International Hellenic University
Department of Information and Electronic Engineering

DIPLOMA THESIS

Chatbot development based on RAG architecture and automated knowledge gathering via bots

Students:
Christos Gousios
Student ID: 185173
Alexios Konstantinidis
Student ID: 185206

Supervisor:
Stefanos Ougiaroglou

29 May 2026

Chatbot development based on RAG architecture and automated knowledge gathering via bots
Code of Dissertation: 24198
Christos Gousios and Alexios Konstantinidis
Supervisor: Stefanos Ougiaroglou
Date of undertaking: 08-04-2024
Date of completion: 29-05-2026

We hereby affirm the authorship of this paper as well as the acknowledgement and credit of whichever assistance we received in its composition. We have, furthermore, noted the various sources from which we extracted data, ideas, visual or written material, in paraphrase or exact quotation. Moreover, we affirm the exclusive composition of this paper by ourselves, for the purpose of it being a dissertation, in the Department of Information and Electronic Engineering of the I.H.U.

This paper constitutes the intellectual property of Christos Gousios and Alexios Konstantinidis the students that composed it. According to the open-access policy, the author/composer offers the International Hellenic University authorisation to use the right to reproduce, borrow, publicly present and digitally distribute the paper globally, in electronic form and media of all kinds, for teaching or research purposes, voluntarily. Open access to the full text by no means grants the right to trespass the intellectual property of the author/composer, nor does it authorise the reproduction, republication, duplication, selling, commercial use, distribution, publication, downloading, uploading, translation, modification of any kind, in part or summary of the paper, without the explicit written consent of the authors.

The approval of this dissertation by the Department of Information and Electronic Engineering of the International Hellenic University does not necessarily entail the adoption of the authors' views, on behalf of the Department.

Dedication

This thesis is dedicated to our families, whose continuous support and understanding have been essential throughout the completion of this work. Their encouragement and stability provided the necessary environment for us to focus on our studies and successfully meet the demands of this academic endeavor.

We would also like to express our sincere appreciation to our teacher, Professor Stefanos Ougiaroglou, for his guidance, constructive feedback, and academic support, which contributed significantly to both the development of this thesis and our academic growth.

Finally, we dedicate this work to our fellow student George Maniatakos, for their collaboration, meaningful discussions, and shared effort throughout our studies, as well as to our friends Calliope and Alexia, whose support and positivity made this journey more meaningful.

Abstract

This thesis describes the design and implementation of UniMentor, a university platform developed to make academic information easier to access from one place. The main idea behind the system is to connect a conversational AI assistant with the university's own material, instead of depending only on the general knowledge of a language model. For this reason, the platform uses Artificial Intelligence together with Retrieval-Augmented Generation (RAG), so that student questions can be answered with information taken from controlled academic sources. The system is also designed to support teachers and secretariat staff, since they need a practical way to organize, update, and manage academic and administrative content.

UniMentor is built around a RAG workflow. Documents are inserted into the system, parsed, split into smaller parts, converted into embeddings, and stored so that relevant content can be retrieved when a user asks a question. The retrieved content is then given to the language model as context before the final answer is produced. In this way, the chatbot does not simply generate a generic response, but tries to base its answer on the available university material. The platform supports course documents, announcements, and secretary-managed information, while the administration area allows authorized users to manage subjects, documents, language models, and the knowledge base used by the assistant.

Another important part of the work is the use of configurable bots for automated content ingestion. These bots were added because university information can change frequently, especially in announcements and course-related pages. If every update had to be inserted manually, the system would become harder to maintain over time. With scheduled bots, selected institutional sources can be checked and their content can be sent to the platform for processing and indexing. This combines manual control with a more automated update process and reduces the amount of repetitive work required from teachers or administrative users.

The final system shows one possible way of using RAG in a real university environment. UniMentor does not replace official university services, but it provides an additional access layer that helps students search for information in a more direct way. At the same time, it gives staff tools for maintaining the content that the assistant uses. Through this approach, the thesis presents a practical implementation of an AI-supported academic platform that focuses on reliable information access, easier knowledge management, and better organization of university-related content.

Περίληψη

Η παρούσα πτυχιακή εργασία περιγράφει τον σχεδιασμό και την υλοποίηση του UniMentor, μίας πανεπιστημιακής πλατφόρμας που αναπτύχθηκε με στόχο να κάνει την πρόσβαση στην ακαδημαϊκή πληροφορία πιο άμεση και συγκεντρωμένη. Η βασική ιδέα του συστήματος είναι η σύνδεση ενός συνομιλιακού βοηθού Τεχνητής Νοημοσύνης με το υλικό του ίδιου του πανεπιστημίου, αντί η απάντηση να βασίζεται μόνο στη γενική γνώση ενός γλωσσικού μοντέλου. Για αυτόν τον λόγο, η πλατφόρμα αξιοποιεί Τεχνητή Νοημοσύνη και Retrieval-Augmented Generation (RAG), ώστε οι ερωτήσεις των φοιτητών να απαντώνται με βάση πληροφορίες από ελεγχόμενες ακαδημαϊκές πηγές. Παράλληλα, το σύστημα σχεδιάστηκε ώστε να υποστηρίζει τους διδάσκοντες και το προσωπικό της γραμματείας, οι οποίοι χρειάζονται έναν πρακτικό τρόπο οργάνωσης, ενημέρωσης και διαχείρισης ακαδημαϊκού και διοικητικού περιεχομένου.

Το UniMentor βασίζεται σε μία ροή λειτουργίας RAG. Τα έγγραφα εισάγονται στο σύστημα, γίνεται η επεξεργασία και ο διαχωρισμός τους σε μικρότερα τμήματα, μετατρέπονται σε embeddings και αποθηκεύονται, ώστε να μπορεί να ανακτηθεί το σχετικό περιεχόμενο όταν ο χρήστης υποβάλλει μία ερώτηση. Στη συνέχεια, το περιεχόμενο που ανακτάται δίνεται στο γλωσσικό μοντέλο ως πλαίσιο πριν παραχθεί η τελική απάντηση. Με αυτόν τον τρόπο, το chatbot δεν παράγει απλώς μία γενική απάντηση, αλλά προσπαθεί να τη βασίσει στο διαθέσιμο πανεπιστημιακό υλικό. Η πλατφόρμα υποστηρίζει έγγραφα μαθημάτων, ανακοινώσεις και πληροφορίες που διαχειρίζεται η γραμματεία, ενώ το διαχειριστικό περιβάλλον επιτρέπει σε εξουσιοδοτημένους χρήστες να διαχειρίζονται μαθήματα, έγγραφα, γλωσσικά μοντέλα και τη βάση γνώσης που χρησιμοποιεί ο βοηθός.

Ένα ακόμη σημαντικό μέρος της εργασίας είναι η χρήση παραμετροποιήσιμων bots για την αυτοματοποιημένη εισαγωγή περιεχομένου. Τα bots προστέθηκαν επειδή η πανεπιστημιακή πληροφορία μπορεί να αλλάζει συχνά, ειδικά στις ανακοινώσεις και στις σελίδες μαθημάτων. Αν κάθε ενημέρωση έπρεπε να εισάγεται χειροκίνητα, το σύστημα θα γινόταν πιο δύσκολο στη συντήρηση με την πάροδο του χρόνου. Με τη χρήση προγραμματισμένων bots, επιλεγμένες πηγές του ιδρύματος μπορούν να ελέγχονται και το περιεχόμενό τους να αποστέλλεται στην πλατφόρμα για επεξεργασία και ευρετηρίαση. Έτσι, συνδυάζεται ο χειροκίνητος έλεγχος με μία πιο αυτοματοποιημένη διαδικασία ενημέρωσης και μειώνεται η επαναλαμβανόμενη εργασία για τους διδάσκοντες ή τους διοικητικούς χρήστες.

Το τελικό σύστημα δείχνει έναν πιθανό τρόπο αξιοποίησης του RAG σε ένα πραγματικό πανεπιστημιακό περιβάλλον. Το UniMentor δεν αντικαθιστά τις επίσημες υπηρεσίες του πανεπιστημίου, αλλά παρέχει ένα επιπλέον επίπεδο πρόσβασης που βοηθά τους φοιτητές να αναζητούν πληροφορίες με πιο άμεσο τρόπο. Ταυτόχρονα, προσφέρει στο προσωπικό

εργαλεία για τη συντήρηση του περιεχομένου που χρησιμοποιεί ο βοηθός. Μέσα από αυτήν την προσέγγιση, η εργασία παρουσιάζει μία πρακτική υλοποίηση μίας ακαδημαϊκής πλατφόρμας με υποστήριξη Τεχνητής Νοημοσύνης, με έμφαση στην αξιόπιστη πρόσβαση στην πληροφορία, στην ευκολότερη διαχείριση γνώσης και στην καλύτερη οργάνωση πανεπιστημιακού περιεχομένου.

Contents

1 Introduction **1**
1.1 Artificial Intelligence for Academic Assistance 1
1.2 Chatbot Technology 2
1.3 Motivation 4
1.4 Contribution 5
1.5 Thesis Structure 6

2 The Evolution of Chatbots **8**
2.1 Rule-Based Chatbots 8
2.2 Retrieval-Based Chatbots 9
2.3 Generative AI Chatbots 11
2.4 RAG-Based Chatbots 12
2.5 Hybrid Chatbots 13

3 Retrieval-Augmented Generation (RAG) Technology **15**
3.1 Introduction to Retrieval-Augmented Generation 15
3.2 Large Language Models and Their Limitations 16
3.3 RAG Architecture and Workflow 18
3.4 Embeddings and Vector Databases 19
3.5 Document Chunking and Context Retrieval 20
3.6 Usefulness of RAG for UniMentor 20
3.7 Advantages and Challenges of RAG Systems 21

4 Related Work **23**
4.1 RAG Chatbots for General University Student Support 23
4.2 RAG Chatbots for Administrative and Institutional Information 24
4.3 RAG Chatbots for Course-Based Learning Support 26
4.4 RAG Chatbots with Feedback and Instructor Involvement 27
4.5 RAG Chatbots for Admissions and Academic Counseling 28
4.6 Comparison with UniMentor 29

5 Programming Languages and Technologies **31**
5.1 Core Programming Languages and Runtime Environments 31
5.1.1 Python 32
5.1.2 TypeScript and JavaScript 32

5.1.3 Node.js 32
5.2 Infrastructure and Deployment Technologies 33
5.2.1 Docker 33
5.2.2 Docker Compose 33
5.2.3 Bash, PowerShell and Makefile Tooling 34
5.3 Database Technologies 34
5.3.1 MySQL 34
5.3.2 Qdrant Vector Database 35
5.4 Artificial Intelligence Provider Technologies 35
5.4.1 Ollama 35
5.4.2 OpenAI API 36
5.4.3 Embedding Models 36
5.5 API Backend Technologies 36
5.5.1 FastAPI and Uvicorn 37
5.5.2 Pydantic 37
5.5.3 SQLAlchemy and PyMySQL 37
5.5.4 API Backend Supporting Libraries 38
5.6 Document Processing and OCR Technologies 38
5.6.1 PDF, Office, CSV, and Text Parsing 38
5.6.2 OCR Technologies 39
5.7 Automation and Browser Interaction Technologies 39
5.7.1 Puppeteer Core and Browser Management 39
5.7.2 puppeteer-extra and Stealth Plugin 40
5.7.3 Cheerio 40
5.7.4 Dockerode and Automation Supporting Libraries 40
5.8 Frontend Development Technologies 41
5.8.1 React 41
5.8.2 Vite 41
5.8.3 Tailwind CSS 42
5.8.4 Radix UI and Local UI Components 42
5.8.5 Frontend Supporting Libraries 42

6 Design and Implementation of UniMentor **44**
6.1 Functional Requirements 44
6.2 Architecture 47
6.3 Relational Database Design and Development 51
6.4 Vector Database Design and Development 57
6.5 AI Providers 60
6.6 Backend Development (API) 63
6.6.1 Backend Role in the System Architecture 63
6.6.2 Project Structure and API Organization 64
6.6.3 Database Session Handling 66
6.6.4 Authentication and Role-Based Authorization 67
6.6.5 LLM Usage in Backend Services 68
6.6.6 Streaming Responses and Error Handling 69

6.6.7 RAG Flow inside the Backend 70
6.6.8 Document Uploading and Queueing 71
6.6.9 Administration API 72
6.6.10 Bot-Facing API 74
6.6.11 Ingestion Worker Integration 75
6.6.12 User Service and Shared User Data 76
6.6.13 Conversation Management and AthenaAI Chat 76
6.6.14 AInsights and Career Mentor Tools 78
6.7 Bot Development 79
6.7.1 Orchestrator 80
6.7.2 Worker 83
6.7.3 Templates 85
6.7.4 Security Concerns and Design Choices 88
6.8 Frontend Development 90
6.8.1 Frontend Role in the System Architecture 90
6.8.2 Project Structure and Routing 91
6.8.3 Authentication Context and Protected Routes 91
6.8.4 API Communication Layer 92
6.8.5 Streaming Response Handling 93
6.8.6 Reusable UI Components and Styling 93
6.8.7 Home Page and Navigation 93
6.8.8 AthenaAI Chat Interface 94
6.8.9 AInsights Interface 94
6.8.10 Career Mentor Interface 95
6.8.11 Admin Panel Interface 95
6.9 Source Code Availability 96
6.9.1 Repository Structure 97
6.9.2 Required Environment Files 97
6.9.3 Local Development Requirements 98
6.9.4 Local Setup Steps 98
6.9.5 Remote Server Requirements 100
6.9.6 Remote Deployment Steps 101
6.9.7 Apache and Backend Runtime 102
6.9.8 Cron Jobs and Background Processing 103
6.9.9 Docker Services 103
6.9.10 Logs and Maintenance Commands 104
6.9.11 Common Issues and Troubleshooting 104

7 UniMentor Presentation 106
7.1 Basic User 107
7.2 Admin User - Secretary 113
7.3 Admin User - Teacher 116

8 User Experience 123
8.1 Evaluation Methodology 123

8.2 System Usability Scale 124
8.3 Questionnaire Structure 124
8.4 Question-Level Results 125
8.5 Overall SUS Results 130

9 Future Extensions and Conclusions **132**
9.1 Future Extensions 132
9.2 Conclusions 133

Bibliography **135**

List of Figures

3.1 Comparison between a classic LLM-only chatbot and a RAG-based chatbot. 17

6.1 General architecture of UniMentor 48
6.2 General architecture of UniMentor with emphasis on the relational database layer. . . 51
6.3 Relational database structure of UniMentor. 52
6.4 General architecture of UniMentor with emphasis on the vector database layer. . . . 57
6.5 General architecture of UniMentor with emphasis on the AI providers layer. 60
6.6 General architecture of UniMentor with emphasis on the FastAPI layer. 63
6.7 General architecture of UniMentor with emphasis on the Bots layer. 79
6.8 General architecture of UniMentor with emphasis on the FrontEnd layer. 90

7.1 Home page of UniMentor. 106
7.2 Initial AthenaAI Chat interface for a basic user. 108
7.3 AthenaAI Chat with conversation history, options menu, and formatted answer. . . . 108
7.4 Initial AInsights interface for course-based material generation. 109
7.5 Course selection list in AInsights. 110
7.6 Template selection list in AInsights. 110
7.7 Generated AInsights response based on selected course and template. 110
7.8 Career Mentor proof-of-concept job listing interface. 111
7.9 Career Mentor filter panel with normal and AI-assisted filtering options. 111
7.10 Career Mentor job details dialog. 112
7.11 Secretary view of the Admin Panel models tab. 113
7.12 Secretary view of the Knowledge Base tab. 114
7.13 Create subject dialog for secretary users. 114
7.14 Create document dialog for secretary users. 114
7.15 Delete subject confirmation dialog. 115
7.16 Delete document confirmation dialog. 115
7.17 Upload document dialog for secretary users. 115
7.18 Teacher view of the Knowledge Base tab. 116
7.19 Upload document dialog in the teacher Knowledge Base view. 117
7.20 Teacher view of the Models tab. 118
7.21 Dialog for adding a new LLM. 118
7.22 Dialog for editing an existing LLM. 118
7.23 Delete confirmation dialog for an LLM. 119
7.24 Teacher view of the Bots tab. 119

7.25 Create bot dialog with one-time execution pattern. 120
7.26 Create bot dialog with repeat-interval pattern. 120
7.27 Edit bot dialog for an existing bot schedule. 121
7.28 Delete confirmation dialog for a bot schedule. 121
7.29 Bot logs dialog showing execution history and messages. 122

8.1 Responses to SUS Question 1: I think that I would like to use UniMentor frequently. 125
8.2 Responses to SUS Question 2: I found UniMentor unnecessarily complex. 126
8.3 Responses to SUS Question 3: I thought UniMentor was easy to use. 126
8.4 Responses to SUS Question 4: I think that I would need the support of a technical person to be able to use UniMentor. 127
8.5 Responses to SUS Question 5: I found the various functions in UniMentor were well integrated. 127
8.6 Responses to SUS Question 6: I thought there was too much inconsistency in UniMentor. 128
8.7 Responses to SUS Question 7: I would imagine that most people would learn to use UniMentor very quickly. 128
8.8 Responses to SUS Question 8: I found UniMentor very cumbersome to use. 129
8.9 Responses to SUS Question 9: I felt very confident using UniMentor. 129
8.10 Responses to SUS Question 10: I needed to learn a lot of things before I could get going with UniMentor. 130
8.11 Distribution of final SUS scores for UniMentor participants. 131

List of Tables

2.1 Comparison of chatbot paradigms and their core characteristics. 14

6.1 Functional requirements expressed as user stories 45
6.2 Main FastAPI endpoints of UniMentor grouped by router and accepted roles. 65

8.1 Summary of SUS results for UniMentor 130

Chapter 1

Introduction

1.1 Artificial Intelligence for Academic Assistance

In recent years, universities have started to depend more and more on digital platforms for everyday academic communication. Information about courses, regulations, administrative procedures, announcements, exams, and learning material is now usually published online. This is clearly more convenient than older paper-based communication, because students can access many resources without being physically present at the university. However, this does not automatically mean that the information is easy to find. As more services have become digital, the information has also become distributed across different systems, instead of being available through one clear point of access.

This situation is common in many universities, including our own academic environment. A student may need to use an e-learning platform for course material, the department website for announcements, downloadable PDF files for regulations, and other university services or intranet pages for additional information. Furthermore, each department, secretariat, laboratory, or academic unit may publish and update information in its own way. This means that students often have to search through different websites, course pages, documents, and announcements before they can answer even a simple academic question.

Each of these systems may be useful on its own, but the problem appears when the student has to combine them. Information may be available, but it can be fragmented, repeated in different places, outdated, or difficult to locate. A student may know that an answer exists somewhere in the university's digital environment, but still not know where to search, which version is correct, or whether the information is still valid. This becomes more difficult when the answer depends on more than one source. For example, a student may need to check a course requirement, an administrative deadline, and a regulation document before making a decision.

For this reason, the difficulty is not only the lack of information, but also the way the information is organized and accessed. Static websites, separate platforms, and document repositories can provide the necessary material, but they do not always help the student understand which information is relevant to a specific question. In many cases, the student has to locate, compare, filter, and interpret the information alone. This can lead to delays, wrong assumptions, or unnecessary communication with classmates and informal groups, especially when the official source is not easy to find.

Artificial intelligence can help with this problem by changing the way students interact with academic

information. Instead of forcing the student to move between disconnected systems, an AI-based assistant can provide a more direct way to ask questions and receive guidance. Through natural language, students can describe what they need in their own words, without first knowing the exact platform, document title, or administrative category where the answer may be located. In this role, AI does not replace the official university sources. It works as an additional layer that helps the user reach and understand them more easily.

This is important because students usually do not ask questions using the same formal language that appears in university documents. A student may ask what is needed to graduate, how to submit a request, which documents are required for a procedure, or where to find information about a course. A traditional search system may depend on exact keywords and return many results that still need to be checked manually. A conversational assistant can make this process simpler, because it can interpret the question more naturally and guide the user toward the most relevant available information.

At the same time, academic assistance cannot be treated like a general conversation system. University-related answers may affect important decisions, such as course selection, exam participation, administrative requests, internships, and graduation planning. For this reason, an AI assistant in this environment must not only produce fluent text. It must also be connected to verified institutional content, so that the answer is based on sources that the university controls or approves. This requirement makes the design more demanding than a normal chatbot, because the system needs to retrieve the correct information before producing the final response.

Based on these needs, intelligent academic assistance becomes useful mainly when it improves access to reliable institutional information. Conversational systems can make the interaction more natural, but their value in a university environment depends on how well they connect the student's question with the correct academic sources. As universities continue to use more digital services, the need for tools that organize access to this information becomes stronger. This is the direction followed in this thesis, where Artificial Intelligence is used as a practical way to support students in navigating university-related knowledge.

1.2 Chatbot Technology

Chatbot technology refers to systems that allow users to communicate with a digital service by writing questions or messages in natural language. In an academic environment, this can be useful because the chatbot can stand between the student and the university's information systems. Instead of asking the student to search through menus, pages, documents, or repositories, the system gives them a simpler starting point: they can write what they need and receive an answer or direction based on that request. This is especially helpful when the student does not already know where the information is located or what exact terminology is used by the institution.

Modern chatbots are not limited only to answering simple questions. Depending on their design, they can guide users through procedures, summarize existing material, explain requirements, assist with forms, or help the user understand which university service is relevant to their case. For students, this can reduce the effort needed to find academic guidance. A student should not always have to know whether a question belongs to a course page, a department announcement, a regulation document, or a secretariat service. A chatbot can provide one common point of communication from which the

search for information can begin.

In a university setting, chatbot technology can support different types of student needs. One use case is academic information retrieval, such as questions about courses, learning material, prerequisites, examination procedures, or study program requirements. Another use case is administrative guidance, for example, when a student needs to know where to find a form, which documents are needed for a request, or which office is responsible for a specific procedure. It can also help with general orientation, such as information about university services, announcements, events, and available resources. These are not rare cases, but questions that can appear repeatedly during a student's studies.

The main advantage of this type of system is that the interaction starts from the student's question, not from the internal structure of the university. Many university platforms are organized in a way that makes sense administratively. Information may be separated by department, office, course, document type, or announcement category. This structure is useful for the institution, but it is not always how students think when they need help. A student usually starts with a practical problem, not with knowledge of where the answer has been classified. A conversational interface can reduce this gap because it allows the user to describe the problem directly.

Availability is another important benefit. A digital assistant can be accessed outside office hours and can respond immediately to common questions. This can be useful when students are studying, preparing documents, checking requirements, or planning academic tasks. Of course, such a system cannot replace the secretariat, teachers, or academic advisors, especially for cases that require official approval or personal judgment. However, it can handle part of the repetitive information search and reduce the number of simple questions that would otherwise need to be answered manually.

The value of a chatbot, however, depends on the information it uses. A chatbot that only gives general answers is not enough for academic support, because university information is usually local, specific, and often updated. It may include internal regulations, course announcements, department procedures, secretary-managed documents, or other material that is not part of public general knowledge. For this reason, a university chatbot must be connected to the institution's own knowledge sources, and this connection must be controlled so that the system does not answer based only on assumptions.

This is important because students may use chatbot answers as real guidance. If the system gives an incomplete or unsupported answer, it may create more confusion rather than offering assistance. Therefore, a chatbot for academic use should not only provide a natural language interface. It also needs grounding, maintainability, and a way to follow updates in the university's documents. In practice, this means that the chatbot should work as an access layer over the university knowledge base. It should retrieve information from approved sources, use the retrieved content when forming the answer, and avoid presenting unsupported information as if it were official.

In this thesis, chatbot technology is treated as a practical tool for improving access to academic information. Its usefulness comes from the fact that it gives students a more direct and familiar way to interact with university knowledge, especially when that knowledge is distributed across different systems and files. A university chatbot should help the student ask questions naturally, locate relevant information from official sources, and receive a clear answer without manually checking many different documents and platforms. For this reason, chatbot technology is a suitable foundation for the development of UniMentor as an intelligent academic support platform.

