5.2 Large Language Model (LLM)

The **Large Language Model (LLM)** is loaded via get_llm() and forms the reasoning and generative engine of the entire chain. Once chunked documents from the Chroma knowledge base have been retrieved and reranked, the LLM synthesises this context with the user's query to generate coherent, factually grounded responses.

4.2.2.2.5.3 Prompt Template

The chain standardises inputs to the LLM using two structured prompt templates:

- **History retriever prompt** (history_retriever_prompt): reformulates follow-up questions into standalone queries using create_history_aware_retriever().
- **QA prompt** (qa_prompt), which combines:
    1. The **user query** (standalone or reformulated),
    2. The **summarised memory** of prior interactions, and
    3. The **retrieved and reranked context**
    4. **Prompt template** (Custom prompt-engineered prompt template for displaying contextual, structured results tailored to users based on gathered feedback)

The QA prompt explicitly instructs the model to generate structured, concise answers tailored to teams at YTL Communications.

4.2.2.2.5.4 Conversational Memory

As previously described in the **Session section of the report**, conversational memory is closely tied to session management, ensuring that context persists only for the lifetime of a session.

Memory is managed through the get_session_memory(session_id) function in memory_service.py, which relies on LangChain's **ConversationSummaryBufferMemory**. It maintains:

- **Summarised history**, which are essentially older turns that are condensed into concise summaries, preserving continuity while keeping token usage low.
- **Recent verbatim turns**, which are just a buffer of the latest exchanges, providing immediate grounding and full context for follow-up queries.

This dual-layer memory design is essential for **reducing the token cost per response** and for preventing the model from reaching its **maximum context window**. By combining summarised long-term history with short-term verbatim context, YEScribe achieves efficient and scalable multi-turn interactions without compromising accuracy.

4.2.2.2.5.5 Multimodal Context

If retrieved document metadata contains linkage to images (e.g., flowcharts indexed with image_path), the chain attempts to resolve and load these assets via resolve_image_path_from_metadata(). Successfully loaded images are passed to the LLM as part of multimodal context.

4.2.2.2.6 LLM Response Streaming

To improve responsiveness and avoid making users wait for the entire answer to be generated, YEScribe streams responses incrementally. This is achieved through **WebSockets**, which enable real-time data streaming between the backend and frontend. By streaming **LLM responses token by token**, the user can begin reading the answer immediately as it is generated, instead of waiting for the complete response. The following section below explains the underlying logic of how streaming of responses is implemented and connected to the frontend (from backend):

1. **Connection & keep-alive.** The server accepts and opens the WebSocket, and starts a heartbeat task that periodically sends { "type": "keepalive" } to prevent idle disconnections during long responses.
2. **Receive payload.** The client submits the query, (optional) CP/version filters, and chat history in JSON format when the WebSocket is first opened.
3. **Chain setup.** The backend constructs the RAG QA chain via get_chain() (Explanation provided in previous section)
4. **Run QA chain.** The chain is executed against the user query. As the LLM produces output, the server streams it back token by token in JSON frames, such as { "token": "partial_text" }, which are rendered and handled by frontend logic of the chatbot.
5. **Completion.** Once generation is complete, the backend sends one final frame containing the full response and the supporting document metadata with confidence scores:

On the client side, Streamlit opens the WebSocket, sends the query payload, ignores heartbeat messages, renders tokens in real time as they arrive, and finally appends the retrieved document metadata under a “Sources” panel once the "done": true frame is received then only closing the WebSocket connection afterwards.

4.2.2.2.7 Retrieved Sources

To ensure transparency and auditability, YEScribe displays the **retrieved sources** that were used to ground each response. These are generated during the retrieval and reranking phase of the QA chain and are only shown after the answer has been fully streamed and finalised. Display for this is handled by the frontend, under the "Show Retrieved Documents" accordion. The following section details the flow for the creation of the retrieved documents JSON data sent and handled by the frontend for this feature.

1. **Retrieval & reranking.** Candidate chunks are retrieved from ChromaDB and rescored by the cross-encoder reranker (RerankingRetriever). The top-k results are selected for inclusion.
2. **Metadata extraction.** Each selected chunk carries metadata such as:
    - cp_number and cp_version (to identify the Concept Paper version)
    - file or source_file (the original filename or embedded object reference)
    - type (e.g., table, flowchart, approval, section summary, OLE object, or plain text)
    - Optional **headings** (heading_2, heading_3) if extracted during ETL
    - **Confidence score** normalised from the reranker's output (scale of 0-100)
3. **Final WebSocket frame.** This metadata is compiled into a confidence_scores list and sent to the frontend alongside the final response

Figure 26: Sample JSON data for retrieved documents/chunks

4.2.3. Third-Party API Integrations (Gemini & LangSmith)

The YEScribe system integrates two external services, **Google Gemini API** and **LangSmith**, to enable advanced language model processing and runtime tracing. Both are connected directly into the FastAPI backend through defined environment variables then applied in wrappers across several service files.