1.3 Motivation

Although universities now use many digital systems, students still often struggle to find accurate and updated academic information. Having information online does not automatically make it easy to access or understand. In practice, important university content may be split between the central website, department pages, intranet systems, e-learning platforms, administrative repositories, PDF files, and announcement sections. This creates a practical problem: the information may exist, but the student still has to spend time locating it and deciding whether it is the correct version.

This problem can also be seen in our own academic environment. Course material may be uploaded to an e-learning platform, while general information about the same course may be available on the department website. Administrative instructions may come from the secretariat, announcements may be published in a different area, and regulations may be stored as separate documents. Some information exists as web content, some as files, and some inside internal university services. Because of this, there is no single place that covers the full range of information a student may need during their studies.

The lack of centralization makes the search process slower than it should be. In many cases, students first need to know where to look before they can even start searching. If they choose the wrong platform or use different wording from the one used in the official source, they may not find the answer at all. Even when they find something relevant, they may still need to compare it with other sources to check if it is complete or still valid. This adds unnecessary effort to simple academic tasks and can make students depend less on official channels.

Another reason behind this work is that many students do not have unlimited time available for university activities. Some students work while studying, either because they need the income or because they want to gain professional experience. This can make it harder for them to attend every lecture, follow every laboratory session, or keep up with all course updates at the correct time. When this happens, they may need extra help to understand specific parts of a course, clarify material they missed, or connect the available study resources with what was discussed during teaching.

This becomes more important when course-related information is also spread across different sources. A student who missed a lecture may not only need the lecture notes. They may also need the related announcement, an exercise file, a laboratory instruction, or an explanation that connects these resources together. If all these pieces are stored in different places, the student loses time before the actual studying even begins. For this reason, an academic support system should not only help with administrative information, but should also help students navigate course-related knowledge more efficiently.

Another issue is that academic information changes over time. Course documents may be replaced, announcements may expire, procedures may be updated, and new decisions may override older instructions. In a fragmented digital environment, older material can remain accessible even after newer information has been published elsewhere. This makes it difficult for students to know which source is the most recent or the most reliable. As a result, they may unintentionally use outdated or incomplete information when planning their academic actions.

Simple chatbot systems are not enough to solve this problem on their own. A chatbot that depends only on predefined answers or a manually maintained knowledge base can become difficult to manage when the amount of information grows. Every change would require someone to update the answers man-

ually, and this is not practical when courses, announcements, and administrative procedures change often. Furthermore, these systems usually cannot combine information from different documents or handle the many different ways students may phrase the same question.

General-purpose AI assistants also have limitations in this case. They can produce fluent and useful-looking answers, but they do not automatically know the private or institution-specific information of a particular university. They may not have access to current announcements, local regulations, internal procedures, or course-specific material. This means that a language model by itself is insufficient for academic assistance, because the answer must be connected to the actual sources of the institution and not only to the model's general knowledge.

These problems show the need for a system that combines natural language interaction with access to verified university content. The student should be able to ask a question in a simple way, but the system should still search the correct academic sources before producing an answer. This is the main difference from a general chatbot. The goal is not only to make the interaction easier, but also to keep the answer grounded in information that belongs to the university environment.

The motivation of this thesis therefore comes from a practical problem: university information is available, but it is often scattered across different systems, formats, and services. Students need a tool that reduces the time spent searching, while still keeping the answer reliable. This is important because academic information is not just general guidance. It can affect course choices, examination participation, administrative requests, internship procedures, and graduation planning.

Based on this need, this thesis focuses on the design and development of UniMentor, a platform based on Retrieval-Augmented Generation (RAG). UniMentor allows students to interact with university-related information through a conversational AI interface, while also using institution-specific knowledge sources during answer generation. The purpose of the system is to make academic information easier to access, reduce the search effort for students, and provide answers that are more closely connected to the available university material.

1.4 Contribution

The main contribution of this thesis is the design and implementation of UniMentor, a university-oriented AI platform that connects a conversational assistant with controlled academic and administrative knowledge through Retrieval-Augmented Generation. The work does not focus only on building a general chatbot. Its main purpose is to show how a chatbot can be connected to the internal data of the information and electronics engineering department, so that the answers are based on material that is available, managed, and updated within the platform.

A general language model cannot reliably answer questions about the internal information of the department on its own. For example, it cannot automatically know the latest Moodle material, department announcements from Aboard, information from the department website, secretary-managed documents, or internal academic procedures. In UniMentor, this information can be inserted into the system, processed, indexed, and later used as context during answer generation. This allows the assistant to produce answers that are based not only on the model's general knowledge but also on department-specific sources.

A central contribution of UniMentor is that it addresses the problem of scattered university informa-

tion. In a real academic environment, students may need to search across Moodle, the department website, Aboard announcements, PDF files, course documents, and secretariat-related information before finding the answer they need. UniMentor was designed as an additional access layer on top of these sources. Instead of forcing students to manually move between different systems, the platform allows them to ask questions in natural language and receive answers based on the material that is available to the system.

Another contribution is the development of a RAG-based workflow for turning university material into searchable knowledge. The system includes document uploading, parsing, chunking, embedding generation, vector storage, retrieval, and language model interaction. Through this flow, documents and collected content are transformed into smaller, searchable parts that can be retrieved when a user asks a related question. This makes the platform more suitable for university use, because new material can be added or updated without redesigning the chatbot from the beginning.

The thesis also contributes through the use of automated knowledge gathering with bots. This was added because university information is not always static, and not all content is practical to insert manually. Some sources, such as Moodle pages, department announcements, and the department website, may change during the semester. The bots can be configured to collect information from selected trusted sources and send it to the backend, where it can be processed and indexed. This reduces repeated manual work and helps the knowledge base remain easier to maintain over time.

Another important part of the contribution is the administrative environment of the platform. Since the assistant uses academic and administrative knowledge, the content cannot be uncontrolled. For this reason, UniMentor includes an Admin Panel where authorized users can manage subjects, documents, language models, and bot configurations. This gives teachers and secretariat users a practical way to supervise what information enters the system and what material can later be used by the assistant.

UniMentor also extends the same RAG logic beyond the main chatbot. The platform includes additional AI-supported tools, such as the templates functionality, where users can generate structured academic or informational content based on selected courses and available knowledge. This shows that the architecture is not limited to simple question answering. The same retrieval and generation flow can support more than one academic support function.

Overall, the contribution of this thesis is both practical and technical. On the technical side, it presents a working RAG-based system that combines backend services, document processing, vector retrieval, language model communication, automated ingestion, database management, and frontend interfaces. On the practical side, it focuses on a real problem faced by students and staff: university information exists, but it is often spread across different systems and is not always easy to find. UniMentor shows how this information can be connected through a controlled AI-assisted platform, giving students a more direct way to access academic knowledge while still keeping the official sources and authorized users in control.

1.5 Thesis Structure

The thesis is divided into nine chapters. Chapter 1 introduced the topic of the thesis and explained the problem that motivated the development of UniMentor. It presented the need for better access to academic information, the role of chatbot technology in this context, the motivation behind the work,

and the main contribution of the proposed platform.

Chapter 2 gives the background needed for the rest of the work by presenting the main stages in the evolution of chatbot systems, from earlier conversational approaches to more recent AI-based methods. Chapter 3 focuses on Retrieval-Augmented Generation and explains the basic ideas behind the technology, including how external knowledge is retrieved and used during answer generation. Chapter 4 presents related work and discusses examples of chatbot and RAG-based systems that are connected to academic support or similar use cases.

Chapter 5 describes the main technologies that were used during the development of UniMentor. This includes the programming languages, frameworks, databases, AI-related tools, document processing libraries, automation tools, and frontend technologies that supported the implementation. Chapter 6 then presents the design and implementation of the platform in more detail, including the system architecture, relational database, vector database, backend API, bot mechanism, frontend development, and deployment process.

Chapter 7 presents the final platform from the user's perspective and explains the main functionality available to the different types of users. Chapter 8 discusses the user experience of the system and the way the interface supports interaction with the platform. Finally, Chapter 9 concludes the thesis by summarizing the work that was completed and by presenting possible future extensions that could improve UniMentor further.

Chapter 2

The Evolution of Chatbots

Chatbot technology did not appear in its current form immediately. It developed gradually, following progress in artificial intelligence, natural language processing, machine learning, and information retrieval [1], [2]. Over time, conversational systems changed not only in the way they were implemented, but also in the kind of problems they could support. Early chatbots were mostly based on simple rules and predefined patterns, while newer systems can produce more flexible answers and use context in a more advanced way [1].

This development can be understood through different chatbot paradigms. Each one handles the conversation in a different way. Rule-based systems depend on manually written logic, retrieval-based systems choose responses from an existing set of answers, and generative systems create new text with the help of language models [1]. More recent systems, such as RAG-based chatbots, combine retrieval and generation so that the final answer can be connected with external knowledge sources and not only with the model's internal knowledge [3], [4].

This evolution is important for this thesis because university environments need more than a simple question-answer system. Students may ask about courses, announcements, procedures, regulations, study material, or university services. A chatbot in this context must be able to understand different types of questions, give clear answers, and remain connected to reliable information. For this reason, examining the main chatbot approaches helps explain why more recent architectures are more suitable for academic support systems such as UniMentor.

2.1 Rule-Based Chatbots

Rule-based chatbots are one of the earliest types of conversational systems. Their operation is based on predefined rules, patterns, and scripted responses that are created before the system is used [2], [5]. The general idea is simple: the chatbot receives the user's input, tries to match it with a known pattern, and then returns the response that has been assigned to that pattern. A well-known early example is ELIZA, which showed that even simple pattern matching could give users the impression of a conversation [6].

In this type of chatbot, the behavior of the system is mostly controlled by the developer [1]. The implementation may use decision trees, keyword matching, regular expressions, or predefined dialogue flows [2]. For example, if the user's message contains a specific word or phrase, the chatbot can return

a prepared answer. If the system is menu-based, the next response may depend on the option selected by the user. This makes rule-based chatbots easy to understand and relatively easy to control when the use case is small and clearly defined.

The main benefit of rule-based systems is their simplicity. Since the answers are written in advance, the system can provide stable and consistent responses for the questions it was designed to handle. This makes them useful for limited scenarios where the possible interactions are predictable. A simple rule-based chatbot, for example, could answer frequently asked questions, guide a user through a fixed process, or provide basic information such as contact details, opening hours, or general instructions.

The same simplicity, however, also creates important limitations [2], [5]. A rule-based chatbot usually fails when the user asks something that was not expected during development. Even a small change in wording can be enough for the system to miss the user's intention, unless that variation has also been included in the rules. These systems also have difficulty with context. If the user asks a follow-up question or refers to something from a previous message, a simple rule-based chatbot may not be able to continue the conversation in a useful way.

Another issue is maintenance [2], [5]. As the number of possible questions increases, the number of rules also grows. Developers or administrators have to write, update, and test these rules manually. This may be manageable for a small set of questions, but it becomes harder in a larger domain. If the information changes often, the rules and answers must also be updated often. Otherwise, the chatbot may continue giving old or incomplete responses.

This limitation is especially clear in academic environments. Students do not always use the same words that appear in official university documents. They may write informally, use abbreviations, ask incomplete questions, or describe a problem without knowing the official name of the procedure. For example, a student may not ask for the exact title of a regulation, but may ask what they need to do to submit a request, pass a course, or complete a requirement. A rule-based system would need many manually prepared variations in order to cover these cases.

University information is also broad and changes over time. Course details, announcements, examination information, and administrative procedures may change from one semester to another. Keeping a rule-based chatbot accurate in such an environment would require continuous manual updates. If these updates are not done on time, the system can easily provide outdated information. For this reason, rule-based chatbots are not enough for complex and changing information environments, except when they are used for very narrow and controlled tasks.

Even with these weaknesses, rule-based chatbots are still important in the history of conversational systems [5], [6]. They showed that users can interact with software through language instead of only through buttons, forms, or menus. They also showed that a conversational interface can make a digital system feel more approachable. However, because they cannot easily generalize beyond the rules written for them, they created the need for more flexible chatbot approaches.

2.2 Retrieval-Based Chatbots

Retrieval-based chatbots appeared as a more flexible approach compared to purely rule-based systems [1], [7]. Instead of depending only on manually written rules or fixed dialogue paths, these systems search inside an existing collection of possible answers and select the one that best matches the user's

message [2], [7]. To do this, they may use similarity calculations, statistical methods, or machine learning techniques, depending on how the chatbot has been designed.

The basic idea is that the user's input is compared with stored questions, dialogue examples, answers, or documents [7]. After this comparison, the system retrieves the response that appears to be the most relevant. This gives retrieval-based chatbots more flexibility than strict rule-based systems, because the user does not always need to write the exact phrase that the developer expected. Two users may ask the same thing with different wording, and the system may still be able to connect both questions with the same stored answer.

This approach became useful because it reduced the need to manually design every possible conversation path [7]. Instead of building a large decision tree with many branches, developers or administrators could prepare a collection of question-answer pairs, common dialogue examples, or frequently asked questions. The chatbot would then search this collection and return the closest matching response. In this way, retrieval-based chatbots became more practical than simple rule-based chatbots for domains where the number of possible questions was larger.

Another benefit is that the responses of retrieval-based systems are usually more controlled [2]. Since the chatbot selects from answers that already exist, it does not freely generate new text. This can be useful in cases where the organization wants the answers to remain consistent and approved in advance. For example, in customer service or institutional information systems, it is often better to return a reviewed answer than to produce a new response that may be phrased incorrectly or include unsupported information.

However, retrieval-based chatbots are still limited by the content that exists in their response collection [7]. If the correct answer has not been included, the system cannot create it by itself. It may return a weak match, give a generic fallback answer, or fail to help the user. This means that the quality of the chatbot depends heavily on the size, structure, and completeness of the stored knowledge that it searches.

This limitation becomes more serious in academic environments. Students may not always ask questions that correspond to one stored answer. Some questions may require information from more than one place. For example, a student may ask whether a course requirement affects graduation, whether a missed laboratory session can be replaced, or how a specific announcement relates to a course procedure. A retrieval-based chatbot may find a related answer or document, but it may not be able to combine several pieces of information into one clear explanation.

Maintenance is also an important issue. When new announcements, regulations, course documents, or administrative instructions are published, the knowledge base must be updated so that the chatbot can retrieve the new information. If this update process is manual, the same problem appears again: someone has to keep the stored answers and documents current. If this is not done regularly, the chatbot may continue retrieving older information even though newer material exists elsewhere.

Even with these limitations, retrieval-based chatbots were an important step in the development of conversational systems [5], [7]. They moved chatbot behavior closer to stored knowledge instead of relying only on predefined rules. This made them more useful for larger domains and for cases where different users may ask similar questions in different ways. However, because they mainly select existing responses, they still have difficulty producing flexible explanations or handling questions that require reasoning over multiple sources. This weakness created the need for more advanced

approaches, such as generative and retrieval-augmented systems.

2.3 Generative AI Chatbots

Generative AI chatbots marked an important change in the way conversational systems are designed. In contrast to rule-based and retrieval-based chatbots, they do not only follow predefined scripts or select an existing answer from a stored collection. Instead, they can generate new responses based on the user's input and the context of the conversation [8], [9]. This became possible mainly because of progress in natural language processing and large language models. These models are trained on large amounts of text and learn language patterns that allow them to produce fluent and coherent answers on many different subjects [9], [10].

The main advantage of generative chatbots is their flexibility [8], [9]. They can respond to many types of questions, adjust the style of the answer, and provide explanations that feel more natural than the responses of older chatbot approaches. The user does not need to follow a fixed menu or write the question in a very specific form. They can ask in everyday language, and the chatbot can usually produce an answer that fits the question and the surrounding context.

This flexibility makes generative chatbots useful for educational and academic scenarios. Students often need more than a short factual answer. They may need an explanation, a summary, an example, or a simpler version of a difficult concept. A generative chatbot can support this type of interaction by rephrasing information, organizing an answer into steps, or explaining a topic in a way that is easier to follow. Compared to earlier systems, this makes the interaction feel more supportive and closer to the way a student would ask for help.

Another useful characteristic is that generative chatbots can handle multi-turn conversations more naturally [10]. When the user asks a follow-up question, the system can often use the previous messages to understand what the user is referring to. This makes the conversation less mechanical. In an academic setting, this matters because student questions are often connected. A student may first ask for a definition, then ask for an example, and then ask how the same idea applies to an exercise, a course topic, or a university procedure.

However, generative AI chatbots also have serious limitations when they are used alone [3], [4]. The most important limitation is that their knowledge comes from the data used during training and from the model's internal parameters [3], [4]. This means that the model is not automatically connected to current information, private documents, or institution-specific knowledge. For example, a general language model may not know the latest university announcements, internal regulations, course instructions, or administrative procedures of a specific department.

Another problem is that the answer can sound correct even when it is not [4], [11]. Generative models are good at producing plausible language, but plausible wording is not the same as verified information [11]. In some cases, the model may give an answer with confidence while the content is inaccurate or not supported by a real source. This is especially risky in a university environment, because students may use the answer when making decisions about courses, exams, administrative requests, or graduation requirements.

Transparency is also a challenge for generative chatbots [3], [4]. When a model generates an answer, it is not always clear where the information came from or whether it was based on an approved source.

For general conversation this may not always be critical, but in an academic system it is a major issue. University information should be traceable to official material, because students need to know that the answer is based on current and valid institutional content, not only on the general knowledge of the model.

For these reasons, generative AI chatbots are powerful, but they are not enough by themselves for institutional academic support. They provide natural interaction, flexible explanations, and better conversation flow, but they still need a way to access verified and updated knowledge. In a university platform, the chatbot must be connected to the actual academic sources that it is expected to use. This need led to approaches that combine language generation with information retrieval, so that the response can be both natural and grounded in relevant external content.

2.4 RAG-Based Chatbots

RAG-based chatbots combine two main ideas: information retrieval and language generation [3], [4]. The goal is to make the answer of the chatbot more reliable by giving the language model relevant external information before it generates the final response [3], [4]. This means that the system does not depend only on what the model already knows from training. It first searches a knowledge base, retrieves useful content, and then uses that content as part of the answer generation process.

This approach is useful in cases where the information must be accurate, updated, and connected to a specific organization [3], [4]. In a university environment, this information may include course descriptions, announcements, study material, administrative procedures, regulations, and other official documents. A general-purpose language model may not have this information, or it may have old or incomplete knowledge about it. A RAG-based chatbot can reduce this problem by retrieving content from the institution's own sources before answering.

The basic flow of a RAG-based chatbot can be described in a simple way [3]. First, the user submits a question. Then, the system searches the available knowledge base and finds content that is related to that question. This retrieved content is added to the prompt as context and is given to the language model together with the user's question. After that, the model generates the final answer using both the question and the retrieved material. In this way, the answer is not produced only from the model's general knowledge, but is also influenced by the external sources that were found.

The main benefit of this approach is grounding [3], [4]. If the retrieved content comes from approved sources, the generated answer has a better chance of following the actual information contained in those sources. This does not remove every possible error, but it reduces the risk of answers that are unsupported, outdated, or based only on general assumptions. This is especially important for academic assistance, because students usually need reliable information and not only a fluent or general suggestion.

RAG-based chatbots are also easier to maintain in changing information environments [3], [4]. When institutional information changes, the knowledge base can be updated with new or modified documents, instead of retraining the language model. Then, when a user asks a related question, the retrieval process can use the newer content. This is useful in universities, where course material, announcements, schedules, and procedures may change during the academic year. In this case, updating the retrieval layer is more practical than depending on a static chatbot or on the fixed knowledge of a

model.

Another important advantage is that RAG can work with private or institution-specific information [4]. Many university documents are not part of public training data. They may exist in internal repositories, e-learning platforms, department pages, or secretariat systems. By indexing this content into a controlled knowledge base, the chatbot can answer questions that a general language model would not answer accurately on its own. This is one of the main reasons why RAG is suitable for systems that need to work with local academic knowledge.

At the same time, RAG-based chatbots are not automatically reliable just because they use retrieval [4]. Their quality depends on the documents that are inserted, the way these documents are processed, the chunking strategy, the embeddings, and the relevance of the retrieved results [4]. If the system retrieves weak, incomplete, or unrelated content, the final answer may still be poor. This means that the document processing pipeline and the retrieval mechanism are not secondary details, but central parts of the chatbot's quality.

In this thesis, RAG-based chatbots are important because they match the needs of a university-oriented assistant. Students need answers that are understandable, but they also need those answers to be connected to official or controlled information. RAG provides a way to combine the natural interaction of generative AI with retrieval from institutional sources. For this reason, it forms the basic technological direction followed in UniMentor.

2.5 Hybrid Chatbots

Hybrid chatbots combine more than one conversational approach inside the same system [1], [2]. Instead of depending only on a single method, they may use rule-based logic, retrieval, generative models, and extra control mechanisms depending on what the user is asking [1], [2]. This kind of design is useful because different requests do not always need the same type of processing. The goal is to keep the system flexible, while still maintaining reliability and control where they are needed.

For example, a hybrid chatbot can use simple rule-based behavior for predictable actions, such as greetings, menu options, or basic system commands. For questions that require information, it can use retrieval to find relevant content from a knowledge base. If the answer needs to be written in a more natural way, a generative model can then use the retrieved content to produce the final response. This gives the system the ability to use the strengths of each approach, without depending completely on one technique.

This is useful in academic environments because student requests can be very different from each other [1], [5]. Some questions are simple and direct, such as asking where a document is located or when a service is available. Other questions need more explanation, summarization, or connection between multiple pieces of information. A single chatbot approach is not always suitable for all of these cases. A hybrid design allows the system to respond differently depending on the type and complexity of the request.

Another important benefit is that hybrid chatbots can include safeguards and administrative control [4]. In an institutional system, it is not enough for the chatbot to only produce a fluent answer. The system must also control what information is used and how the answer is formed. For example, it may restrict answers to approved sources, guide users to official services when needed, or avoid giving an

answer when no reliable information has been found. This makes the chatbot more appropriate for environments where accuracy and responsibility are important.

Hybrid chatbots can also support different user roles and different platform functions. In a university platform, students, teachers, secretariat staff, and administrators may not need the same type of access or assistance. A hybrid system can combine conversational interaction with role-based functionality, document management, administrative tools, and knowledge base control. In this way, the chatbot becomes part of a wider academic support platform rather than functioning solely as an isolated question-answering tool.

For these reasons, hybrid chatbot architectures are a practical direction for modern conversational systems [1], [4]. They recognize that no single chatbot paradigm is sufficient for every situation. Rule-based logic can provide control for simple actions, retrieval can connect the system with stored knowledge, generative models can produce natural answers, and RAG-based mechanisms can ground responses in external sources. When these elements are combined carefully, the result can be more useful for complex domains such as higher education.

Table 2.1 summarizes the main chatbot paradigms discussed in this chapter and highlights their respective characteristics.

Paradigm	Response Type	Adaptability	Knowledge Scope
Rule-based	Scripted	Low	Static
Retrieval-based	Selected	Medium	Dataset-bound
Generative AI	Generated	High	Parametric
RAG-based	Generated with retrieved context	High	External and dynamic
Hybrid approaches	Combined	High	Controlled and extensible

Table 2.1: Comparison of chatbot paradigms and their core characteristics.

The evolution of chatbots shows a movement from fixed and predefined systems toward more flexible systems that can use context and external knowledge [1], [3], [5], [6], [10]. Rule-based chatbots introduced conversational interaction, but they were limited by manually written rules. Retrieval-based chatbots improved this by searching existing data, but they still depended on the available response collection. Generative AI chatbots made the interaction more natural and flexible, but they also introduced problems related to factual reliability, source transparency, and access to private information. RAG-based and hybrid approaches try to handle these issues by combining language generation with retrieval and system-level control.

This development is directly connected to academic support platforms. A university chatbot should not only communicate in a natural way. It also needs to answer based on reliable institutional knowledge and behave safely when the information is missing or uncertain. For this reason, chatbot systems for higher education benefit from architectures that combine conversational AI, retrieval mechanisms, and controlled knowledge management. This is also the direction followed in UniMentor, where the chatbot is designed as part of a broader platform for academic information access and support.

Chapter 3

Retrieval-Augmented Generation (RAG) Technology

Retrieval-Augmented Generation (RAG) is a modern approach for building language-model-based systems that can produce answers using external knowledge sources. Instead of depending only on the information stored inside the parameters of a large language model, a RAG system retrieves relevant information from a separate knowledge base and uses it as context during response generation [3], [4]. This makes RAG particularly suitable for knowledge-intensive applications where answers must be accurate, current, and connected to specific documents or institutional sources.

In the context of UniMentor, RAG is an essential technology because university information is not static or fully contained in the general knowledge of a language model. Course material, administrative announcements, regulations, schedules, and internal procedures may change frequently and may exist only inside the university's own digital environment. Therefore, a general-purpose chatbot cannot reliably answer student questions unless it is connected to the university's actual knowledge sources. RAG provides this connection by combining document ingestion, semantic retrieval, and controlled language generation into a single workflow.

This chapter presents the main concepts behind RAG technology. It first explains the limitations of standalone large language models, then describes the architecture of a RAG system, including ingestion, indexing, retrieval, and generation. It also discusses embeddings, vector databases, document chunking, and the advantages and challenges of using RAG in academic environments such as UniMentor.

3.1 Introduction to Retrieval-Augmented Generation

Large language models have demonstrated strong natural language generation abilities and can produce fluent responses across many domains [10]. However, their internal knowledge is limited by the data used during training and by the time at which the model was trained. This means that a model may not know recently updated information, private institutional content, or local academic procedures. In addition, even when a model generates a response that appears confident and well-written, the response may still be unsupported or factually incorrect [11].

Retrieval-Augmented Generation addresses this limitation by introducing an external knowledge re-

trieval step before generation. When a user submits a question, the system does not immediately ask the language model to answer from memory. Instead, it searches a knowledge base for relevant documents or passages. These retrieved passages are then inserted into the model's context, allowing the model to generate an answer based on the provided information [3]. In this way, RAG combines the flexibility of generative models with the reliability of information retrieval.

The main idea behind RAG is the separation between language ability and knowledge access. The language model is responsible for understanding the question and producing a readable answer, while the retrieval system is responsible for locating the information that should support the answer. This separation is useful because the knowledge base can be updated independently from the language model. For example, if a university regulation changes or a new announcement is published, the updated document can be re-indexed without retraining the model.

For UniMentor, this separation is especially important. The platform must answer questions based on university-specific information, such as course documents, announcements, secretary-managed content, and other academic material. These sources may change during the academic year, and some of them may not be publicly available. A RAG-based architecture allows UniMentor to use these sources directly, making the assistant more suitable for the needs of students than a standalone chatbot.

3.2 Large Language Models and Their Limitations

Large language models are trained on very large collections of text and learn patterns that allow them to produce coherent natural language responses [10]. This is what makes them useful in dialogue systems, summarization, explanation, and question answering. In an academic support platform, the language model can help turn retrieved material into an answer that is easier for the student to understand. It can also explain difficult concepts, organize information, and respond in a more conversational way.

However, a language model by itself has important weaknesses when the system requires factual accuracy and domain-specific knowledge. One of the main problems is knowledge freshness. After training, the model's knowledge is mostly fixed, so it does not automatically know about new announcements, updated course material, changed deadlines, or revised university procedures [3], [4]. This is a serious limitation in a university environment, because academic information can change often during the semester.

Another limitation is that the model does not automatically have access to private or local knowledge. Many university resources are stored in internal systems, course platforms, intranet pages, PDF files, or administrative repositories. This type of information is usually specific to the institution and may not exist in the training data of a general model. Because of this, even a strong language model may not be able to answer questions about a specific university correctly unless the relevant material is provided during the user's request.

A further limitation is hallucination. Hallucination refers to cases where the model generates information that sounds possible, but is not supported by the available facts [11]. In a normal conversation this may only be inconvenient, but in academic assistance it can create real problems. Students may use the answer to decide about courses, examinations, applications, or graduation requirements. For this reason, a university chatbot should avoid unsupported claims as much as possible and should not present uncertain information as official guidance.

RAG helps reduce these limitations by giving the language model relevant evidence before the final response is produced. The model is not forced to answer only from its internal knowledge. Instead, it can use selected passages from the university's knowledge base and form the answer around that material. This increases the chance that the response is connected to official or approved content. Still, RAG is not a perfect solution by itself. The final answer also depends on the quality of the inserted documents, the retrieval process, the chunking strategy, and the way the prompt tells the model to use the retrieved context [4].

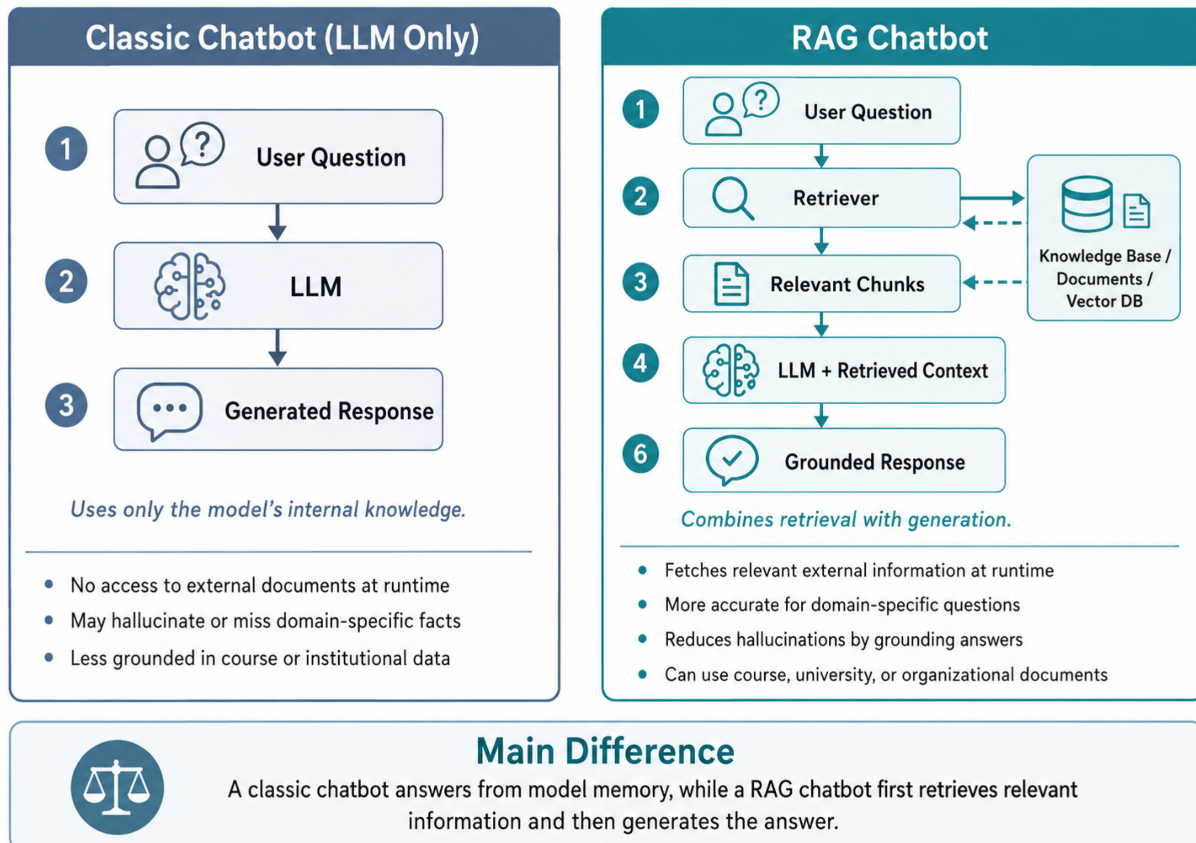

Figure 3.1: Comparison between a classic LLM-only chatbot and a RAG-based chatbot.

Figure 3.1 illustrates the main difference between a classic LLM-only chatbot and a RAG-based chatbot. In the first case, the model answers only based on its internal knowledge. In the second case, the system first retrieves relevant information from an external knowledge source and then uses this retrieved context during answer generation. This makes RAG more suitable for domain-specific environments, such as universities, where the quality of the answer depends on access to updated and controlled information.

3.3 RAG Architecture and Workflow

A RAG system can be understood through two main workflows: the ingestion workflow and the query workflow. The ingestion workflow prepares the documents before users interact with the system, while the query workflow starts when a user submits a question. This separation is useful because document processing and user interaction do not happen in the same step. The system can first organize the available knowledge, and later use that knowledge when a question needs to be answered.

The ingestion workflow starts with the collection of documents from one or more knowledge sources. These sources can include PDF files, text files, web pages, structured records, announcements, and course material. In UniMentor, this stage is important because university information does not exist in one format or in one location. Some content is uploaded manually by authorized users, while other content can be collected through automated bots. The purpose of this stage is to bring these different sources into a form that the retrieval system can later search.

After a document is collected, the system must parse and clean its content. Parsing involves extracting readable text from the original file or web page. Cleaning removes unnecessary formatting, repeated spaces, irrelevant symbols, or other noise that may appear during extraction. This stage affects the whole RAG pipeline. If the extracted text is incomplete, badly formatted, or corrupted, the retrieval system may fail to locate the correct information later, even if the original document contained it.

The next step is document chunking. Many documents are too large to be given directly to a language model as one complete text. For this reason, they are divided into smaller passages called chunks. Each chunk needs to be large enough to keep useful meaning, but not so large that it brings too much unrelated information into the answer. This makes chunking an important design decision in RAG systems, because it can affect both retrieval precision and the quality of the generated response [4].

After chunking, every chunk is converted into an embedding. An embedding is a numerical representation of text that tries to capture its meaning. Texts with similar meaning usually have similar embedding representations, even when they are written with different words [12]. This is useful in an academic system because students may not use the same wording as the official university documents. A student may describe a problem informally, while the document may use a more formal or administrative expression.

The embeddings are then stored in a vector database together with metadata. This metadata can include the document name, subject, document type, chunk index, version, or other useful information. In practice, metadata helps the system keep retrieval results inside the correct academic context. For example, UniMentor can use metadata to retrieve content from a specific subject, document category, or active version. Vector databases are used for this purpose because they support efficient similarity search across large collections of embeddings [13].

The online query workflow begins when the user asks a question. The question is converted into an embedding using the same or a compatible embedding model. The system then compares this query embedding with the stored chunk embeddings and returns the chunks that are closest in meaning. Dense retrieval methods are useful in this step because they can select relevant passages based on semantic similarity and not only on exact keyword matching [14].

After the relevant chunks are retrieved, they are added to the prompt that is sent to the language model. The prompt normally contains the user's question, the retrieved passages, and instructions about how

the model should use that context. The model then produces a natural language answer based on the provided information. Depending on the system design, the answer may also include references or source details, so that the user can understand which material was used to support the response.

In UniMentor, this workflow is used to support the main goal of the platform. The student can ask a question in a natural way, but the system still searches the university-controlled knowledge base before producing the answer. This means that the student does not have to manually search through different websites, documents, announcements, or course pages. The assistant retrieves the relevant academic content and uses it to create a clearer response.

3.4 Embeddings and Vector Databases

Embeddings are an important part of modern RAG systems because they make semantic search possible. A traditional keyword search mainly looks for exact words or very close matches. This can be limiting in a university environment, because a student may describe a question differently from the way it is written in an official document. Embeddings help reduce this problem by converting text into a numerical representation that captures meaning, not only the exact words used [12].

For example, a student may ask, “What do I need to pass this course?” while the official document may use terms such as “assessment criteria”, “laboratory requirements”, or “minimum passing grade”. The wording is different, but the meaning is related. A vector-based retrieval system can identify this relationship better than a simple keyword search, because it compares the semantic similarity between the student’s question and the stored document chunks.

A vector database stores these embeddings and allows the system to search them efficiently. In a RAG system, the vector database works as the searchable memory of the application. When new documents are inserted or updated, their chunks are converted into embeddings and stored in the database. When a user asks a question, the system creates an embedding for the question and compares it with the stored vectors. The closest matches are then retrieved and used as context for the answer.

Metadata is also important because embeddings alone are not always enough. Without metadata, the system may retrieve content that is semantically similar but belongs to the wrong context. For example, two courses may use similar terminology, but the answer should come from the course that the student is actually asking about. The same issue can appear with old document versions. By storing metadata such as subject name, document name, document hash, content type, or version information, the system can filter results and keep the retrieved content more accurate for the specific case.

In UniMentor, this combination is useful because the platform manages different types of institutional content. Some content belongs to courses, some is managed by the secretariat, and some may be collected automatically from announcements or web pages through bots. Embeddings allow the system to search based on meaning, while metadata keeps the content organized and controlled. This means that retrieval is not only based on semantic similarity, but also on the academic context in which the information belongs.

3.5 Document Chunking and Context Retrieval

Document chunking is the step where large texts are divided into smaller passages before they are embedded and indexed. This is necessary because most documents are too large to be used as a whole within the context of a language model. Also, retrieving an entire document would usually bring too much unrelated information into the answer. A good chunk should therefore keep enough context to be meaningful, but it should also remain focused on a specific part of the document.

Chunking has a direct effect on retrieval quality. If the chunks are too small, important context may be lost. For example, one paragraph may refer to a requirement or regulation that was explained in the previous paragraph. If these parts are separated too aggressively, the retrieved chunk may not contain enough information for the model to answer correctly. On the other hand, if the chunks are too large, they may include several different topics together. This can make retrieval less precise and can pass unnecessary text to the language model.

Context retrieval is the process of choosing which chunks will be given to the language model for a specific question. In most RAG systems, the system retrieves the top-ranked chunks based on the similarity between the user's query and the stored embeddings. However, similarity alone is not always enough. The result must also belong to the correct subject, document type, version, or category. A chunk may be semantically close to the question, but still be wrong for the specific academic context if it comes from another course or from an outdated document.

For UniMentor, chunking and retrieval are especially important because university documents are not all written in the same way. A course document may include theory, exercises, grading rules, laboratory instructions, and bibliography in the same file. Administrative documents may include formal explanations, deadlines, exceptions, and step-by-step procedures. Announcements, on the other hand, may be shorter but more time-sensitive. Consequently, the system must process each type of content carefully, so that the retrieved chunks still contain the information needed for a useful answer.

Versioning is another important part of this process. Academic information can change over time, while older files may still exist in the system for technical, historical, or administrative reasons. A RAG system should avoid using an older chunk when a newer version of the same information is available. UniMentor supports this by connecting indexed chunks with document version metadata and retrieving only the active version when needed. This helps the assistant answer from the current knowledge base, instead of using material that may no longer be valid.

The quality of context retrieval also affects the risk of hallucination. When the system retrieves relevant and complete information, the language model has a better basis for producing an accurate answer. When retrieval fails or returns weak context, the model may try to fill the gaps using its internal knowledge, which can lead to incomplete or unsupported responses. For this reason, a RAG system should also include prompt instructions that tell the model not to answer confidently when the retrieved context is not enough.

3.6 Usefulness of RAG for UniMentor

RAG is useful for UniMentor because the platform is not trying to answer general questions only. Its purpose is to help with a practical university problem: students need information that is specific to

their institution, department, course, or current academic period. A general language model cannot provide this reliably by itself, because this type of information may not exist in its training data or may have changed. By using RAG, UniMentor connects the conversational interface with the university's own knowledge base.

One benefit of this approach is that it makes academic information easier to access. Students do not need to know the exact title of a document, the correct website section, or the formal administrative wording before they can start searching. They can ask a question in natural language, and the system can search the indexed knowledge base for semantically related content. This reduces the effort needed to locate information and makes the platform easier to use, especially when the student is not sure where the answer is stored.

Another important benefit is grounding. Since the answer is generated using retrieved university material, it can stay closer to the official sources that the platform has indexed. This matters in academic support because the answer may affect real student decisions. For example, a student may ask about exam participation, course requirements, document submission, or an administrative procedure. In these cases, the answer should not be based only on a general assumption from the model, but on material that belongs to the university knowledge base.

RAG also makes the system easier to maintain. University information changes over time, and retraining a language model every time a document is updated would not be realistic. In a RAG-based system, the knowledge base can be updated by adding, replacing, or re-indexing documents. After the new content is indexed, the retrieval process can use it in future answers. This makes the approach suitable for a university environment, where announcements, course files, schedules, and procedures may change during the academic year.

UniMentor also benefits from RAG because the same retrieval and generation process can be reused in different parts of the platform. The chatbot can retrieve context to answer student questions, while other AI-supported tools can use academic material to create structured text, summaries, or guidance. This means that RAG is not only used as a chatbot feature. It works as a general knowledge access layer that can support multiple functions of the platform.

Finally, RAG fits the controlled and role-based design of UniMentor. Since the indexed content is managed through the platform, authorized users can decide which documents are available, which subjects are included, and how updates enter the system. This is important because an academic assistant should not depend on unknown or uncontrolled sources. It should use information that is part of the institution's managed knowledge base and can be supervised by the appropriate users.

3.7 Advantages and Challenges of RAG Systems

RAG systems offer important advantages compared to standalone language models. One of the main advantages is grounding. Since the system retrieves relevant documents before generating the answer, the final response can be connected more closely to external evidence [3], [4]. This is especially useful in systems that answer knowledge-based questions, where the response should not depend only on the model's general knowledge.

Another advantage is that RAG can use updated or private information. The knowledge base may include documents that were never part of the language model's training data. In a university system,

this can include internal files, recent announcements, course-specific material, and administrative procedures. As long as this content can be processed, chunked, embedded, and indexed, it can become available to the retrieval system and used during answer generation.

RAG also helps with maintainability. In many cases, updating a knowledge base is more realistic than retraining or fine-tuning a language model whenever information changes. Documents can be inserted, replaced, or removed, and the retrieval system can then work with the updated content. This makes RAG practical for institutions where information changes regularly and where the system must remain aligned with current material.

At the same time, RAG introduces its own challenges. One of the most important is retrieval quality. If the system retrieves chunks that are irrelevant, incomplete, or taken from the wrong context, the generated answer may also be weak. For this reason, the quality of a RAG system depends strongly on the whole document pipeline, including parsing, chunking, embeddings, indexing, filtering, and ranking [4].

Another challenge is response time. A RAG system has to perform several steps before producing the final answer. It must embed the user's question, search the vector database, select the retrieved chunks, prepare the prompt, and then generate the response. These extra steps can make the system slower than a simple chatbot that answers directly. In a practical platform such as UniMentor, this means that the architecture must balance answer quality with acceptable performance.

Source quality is also a major issue. RAG can only retrieve information that exists in the knowledge base. If the inserted documents are outdated, incomplete, duplicated, or badly structured, the system may still produce poor answers. This is why the ingestion and administrative parts of UniMentor are important. They help keep the knowledge base organized, controlled, and updated, so that the retrieval process has better material to work with.

Despite these challenges, RAG remains a suitable architecture for UniMentor because it matches the main problem that the platform tries to solve. Students need to ask questions in natural language, but the answer must still be connected to reliable university information. By combining document ingestion, semantic retrieval, vector databases, metadata filtering, and language generation, RAG provides the technical foundation for an academic assistant that is flexible, maintainable, and grounded in institutional knowledge.

Chapter 4

Related Work

Retrieval-Augmented Generation (RAG) has become an important direction for educational chatbot systems because it allows language models to work with external and domain-specific knowledge sources. This is useful in education because the information students need is often not general. It may be connected to a specific course, department, regulation, academic period, or institutional procedure. Recent surveys show that RAG-based educational systems are being used in different ways, such as student support, course learning, question answering, content generation, and access to institutional information [15], [16].

The basic idea behind these systems is similar, even when the implementation changes. The language model is not left to answer only from its internal knowledge. Instead, the system first searches a controlled knowledge base and retrieves information that is relevant to the user's question. This knowledge base can include university documents, course material, regulations, frequently asked questions, announcements, website content, or other institutional data. The retrieved content is then given to the language model, so the final answer can be more contextual and better connected to the available sources.

This chapter presents related work on RAG-based chatbot systems in education and university environments. The goal is not only to describe existing systems, but also to explain how their use cases connect with UniMentor. The selected examples include university student support, administrative information retrieval, course-based learning assistance, feedback-supported educational assistants, and admission-related academic counseling. These examples are relevant because UniMentor also focuses on student support through a conversational interface, while also giving teachers and secretariat staff a way to manage and transfer academic knowledge more effectively.

4.1 RAG Chatbots for General University Student Support

One relevant example of a RAG-based academic assistant is the Unimib Assistant, which was designed for students of the University of Milano-Bicocca. The system was developed as a student-friendly chatbot that could answer university-related questions through a conversational interface. Its purpose was not simply to provide a generic chatbot experience, but to offer information and solutions based on the actual needs of students in that university environment [17].

The Unimib Assistant used Retrieval-Augmented Generation by specializing ChatGPT with university-

related documents and links. These sources were selected after a need-finding phase, where students were interviewed in order to understand what kind of information they expected from the assistant [17]. This detail is important because it shows that an academic chatbot should not be designed only from the technical side. It also has to reflect the real information needs and habits of the students who will use it.

The main use case of the system was general university question answering. A student could ask about services, procedures, locations, or other information related to university life. Instead of searching manually through different websites, documents, or pages, the student could write a question in natural language and receive an answer from the assistant. This type of interaction is useful because students do not always know where the information is stored, and they may also not know the official term they should use when searching.

The study reported that students valued the assistant for its conversational tone, structured answers, and ease of use [17]. This shows that the usefulness of an academic chatbot is not only measured by technical accuracy. The way the answer is presented also matters. In a university environment, official information is often written in formal language, and students may need it to be explained in a clearer and more direct way. For this reason, the user experience of the chatbot becomes part of the overall value of the system.

At the same time, the Unimib Assistant also showed some limitations. The study mentioned cases where the chatbot did not always provide fully accurate information, missed relevant content that existed in the uploaded material, or generated links that were not clickable or reliable [17]. These issues are important because they show that a RAG chatbot does not become reliable automatically just because documents are available. The quality of retrieval, the structure of the sources, and the way the final answer is generated all affect the result.

This example is closely related to UniMentor. Both systems try to make university information easier to access through a conversational interface. However, UniMentor extends this idea by treating the chatbot as part of a wider academic platform. In UniMentor, the knowledge base is not only uploaded once and then used passively. It is managed through an administrative environment and can be updated through manual document management as well as automated ingestion mechanisms. This is important because student support depends not only on the chatbot interface, but also on the continuous maintenance of the information behind it.

The Unimib Assistant is therefore a useful reference point for UniMentor. It shows that students can benefit from conversational tools that reduce the effort needed to find university information. It also shows that grounding, source handling, and answer reliability must be designed carefully. UniMentor follows the same general direction, but gives more emphasis to controlled knowledge management, role-based administration, and automated knowledge ingestion.

4.2 RAG Chatbots for Administrative and Institutional Information

Another important category of related work includes RAG chatbots that focus on administrative and institutional questions. Oreški and Vlahek presented the development of an AI chatbot for student

support using Retrieval-Augmented Generation with large language models. Their work focused on helping students find information about faculty rules, procedures, documents, and general institutional services [18].

This type of system is relevant because administrative information is often spread across many official sources. A student may need to understand exam rules, graduation requirements, financial aid procedures, transfer rules, application documents, or student obligations. In many cases, this information already exists on websites, in PDF regulations, or on official faculty pages. The problem is that availability alone does not make the information easy to use. Students may still struggle to find the correct document or understand the exact procedure they need to follow.

The chatbot described by Oreški and Vlahek was designed with both functional and non-functional requirements. The functional requirements included starting interaction with the user, offering predefined inquiry options, answering open-ended questions using documents and website content, maintaining conversation context, supporting multilingual interaction, and collecting user feedback [18]. The non-functional requirements included availability, acceptable response speed, clear answers, ease of use, and integration with the official faculty website [18].