4.2.3.1 Gemini API

The Gemini API is the core LLM provider for YEScribe and is called through **LangChain service wrappers** in the backend:

- **Chatbot QA Chain:** The main chatbot uses get_llm() in llm_service.py to load **Gemini 2.5 Flash** for low-latency generation. This model streams answers token-by-token via WebSockets, guided by prompt templates and enriched with retrieved context in the QA chain.
- **Flowchart Pipeline:** The Flowchart ETL pipeline (flowcharts.py) uses **Gemini 2.5 Pro** for higher-accuracy summarisation. Images of embedded flowcharts are passed to Gemini with a structured flowchart_prompt, and the model outputs natural-language summaries that are chunked and embedded into Chroma.
- **Section 1 Pipeline:** The Section 1 summarisation pipeline (section_1_tables.py) also calls **Gemini 2.5 Pro**, analysing previously rasterised pages of Section 1 in the CP then generating concise summaries. These are embedded for retrieval and compared against previous versions, if available to record change histories.
- **Approval Pipeline:** Within the **Approval ETL pipeline**, Gemini **2.0 Flash** is used to classify extracted .msg emails, identifying whether each represents an approval or rejection, and justifying the decision in structured metadata.
- **OLE Pipeline:** The **OLE pipeline** (ole.py) calls Gemini **2.0 Flash** to classify and rename extracted icons from embedded files, ensuring accurate metadata labelling.
- **Flowchart Detection (Pre-Summarisation):** Before full summarisation, Gemini **2.0 Flash** is also invoked to quickly detect whether an extracted image is indeed a flowchart, preventing unnecessary calls to the Pro model.

4.2.3.2 LangSmith

LangSmith is integrated for **tracing and observability** of all LangChain-driven components in YEScribe.

- **Configuration**

  In .env, the following variables are defined once:

```
LANGCHAIN_TRACING_V2=true
LANGCHAIN_API_KEY=<your_key>
LANGCHAIN_PROJECT=yescribe-rag-chatbot-traces
```

Figure 27: .env setup for LangSmith

- **Captured Data:** Every invocation of a LangChain component (QA chain, retriever, reranker, memory summarisation) is logged to LangSmith, including:
  - Latency of each step (retriever, reranker, prompt, LLM).
  - Token usage (prompt, completion, total).
  - Chain structure (sub-components, dependencies, runtime order).
  - Inputs and outputs of each step, subject to LangSmith workspace redaction settings.
- **Scope in YEScribe**
  - **Chatbot QA Chain**: Each user query generates a trace showing retriever, reranker, memory, and LLM components
  - **Conversation Memory**: Summarisation updates from ConversationSummaryBufferMemory are logged as independent runs.

Chapter 5: Deployment

5.1 Deployment Environment

As previously proposed, the deployment for YEScribe is fully containerised using **Docker**. This ensures that the application remains reproducible and consistent across different operating systems. Development was primarily conducted on **Windows**, while the production environment is **Linux-based**. Thus, Docker was chosen to eliminate compatibility issues between heterogeneous environments and in guaranteeing environment parity.

The deployment environment consists of a **Python 3.13.3-slim** base image, with additional **system dependencies** and **fonts** installed to support the ETL pipelines. These include:

Table 12: List of System Dependencies in order to run YEScribe

Category	Dependency	Purpose in YEScribe	Usage
Build Tools	build-essential, curl, unzip	Compiling Python packages, handling archives	General setup
Document Conversion	libreoffice	Headless DOCX → PDF conversion	Section 1 pipeline
PDF Tooling	ghostscript, poppler-utils	PDF rendering, parsing, and rasterisation	Section 1, Tables
Image Processing	imagemagick, libmagickwand-dev	Image conversion & manipulation	Flowcharts, OLE
Vector Conversion	inkscape	EMF/WMF → PNG for flowcharts and OLE icons	Flowcharts, OLE
Rendering Libraries	libglib2.0-0, libsm6, libxext6, libxrender-dev, libcairo2, libpangocairo-1.0-0	Rendering support for headless graphics	PDF/image handling

Font Libraries	fontconfig, fonts-liberation, fonts-dejavu, fonts-crosextra-carlito, fonts-crosextra-caladea, fonts-noto, fonts-noto-cjk	Ensures consistent text rendering across platforms	DOCX/PDF → image conversions

All dependencies are installed during image build using apt-get, after which caches are cleaned to reduce image size. The environment also enforces **non-root execution** by creating a dedicated appuser account inside the container.

5.2 Deployment Architecture

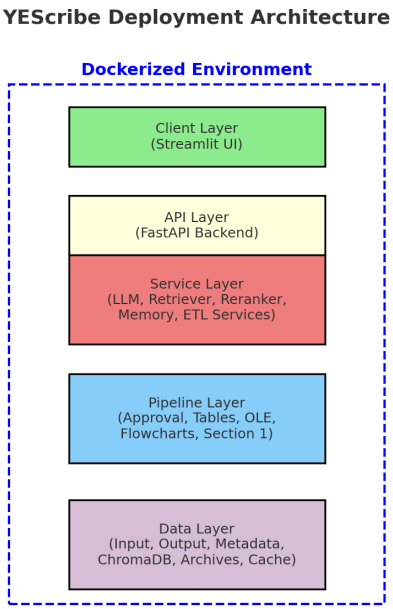


Figure 28: YEScribe Deployment Architecture

The deployment architecture is structured into **five logical layers**, encapsulated within a single Docker container:

1. **Client Layer (Frontend)**
    - Streamlit Chatbot UI (port 8XXX).
    - Streamlit CP Upload Portal (port 7XXX).