A clear usage example from this work is a student asking an open-ended administrative question, such as how to transfer from another faculty. The chatbot searches the available knowledge base, including documents and faculty website content, and returns an answer based on the retrieved information [18]. This is close to the type of support that UniMentor aims to provide for secretary-managed information. The student should be able to ask a normal question and receive guidance based on official institutional material, without first knowing which document contains the answer.

The work also includes examples of predefined questions, such as questions about graduation requirements or enrollment in the following year [18]. This shows that a university chatbot does not have to support only one interaction style. Some students may prefer to choose from suggested questions, especially when they are not sure how to phrase their request. Others may already know what they need and may prefer to write an open-ended question directly. Supporting both cases can make the system easier to use for different types of users.

The implementation described in this work used a RAG component based on a knowledge base that included the faculty website and PDF regulations [18]. This shows a practical workflow for institutional information retrieval. Documents and web content are collected, processed, indexed, and then used by the chatbot during the conversation. This approach is useful because the chatbot is connected to official content, instead of depending only on the general knowledge of the language model.

This related work is important for UniMentor because the platform also needs to support administrative and institutional information. UniMentor includes secretary-managed content, announcements, and documents that may be relevant to student procedures. However, UniMentor extends this idea by adding an administrative environment where authorized users can manage the information used by the system. This allows secretariat staff and other authorized roles to participate directly in the process of maintaining and transferring academic knowledge.

The comparison also shows why automated ingestion is useful. In a university environment, administrative information may change often. New announcements may be published, older procedures may be replaced, and new documents may become available. If the chatbot depends only on manual updates, the knowledge base can become outdated over time. UniMentor addresses this issue by in-

troducing configurable bots that collect information from predefined institutional sources and send it for processing and indexing. In this way, the system supports both the student who needs the answer and the staff who are responsible for keeping the information available and updated.

4.3 RAG Chatbots for Course-Based Learning Support

RAG chatbots are also used to support students inside specific courses. Lang and Gürpinar studied the effectiveness of a RAG chatbot in a self-paced online R programming course. The chatbot was designed to answer student questions using the course material and to provide explanations and code examples that were connected to the course content [19].

This work is relevant because it shows that RAG can be used not only for general university information, but also as part of the learning process. Inside a course, students often need help with concepts, examples, exercises, and technical details. A general chatbot may still produce an answer, but the answer may not follow the material, structure, or expectations of that specific course. A RAG-based course assistant can reduce this issue by searching the actual course content before generating the response.

The main usage example in this study is a student asking questions about R programming topics. The chatbot can return conceptual explanations and code examples based on the course materials [19]. This is useful in self-paced or asynchronous learning, where the student may not have direct access to an instructor at the moment they need help. Instead of waiting for a reply, the student can ask the chatbot and receive support that is connected to the same material used in the course.

The study found that students mainly used the chatbot for help with more advanced topics and appreciated its ability to provide useful explanations based on the course material [19]. This shows that RAG chatbots can support more than simple factual questions. A student may ask for clarification, an example, or a simpler explanation of a difficult topic. In this case, the chatbot is not only pointing to where information exists, but helping the student understand the content more easily.

This is directly related to UniMentor because course material is one of the main knowledge categories of the platform. Students may ask questions about lecture notes, exercises, laboratory instructions, assignments, or course requirements. If UniMentor retrieves the relevant chunks from the selected course and provides them to the language model, the final answer can follow the material that the teacher has uploaded. This makes the assistant more useful than a generic chatbot that may explain the topic correctly in general, but not in the same way as the course expects.

Course-based RAG support is also helpful for students who cannot always attend every lecture or who need extra explanations while studying. In many cases, students do not only need to download a file. They need help understanding what a part of that file means, how it connects with an exercise, or how it applies to a question they have. A RAG chatbot can work as an access layer over the course material, helping students move from simply searching for documents to asking questions about their content.

However, this type of system also has limitations. If the course material is incomplete, outdated, or poorly structured, the chatbot may retrieve weak context. If chunks are too small, important explanations may be missing. If chunks are too large, the model may receive too much unrelated information. For this reason, the quality of the ingestion and indexing pipeline is important. UniMentor addresses this through document parsing, chunking, embeddings, and vector-based retrieval as part of its RAG

workflow.

The work of Lang and Gürpinar shows that RAG chatbots can support course learning when they are connected to the correct educational material. UniMentor follows this idea but applies it within a broader university platform. Instead of focusing on only one online course, UniMentor is designed to support multiple subjects and different types of academic knowledge. This makes knowledge management more demanding, but it also makes the platform more useful as a central academic assistant.

4.4 RAG Chatbots with Feedback and Instructor Involvement

A further relevant example is MARK, which stands for Machine Assistant with Reliable Knowledge. MARK is a RAG-based question-answering system designed to support student learning by producing accurate and context-aware answers. The system uses a curated knowledge base and combines retrieval with language generation, so that the answers remain more consistent with the available educational material [20].

One important part of MARK is that the chatbot is not treated as a system that works completely on its own. It includes feedback and improvement mechanisms. Students can rate the answers they receive, while instructors can review and correct responses when needed. These instructor corrections can then be added back into the retrieval corpus, allowing the system to improve over time [20]. This is useful because educational chatbots need supervision, especially when they are used to support learning and not only to provide general information.

A typical usage example is a student using the chatbot as a substitute or addition to office hours. The student asks a question about the course material, the system retrieves relevant content, and the language model produces an answer. If the answer is incomplete, unclear, or not fully correct, feedback can be used to improve future responses [20]. This creates a useful cycle between student questions, instructor review, and gradual improvement of the knowledge base.

MARK also uses a hybrid retrieval strategy that combines dense vector retrieval with sparse keyword-based retrieval [20]. This is important because educational questions are not all the same. Some questions are broad and depend mostly on meaning, while others depend on exact terms, formulas, commands, or named concepts. Dense retrieval helps when the question is phrased differently from the source material, while keyword-based retrieval is useful when precise terminology matters. This makes the design relevant for academic systems, where both semantic understanding and exact course language may be needed.

This example is connected to UniMentor because teachers are not only users who upload content. They are also part of the knowledge transfer process. In UniMentor, teachers can manage course-related material and keep the content that students use more organized. The idea of instructor involvement in MARK supports the same general direction. Educational AI systems should not remove teachers from the process. They should help teachers make their material easier to access, reuse, and maintain.

The feedback-based approach also shows an important limitation of RAG systems. Even when the system retrieves information from trusted documents, the final answer may still need correction. The retrieved context may not be enough, the model may misunderstand the student's question, or the response may not match the explanation that the teacher would prefer. Feedback mechanisms can help identify these problems and improve the system step by step.

For UniMentor, this suggests possible future extensions such as answer feedback, teacher review, or correction workflows. The current focus of the platform is on document ingestion, retrieval, and answer generation. However, feedback could later make the system more adaptive. Instead of only giving access to a static knowledge base, UniMentor could gradually improve based on student interactions and teacher corrections.

MARK is therefore relevant because it shows how RAG chatbots can become more reliable when humans remain involved in the process. This is important in education, where accuracy, explanation quality, and trust matter a lot. UniMentor follows a similar idea by giving authorized users control over the knowledge base and by using automated ingestion to make academic knowledge easier to maintain over time.

4.5 RAG Chatbots for Admissions and Academic Counseling

Another related example is URAG, a unified hybrid RAG framework developed for university admission chatbots. The work presents a case study at Ho Chi Minh City University of Technology and focuses on answering admission-related and academic counseling questions with higher precision [21]. This use case is important because admission information is usually detailed, complex, and sensitive to mistakes. A wrong answer in this context can affect how a prospective student understands their options or prepares their application.

Admission chatbots need to answer questions about programs, entry requirements, deadlines, procedures, eligibility, and academic pathways. These answers must be reliable because prospective students may use them when making real decisions about where and how to apply. The URAG work shows that educational question-answering systems need strong answer reliability, especially for critical queries. It also highlights that standard LLM-based chatbots may not be precise enough when they are not supported by suitable retrieval mechanisms [21].

A typical usage example is a prospective student asking a question about admission or academic counseling. The system retrieves information from university-specific data and then generates an answer based on that retrieved content. This is similar to the general goal of UniMentor, because both systems depend on institution-specific knowledge rather than only on the general knowledge of a language model. The difference is that UniMentor focuses mainly on current students, teachers, and secretariat staff, while URAG is centered on the admissions process.

URAG is also relevant because it uses a hybrid approach to improve answer precision. The system is designed to improve responses for critical questions, while still remaining realistic for implementation inside an educational institution [21]. This matters because university systems cannot be evaluated only as theoretical AI experiments. They also need to be deployable, maintainable, and usable within the practical limits of an institution.

This connects with UniMentor in several ways. First, both systems handle information that belongs to a specific institution. Second, both use RAG to reduce the weaknesses of standalone language models. Third, both are concerned with answer reliability, because students may use the information when making academic decisions. Finally, both examples show that educational chatbots become more useful when they are connected to controlled knowledge sources instead of depending only on open-ended model knowledge.

The main difference is the scope of the systems. URAG focuses on admission-related question answering, while UniMentor is designed as a broader academic support platform. UniMentor includes student support for academic information, course material, secretary-managed content, announcements, and knowledge transfer from teachers and administrative staff. It also includes automated ingestion through bots, which helps the platform keep the knowledge base updated from predefined institutional sources.

The URAG example therefore supports the argument that RAG is useful in university environments where accuracy and institutional specificity are important. It also shows that hybrid retrieval and practical deployment should be considered when designing real educational systems. For UniMentor, these points are useful because the platform has to support more than natural language interaction. It also has to handle document management, vector retrieval, automated ingestion, and role-based control over the knowledge used by the assistant.

4.6 Comparison with UniMentor

The related systems discussed in this chapter show that RAG chatbots are becoming more common in educational and university environments. The examples cover different needs, such as general student support, administrative information retrieval, course-based learning assistance, instructor-supported feedback, and admission counseling. Even though each system focuses on a different case, they all follow the same basic direction: language models become more useful for education when they are connected to trusted, specific, and updated knowledge sources.

Compared with these systems, UniMentor brings several of these ideas together in one platform. Like the Unimib Assistant, it helps students access university information through natural language. Like the student support chatbot developed by Oreški and Vlahek, it supports administrative and institutional information. Like the course chatbot studied by Lang and Gürpinar, it can work with course-related knowledge and learning material. Like MARK, it recognizes that teachers and controlled knowledge improvement are important. Like URAG, it treats accuracy and institution-specific retrieval as important requirements, especially when students may use the answers for academic decisions.

The main difference is that UniMentor is not designed around only one chatbot use case. It functions as a platform that connects students, teachers, and secretariat staff through a shared knowledge base. Students can ask questions and receive answers grounded in the available university material. Teachers and secretariat staff can manage information and transfer knowledge more efficiently. The platform also includes automated ingestion bots, which support the continuous collection and indexing of content from predefined institutional sources. Because of this, UniMentor is not only a conversational assistant. It also works as a knowledge management and academic support system.

The reviewed works also show the main challenges that UniMentor has to consider. These include retrieval quality, source reliability, outdated information, incomplete answers, feedback collection, and clear response presentation. A RAG architecture can reduce hallucinations and improve grounding, but it does not solve everything by itself. The final quality of the system still depends on the documents that are inserted, the indexing process, the retrieval mechanism, the prompt design, and the way authorized users maintain the knowledge base.

Overall, the related work supports the design direction followed in UniMentor. RAG-based chatbots are suitable for educational environments because they combine natural language interaction with access to controlled knowledge sources. UniMentor builds on this idea and applies it to a broader university platform. The goal is not only to answer student questions, but also to support the continuous transfer, maintenance, and accessibility of academic and institutional knowledge.

Chapter 5

Programming Languages and Technologies

The development of UniMentor required the use of several programming languages, frameworks, libraries, databases, runtime environments, and deployment tools. This was necessary because the platform does not cover only one technical area. It includes a frontend interface, backend API services, artificial intelligence integration, semantic search, document processing, browser automation, database management, and deployment configuration. Because of this, the technologies were selected based on the role they had to serve in the system, rather than forcing every part of the project into one single approach.

The technology choices were mainly based on practical needs such as compatibility, maintainability, performance, deployment control, and cost. Compatibility was important because the different parts of UniMentor had to communicate with each other without adding unnecessary complexity. Maintainability was also an integral criterion, as the platform includes different services and tools that need to remain organized and easy to extend. Performance had to be considered in areas such as the frontend, API execution, model inference, document retrieval, and vector search. Deployment control was also a key factor, especially for technologies that needed to run locally or inside the target server environment.

Some technologies were selected because they gave more control over execution and data handling. Ollama, Qdrant, and Tesseract can run locally, which is especially useful when privacy, independence from external services, and cost control are important. The trade-off is that local tools also require more manual setup, configuration, and maintenance. Docker and Docker Compose helped reduce this difficulty by making services easier to package, run, and reproduce across different environments.

5.1 Core Programming Languages and Runtime Environments

The basic technology stack of UniMentor starts with the programming languages and runtime environments used in the different parts of the system. The platform includes backend services, frontend development, document processing, AI-related communication, and browser automation. Because these parts have different needs, using only one programming environment for everything would not be practical. For this reason, Python, TypeScript, JavaScript, and Node.js were used where each one fitted better.

5.1.1 Python

Python is a high-level programming language that is widely used for backend development, automation, data processing, and artificial intelligence-related systems. Its syntax is readable and its ecosystem is large, which makes it helpful to projects that need to connect different operations without making the implementation unnecessarily complicated [22].

A main reason for using Python in UniMentor was its library support. The same language environment could be used for API development, database communication, document parsing, OCR processing, and communication with artificial intelligence services. Libraries such as FastAPI, Pydantic, SQLAlchemy, pypdf, python-docx, pytesseract, and HTTP clients made it possible to cover many server-side needs without moving each responsibility to a different stack.

This was essential for UniMentor as several backend processes had to work together. The API had to communicate with the database, the document processing logic had to extract text from uploaded files, and the AI-related parts had to send requests to model services. Python allowed these parts to be connected in one backend environment, using mature libraries that are already common in similar types of systems.

5.1.2 TypeScript and JavaScript

JavaScript is one of the main languages used for web development and is supported by all modern browsers. It is used to build dynamic web applications and to control the behavior of interactive pages. JavaScript is also used outside the browser through runtime environments such as Node.js, which makes it useful for tooling, scripts, and automation tasks [23].

TypeScript extends JavaScript by adding static typing. This allows the developer to define the expected structure of variables, function parameters, return values, and objects before the code runs. In UniMentor's frontend, this helped keep the codebase more organized because API responses, component props, application state, and reusable types could be described more clearly [24].

The two languages were used in connected but different ways. TypeScript gave more structure to the frontend application and helped reduce mistakes during development. JavaScript remained relevant considering that it is the base language of the web and is also used by many browser and automation tools. Together, they supported the parts of UniMentor that needed browser interaction, frontend logic, and development tooling.

5.1.3 Node.js

Node.js is a runtime environment that allows JavaScript to run outside the browser. It is commonly used for command-line tools, backend utilities, build processes, and automation scripts. Its asynchronous model is useful for tasks that involve network requests, file handling, and communication with external processes [25].

The browser automation part of UniMentor relied on Node.js because tools such as Puppeteer are designed to run in this environment. Modern websites depend heavily on JavaScript, so using a JavaScript runtime to control browser behavior made sense for this part of the system. It allowed the automation logic to interact with pages in a way that is close to how a real browser loads and executes them.

Node.js also provides access to libraries for browser control, HTML handling, logging, stream processing, and container interaction. For this reason, it was a better fit for the automation-related parts of the project. Python remained more suitable for the backend API, document processing, and AI service communication, while Node.js was used where browser-based automation was needed.

5.2 Infrastructure and Deployment Technologies

After the main programming environments, another important part of the stack is how the required services are executed, organized, and deployed. UniMentor depends on different components with different runtime needs, so the infrastructure had to provide a controlled way to run them. Docker, Docker Compose, and scripting tools helped make the setup more reproducible and reduced the amount of manual configuration needed during development and deployment.

5.2.1 Docker

Docker is a containerization platform that allows applications and services to run inside isolated environments called containers. Each container includes the dependencies and runtime configuration needed by the service it runs. This makes the behavior of the service more predictable, because it does not depend as much on the exact setup of the host machine.

One of the main reasons for using Docker is that it reduces differences between development and deployment environments. Instead of installing every dependency directly on the server, the container can describe what the service needs in order to run. This improves reproducibility and makes it easier to move services between machines without recreating the whole environment manually [26].

Docker also helped with the browser automation part of UniMentor. Browser automation workers may visit external pages and run browser-based processes, which can create stability or security concerns if they run directly on the host environment. By running these workers inside containers, their execution can be separated from the rest of the system. This also makes their dependencies easier to control and limits the way they interact with the host machine.

Another reason for using Docker is that some parts of the platform can be more demanding in terms of resources. Vector database services, local AI-related tools, and browser automation workers may require more CPU, memory, or storage compared to simpler backend services. Running them in separate containers makes it easier to isolate their execution and control their resource usage. UniMentor had to run in a server environment where resources should be managed carefully, therefore such a functionality was necessary.

Compared to fully managed cloud services, Docker gives more control over the execution environment and reduces dependence on external infrastructure. The trade-off is that configuration, monitoring, storage, resource limits, and service maintenance remain the responsibility of the developer or administrator.

5.2.2 Docker Compose

Docker Compose is a tool that allows multiple Docker containers to be defined and started from a single configuration file. While Docker can run individual containers, Docker Compose is more

practical when several services need to work together as part of the same application [27].

Since UniMentor utilizes vector databases, AI-related services, and supporting components, the capability for all of them to run at the same time and communicate with the backend application is vital. Docker Compose allowed services, networks, ports, volumes, and environment variables to be described in one place. In practice, this made the setup easier because each container did not have to be started and configured separately.

Docker Compose is simpler than larger orchestration systems such as Kubernetes, but this simplicity was useful for the scope of the project. It provided a practical way to manage a multi-service environment without adding the extra complexity of a full orchestration platform.

5.2.3 Bash, PowerShell and Makefile Tooling

Bash, PowerShell, and Makefile tooling were used to support repeated development and deployment tasks. These tools are not part of the main application logic, but they make the project easier to operate. They can be used for tasks such as starting services, preparing environments, running scripts, building the frontend, or executing deployment commands.

Bash is commonly used in Linux environments and is suitable for server-side scripts. PowerShell serves a similar purpose in Windows environments, which is useful during local development. Makefile tooling adds another layer of convenience by grouping common commands under named targets. This allows repeated tasks to be executed with shorter and more consistent commands.

At the operational level, these tools helped reduce manual repetition. They made common setup, execution, and deployment steps easier to reproduce, which is important in a project that includes several services and technologies.

5.3 Database Technologies

Different database technologies were needed since UniMentor stores different kinds of information. Some data has a clear structure and relationships, while other data is used for semantic search. MySQL was used for structured operational data, where tables, relationships, and consistency are important. Qdrant was used for vector data, where the main requirement is to search document embeddings based on similarity.

5.3.1 MySQL

MySQL is a relational database management system used to store structured data in tables. It organizes data through schemas, rows, columns, primary keys, foreign keys, and relationships between tables. This makes it suitable for information that has a clear structure and must remain consistent across the application. It also helps reduce problems such as duplicated data, missing relationships, or invalid records [28].

In UniMentor, MySQL made the structured part of the system easier to organize as the platform grew. By defining tables and relationships clearly, the application could manage different categories of information in a more controlled way. This was important because several backend services depend

on the same operational data, such as users, roles, subjects, documents, conversations, models, bots, and logs.

Using MySQL in a self-managed environment also gave more direct control over configuration, storage, and deployment. The trade-off is that maintenance, backups, security settings, and monitoring have to be handled by the administrator. In this project, MySQL was suitable because UniMentor needed a reliable relational database that could run inside the existing server environment and integrate with the backend stack without depending on an external managed database service.

5.3.2 Qdrant Vector Database

Qdrant is a vector database designed to store and search high-dimensional vector representations. It is different from a relational database because its main purpose is not to store structured records in tables, but to support similarity search. This makes it useful when the system needs to find text chunks that are close in meaning to a user's question [13], [29].

In a RAG system, text is converted into embeddings, which are numerical representations of meaning. These embeddings must be stored in a way that allows fast comparison between the user's question and the indexed document chunks. Qdrant supports this by storing vectors together with metadata and by allowing searches based on vector distance or similarity.

For this reason, Qdrant was appropriate for the semantic retrieval part of UniMentor. MySQL handled the structured application data, while Qdrant handled embedded document chunks and similarity search. Metadata filtering also made retrieval more controlled, because searches could be limited to a specific subject, document type, or other relevant category instead of searching everything without context [29].

Qdrant can also be self-hosted, which matched the need for deployment control in UniMentor. Compared to managed vector database services, this reduces dependence on external providers and gives more control over where the data is stored. The disadvantage is that configuration, storage management, performance tuning, and monitoring become part of the deployment and maintenance work.

5.4 Artificial Intelligence Provider Technologies

The artificial intelligence provider technologies gave UniMentor access to language and embedding models. These technologies do not store the main application data. Their role is to provide model execution, so that text can be generated, processed, or converted into vector representations. The stack included both local and cloud-based options, because the platform needed a balance between deployment control, performance, flexibility, and cost.

5.4.1 Ollama

Ollama is a platform that allows large language models and embedding models to run locally. Instead of using only cloud-based AI providers, Ollama makes it possible to execute supported models on the machine or server where it is installed [30].

The main benefit of Ollama is local execution. This can reduce dependence on external AI providers

and can also help control usage costs, because requests are handled by the local infrastructure instead of a paid hosted service. It also gives more direct control over which models are installed, tested, replaced, and used by the application.

The trade-off is that performance depends heavily on the available server hardware, especially CPU, memory, and GPU resources. Compared to managed providers, local models may be slower, harder to scale, and more sensitive to server configuration. In UniMentor, Ollama was useful in cases where local execution and deployment control were important, although self-hosted model execution also requires more setup and maintenance.

5.4.2 OpenAI API

The OpenAI API provides access to hosted artificial intelligence models through an external cloud service. In this case, the application does not run the model locally. It sends a request to the provider and receives the generated response or model output through the API [31].

This approach can provide stronger performance, faster responses, and access to more capable models than those that can realistically run on a local server. When local execution through Ollama is limited by hardware resources, model quality, or response speed, the OpenAI API can be used as an alternative provider.

The disadvantage is that the OpenAI API depends on an external service, requires API credentials, and introduces usage costs. Since requests leave the local environment, privacy and data handling also need to be considered. Even with these limitations, it was useful as a faster and more capable option when local execution was not the most efficient solution for a specific use case.

5.4.3 Embedding Models

Embedding models convert text into numerical vector representations. These vectors capture semantic information, so pieces of text with similar meaning can be placed closer together in vector space [12].

Their role in UniMentor was to prepare text for similarity-based retrieval. After documents are parsed and divided into smaller chunks, each chunk can be converted into an embedding. These embeddings are then stored in Qdrant and later compared with the embedding of the user's question [12], [13].

Embeddings connect the AI provider layer with the vector database layer. They are not a database by themselves, but they define the format that makes semantic retrieval possible. This allowed UniMentor to compare text based on meaning instead of relying only on exact keyword matching.

5.5 API Backend Technologies

The API backend is the part of the system that exposes the main application functionality through web endpoints. It receives requests from the frontend and connects them with the internal services of the platform. This includes database operations, authentication logic, file handling, conversation management, administration features, and AI-related processes. For this layer, the selected technologies had to support clear endpoint structure, request validation, database communication, and asynchronous operations.

5.5.1 FastAPI and Uvicorn

FastAPI is a Python web framework used for building APIs. It is based on modern Python features, such as type hints, asynchronous functions, and automatic request validation. These characteristics made it suitable for UniMentor, because the backend had to expose structured endpoints and communicate with several different services [32].

FastAPI allows routes, request parameters, request bodies, and response structures to be defined clearly through Python types and schema models. This reduced the amount of manual validation code needed in the endpoints. It also made the communication between frontend and backend easier to follow, because the expected structure of each request and response could be described more directly.

The API layer had to support several responsibilities, such as authentication checks, document management, conversation handling, streaming responses, model selection, and administrative actions. FastAPI's support for asynchronous execution was useful for operations that wait for external services, such as AI providers, embedding services, or vector database calls.

Uvicorn is the Asynchronous Server Gateway Interface (ASGI) server used to run FastAPI applications. FastAPI defines the application logic and the API routes, while Uvicorn serves the application and handles incoming HTTP requests. Together, they provided a lightweight and efficient foundation for the backend API.

5.5.2 Pydantic

Pydantic is a Python library used for data validation and schema definition. It allows developers to define models that describe the expected structure and types of data. When data is received by the application, Pydantic can validate it and convert it into the correct Python objects when possible [33].

This was chosen as many UniMentor endpoints exchange structured data with the frontend. Request bodies, response objects, form data, and configuration values need to follow specific formats. Pydantic made the contract between frontend and backend clearer and reduced the need for repeated manual checks inside the endpoint logic.

Pydantic was also useful for configuration management through typed settings loaded from environment variables. This was needed because the backend depends on several external services and configuration values, such as database connections, AI provider settings, authentication parameters, and vector database configuration.

5.5.3 SQLAlchemy and PyMySQL

SQLAlchemy is a Python library used for database interaction through object-relational mapping. Instead of writing every SQL query manually, the developer can define Python classes that represent database tables. These models can then be used to create, read, update, and delete records through a consistent programming interface [34].

The UniMentor backend relied on SQLAlchemy because a large part of the application state is stored in MySQL. Entities such as users, sessions, conversations, messages, subjects, documents, document queue entries, bots, and logs are structured as relational data. SQLAlchemy made it possible to represent these tables as Python models and work with them from the API layer in a more organized

way.

PyMySQL was the database driver that allowed the Python application to communicate with MySQL. SQLAlchemy provided the higher-level ORM structure, while PyMySQL handled the actual connection between the backend and the MySQL database. This combination allowed the backend to use MySQL while keeping the database interaction integrated with the Python codebase.

5.5.4 API Backend Supporting Libraries

Several smaller libraries were also used to support the backend with practical tasks around HTTP communication, configuration, JSON handling, and file uploads. HTTPX was used as an HTTP client library, allowing the backend to communicate with AI providers, embedding services, and other APIs. Its support for asynchronous requests made it suitable for operations where the backend has to wait for a response from another service.

Python-dotenv was used to load environment variables from configuration files, so sensitive or environment-specific values did not have to be written directly inside the source code. Orjson was used for fast JSON serialization and deserialization, which is useful because the API exchanges data in JSON format. Python-multipart supported file upload handling, allowing FastAPI to receive and process uploaded files correctly.

Together, these supporting libraries helped cover specific backend requirements without changing the main architecture. They handled common operational needs, while the main API logic remained focused on the behavior of the UniMentor platform.

5.6 Document Processing and OCR Technologies

Document processing technologies were needed because a large part of the information used by UniMentor comes from files, not only from manually written database entries. These files can have different formats, such as PDFs, Word documents, PowerPoint presentations, Excel files, CSV files, and plain text. Because of this, the system needed libraries that could read these formats and extract usable text before the content could be cleaned, chunked, embedded, and indexed.

5.6.1 PDF, Office, CSV, and Text Parsing

File parsing is the process of reading a document and extracting its textual content in a form that can be processed by software. Most documents are created mainly for human reading and presentation, not for direct use in an automated pipeline. A PDF may contain selectable text, images, tables, or scanned pages, while Office files may contain paragraphs, slides, spreadsheet cells, or structured tables.

For PDF documents, pypdf was used to extract text from files that already include selectable text. For Microsoft Office formats, python-docx supported Word documents, python-pptx supported PowerPoint presentations, and openpyxl supported Excel files. These libraries made it possible to access the internal structure of each format and extract information from paragraphs, slides, cells, and tables [35], [36], [37], [38].

CSV and plain text files required simpler handling because their content is already stored in a more

direct textual form. Even in these cases, parsing was still needed so that the extracted content could be normalized and prepared for the next stages of the pipeline.

Using multiple parsing libraries was necessary because no single tool handles every document format correctly. Each format stores information in a different way, so specialized libraries were needed to improve compatibility with the different file types supported by UniMentor.

5.6.2 OCR Technologies

Optical Character Recognition, commonly referred to as OCR, is the process of converting text that appears inside images into machine-readable text. This is needed when a document does not contain selectable digital text, such as scanned PDFs or photographed pages [39].

Tesseract OCR was used as the main OCR engine. It is an open-source OCR technology that can recognize text in images and convert it into textual output. Pytesseract provided the Python interface to Tesseract, while pdf2image was used to convert PDF pages into images when OCR processing was required [39].

This combination allowed the system to support documents that would otherwise return little or no extracted text. When normal PDF extraction was not enough, the pages could be converted into images and then processed with OCR.

Tesseract also matched the need for deployment control and cost practicality. Compared to commercial OCR services, it can run locally and does not require every document to be sent to an external provider. The trade-off is that local OCR requires system configuration, language packs, and image conversion tools, while the final accuracy also depends on the quality of the scanned document.

5.7 Automation and Browser Interaction Technologies

Automation and browser interaction technologies were used when information had to be collected from web-based sources and simple file input or standard API access was not enough. Some pages can be parsed directly from their HTML, while others need to be rendered in a browser first because their content is loaded dynamically. For this reason, the automation stack had to support both full browser interaction and lighter HTML parsing.

5.7.1 Puppeteer Core and Browser Management

Puppeteer Core is a Node.js library that allows Chromium-based browsers to be controlled through code. With Puppeteer, a script can open pages, visit URLs, wait for content to load, interact with page elements, and extract information from the rendered page [40].

This was necessary for web pages that depend on JavaScript execution, client-side rendering, or interactions that are closer to normal browser use. In these cases, the required content may not exist in the raw HTML returned by a simple request. The page first has to run inside a browser environment before the information can be accessed correctly.

Puppeteer Core does not include a bundled browser installation by default. This provides flexibility because the browser version can be managed separately. Browser management was supported

by @puppeteer/browsers, which helps install and manage the browser binaries used by Puppeteer. Together, these tools made browser automation more predictable across different environments.

5.7.2 puppeteer-extra and Stealth Plugin

Puppeteer-extra is a wrapper around Puppeteer that allows plugins to be added to browser automation scripts. It extends Puppeteer without changing the core automation library directly, which makes it useful when extra browser behavior or configuration is needed.

The stealth plugin tries to reduce some of the visible differences between an automated browser and a normal browser session. This can help in cases where a website behaves differently when it detects automation. Its purpose is not to modify the content of the page, but to make the browser session closer to a normal browsing environment.

This type of tooling needs to be used with care. In UniMentor, its role is to support reliable access to web content as part of the automation process, while still respecting the rules, restrictions, and intended access limits of the target website.

5.7.3 Cheerio

Cheerio is a lightweight HTML parsing library for Node.js. It allows HTML content to be loaded and queried with a syntax similar to working with the Document Object Model, but without opening or controlling a real browser [41].

It was used because not every page needs full browser automation. If a page is mostly static and the required content is already available in the returned HTML, launching a Chromium browser would add unnecessary overhead. In these cases, Cheerio provides a faster and simpler way to extract the needed information.

The distinction between Cheerio and Puppeteer made the automation layer more efficient. Puppeteer was suitable for pages that required rendering, navigation, waiting, or interaction. Cheerio was better for pages where the content could be read directly from the HTML response. This allowed the system to avoid using heavier browser automation when a simpler parsing method was enough.

5.7.4 Dockerode and Automation Supporting Libraries

Dockerode is a Node.js library used to interact with the Docker API. It allows scripts to communicate with the Docker engine, manage containers, and perform container-related operations programmatically. This helped with browser automation workers, enabling their execution inside containerized environments instead of being treated only as local scripts.

Browser automation can create stability and security concerns because it involves opening browser instances and interacting with external web pages. By connecting the automation logic with Docker, workers can be managed in isolated environments. This reduces direct exposure of the host system and makes the execution of automation tasks easier to control.

Other supporting libraries handled smaller but necessary tasks. Mysql2 provided MySQL connectivity from the Node.js side of the stack, tar-fs supported tar archive handling, is-stream helped identify

stream objects, and content-disposition supported file metadata handling from HTTP responses. Pino and pino-pretty were used for structured and readable logging.

These libraries did not define the main automation logic by themselves. Their role was to support the surrounding operations needed for browser automation to work as part of a larger system, including database access, container handling, file processing, HTTP metadata handling, and logging.

5.8 Frontend Development Technologies

Frontend development technologies were used to build the part of UniMentor that users interact with directly. This layer had to support interactive pages, reusable interface elements, client-side routing, formatted content rendering, and communication with the backend API. For this reason, the frontend stack was based on technologies that support component-based development, fast development iterations, consistent styling, and organized communication with backend services.

5.8.1 React

React is a JavaScript library used for building user interfaces with reusable components. Instead of writing each page as one large block of HTML and JavaScript, React allows the interface to be separated into smaller parts. Each component can manage its own structure, properties, and state, which makes the frontend easier to organize as the application grows [42].

This approach was suitable for UniMentor because the interface is not made of static pages only. The frontend loads data from the API, reacts to user input, updates conversations, uploads files, displays generated responses, and manages administrative information through interactive views. React provided a clear way to organize these behaviors inside reusable components instead of repeating the same logic across different pages.

React DOM was also part of the frontend stack because it is responsible for rendering React components in the browser. React defines how the components are structured, while React DOM connects those components with the actual web page that the user sees.

5.8.2 Vite

Vite is a frontend build tool and development server for modern JavaScript and TypeScript applications. It provides fast startup during development, quick updates while coding, and optimized builds for production [43].

Vite was appropriate because UniMentor's frontend was built with React and TypeScript, which are directly supported by the tool. It also made the build configuration simpler by supporting development and production settings, path aliases, environment variables, and production optimization from one place.

This helped both development speed and maintainability. The frontend could be tested and updated quickly during development, while the final version could be built into static files and deployed to the server.

5.8.3 Tailwind CSS

Tailwind CSS is a utility-first CSS framework used for styling web interfaces. Instead of creating many separate custom CSS classes, Tailwind provides small utility classes for spacing, layout, typography, colors, borders, and responsive behavior [44].

This worked well with a component-based frontend. Each component could include its structure and most of its styling rules in the same place, which made interface changes faster during development. This was practical for UniMentor because the frontend includes dashboards, forms, dialogs, tables, cards, and responsive layouts.

Tailwind CSS also helped keep the interface visually consistent. By reusing the same utility classes across components, the project avoided large separate style sheets and made layout changes more controlled.

5.8.4 Radix UI and Local UI Components

Radix UI is a collection of low-level interface primitives for React. It provides accessible building blocks for common interface patterns such as dialogs, dropdown menus, tabs, scroll areas, select menus, switches, checkboxes, labels, and tooltips [45].

The frontend used local reusable UI components based on Radix UI primitives. These components were stored inside the project codebase instead of being used only as a separate prebuilt design system. This way, common interface elements could be reused, while still keeping direct control over their structure and styling.

Supporting libraries such as class-variance-authority, clsx, and tailwind-merge were used together with these components. They helped define reusable styling variants, combine conditional classes, and resolve conflicts between Tailwind CSS classes. This improved consistency across repeated interface elements such as buttons, cards, dialogs, forms, and other UI patterns.

5.8.5 Frontend Supporting Libraries

Several additional frontend libraries supported routing, formatted text rendering, icons, date handling, and export functionality. React Router was used for client-side routing, allowing the frontend to show different pages without forcing a full page reload.

React Markdown and remark-gfm were used to render Markdown content in the interface, including paragraphs, lists, tables, links, and code blocks. React-syntax-highlighter improved the display of code blocks by adding syntax highlighting, which made generated or displayed code easier to read.

Lucide React provided the icon set used across the interface, helping navigation, buttons, and status indicators remain visually consistent. The jsPDF library supported PDF export functionality, allowing displayed or generated content to be saved as a PDF document. Date-related functionality was supported by react-day-picker and date-fns, which helped with calendar elements, date formatting, and date manipulation.

Together, these supporting libraries covered specific frontend needs without changing the main architecture. React provided the component model, Vite handled development and builds, Tailwind CSS

handled styling, Radix-based local components provided reusable interface elements, and the supporting libraries added routing, Markdown rendering, icons, export functionality, and date handling.

Chapter 6

Design and Implementation of UniMentor

6.1 Functional Requirements

The functional requirements of UniMentor were defined based on the main roles that use the platform and the actions that each role needs to perform. UniMentor was not designed only as a chatbot. It was designed as a university support platform that combines chat interaction, study tools, knowledge base management, document ingestion, model configuration, and automated data collection.

The main human roles of the platform are students, teachers, and secretaries. Students use the learning and assistance features of the system. Teachers and secretaries can also use these same features, but they have extra access to the Admin Panel depending on their responsibilities. Automated bots are also part of the system, but they are not normal users of the UniMentor interface. They work in the background and collect information from external university systems, so that this information can later be stored, processed, and indexed.

The requirements are presented through user stories. This format was useful because it connects each requirement with the role that needs it and with the reason the feature exists. In this way, the requirements are not shown only as technical functions, but also as actions that support the real use of the platform.

Role	User Story
Student	As a student, I want to use AthenaAI to ask questions, so that I can receive support based on university and course-related information.
Student	As a student, I want to manage my chat conversations, so that I can continue previous discussions, pin important conversations, and delete conversations that are no longer needed.
Student	As a student, I want chatbot answers to support Markdown formatting and copyable code snippets, so that technical responses are easier to read and reuse.
Student	As a student, I want to use AInsights by selecting a course and a template, so that I can generate structured study material based on the course knowledge base.
Student	As a student, I want to optionally provide a custom prompt in AInsights, so that the generated material can be adjusted to my specific needs.
Student	As a student, I want to download generated insights as PDF files, so that I can keep them for offline study.
Student	As a student, I want to use AI-based filtering for job positions, so that I can describe my skills and receive more relevant results.
Teacher	As a teacher, I want to access the Admin Panel for my own courses, so that I can manage the course material available in the platform.
Teacher	As a teacher, I want to upload actual files for existing course documents, so that they can be added to the parsing and indexing pipeline.
Teacher	As a teacher, I want to view the processing status of course documents, so that I can know whether a document is missing, processing, or processed.
Teacher	As a teacher, I want to configure Moodle document collection bots for my courses, so that course files can be collected automatically when needed.
Teacher	As a teacher, I want to manage LLM configuration records, so that the platform can offer different active models for its AI features.
Secretary	As a secretary, I want to manage secretary-related subjects and documents, so that institutional information can be available through UniMentor.
Secretary	As a secretary, I want to manually upload files for secretary-related documents, so that they can be inserted into the ingestion pipeline.
Secretary	As a secretary, I want to view the available LLM configurations, so that I can provide basic information about the system when needed.
System	As the system, I want to collect course, teacher, document, university, and announcement information automatically, so that the knowledge base can stay updated with less manual work.

Table 6.1: Functional requirements expressed as user stories

For students, the main requirement is access to the user-facing AI features of UniMentor. Through AthenaAI, students can ask questions and receive answers that can be supported by retrieved university or course-related material. The chat also keeps conversation history. This is important because previous questions and answers can remain available, and the chatbot can use the earlier parts of a discussion when the user continues the same conversation. Students can create new conversations, open previous ones, pin important conversations, and delete conversations that are no longer needed.

AthenaAI also needs to present its answers in a readable way. Since students may ask technical questions, especially for programming or course-related topics, plain text is not always enough. For this reason, the chatbot supports Markdown formatting. This allows answers to include structured text, lists, explanations, and code blocks. Code snippets can also be copied more easily, which makes the chat more practical for study and technical support.

AInsights is another important student feature. It is designed to generate curated study material from predefined templates. The student selects a course and then chooses a template, such as a theory summary or another type of educational output. The system then retrieves relevant context from the selected course before generating the final result. This is useful because the answer is not produced only from the general knowledge of the language model. It is also connected with the indexed material that exists for that course. The student can also provide an optional custom prompt, which gives more control over the final output. After the material is generated, it can be downloaded as a PDF and used later for offline study.

Career Mentor is the third main student-facing part of the platform. Its purpose is to help students explore job opportunities. Students can view available positions, filter them with normal filtering options, open the details of a position, and navigate to the original page in order to apply. The platform also includes AI-based filtering. In this case, the student can describe their skills, interests, or experience in a custom prompt, and the system filters the available positions based on that description. This feature does not use RAG, because it is based on matching the user's description with the existing job position data. In the current implementation, the frontend and backend logic for Career Mentor are ready, but the connection with a third-party job positions API has not been completed. For demonstration purposes, sample data are used.

Across AthenaAI, AInsights, and Career Mentor, users can select which active LLM will be used. The available models are not written directly inside the frontend code. They are stored as configuration records, and only the active models are shown to the user. This makes the system easier to maintain, because different models or providers can be added without changing the main user interface each time.

Teachers can use all student-facing features, but they also have access to the Admin Panel. Their administrative access is focused on the courses assigned to them. In the Knowledge Base tab, teachers can view their own courses and the documents connected with them. They can filter the documents, check their processing status, and manually upload the actual file for an existing document. When a file is uploaded, it is added to the queue so that it can be parsed and later indexed. Teachers cannot create, delete, or rename courses and subject names. This information is not managed manually by them, because it is generated by the default Moodle synchronization bot.

Teachers also have access to the Models tab. In this part of the Admin Panel, they can view, create, update, delete, and activate or deactivate LLM configuration records. This requirement exists because

UniMentor must be able to control which models are available for chat, insight generation, and AI filtering. Instead of changing the source code whenever a model changes, the platform stores the model configuration in the relational database and exposes the active models to the frontend.

The Bots tab is also available to teachers. Through this tab, teachers can configure Moodle document collection bots for their courses. These bots collect the actual course files from Moodle and send them to the backend API. From there, the files are enqueued for parsing and indexing. This means that teachers have two ways to populate course material in the platform. They can upload files manually, or they can configure a bot to collect them automatically. To keep this controlled, each course can have at most one one-time bot and one repeat-interval bot.

Secretaries can also use all student-facing features and have access to the Admin Panel, but their permissions differ from those of the teachers. They can see the Knowledge Base and Models tabs, but they do not have access to the Bots tab. In the Knowledge Base area, secretaries can view secretary-related subjects and documents. They can add new secretary-related records when needed, delete records that are outdated or incorrect, and manually upload files so that they can be inserted into the ingestion pipeline. Updating names was not included as a separate requirement, because in most cases these records are either created automatically by system bots or can be recreated manually when needed.

In the Models tab, secretaries have read-only access. They can view the available model configuration records, but they cannot create, update, delete, activate, or deactivate models. This was included mainly for information purposes. For example, if students ask which models are available in the system, secretaries can view the relevant information, but the actual management of model records remains a teacher responsibility.

The final group of requirements concerns the automated bots. These bots are not users of the frontend. They are background workers that help keep the knowledge base updated. One predefined bot synchronizes information from Moodle by collecting course names, document names, and the teachers assigned to each course. This information is sent to the backend and is used to populate the subjects, documents, and teaches course records. Another predefined bot collects information from the university website, such as section titles, document names, and document content. A third predefined bot connects to the announcement system, retrieves recent announcements, and sends their information and content to the platform. These bots reduce the amount of manual work needed and make the knowledge base easier to maintain.

Overall, the functional requirements show that UniMentor is built around both direct user interaction and background automation. Students need simple access to AI assistance, study material generation, and career support. Teachers need controlled access to course material management, model configuration, and document collection bots. Secretaries need a more limited administrative area for institutional information. At the same time, automated bots support the platform by collecting and preparing information that can later be used by the ingestion and RAG pipeline.

6.2 Architecture

The architecture of UniMentor is based on separating the main parts of the system according to their role. The platform has a frontend for user interaction, a backend API for the main logic, a relational

database for structured data, a vector database for semantic search, an ingestion worker for document processing, AI providers for response generation, and bots for collecting data from external university systems. Figure 6.1 shows the main components and how they communicate with each other.

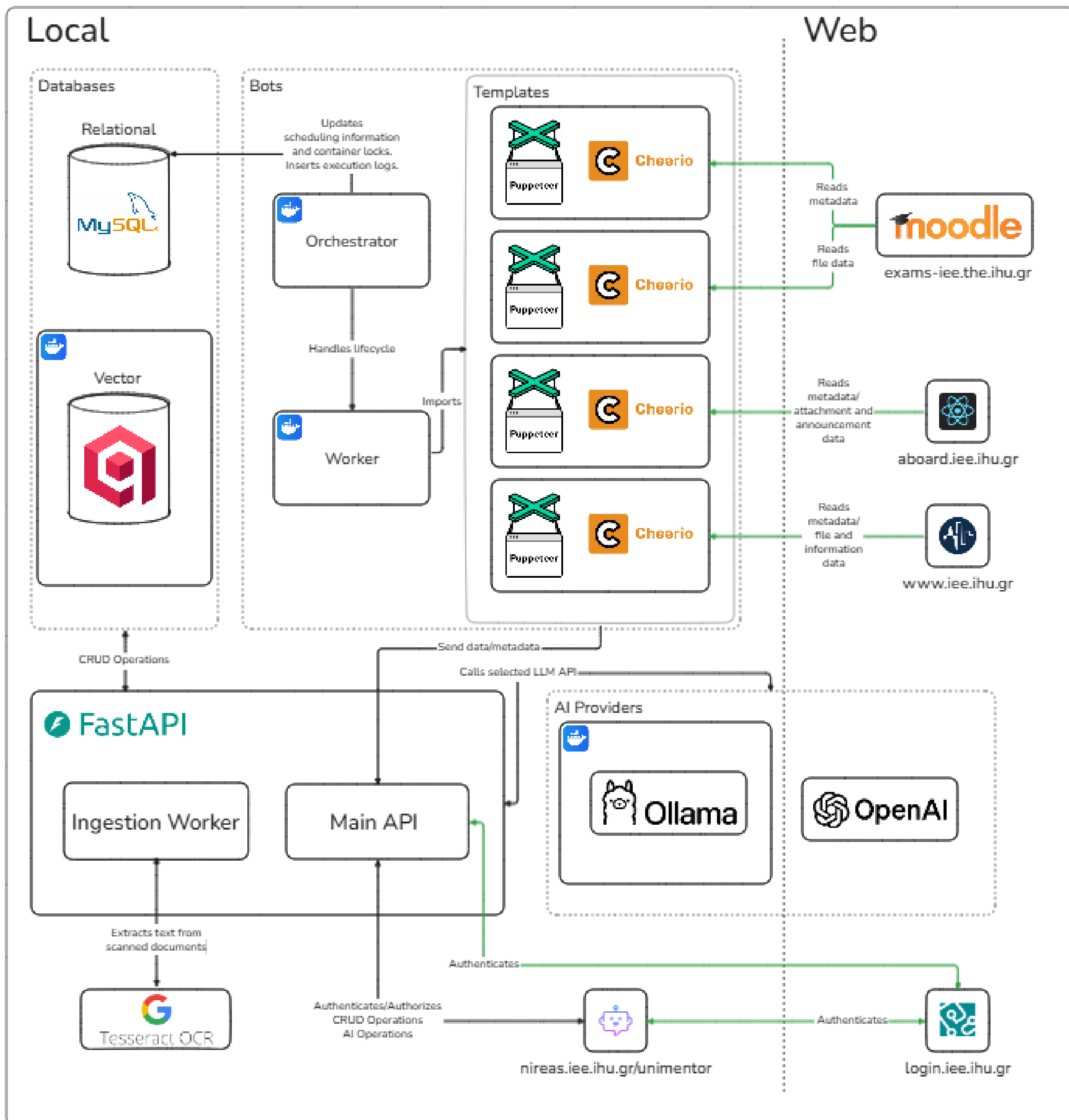

Figure 6.1: General architecture of UniMentor

The frontend is the main entry point of the platform. It is served through the university server and is used by students, teachers, and secretaries. Through the frontend, users can access AthenaAI, AInsights, Career Mentor, and, depending on their role, the Admin Panel. The frontend does not connect

directly to the databases, the vector database, or the AI providers. All requests go through the backend API. This makes the system easier to control, because permissions, database operations, retrieval, and AI calls are handled from one central point.