2. **API Layer (FastAPI Backend)**
    - REST and WebSocket endpoints (port 5XXX).
    - Responsible for chat sessions, approvals, CP uploads, and retrieval services.
3. **Service Layer**
    - Retrieval and reranking services (Chroma + BGE reranker).
    - Memory service (summarised conversational history).
    - LLM orchestration using Google Gemini models.
    - ETL services for CP uploads and metadata processing.
4. **Pipeline Layer**
    - Five sequential pipelines: Approvals, Tables, OLE, Flowcharts, and Section 1.
    - Each pipeline extracts structured data and embeds it into the Chroma knowledge base.
5. **Data Layer**
    - **Input directory**: Raw CP files (api/data/CPs).
    - **Output directory**: Pipeline extracted outputs.
    - **Metadata**: Joined and approval JSON metadata.
    - **Chroma**: Serves as CP knowledge base and persisted embeddings for retrieval for RAG QA Chain.
    - **Archive directory**: Stores processed CPs with audit logs.
    - **Cache directories**: Embeddings, reranker weights, and LangChain LLM cache.

All components are orchestrated within the Docker runtime, which are illustrated in the architectural diagram in Figure 28.

5.3 Deployment Process

5.3.1 Preparing the Environment

1. Create a .env file in the project root to configure:
    - API keys (GOOGLE_API_KEY, LANGCHAIN_API_KEY).
    - File system paths (INPUT_DIR, OUTPUT_DIR, CHROMA_DIR, ARCHIVE_DIR).
    - Model caches (CACHED_MODELS_DIR, CACHED_EMBEDDING_DIR, CACHED_RERANK_DIR).
2. Ensure persistent directories exist on the host for archives, caches, and logs.

5.3.2 Building the Image

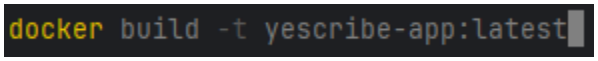


Figure 29: Command for building YEScribe project Docker Image

5.3.3 Running in Development Mode

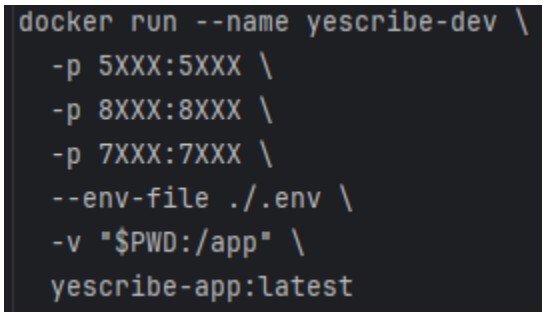


Figure 30: Command for creating new Docker container for YEScribe project Docker Image (Dev Mode)

This setup mounts the source code for live reloading during development or testing mode in IOT server.

5.3.4 Running in Production Mode

```
docker run -d --name yescribe \
  -p 5XXX:5XXX \
  -p 8XXX:8XXX \
  -p 7XXX:7XXX \
  --restart unless-stopped \
  --env-file ./.env \
  -v "$PWD/api/data:/app/api/data" \
  yescribe-app:latest
```

Figure 31: Command for creating new Docker container for YEScribe project Docker Image (Prod Mode)

Here, only persistent volumes are mounted. This ensures that archives, embeddings, and caches are preserved across container restarts.

Chapter 6: Documentation

6.1. Technical Documentation

6.1.1. Overview

The technical documentation is a critical component of the project as it provides a structured, detailed reference of the entire system. It ensures that developers, maintainers, business analysts, and system administrators can fully understand the system's architecture, workflows, and components without relying solely on the source code. This documentation also supports future enhancements, debugging, and integration efforts by offering clear explanations of how each part of the system operates. The report consolidates the technical aspects into one comprehensive reference, covering the following sections:

1. Introduction
2. System Overview (High-Level)
3. Technical Architecture (Low-Level)
4. Constants and Utilities
5. ETL Pipelines
6. CP Upload Portal Internal Architecture
7. RAG Chatbot Internal Architecture
8. Metadata Management
9. Data Flow
10. FastAPI & Schema Reference (Note: All API endpoints removed)
11. Deployment Guide (Docker) (Note: all port numbers removed)
12. Current Known Issues & Limitations
13. Future Enhancements

The technical documentation also includes the user manual, which was created to guide users in operating the YEScribe platform. It covers the following sections:

1. Overview
2. Uploading Concept Papers
3. Using the Chatbot & Sidebar
4. Troubleshooting
5. Getting Help

6.1.2. Accessing the Technical Documentation

The finalised technical documentation can be accessed as part of the submission of this report afterwards.

Chapter 7: References

1. Yes 5G, Uncap Yourself - No Data Cap, No Speed Cap. (2024, March 13). *About us - Yes 5G | Uncap yourself - no data cap, no speed cap.* https://www.yes.my/about-us/
2.
3. Lin, B. (2024, May 21). *From RAGs to Vectors: How Businesses Are Customizing AI Models*. The Wall Street Journal. https://www.wsj.com/articles/from-rags-to-vectors-howbusinessesare-customizingai-models-beea4f11?reflink=desktopwebshare_permalink
4. Data, P. (2024). *Step-by-Step Guide: Building a RAG System using Azure AI Search*. Pondhouse-Data.com; Pondhouse Data. https://www.pondhouse-data.com/blog/rag-with-azure-ai-search
5. deepset-ai. (2022). *GitHub - deepset-ai/haystack-tutorials: Here you can find all the Tutorials for Haystack* 📓. GitHub. https://github.com/deepset-ai/haystack-tutorials
6. onyx-dot-app. (2025, August 18). *GitHub - onyx-dot-app/onyx: Gen-AI Chat for Teams - Think ChatGPT if it had access to your team's unique knowledge.* GitHub. https://github.com/onyx-dot-app/onyx
7. **Ahmed, Z.** (2024, April 28). *What is FastAPI and its LLM Applications?* Medium. https://medium.com/@zahmed333/what-is-fastapi-and-its-llm-applications-4b30ae2d43a5
8. GeeksforGeeks. (2020, November 24). *A Beginners Guide To Streamlit*. GeeksforGeeks. https://www.geeksforgeeks.org/python/a-beginners-guide-to-streamlit/