Authentication is connected with the university login system. After the user logs in, the backend can identify the user and apply the correct permissions. This is needed because the same platform is used by different types of users. Students can use the main AI tools, teachers can also manage course-related data, and secretaries can manage secretary-related information. For this reason, access control is not handled only by the interface. The backend also also verifies each user's permissions.

The backend is implemented with FastAPI and works as the main coordinator of the platform. It receives the requests from the frontend, checks the role of the user, communicates with the required database or service, and returns the response. The Main API handles the normal operations of the system, such as chat requests, AInsights generation, Career Mentor filtering, conversation management, model configuration, knowledge base actions, and bot configuration.

UniMentor uses two different databases because the platform needs to store two different types of data. MySQL is used as the relational database. It stores structured data such as users, conversations, messages, subjects, documents, document queue records, teacher-course relationships, LLM records, bot configurations, and bot logs. This data has clear relationships, so it makes sense to store it in relational tables with primary keys, foreign keys, and constraints.

Qdrant is used as the vector database. Its role is different from MySQL. It stores the vector representations of document chunks, which are used for semantic search. After a document is processed, its text is split into smaller chunks and each chunk is converted into an embedding. These embeddings are inserted into Qdrant. Later, when AthenaAI or AInsights needs relevant context, the backend searches Qdrant and retrieves the chunks that are closest to the user's request.

This separation between MySQL and Qdrant was important for the design. MySQL keeps the structured state of the platform, while Qdrant supports the retrieval part of the RAG pipeline. For example, MySQL can store that a document belongs to a specific subject and whether its status is missing, processing, or processed. Qdrant stores the searchable vector chunks that were created from the content of that document. In this way, each database is used for the task it fits better.

Document processing is handled by the ingestion worker. When a teacher or secretary uploads a file, or when a bot sends a file to the backend, the file is not processed directly inside the same request. Instead, it is added to the document queue. The ingestion worker then consumes queued documents, extracts their text, uses Tesseract OCR when scanned content needs to be handled, splits the text into chunks, creates embeddings, and stores them in Qdrant. During this process, the document status is also updated in MySQL, so that users can see the current state of each document from the Admin Panel.

The backend also communicates with AI providers. UniMentor can use local models through Ollama or external models through OpenAI. When a response needs to be generated, the backend calls the selected LLM provider API. The available models are stored as configuration records in the relational database, and only active models are shown in the frontend. This makes the system more flexible, because new models can be added or disabled without changing the main frontend code.

For the RAG-based features, the backend does not send the user's request directly to the language

model. First, it retrieves useful context from Qdrant. In AthenaAI, this retrieval is based on the user's question and the available indexed information. In AInsights, the retrieval is also filtered by the selected course before similarity search is applied. This makes the generated result more focused, because the context comes from the selected course and not from unrelated documents. After the context is retrieved, the backend builds the prompt, calls the selected AI provider, and returns the generated response to the frontend.

The bot system works separately from the normal user flow. Its purpose is to collect information from external university systems and send it to UniMentor. The bot part of the architecture includes an orchestrator, a worker, and several templates. The orchestrator handles scheduling information, execution timing, and container locks. The worker handles the execution lifecycle and imports the correct template. The templates contain the actual logic for each external source.

Some bot templates are predefined system automations. One template connects to Moodle and collects course names, document names, and teacher-course relationships. This information is sent to the backend and is used to populate the relational database. Another template collects information from the university website, such as section titles, file information, and page content. A third template connects to the announcement system and retrieves recent announcements. These automations reduce manual work because the platform can collect important information from existing university systems.

There is also a teacher-configurable bot template. This template collects actual course files from Moodle. A teacher can configure it for a course as a one-time or a repeated execution. When the bot collects files, it sends them to the backend API, where they are added to the same queue used by manual uploads. This means that manually uploaded files and files collected automatically follow the same ingestion process.

The bots use Puppeteer and Cheerio depending on the type of source they need to read. Puppeteer is used when browser automation is needed, for example when login or dynamic page interaction is required. Cheerio is used when the information can be extracted from HTML content in a lighter way. The external systems act only as data sources. UniMentor receives the collected metadata and files, stores the needed information, and then processes the files through the same backend and ingestion pipeline.

Overall, the architecture keeps each part of UniMentor responsible for a specific task. The frontend handles user interaction. The FastAPI backend handles requests, permissions, retrieval, database operations, and AI provider calls. MySQL stores structured data, while Qdrant stores vector chunks for semantic retrieval. The ingestion worker prepares documents for the RAG pipeline, and the bot system collects information from external university systems. This structure makes the platform easier to develop and extend because new models, new document sources, or new features can be added without changing the whole system.

6.3 Relational Database Design and Development

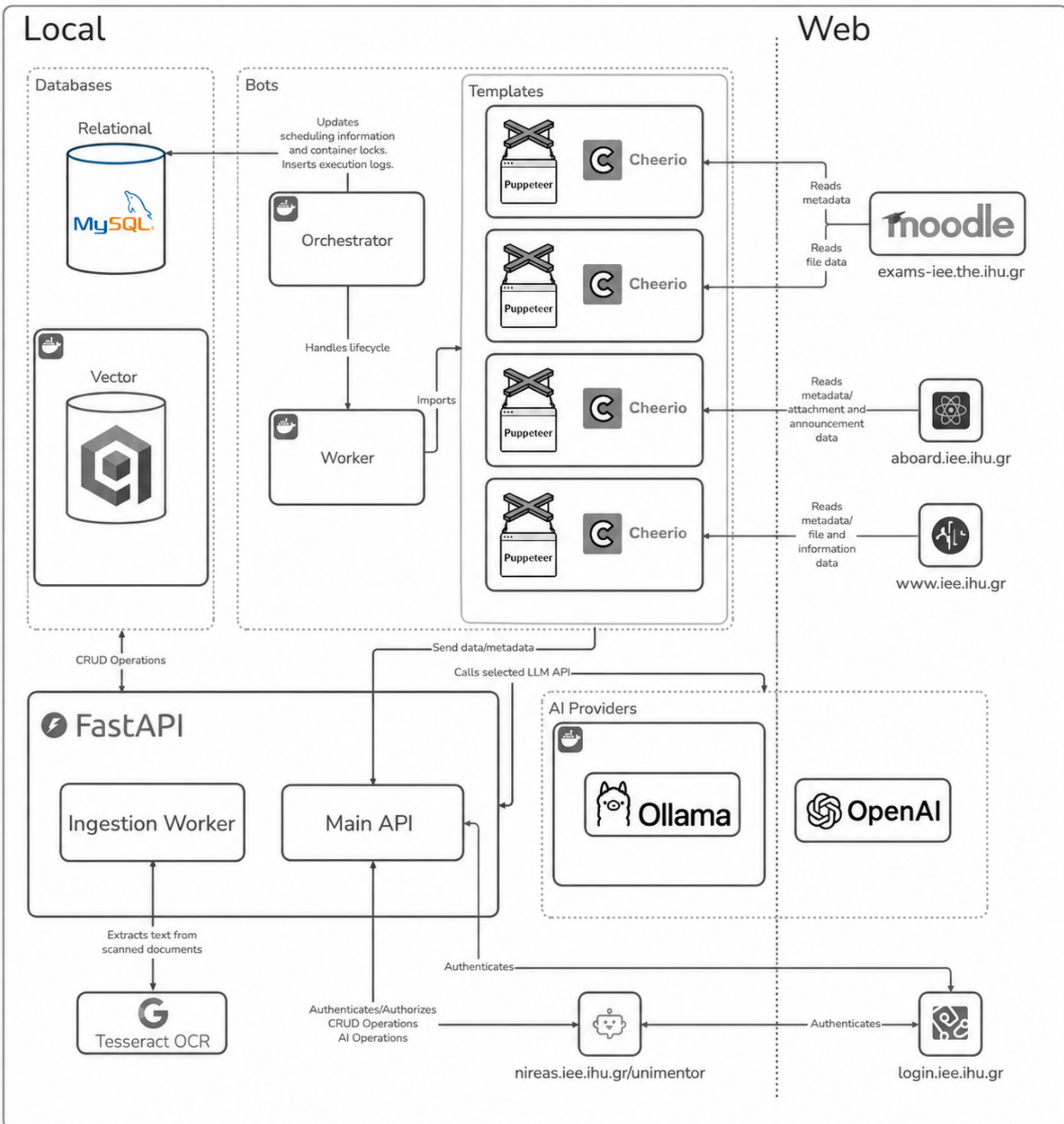

Figure 6.2: General architecture of UniMentor with emphasis on the relational database layer.

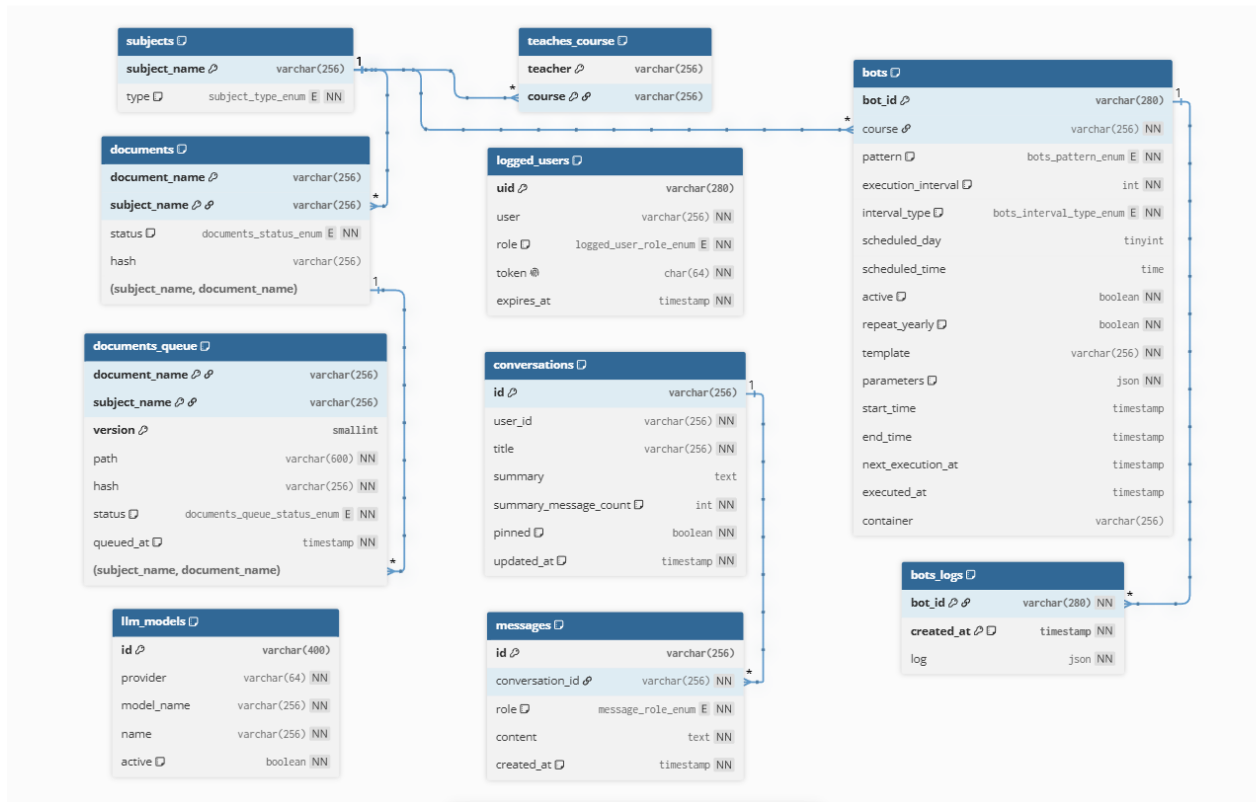

Figure 6.3: Relational database structure of UniMentor.

The relational database of UniMentor was designed to store the structured data of the platform. As shown in Figure 6.2, the relational database is one of the main supporting components of the system architecture, while Figure 6.3 presents its internal structure. This includes users, conversations, subjects, documents, document processing state, model configurations, bot schedules, and bot logs. This type of information has clear relationships, so a relational database was suitable for it. For example, a document belongs to a subject, a message belongs to a conversation, a bot belongs to a subject, and a teacher can be connected with one or more courses.

MySQL was used for this part of the system. The goal was not only to store data, but also to protect the consistency of the platform. For this reason, the schema uses primary keys, foreign keys, indexes, enums, stored procedures, triggers, and scheduled events. Some rules are handled from the backend, but important database-level rules are also enforced inside MySQL. This was especially useful for bot scheduling, because bot execution needs to be controlled carefully and should not depend only on application-side checks.

The database can be separated into a few main groups. The first group stores users and conversations. The second group stores the knowledge base structure and the document ingestion queue. The third group stores model configuration. The fourth group stores bot configuration, scheduling information, and bot execution logs. These groups are connected to each other where needed, but each one has a specific responsibility inside the platform.

The `logged_users` table stores temporary login information for users that have been authenticated by the platform. It includes the user's unique identifier, display name, role, token, and expiration time.

The supported roles are student, teacher, secretary, and bot. The human roles are used for normal platform access, while the bot role is used internally for bot execution. The token is unique, so the same authentication token cannot be reused by multiple logged users.

The table also includes an expiration timestamp. This means that user login records are not meant to stay in the database permanently. They represent active or recently active sessions. To support this, the database includes the `purge_logged_users` procedure. This procedure deletes records whose expiration time has passed. It is executed automatically by the `ev_purge_logged_users` event every thirty minutes. This keeps the table clean and avoids keeping expired login records longer than needed.

The chat history of the platform is stored through the `conversations` and `messages` tables. A conversation stores the general information of a chat, such as its identifier, user identifier, title, optional summary, number of messages included in the summary, pinned state, and last update time. The `messages` table stores the individual messages of each conversation. Each message has its own identifier, the conversation it belongs to, its role, the message content, and the creation time.

The relationship between conversations and messages is handled with a foreign key from `messages` to `conversations`. When a conversation is deleted, its messages are deleted as well through cascade deletion. This matches the behavior of the platform, because messages should not remain in the database without their parent conversation. This design also supports the conversation management features of AthenaAI, where users can create, view, pin, and delete conversations.

The `summary` and `summary_message_count` fields in the conversations table were added to support longer chat history. Instead of always sending the full conversation to the language model, the system can keep a summary of previous discussions and track how many messages are already included in it. This helps the chat keep awareness of previous questions and answers, while avoiding very large prompts every time the user continues a conversation.

The knowledge base structure starts from the `subjects` table. A subject represents an area of information that can contain documents. It can be a course, secretary-managed information, or bot-managed information. The subject type is stored as an enum with values such as Course, SecretaryManaged, BotManaged, BotManaged-Info, and BotManaged-Announcements. This distinction is important because not all subjects are managed in the same way. Course subjects are connected with teachers, secretary-managed subjects are handled by secretaries, and bot-managed subjects are populated by predefined system automations.

The `documents` table stores the document records that belong to subjects. A document is identified by the combination of its subject name and document name. This composite primary key was used because the same document name could theoretically appear under different subjects. Each document also has a status and a hash. The status can be missing, processing, or processed. This is useful because the platform often knows that a document exists before it has the actual file available for indexing.

For example, the Moodle synchronization bot can collect the names of documents that exist in a course. At that point, the platform can create records in the `documents` table, but the actual file may not have been uploaded or collected yet. In that case, the document remains in a missing state. Later, a teacher can manually upload the file, or a configured bot can collect it automatically. Then the document can move into processing and finally processed after the ingestion pipeline finishes.

The `documents_queue` table supports this ingestion pipeline. It stores documents that have been added for parsing or indexing. Each queue record is identified by subject name, document name, and version. The version field allows the platform to distinguish different submitted versions of the same document. The table also stores the file path, hash, status, and queue time. The queue status can be waiting_to_be_parsed or waiting_to_be_indexed. This separates the ingestion process into stages, instead of treating document processing as a single immediate action.

The queue table has a foreign key to the `documents` table. This means that a queued file must belong to an existing document record. If the document record is deleted, its queue entries are also deleted through cascade deletion. This keeps the database consistent and avoids queue records that refer to documents that no longer exist. Indexes were also added on fields such as document name, version, status, and queue time, because these fields are useful when the ingestion worker searches for pending work.

The `teaches_course` table connects teachers with course subjects. It uses a composite primary key with teacher and course. This allows a teacher to connect with multiple courses and a course to connect with multiple teachers if needed. The course field references the `subjects` table. This relationship is important for the role-based behavior of the Admin Panel. Teachers should only see and manage the courses assigned to them, and this table provides the database structure for that restriction.

The `llm_models` table stores the model configurations available in UniMentor. Each model record includes an identifier, provider, model name, display name, and active flag. The provider can represent where the model comes from, such as OpenAI or Ollama. The model name is the technical model identifier used when the backend calls the provider API, while the name is the user-friendly label shown in the frontend.

This table was added so that model options are not hardcoded in the frontend. Teachers can manage model records through the Admin Panel, and users can select from the active models in AthenaAI, AInsights, and Career Mentor. This makes the platform easier to maintain. If a new model is added or an old one should no longer be available, the change can be made through the database record instead of changing the application code. The schema also inserts one initial model record, so the platform has a default active model after setup.

The bot system uses the `bots` and `bots_logs` tables. The `bots` table stores the configuration and scheduling information of each bot. A bot belongs to a subject through the course field. Even though the field is named course, it can also point to bot-managed subjects such as general information or announcements. Each bot has a pattern, which can be one_time or repeat_interval. This allows the system to support both bots that run once and bots that run repeatedly.

The scheduling fields include the execution interval, interval type, scheduled day, scheduled time, start time, end time, next execution time, and executed time. These fields give the system enough flexibility to support different execution patterns. For example, a bot can run daily at a specific time, monthly on a specific day, or once at the next available time. The `repeat_yearly` field also allows a repeated bot to operate within a yearly time frame when this behavior is needed.

The `template` field defines which bot template should be executed. This is important because the platform uses template-based bots. A template can represent Moodle course synchronization, Moodle document collection, university website information collection, or announcement collection. The `parameters` field is stored as JSON, so each bot can have flexible configuration data without

requiring a different table for every template. The `container` field is used to track whether a bot is waiting for a worker or is already assigned to a running container.

The `bots_logs` table stores execution logs for bots. Each log is connected with a bot and has a creation timestamp. The log itself is stored as JSON, because bot logs may contain different types of information depending on the template and the execution result. This makes the logging structure flexible. Since logs can grow over time, the database also includes a cleanup procedure and event for them. The `purge_bots_logs` procedure deletes logs older than thirty days, and the `ev_purge_bots_logs` event runs this cleanup once per day.

A large part of the relational design is connected with bot scheduling. This was implemented inside the database through stored procedures and triggers. The reason for this design is that scheduling rules should remain consistent regardless of whether a bot is inserted or updated from the backend or from another internal process. By putting this logic close to the data, the database can reject invalid schedules and automatically compute execution fields.

The `compute_one_time_schedule` procedure handles scheduling for one-time bots. It validates that one-time bots do not include fields that belong to repeated execution, such as execution intervals, interval types other than the default, start and end times, or yearly repetition. It then calculates the next execution time based on the provided scheduled day and scheduled time. If the bot should run immediately, the procedure marks it as waiting for a worker by updating the container field.

The `compute_repeat_interval_schedule` procedure handles bots that run repeatedly. It validates the interval configuration based on the selected interval type. For example, second-based intervals should not define scheduled time, minute-based and hour-based intervals have specific restrictions, and weekly or monthly intervals require a scheduled day. It also checks that execution intervals are greater than zero and that start and end times are valid. After validation, it calculates the next execution time and decides whether the bot should be queued for execution.

The scheduling logic also uses helper procedures. The `add_interval` procedure adds the correct amount of time to a timestamp based on the interval type. The `calculate_next_execution_time` procedure calculates the next valid execution time using a reference timestamp, scheduled time, scheduled day, and interval type. The `adjust_scheduling_information` procedure moves the next execution time forward when needed and handles cases where a repeated bot reaches the end of its allowed time frame. These procedures keep the scheduling logic organized instead of repeating the same calculations in many places.

The `run_bots_scheduler` procedure is used by the orchestrator. It receives the number of available containers and assigns waiting bots to available container names. It uses a database lock so that two scheduler executions cannot run at the same time. This is important because without locking, two scheduler processes could try to assign the same bot or the same container at the same moment. The procedure also gives priority to bot-managed subjects, older execution times, and one-time bots before repeated bots when selecting which bots should run.

When the orchestrator assigns a bot to a container, a temporary token is also created through the database triggers. This token is inserted into the `logged_users` table with the role `bot`. This allows the bot worker to authenticate temporarily when it sends data back to the backend API. The token has a short expiration time, so it is not a permanent user account. This design keeps bot access limited to the period of execution.

The `cleanup_bots` procedure is used after a bot finishes or when cleanup is needed. It removes the container assignment from the bot and deletes the temporary bot user from `logged_users`. It also uses a database lock based on the bot identifier, so the same bot cannot be cleaned up by multiple processes at the same time. This protects the execution state of the bot and avoids inconsistent cleanup behavior.

Triggers are used to enforce several important rules. The trigger on `logged_users` prevents normal insertions of users with the bot role. Only the orchestrator is allowed to create temporary bot users. This prevents a normal database operation from creating a fake bot session. The triggers on `bots` are more complex. Before inserting a bot, the database automatically generates the bot identifier, validates required fields, prevents manual values for generated fields, and calls the correct scheduling procedure based on the bot pattern.

Before updating a bot, the database prevents users from changing fields that should be controlled automatically, such as next execution time, executed time, and container. It also prevents changing the bot identifier, course, or pattern after creation. This is important because these fields define the identity and scheduling type of the bot. If they were changed freely, the scheduler could become inconsistent. The update trigger also calls the scheduling procedures again when scheduling-related fields change, so the next execution time remains correct.

The after-update trigger on `bots` is used mainly for orchestrator behavior. When a bot is assigned to a real container, the trigger creates the temporary bot token and inserts the necessary execution information into a temporary table used by the scheduler. This allows the scheduler to return the data needed by the worker, such as the bot id, template, parameters, token, course, and container. The trigger also removes the temporary bot login when cleanup sets the container back to null.

There is also a delete trigger for bots. It prevents deletion of a bot that is currently running in a container. This avoids removing a bot configuration while its worker is still executing. If a bot is deleted normally, the trigger also removes any related temporary bot login record. Another trigger is used before deleting subjects, so that bot deletion during subject cascade deletion can be handled differently from manual bot deletion. This is needed because subjects can cascade to bots, and the database must know when a delete operation is part of a larger cascade.

The schema also includes initial data. One active LLM is inserted, so that the system has a default model available after setup. Three bot-managed subjects are also inserted: course information, general information, and announcements information. These subjects are used by the predefined system bots. The initial bot records include a daily course information synchronization bot, a monthly general information synchronization bot, and a daily announcements synchronization bot. This means that the system is not empty after deployment. It already contains the basic automated tasks needed to start populating the knowledge base.

Overall, the relational database was designed to support both normal application data and internal system automation. It stores the data needed by users, conversations, knowledge base management, document ingestion, model selection, and bot execution. The use of foreign keys keeps related records connected, while cascade rules remove dependent data when the parent record is deleted. The stored procedures, triggers, and events add another layer of control for scheduling, cleanup, and consistency. This makes the database an active part of the UniMentor implementation, not just a passive storage layer.