9. Awan, A. A. (2023, August 31). *Chroma DB Tutorial: A Step-By-Step Guide*. Datacamp.com; DataCamp. https://www.datacamp.com/tutorial/chromadb-tutorial-step-by-step-guide
10. Hugging Face. (n.d.). *sentence-transformers/all-MiniLM-L6-v2 · Hugging Face*. Huggingface.co. https://huggingface.co/sentence-transformers/all-MiniLM-L6-v2
11. **Qwen Team (Alibaba).** (2025, June 5). *Qwen3 Embedding: Advancing Text Embedding and Reranking Through Foundation Models.* Qwen Model Blog. https://qwenlm.github.io/blog/qwen3-embedding/
12. **Beijing Academy of Artificial Intelligence (BAAI).** (2023). *BGE-Reranker-v2-m3 – Model Card.* Hugging Face. https://huggingface.co/BAAI/bge-reranker-v2-m3 (accessed 2025).
13. Google. (2025). Gemini Developer API Pricing. Gemini API. https://ai.google.dev/gemini-api/docs/pricing
14. Mandal, S., Gupta, N., Talewar, A., Ahuja, P., Juvatkar, P., & Banda, G. (2025). *IDP Leaderboard: A Unified Leaderboard for Intelligent Document Processing Tasks*. Intelligent Document Processing Leaderboard. https://idp-leaderboard.org/
15. **Bhuyan, A. P.** (2024, November 5). *Why Docker is Essential for Modern Development Environments and Its Alternatives.* DEV Community. https://dev.to/adityabhuyan/why-docker-is-essential-for-modern-development-environments-and-its-alternatives-6k7
16. GeeksforGeeks. (2025, July 20). *Introduction to Postman for API Development*. GeeksforGeeks. Retrieved from https://www.geeksforgeeks.org/web-tech/introduction-postman-api-development/

17. Lakshitha. (2025, June 26). *FastAPI vs Django vs Flask: Python Backend for AI-Powered Apps*. Medium; Level Up Python. https://medium.com/level-up-python/fastapi-vs-django-vs-flask-python-backend-for-ai-powered-apps-4e30c4c4f986
18. The Educative Team. (2025, June 23). *How LangChain makes retrieval augmented generation production-ready*. Learning Daily. https://learningdaily.dev/how-langchain-makes-retrieval-augmented-generation-production-ready-8492790d7baa
19. LangChain. (n.d.). *LangSmith*. Www.langchain.com. https://www.langchain.com/langsmith
20. LangChain. (n.d.-a). *LangGraph*. Www.langchain.com. https://www.langchain.com/langgraph

Appendix A

Snippets of Finalised Frontend Design (CP Upload Portal)

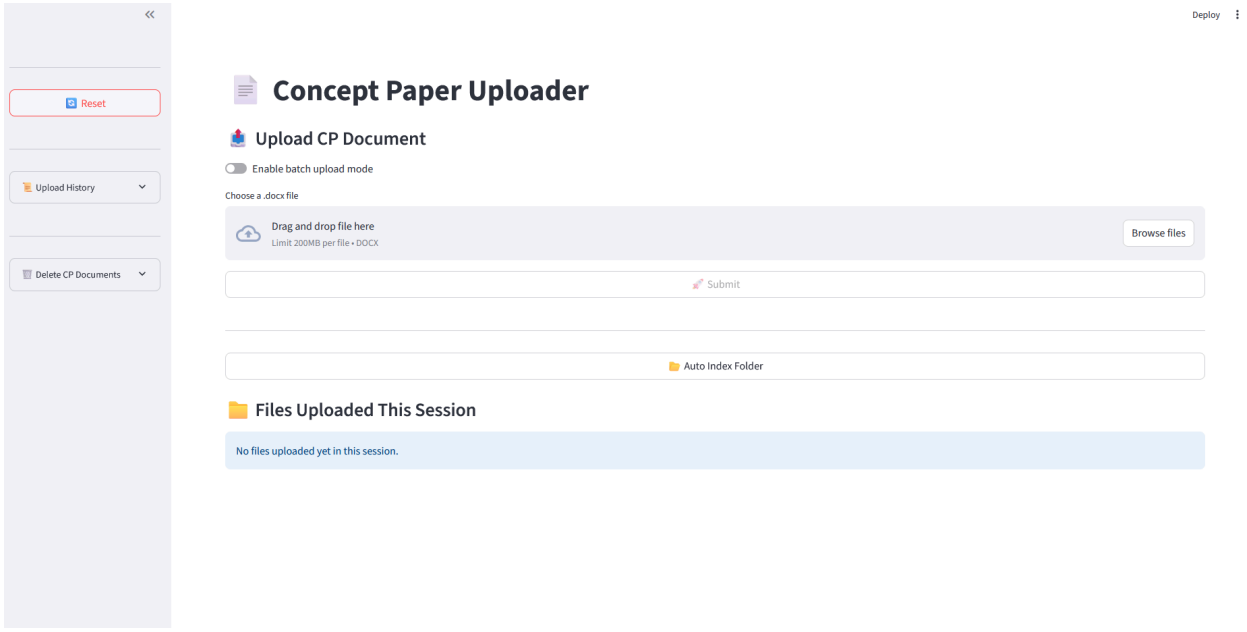


Figure A.1 CP Upload Portal Base Screen

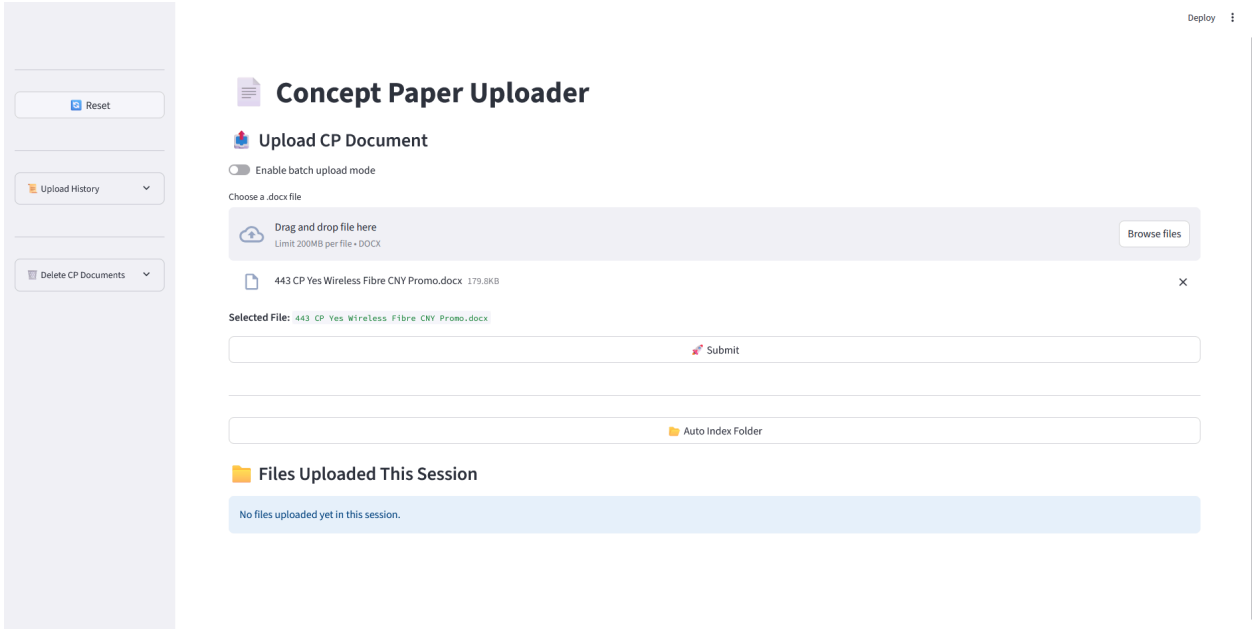


Figure A.1: CP Upload Portal after File Upload

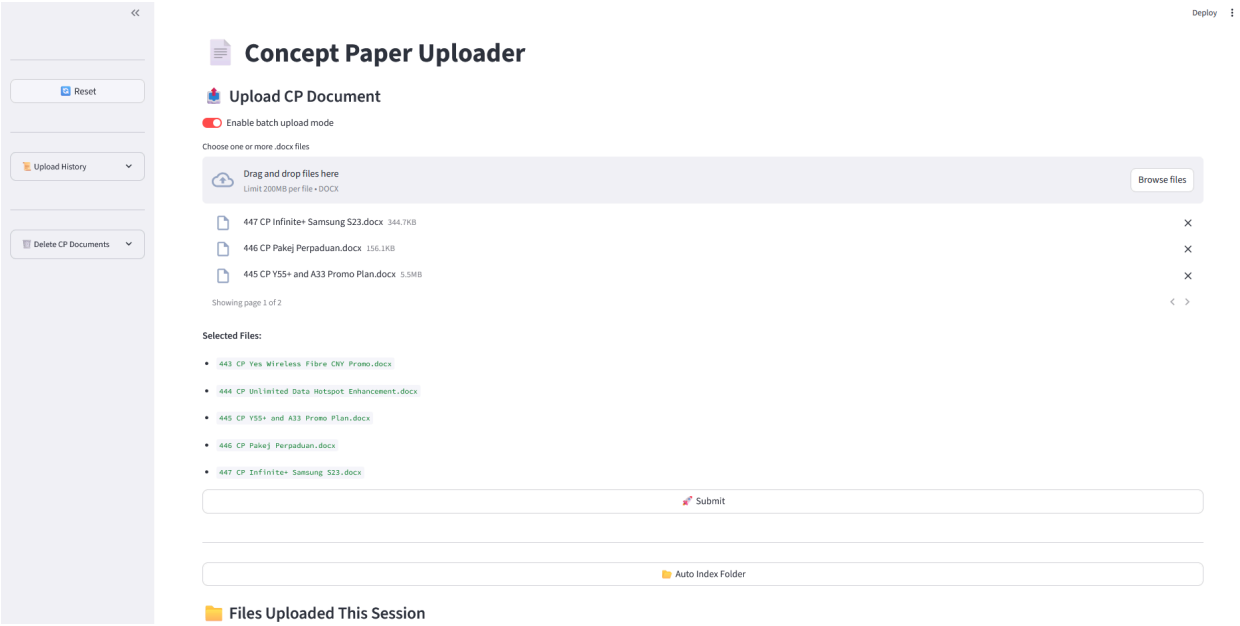


Figure A.2: CP Upload Portal after File Upload (Batch)