6.4 Vector Database Design and Development

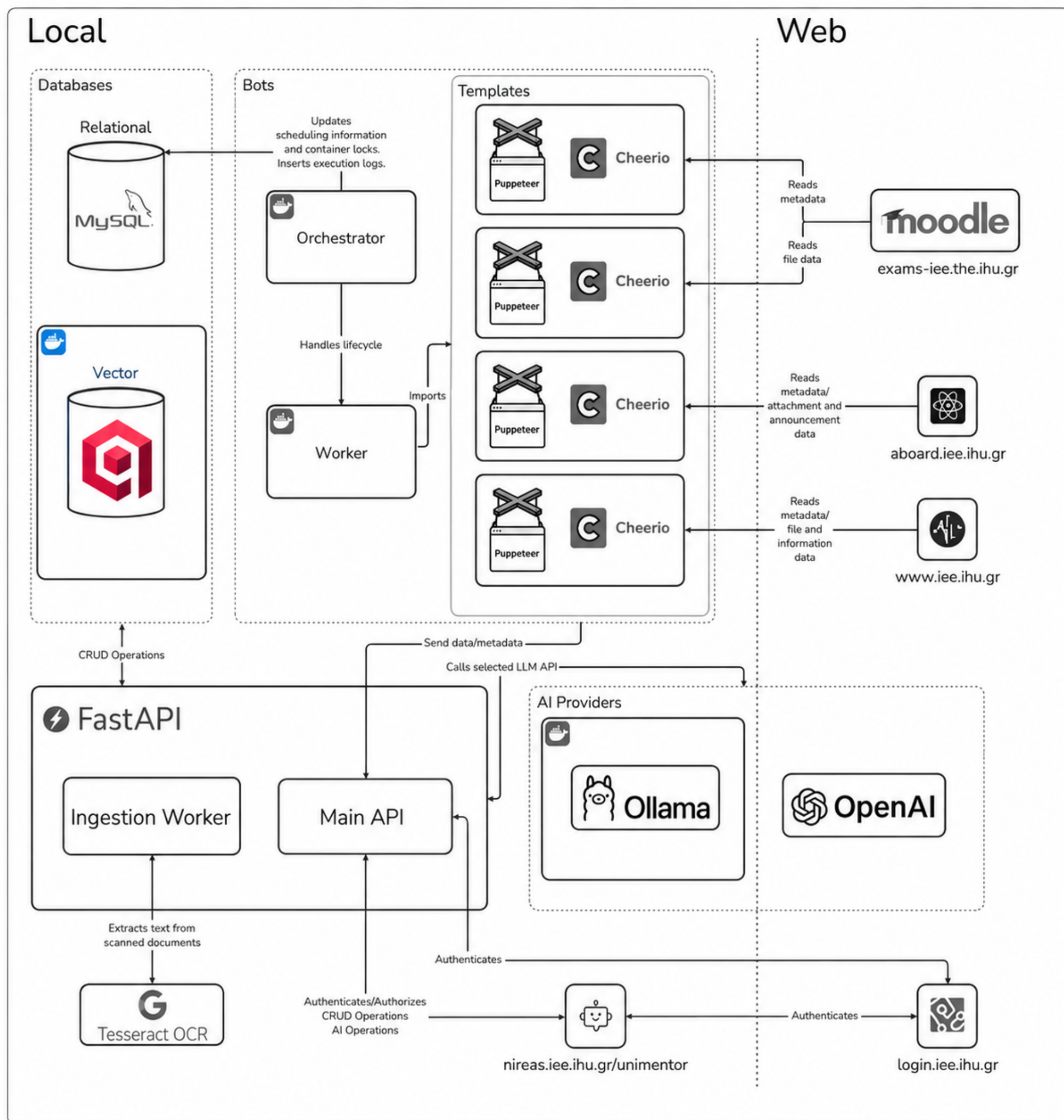

Figure 6.4: General architecture of UniMentor with emphasis on the vector database layer.

The vector database is the part of UniMentor that supports semantic retrieval. While the relational database stores structured information such as subjects, documents, users, conversations, and bot records, the vector database stores the searchable representation of document content. This separation is important because normal relational queries are useful for structured data, but they are not enough when the system needs to find text that is semantically related to a user's question.

For this part of the system, UniMentor uses Qdrant. Qdrant stores vectors that are produced from document chunks. Each vector represents a part of a document, and it can later be compared with the embedding of a user's query. The collection is created automatically if it does not already exist, using the collection name from the application settings. The collection uses cosine similarity as the distance metric, because the goal is to compare the semantic direction of embeddings rather than exact text equality. The vector size is also loaded from the settings, so the collection can match the embedding model used by the platform.

In the deployment setup, Qdrant runs inside a Docker container. The container exposes the Qdrant service only on the local machine, so it is used by the backend without being directly exposed as a public service. Its storage is also connected with a Docker volume, named `unimentor_qdrant_data`. This is important because the indexed vectors should not be lost when the container is restarted or recreated. A memory limit is also defined for the container, so the vector database can run in a more controlled way inside the server environment.

Embeddings are generated through the Ollama embeddings endpoint. The embedding model used in the current implementation is `bge-m3`. When the system needs to embed text, the backend sends the text list to the Ollama embeddings API and receives a vector for each input. The implementation also checks that the response is valid, that the returned embeddings are not empty, and that all returned vectors have the same dimension. These checks are useful because an invalid embedding response would make the indexed data unreliable or could break later retrieval operations.

The indexing process starts after a document has entered the ingestion pipeline. A document may come from a manual upload by a teacher or secretary, or it may be collected automatically by a bot. In both cases, the file is first stored in the document queue. The ingestion worker then processes the queued file and prepares the extracted text for indexing. The detailed parsing process is part of the backend implementation, but in relation to the vector database, the important point is that the final extracted text becomes the source for chunking, embedding, and vector storage.

Before a document is inserted into Qdrant, its text is split into smaller chunks. The chunking method used in the implementation is simple and direct. By default, each chunk has a size of 1000 characters and an overlap of 200 characters. The overlap is useful because it reduces the chance that important information is cut between two separate chunks. The system also validates the chunking parameters, so the chunk size cannot be too small, the overlap cannot be negative, and the overlap must be smaller than the chunk size.

Each chunk is then converted into an embedding and inserted into Qdrant as a point. The point identifier is generated with a UUID. Together with the vector, the system also stores a payload. This payload includes the actual chunk text and metadata about the source document. The stored metadata includes the document key, subject name, document name, file hash, chunk index, and total number of chunks. The document key is created by combining the subject name and the document name. This makes it easier to delete or manage all chunks that belong to the same document later.

The hash value is especially important because it helps the platform distinguish between different versions of the same document. When a new version of a document is processed, the system can activate the new hash in the relational database and remove the old embeddings that belonged to the previous hash. This prevents old document versions from remaining active in the vector database after a newer version has been indexed. In this way, the system avoids mixing outdated content with current

content during retrieval.

Some types of content also receive extra context before indexing. For secretary-managed subjects, bot-managed information, and announcement-related content, the title is added at the beginning of each chunk. This was done because this type of information may be shorter or less connected to a course structure than normal course documents. Adding the title gives the chunk more context and can make retrieval more useful, especially when the user asks about an announcement or an institutional information item.

Retrieval is handled by first embedding the user's query and then searching Qdrant with that query embedding. For AthenaAI, the retrieval searches through the active indexed documents. The system does not simply search every vector that ever existed. It first reads the active document hashes from the relational database and uses them as a filter in Qdrant. This means that missing documents or old document versions are not used as context. The result is a list of matching chunks with their identifiers, text, metadata, and similarity scores.

AInsights uses a more restricted retrieval process. Since AInsights generates material for a selected course, the system should not retrieve context from unrelated subjects. For this reason, the query is created from the selected template name and the optional custom prompt, and the vector search is filtered by the selected subject. It is also filtered by the active hashes of that subject. This allows the generated study material to stay connected to the course that the student selected, instead of being influenced by documents from other courses.

The retrieved chunks do not constitute final answers. They are used as context for the language model. After Qdrant returns the relevant chunks, the backend builds the final prompt and sends it to the selected LLM provider. For RAG-based chat, the retrieved context is used to support answers about university, course, or document information. For AInsights, the context is combined with the selected course, template type, and optional user instructions in order to generate the final structured output.

The vector database is therefore connected closely with both the relational database and the backend services. MySQL keeps track of which documents exist, which document version is active, and what status each document has. Qdrant stores the semantic chunks that can be searched during RAG. The ingestion worker connects the two by preparing the extracted text, creating chunks, generating embeddings, and updating the active document version after successful indexing. This design keeps the retrieval system consistent with the document state shown in the Admin Panel.

Overall, the vector database design of UniMentor focuses on making the knowledge base searchable by meaning rather than by exact keywords. Documents are chunked, embedded, and stored with enough metadata to support filtering, retrieval, version control, and deletion. AthenaAI can use this structure to answer questions with relevant context, while AInsights can generate course-specific material by retrieving only from the selected course. This makes Qdrant a central part of the RAG implementation, while MySQL remains responsible for the structured state and management of the platform.

6.5 AI Providers

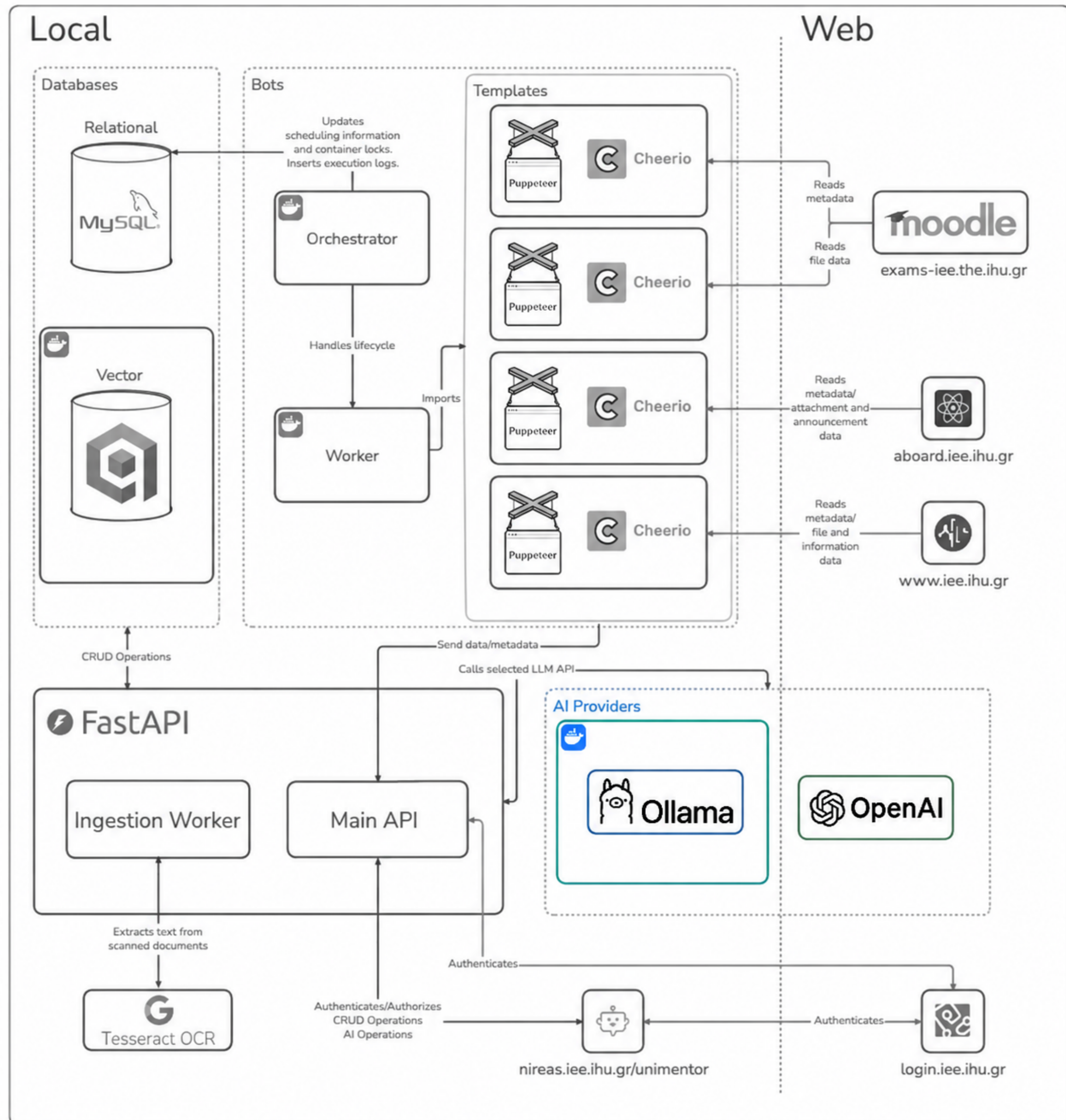

Figure 6.5: General architecture of UniMentor with emphasis on the AI providers layer.

The AI providers layer is the part of UniMentor that connects the backend with the language models used by the platform. This layer was added so that the system is not tied to one specific model or one specific provider. Instead of calling a model directly from each feature, the backend resolves the selected model configuration and then routes the request to the correct provider.

UniMentor currently supports two providers, Ollama and OpenAI. Ollama is used as a self-hosted

provider, while OpenAI is used as an external third-party provider. This gives the platform more flexibility. Local models can be used through Ollama when the server environment supports them, while OpenAI can be used through its API when an external model is preferred.

Ollama runs locally inside a Docker container. In the deployment setup, the container is named `unimentor-ollama` and the service is exposed only to the local machine. This means that the backend can call Ollama, but the Ollama service is not directly exposed as a public endpoint. The container also uses a Docker volume, named `unimentor_ollama_data`, so that downloaded models and related data remain available even if the container is restarted or recreated. Resource limits are also defined for the container, so the local model service can run in a more controlled way inside the server environment.

OpenAI is handled differently because it is not self-hosted. It is accessed as a third-party provider through the OpenAI API. For this provider, the backend uses an API key that is loaded from the application settings or from the server environment. This is important because sensitive credentials should not be written directly inside the source code. The OpenAI base URL can also be resolved from configuration, which keeps the provider connection flexible.

The available models are stored in the relational database as configuration records. Each record contains the provider, the technical model name, a display name, and an active flag. This design makes the model layer easier to manage, because the platform does not need to hardcode all model choices inside the application logic. Only active models are considered available for use by the AI features of the system.

When the backend needs to generate a response, it first resolves the model that should be used. If a specific model identifier is provided, the backend loads that model from the database and checks that it is active. If no specific model is provided, the backend falls back to the first active model. This prevents requests from being sent to missing or inactive model records and keeps the model selection logic centralized.

After the model is resolved, the backend routes the request based on the provider value. If the provider is Ollama, the request is sent to the Ollama connection layer. If the provider is OpenAI, the request is sent to the OpenAI connection layer. If an unsupported provider is configured, the backend returns an error. This approach keeps the provider logic separate from the rest of the system and also makes it easier to add another provider in the future.

The Ollama provider calls the local Ollama API and streams the generated response. The request includes the selected model name and the prompt created by the backend. Some generation settings, such as context size, number of threads, GPU layers, temperature, top-p, repeat penalty, and maximum prediction length, are read from the application configuration. This allows the behavior of local models to be adjusted without changing the main application code.

The OpenAI provider calls the OpenAI chat completions API and also receives the response as a stream. Since OpenAI and Ollama return streamed output in different formats, the provider layer normalizes the response into a common structure. In both cases, the rest of the backend receives generated text pieces and a completion flag. This is useful because AthenaAI, AInsights, and Career Mentor can consume model output in the same way, without needing separate logic for every provider.

This common streaming behavior also improves the user experience. The system can forward gener-

ated text gradually instead of waiting for the full answer to finish first. This is especially useful for longer answers or for local models that may need more time to generate a response. The provider layer therefore supports both flexibility in model choice and a smoother response flow for the frontend.

The AI providers are used by multiple parts of UniMentor. AthenaAI uses them for normal chat responses and RAG-based answers. The same generation layer is also used for supporting tasks, such as conversation title generation and conversation summarization. AInsights uses the selected provider to generate structured study material after the relevant course context has been retrieved. Career Mentor uses the provider for AI-based filtering of job positions.

Embeddings are also connected with this layer, but they are not the main focus of this section. In the current implementation, Ollama is used with the `bge-m3` embedding model for document chunks and user queries. This was discussed in the vector database section because it is directly connected with Qdrant and semantic retrieval. Here, the focus is mainly on how UniMentor routes generation requests to local or external LLM providers.

Overall, the AI providers layer makes UniMentor more flexible and easier to maintain. Ollama allows the platform to use self-hosted models inside the server environment, while OpenAI allows it to use external models through a third-party API. The database model records connect these providers with the rest of the platform, and the shared routing logic keeps the implementation consistent across the different AI features.

6.6 Backend Development (API)

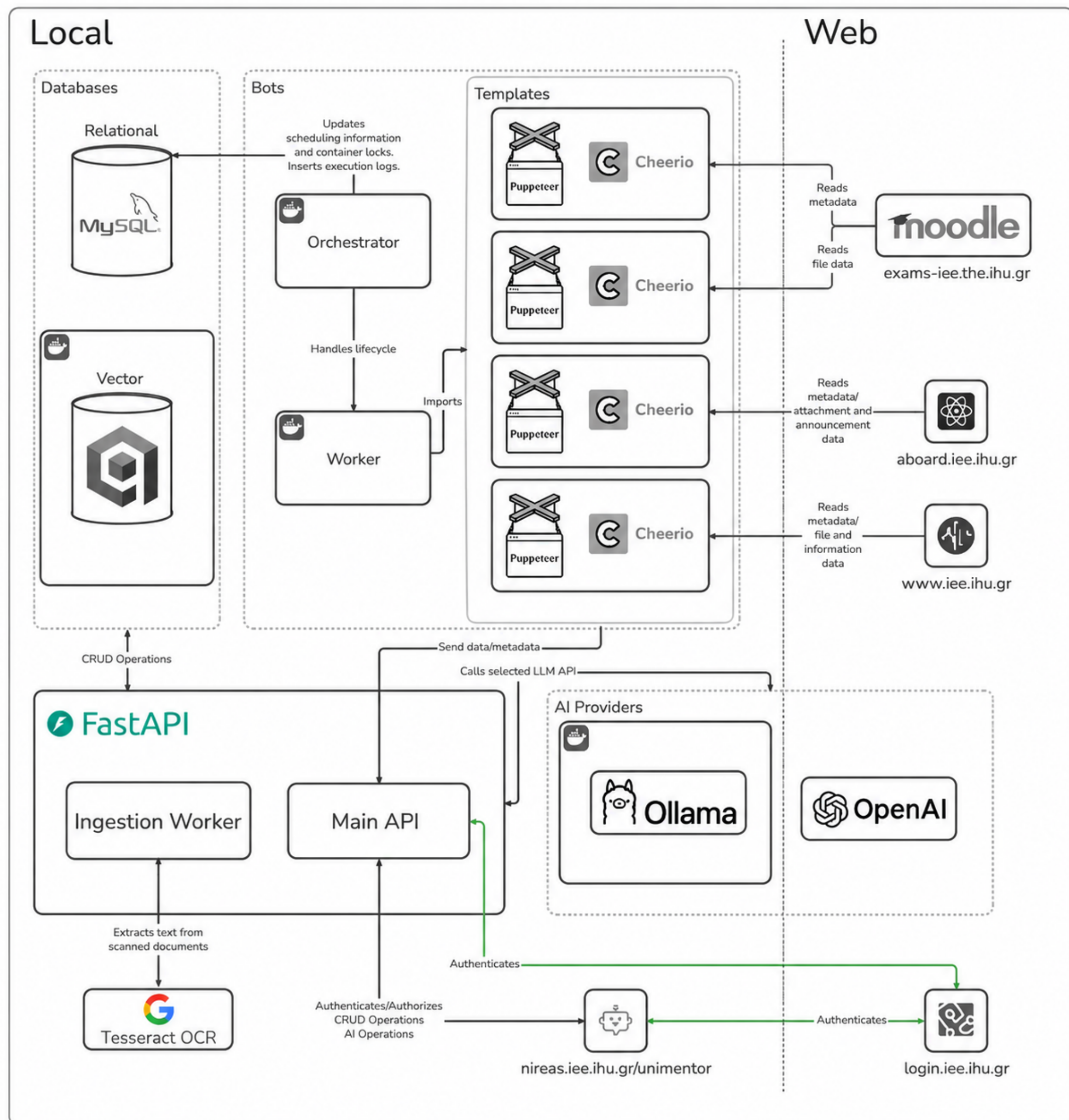

Figure 6.6: General architecture of UniMentor with emphasis on the FastAPI layer.

6.6.1 Backend Role in the System Architecture

The backend of UniMentor was developed with FastAPI and acts as the central coordination layer of the platform. As shown in Figure 6.6, the FastAPI layer is placed between the frontend, the relational database, the vector database, the AI providers, the ingestion worker, and the bot subsystem. This

means that the backend is responsible not only for exposing HTTP endpoints but also for connecting the different parts of the system in a controlled and consistent way.

The frontend does not communicate directly with MySQL, Qdrant, Ollama, OpenAI, or the bot workers. All these operations go through the backend. This was important because the backend is the only layer that can safely enforce authentication, role restrictions, validation rules, and the correct order of operations. For example, when a teacher uploads a file, the frontend only sends the file and the selected document information to the API. The backend then checks whether the teacher is allowed to access that course, validates the document, sends the file to the queue, and later the ingestion worker processes it.

The same idea is used for the AI features. AthenaAI, AInsights, and Career Mentor do not call an AI model directly from the frontend. Instead, they send their request to the backend. The backend prepares the prompt, optionally retrieves context from Qdrant, resolves the selected model, calls the correct AI provider, and streams the response back to the frontend. This design keeps the frontend simpler and keeps sensitive logic, such as model routing and API keys, inside the server environment.

The backend also separates normal user operations from background processing. The Main API handles user-facing and admin-facing requests, while the ingestion worker processes queued documents outside the normal request-response flow. This distinction is important because parsing, OCR, chunking, embedding, and indexing files can take time. If these operations were executed directly during the upload request, the user would have to wait longer and the API would be more likely to fail during heavy processing.

In this architecture, FastAPI is also the controlled entry point for automated bots. Bots do not directly modify the database or insert vectors into Qdrant. Instead, they send collected metadata or files to specific bot-facing endpoints. The backend then validates the request, applies the correct database changes, and uses the same document enqueueing pipeline used by manual uploads. This makes the bot subsystem easier to control and keeps the rest of the platform consistent.

Overall, the backend is the part that turns the separate services into one working system. MySQL stores the structured state, Qdrant stores the semantic search data, the AI providers generate responses, and the frontend presents the platform to the users. FastAPI connects these parts together and applies the rules that make them work as one platform.

6.6.2 Project Structure and API Organization

The backend project is organized around routers, services, database models, schemas, providers, parsers, vectorstore helpers, and scripts. This separation is useful because the backend contains many different responsibilities. If all the code was placed directly inside route functions, the API would quickly become difficult to read and maintain.

The main application is initialized in `main.py`. This file creates the FastAPI application, configures the API title, sets the documentation and OpenAPI paths, applies the deployment root path, registers the database exception handler, enables CORS, and includes the main routers. The root path setting is important for deployment because UniMentor is served under a specific path on the university server. This allows the API to work correctly behind the server configuration instead of assuming that it always runs from the domain root.

The public API surface of the backend is summarized in Table 6.2. The table lists the main endpoints exposed by FastAPI, the router group to which each endpoint belongs, and the roles that are allowed to access it. It is not meant to describe every internal service function, because most of the implementation logic is placed behind these endpoints and is explained in the following subsections.