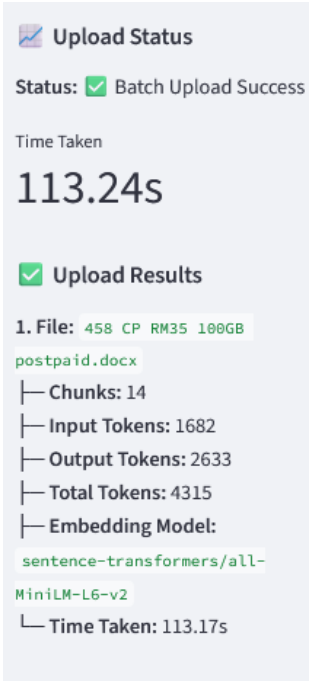


Figure A.3: CP File Indexing Success Status Display + Details

Figure A.4: CP Upload History Display

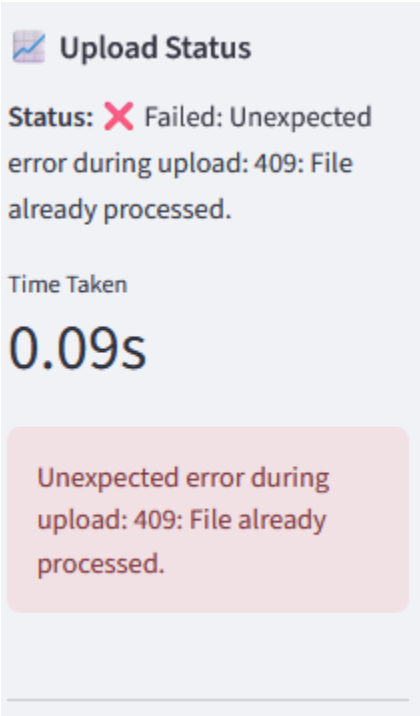


Figure A.5: CP File Indexing Failure Status Display + Details

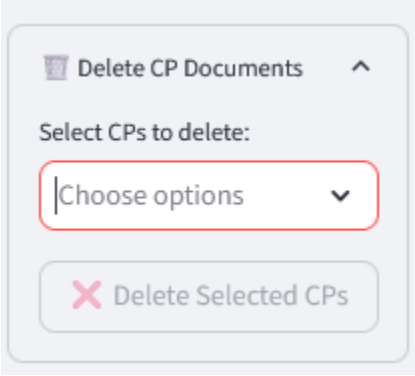


Figure A.6: CP Deletion Section

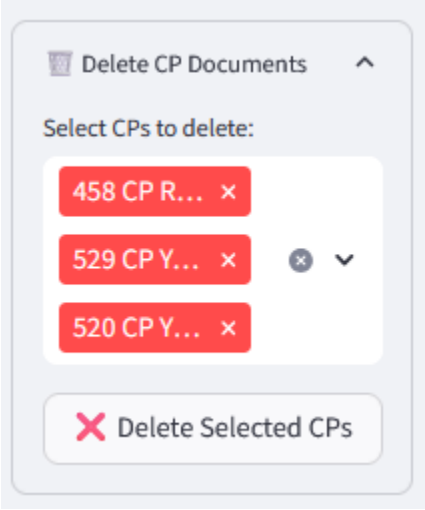


Figure A.7: CP Deletion Section (Selected)

Appendix B

Snippets of Finalised Frontend Design (RAG CP Chatbot)

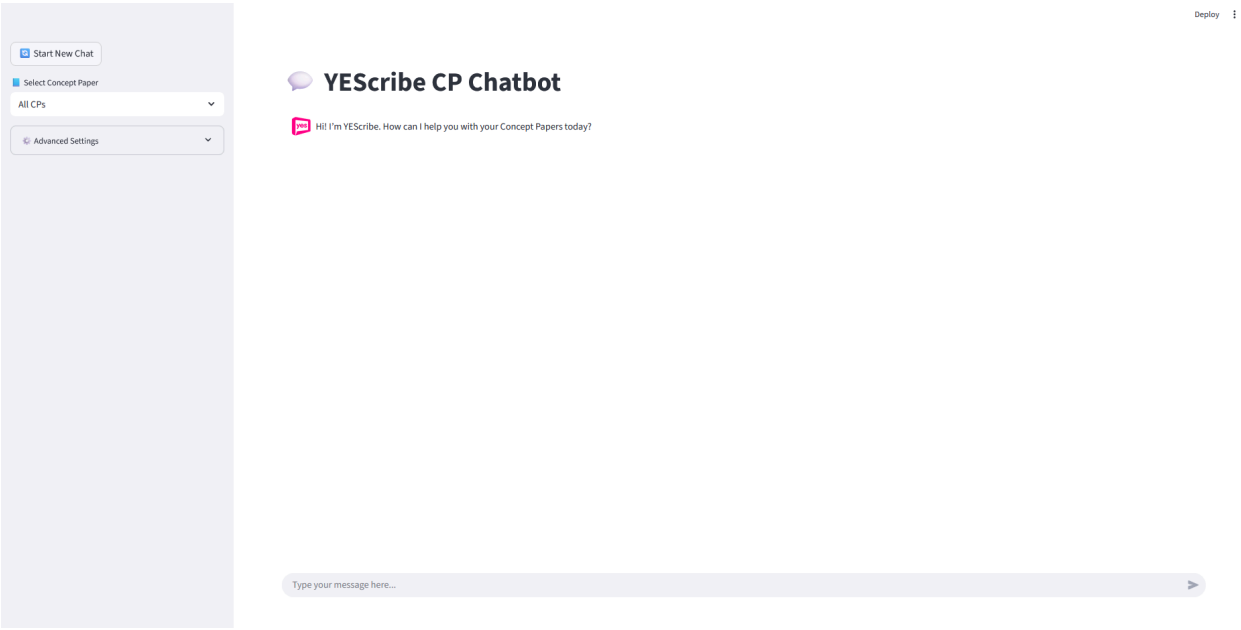


Figure B.1: RAG CP Chatbot Starting Screen

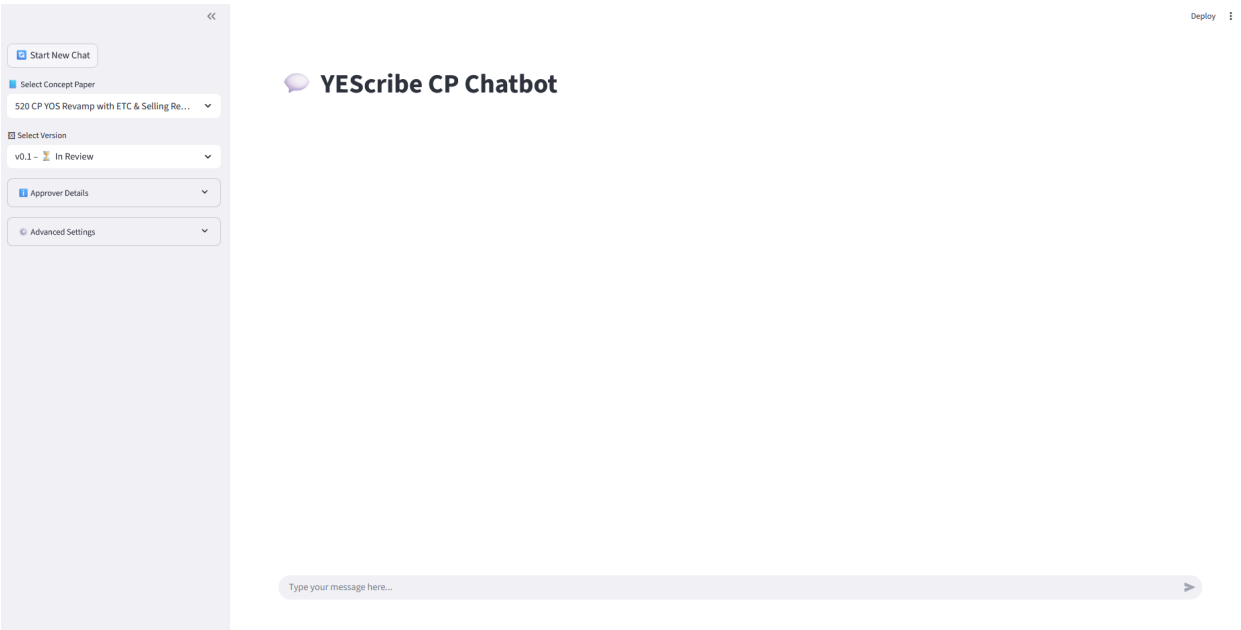


Figure B.2: RAG CP Chatbot Sidebar After selecting CP & Version

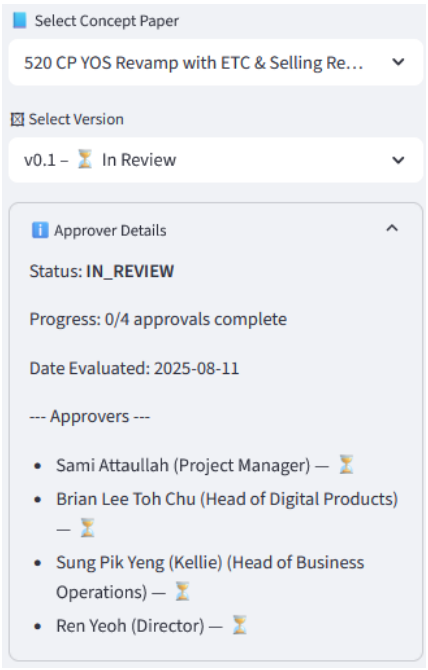


Figure B.3: Approver Details Sidebar Section

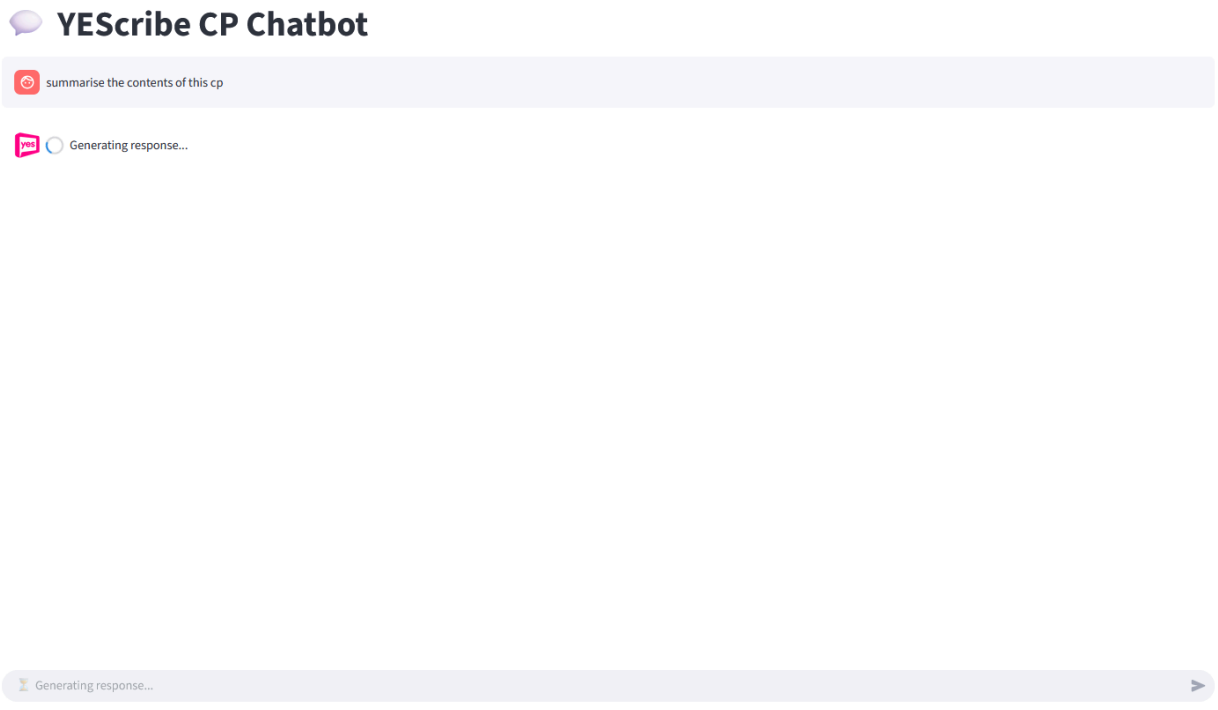


Figure B.4: Chatbot Pending Generation UI

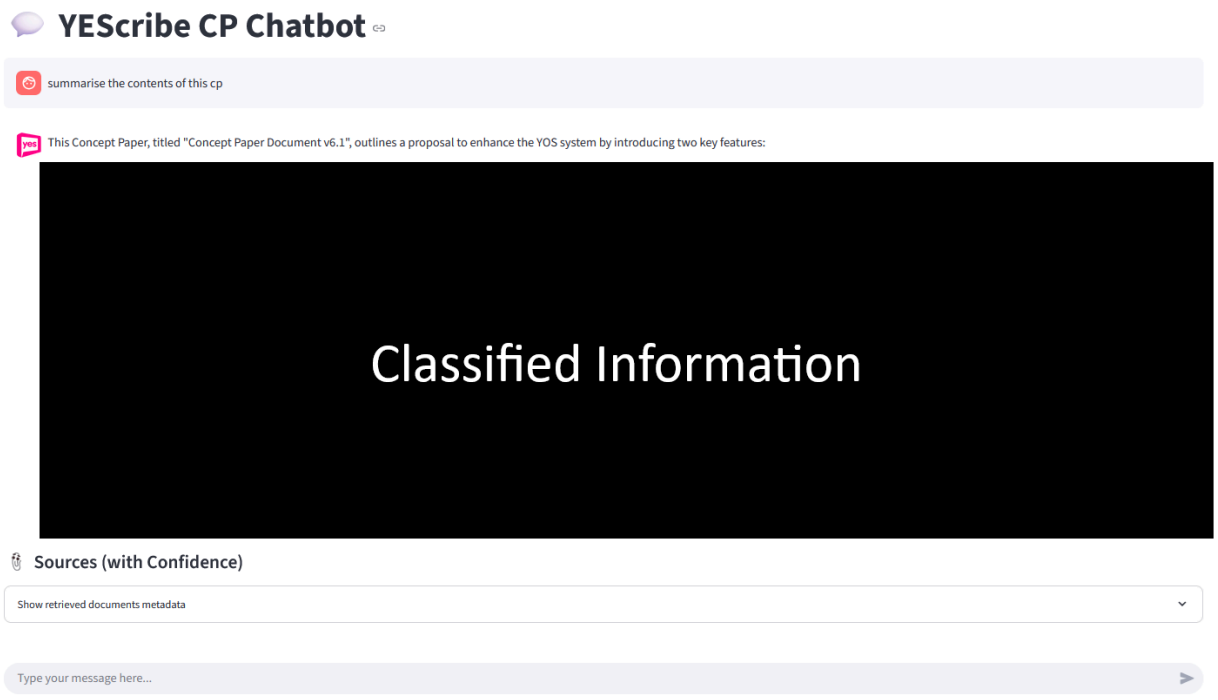


Figure B.5: Sample Chatbot Generation Results

Figure B.6: Show Retrieved Documents Section (Below Generation Results)