Endpoint / Operation	**Router / Group**	**Accepted Roles**
`GET /auth`	Authentication	Public / existing session
`GET /users/courses`	Users	student, teacher, secretary
`GET /users/models/active`	Users	student, teacher, secretary

Conversation management:
`POST /chat/conversations`
`GET /chat/conversations`
`GET /chat/conversations/{conversation_id}`
`PATCH /chat/conversations/{conversation_id}/pinned`
`DELETE /chat/conversations/{conversation_id}`
	Chat	student, teacher, secretary

`POST /chat`	Chat	student, teacher, secretary
`POST /tools/templates`	Tools	student, teacher, secretary
`POST /tools/filter`	Tools	student, teacher, secretary

Subject management:
`GET /administration/subjects`
`POST /administration/subjects`
`DELETE /administration/subjects/{subjectName}`
	Administration	teacher, secretary
		secretary-only for create and delete

Document management:
`GET /administration/subjects/{subjectName}/documents`
`POST /administration/subjects/{subjectName}/documents`
`DELETE /administration/subjects/{subjectName}/documents/{documentName}`
	Administration	teacher, secretary
		secretary-only for create and delete

Model management:
`GET /administration/models`
`POST /administration/models`
`PATCH /administration/models/{model_id}`
`DELETE /administration/models/{model_id}`
	Administration	teacher, secretary
		teacher-only for create, update, and delete

Bot configuration:
`GET /administration/bots`
`POST /administration/bots`
`PATCH /administration/bots`
`DELETE /administration/bots`
`GET /administration/botLogs`
	Administration	teacher

`POST /administration/callDocEnqueuer`	Administration	teacher, secretary
`POST /bots/updateSubjectInfo`	Bots	bot
`POST /bots/callDocEnqueuer`	Bots	bot

Table 6.2: Main FastAPI endpoints of UniMentor grouped by router and accepted roles.

These endpoints are organized into router groups. The authentication router handles login and session creation. The users router exposes shared user-facing data, such as available courses and active models. The chat router handles AthenaAI conversations and chat generation. The tools router handles AInsights and Career Mentor filtering. The administration router handles the Admin Panel operations. The bots router handles requests sent by bot workers. This structure makes the API easier to understand because each group has a clear purpose.

The router files are kept relatively small. They mainly define the endpoint paths and call the appropriate service function. The real logic is placed in service files, such as `auth_service.py`, `chat_service.py`, `tools_service.py`, `admin_service.py`, `bots_service.py`, `doc_enqueuer_service.py`, and the RAG-related services. This makes the code more main-

tainable because the routing layer is separated from the business logic.

The backend also uses schema files for request and response validation. These schemas define the structure of the data that enters or leaves the API. This is useful because FastAPI can validate incoming data before it reaches the service logic. For example, chat requests, template requests, filter requests, model records, bot configuration requests, subject information requests, and document enqueueing requests all follow predefined structures. This reduces the chance of invalid or incomplete data being handled later in the flow.

Database-related code is separated into the `db` folder. The SQLAlchemy models represent the relational database tables, while the session file creates and provides database sessions. The provider files are placed separately and are responsible for connecting to Ollama and OpenAI. The vectorstore file is responsible for connecting with Qdrant and handling vector operations. The parser files are responsible for extracting text from different document formats. Finally, the scripts folder contains the ingestion worker, which runs separately from normal API requests.

This structure follows a practical separation of responsibilities. The routers expose the API, the services decide what should happen, the database layer stores structured data, the vectorstore layer manages embeddings, the provider layer communicates with AI models, and the parsers handle file text extraction. This makes the backend easier to expand because new functionality can be added in the correct layer instead of mixing everything together.

6.6.3 Database Session Handling

Database session handling is one of the important backend details because almost every feature needs access to MySQL. UniMentor uses SQLAlchemy for database access, and the database connection is configured centrally in `db/session.py`. The database URL is loaded from the application settings, and the engine is created with connection options such as `pool_pre_ping` and `pool_recycle`. These options help the backend avoid problems with stale database connections, especially in a server environment where connections may stay open for a long time.

The backend uses a `SessionLocal` factory to create database sessions. The important part is the `get_db()` dependency. This dependency creates one database session for the request and yields it to the endpoint through FastAPI dependency injection. When the request finishes successfully, the dependency commits the transaction. If an exception happens, it rolls back the transaction. Finally, the session is closed. This means that normal database transaction handling is centralized instead of being repeated manually in every endpoint.

This design is useful because many endpoints need to read and write database data. Without a central session dependency, every endpoint would have to remember to commit, roll back, and close the session. That would make the code more repetitive and more error-prone. By handling this in one place, the backend has a more predictable transaction flow. A successful request can commit its changes, while a failed request can roll them back.

However, some parts of the backend still need explicit commits. This happens when a flow is longer than a normal request or when part of the work must be saved before the next step starts. For example, the authentication flow commits when a session is created or extended. The chat streaming flow commits the user message before generation starts, because the conversation state must already exist while

the stream is running. After the assistant response is completed, the backend commits again to store the generated answer and later the updated conversation summary. The ingestion worker also uses explicit commits because it runs outside the normal request dependency and processes queue items step by step.

This means that the backend uses a mixed but controlled approach. Normal request flows can rely on the centralized `get_db()` behavior, while long-running or streaming flows can commit earlier when needed. The trade-off is that the developer must be careful in these special cases. For example, if a chat generation starts and the user message has already been committed, then the backend must still handle errors cleanly if the AI provider fails. In the implementation, failed streaming flows perform a rollback where possible and send an error event to the frontend.

The centralized session dependency additionally helps with consistency. If a service raises an `HTTPException` or another error before the request completes, the transaction is rolled back. This prevents partial changes from remaining in the database after a failed operation. For example, if an admin operation fails during validation, the database should not keep half of the intended change. This is especially important in UniMentor because many operations affect more than one table.

Another detail is that the backend uses a shared request context dependency. The `RequestContext` class collects the login code, login error, session cookie, and database session. This context is then passed to service functions using the same pattern across routers. This avoids repeating the same dependency parameters in every endpoint and makes the service calls more consistent.

6.6.4 Authentication and Role-Based Authorization

Authentication in UniMentor is handled through the university login system. The main authentication endpoint is `GET /auth`. This endpoint is used both for starting the login process and for validating an existing session. The backend supports the login flow by using the university authorization URL, token URL, profile URL, client id, client secret, scope, and redirect URI from the application configuration. When a user is not already authenticated, the backend can return the authorization URL so the frontend can redirect the user to the university login page.

After the user logs in, the university authentication system returns a code to the configured redirect URI. The backend receives this code through the authentication flow and exchanges it for an access token. Then it uses the access token to fetch the user profile. The profile information is used to identify the user and map them to a UniMentor role. The main human roles are student, teacher, and secretary. The backend maps staff users with the title Secretary to the secretary role, other staff users to the teacher role, and student users to the student role.

There is also a special detail for teachers. The user profile may contain both a normal name and a Greek-language name. Since course-teacher relations in the database may be based on the Greek name, the backend checks whether the Greek teacher name exists in the `teaches_course` table. If it exists, the backend uses that resolved teacher name for authorization checks. This is important because the teacher name used by the external university data and the name returned by the authentication profile may not always match in the same format.

After authentication succeeds, the backend creates a session token. This token is generated as a secure random value and stored in the `logged_users` table together with the user id, user name,

role, and expiration time. The token is then returned to the frontend as an HTTP-only cookie named `session_token`. The cookie is configured with a maximum age of 30 minutes, `SameSite` protection, and the secure flag based on the deployment settings. This keeps the session token away from normal frontend JavaScript access.

When a request already contains a valid session cookie, the backend does not need to repeat the full university login flow. Instead, it looks up the token in `logged_users`. If the session exists and has not expired, the backend extends the session expiration time by another 30 minutes and allows the request to continue. If the session is expired, it deletes the old row and continues with the login flow. This gives the system temporary sessions without keeping users logged in forever.

Authorization is handled through a shared helper called `authorize()`. Each protected service function calls this helper and provides the accepted roles for that operation. If the user is not authenticated, the helper returns an unauthorized response. If the user is authenticated but their role is not accepted, it returns a forbidden response. If the user is allowed, the helper returns the user id, user name, role, and refreshed session token.

This design is important because permissions are enforced at the backend level. The frontend may hide buttons or pages depending on the role, but that is not enough for security. A user could still try to call an endpoint manually. For this reason, every sensitive service checks the role again. For example, students can use chat and tools, but they cannot manage subjects. Teachers can manage course-related documents and bots for their own courses. Secretaries can manage secretary-related content, but they cannot configure course bots. Bot workers can call only bot-specific endpoints.

The bot role is handled separately from the human roles. It exists so that automated workers can authenticate and send collected data back to the platform. A bot worker is not a normal frontend user. This distinction matters because the bot-facing API has different responsibilities from the user-facing API. Bot endpoints accept the bot role, while normal chat, tools, and administration endpoints accept human roles depending on the feature.

6.6.5 LLM Usage in Backend Services

LLM usage is centralized in the backend so that the different AI features do not need to know the details of each provider. The model records are stored in the relational database, and each record includes the provider, the technical model name, a display name, and an active flag. This allows the platform to manage available models from the database instead of hardcoding the model choices in multiple places.

The active model list is exposed through the `GET /users/models/active` endpoint. This endpoint returns only active model records and is available to students, teachers, and secretaries.

The complete model management API is placed under the administration router. The `GET /administration/models` endpoint returns all model records to teachers and secretaries. Teachers can also manage these records through the create, update, and delete endpoints of the same router. More specifically, model creation is handled by `POST /administration/models`, model updates are handled by `PATCH /administration/models/{model_id}`, and model deletion is handled by `DELETE /administration/models/{model_id}`.

When an AI feature needs to generate text, it passes the selected model identifier to the generation service. If a model identifier is provided, the backend loads that model from the database and checks that it exists and is active. If no model identifier is provided, the backend falls back to the first active model it can find. This prevents requests from using missing or inactive models and keeps the model selection behavior consistent across the platform.

After the model is resolved, the backend checks the provider value. If the provider is Ollama, the request is routed to the Ollama provider connection. If the provider is OpenAI, the request is routed to the OpenAI provider connection. If the provider is not supported, the backend returns an error. This makes the backend easier to extend because adding another provider would mainly require adding a new provider connection and one more routing case.

The features that use this generation service include AthenaAI chat, RAG-based answers, conversation title generation, conversation summarization, AInsights template generation, and Career Mentor AI filtering. This shared usage is important because it avoids duplicated model-calling logic. Each feature is responsible for building the correct prompt and handling the result, but it does not need to know how Ollama or OpenAI streaming works internally.

The backend also contains prompt-building functions for different use cases. Normal chat uses a general AthenaAI system prompt. RAG chat uses a stricter prompt that tells the model to use only the provided context for factual university, course, or document answers. AInsights uses a template-focused prompt that asks the model to generate structured study material based on the selected course and template type. Career Mentor uses a filtering prompt that asks the model to return only a JSON array of matching job ids. Conversation summarization uses another prompt that keeps older conversation context compact.

This prompt separation is useful because the same model can be used for different tasks, but each task needs different instructions. For example, the chat feature may answer naturally using conversation history, while the RAG prompt must be more strict and avoid unsupported information. The filtering prompt is even more constrained because it expects valid JSON output instead of a normal natural language answer.

Embeddings are also part of backend AI usage, but they are used differently from text generation. For retrieval, the backend embeds the user query or template query and sends the vector to Qdrant. For indexing, the ingestion worker embeds document chunks before storing them in the vector database. This was already discussed in the vector database section, but from the FastAPI implementation side the important point is that embeddings and generation are both accessed through backend services instead of being called directly by the frontend.

6.6.6 Streaming Responses and Error Handling

Streaming is used in UniMentor because AI responses may take time to generate. If the backend waited for the full answer before returning anything, the user experience would feel slow, especially for long responses or local models. For this reason, the `POST /chat` and `POST /tools/templates` endpoints use `StreamingResponse` with Server-Sent Events.

The backend formats stream messages using SSE events. Each event contains a type and a data payload. The main event types used in the platform are `meta`, `token`, `heartbeat`, `sources`, `error`, and

`done`. The `meta` event is sent at the start of the stream and contains information about the request, such as whether RAG is enabled, how many chunks were requested, the conversation id, or the selected sources. The `token` event contains a generated text piece. The frontend appends these pieces to show the answer gradually.

The `heartbeat` event is used to keep the streaming connection alive during longer operations. If the model takes some time before producing the next token, the backend can send a keep-alive event so the connection is not treated as inactive. The `sources` event is used when RAG context is retrieved. It sends source metadata such as the chunk id, subject name, document name, hash, and chunk position. The `done` event marks the end of the stream, while the `error` event is used when something fails during generation.

Streaming is also connected with transaction handling. In a normal JSON response, the request finishes after the service function returns. In a streaming response, the response object is returned first, but the actual generation continues inside an asynchronous generator. Because of this, the chat service performs explicit commits at important points. It saves and commits the user message before generation starts. After the assistant's answer is completed, it saves the message and commits it. If a summary update is needed, that update is also committed after it is generated. If the stream fails, the backend rolls back the current transaction and sends an error event.

The backend also checks whether the client has disconnected. This matters because the frontend may close the page or cancel the request while the backend is still generating. If the client disconnects, the backend stops streaming instead of continuing unnecessary work. This helps reduce wasted processing, especially when local models or long prompts are used.

Error handling is also implemented at the database level. The FastAPI application registers a custom exception handler for SQLAlchemy database errors. This handler extracts MySQL error codes and converts them into cleaner HTTP responses. For example, custom database rule errors raised with MySQL signal code `1644` are returned as bad requests with the database message. Duplicate key errors are returned as conflict responses. Foreign key errors are returned as invalid related record responses. Other database errors are treated as internal server errors.

This is useful because some important rules are enforced inside the relational database, especially around bot scheduling and data consistency. If a trigger or stored procedure rejects a change, the frontend should receive a meaningful message rather than a raw database error. This keeps database-level validation and API-level error handling connected.

The backend also raises normal HTTP errors for application-level problems. Examples include missing subjects, missing documents, inactive models, unsupported file types, forbidden role access, invalid bot configuration access, missing conversation ids, and unsupported AI providers. This makes the API behavior clearer because invalid states are rejected close to the point where they are detected.

6.6.7 RAG Flow inside the Backend

The RAG flow inside the backend is the mechanism that connects user questions or template requests with the indexed knowledge base. The vector database section explained how document chunks and embeddings are stored in Qdrant. In this subsection, the focus is on how FastAPI uses that data during a real request.

For normal AthenaAI RAG chat, the backend receives the user query through `POST /chat` and checks whether RAG is enabled in the request. If RAG is enabled, the retrieval service embeds the query and searches Qdrant for similar chunks. Before querying, the service also checks the active document hashes from the relational database. This is important because a document may have older indexed versions in Qdrant. The backend uses active hashes to make sure retrieval is based only on currently active document versions.

After the relevant chunks are retrieved, the backend normalizes the result into a common structure. Each retrieval hit contains a chunk id, the chunk text, and metadata. The metadata can include the document key, subject name, document name, hash, chunk index, and total chunks. This information is later used both for prompt construction and for source reporting to the frontend.

The retrieved chunk texts are inserted into the RAG prompt as provided context. The RAG system prompt is stricter than the normal chat prompt. It tells AthenaAI to use only the provided context for factual university, course, or document answers. It also tells the model not to invent sources, facts, courses, documents, dates, or policies. This is important because the goal of RAG in UniMentor is not only to produce fluent answers, but to reduce unsupported answers when the user asks about university-specific information.

If no active hashes exist or no relevant chunks are found, the retrieval service can return an empty list. In that case, the prompt still makes it clear that no context was retrieved. This gives the model a safer behavior, because it should not pretend that information exists when the backend did not provide supporting context. This is especially important for academic information, course rules, announcements, and secretary-related information.

AInsights uses a similar RAG flow through `POST /tools/templates`, but with a more restricted retrieval scope. Instead of searching across all active documents, the backend retrieves chunks only for the selected course. It builds a query from the selected template type and the optional custom prompt. Then it embeds that query and searches Qdrant with filters for the selected subject and active hashes. This makes AInsights more focused because a study plan or quiz for one course should not use material from unrelated courses.

The backend keeps the RAG retrieval logic separate from the chat and tools services. The chat service and tools service call retrieval functions, but they do not implement the low-level query logic themselves. This makes the code easier to reuse. AthenaAI and AInsights both need retrieval, but they use it in slightly different ways. The shared RAG service provides the common base, while each feature decides how to build the final prompt.

The RAG flow also connects the relational database with Qdrant. MySQL stores the official state of subjects, documents, and active document hashes. Qdrant stores the semantic chunks. During retrieval, the backend uses both. This prevents the vector database from becoming an uncontrolled source of truth. Qdrant is used for semantic search, but MySQL still decides which document versions are active and valid.

6.6.8 Document Uploading and Queueing

Document uploading is handled through a separate enqueueing service. This was done because uploading a document and fully processing a document are different operations. Uploading should validate

the file and place it in the queue. Parsing, chunking, embedding, and indexing should happen later through the ingestion worker.

A document can enter the queue from two main paths. The first path is manual upload through the administration endpoint `POST /administration/callDocEnqueuer`. This can be done by a teacher for an assigned course or by a secretary for secretary-managed content. The second path is automated upload through the bot-facing endpoint `POST /bots/callDocEnqueuer`. In both cases, the backend uses the same `doc_enqueuer` service. This is important because manual and automated files should not follow two different ingestion pipelines.

The enqueueing service first normalizes the subject and document names by trimming whitespace. Then it checks that both values are provided. It also checks that the uploaded file has a filename and an extension. After that, it searches the `documents` table to confirm that the document exists for the selected subject. If the document does not exist, the backend rejects the request instead of creating an uncontrolled queue item.

The service then calculates a hash of the uploaded file. This hash is used to detect whether the same content has already been processed or is already waiting in the queue. If the hash is the same as the active document hash, the backend returns that the document is already ingested and up to date. If the same hash already exists in the queue for that document, it returns that the document is already waiting for processing. This avoids duplicate processing and prevents the same file from being indexed multiple times.

If the file is new, the backend marks the document status as `processing`. It then calculates the next queue version for that subject and document, creates a storage filename, and saves the uploaded file under the `waiting_to_be_parsed` directory. The stored filename is based on a hash of the subject, document, and version. This avoids using raw document names directly as file names and reduces problems with special characters or very long names.

After storing the file, the backend inserts a row in `documents_queue`. The queue record includes the subject name, document name, version, path, file hash, status, and queued timestamp. The initial status is `waiting_to_be_parsed`. This means that the worker will first extract text from the file before it can index it.

This queueing design is useful because the upload request remains relatively small. The backend does not attempt to OCR a PDF or embed many chunks during the same request that uploads the file. Instead, it records that work needs to happen and lets the ingestion worker process the queue. This makes the API more stable and easier to recover if a document fails during parsing or indexing.

6.6.9 Administration API

The administration API is responsible for the backend side of the Admin Panel. It includes operations for subjects, documents, LLMs, bot configurations, bot logs, and manual document enqueueing. This part of the backend is strongly connected with role-based access because teachers and secretaries do not have the same administrative permissions.

Subject management is handled through endpoints under the administration router. The `GET /administration/subjects` endpoint lists the subjects available to the authenticated admin

user. For teachers, the backend checks the `teaches_course` table and returns only the courses assigned to that teacher. For secretaries, it returns secretary-managed subjects. This keeps the Admin Panel focused on the data each role is allowed to manage.

The secretary role has the ability to create and delete secretary-managed subjects. Subject creation is handled by the `POST /administration/subjects` endpoint, while subject deletion is handled by `DELETE /administration/subjects/{subjectName}`. This is different from course subjects, which are mainly synchronized by bots from external university systems. When a secretary creates a subject, the backend stores it with the secretary-managed type. When a secretary deletes a subject, the backend also checks its related documents and tries to remove the corresponding vector data from Qdrant. This is important because deleting only the relational record would leave old semantic chunks in the vector database.

Document operations follow a similar rule. The administration API includes endpoints for listing, creating, and deleting documents, as shown in Table 6.2. The document listing endpoint is available to both teachers and secretaries, but the returned data depends on the access rules. Secretaries can create and delete documents only under secretary-managed subjects. Teachers cannot create or delete course document records directly in the same way, because course document records are mainly synchronized from Moodle by the relevant bot process. However, teachers can list the documents of their assigned courses and upload actual files for existing document records.

When documents are deleted by an allowed admin operation, the backend also deletes related vectors from Qdrant using the document key. This is another example of the API acting as the consistency layer between MySQL and Qdrant. MySQL contains the structured record of the document, while Qdrant contains the indexed chunks. If the document is removed from MySQL, the backend should also remove the semantic representation that belongs to it.

The administration API also handles model configuration. The model listing endpoint is `GET /administration/models`, and it returns model records to teachers and secretaries. Teachers can create model records through `POST /administration/models`. They can update them through `PATCH /administration/models/{model_id}`, and delete them through `DELETE /administration/models/{model_id}`. Before creating a model, the backend checks whether another model with the same provider and technical model name already exists. This prevents duplicate model definitions. Secretaries can view model records, but they do not manage them.

Bot configuration is also exposed through the administration API, but only for teachers. A teacher can list bots for their assigned courses through `GET /administration/bots`. Bot configurations are created through `POST /administration/bots`, updated through `PATCH /administration/bots`, and deleted through `DELETE /administration/bots`. Bot logs are viewed through `GET /administration/botLogs`. Every bot configuration operation checks that the teacher is actually assigned to the course through `teaches_course`. This prevents a teacher from configuring bots for courses that do not belong to them.

The backend builds bot identifiers based on the course and pattern. It also returns a simplified bot status to the frontend. For example, if the container field contains the waiting marker, the bot is shown as pending. If a container is set, it can be shown as running. The actual bot scheduling rules are enforced mainly by the database procedures and triggers, but the API is responsible for receiving

the configuration request, checking permissions, and returning the result to the frontend.

Manual document enqueueing is also part of the administration API. Teachers and secretaries can call `POST /administration/callDocEnqueuer`, but the backend checks their role and access before accepting the file. For teachers, access is checked through the teacher-course relation. For secretaries, access is checked by confirming that the subject is secretary-managed. If the user is allowed, the backend sends the file to the shared document enqueueing service.

This design keeps the Admin Panel flexible but controlled. Teachers can manage their course-related data and bot collection settings. Secretaries can manage secretary-related information. Both roles can use the same backend area, but the backend decides which records and operations are valid for each role.

6.6.10 Bot-Facing API

The bot-facing API is separate from the administration API. This distinction is important because teachers use the administration API to configure bots, while bot workers use the bot-facing API to send collected data back to UniMentor. These are different operations and they require different permissions.

The bot-facing API accepts requests only from the `bot` role. This means that a normal student, teacher, or secretary cannot call these endpoints as if they were an automation worker. This keeps automated data collection separated from human platform usage.

The bot-facing metadata update flow is handled through `POST /bots/updateSubjectInfo`. This endpoint receives a subject type and a list of collected subject items. It supports course information, bot-managed general information, and bot-managed announcements. The backend first validates that the subject type is one of the supported automated types and that the subject list is not empty. Then it compares the posted subjects with the existing subjects of the same type in the database.

If a posted subject does not already exist, the backend creates it. If an existing subject of that type is no longer present in the posted data, the backend deletes it. Before deleting old subjects, the backend collects their documents and deletes the related vector data from Qdrant. This prevents outdated bot-managed subjects from remaining searchable after they have disappeared from the external source.

For each subject item, the backend also updates its documents. It compares the posted document names with the existing document names for that subject. New documents are added, and documents that are no longer present are deleted. When a document is deleted, the backend also deletes its Qdrant vectors using the document key. This keeps the knowledge base aligned with the latest data collected by the bots.

For course subjects, `POST /bots/updateSubjectInfo` also updates the `teaches_course` relation. It compares the posted teachers with the existing teachers for the course. New teacher-course relations are added, and old ones are removed. This is important because teacher permissions in the Admin Panel depend on this relation. If the external course data changes, teacher access should also change accordingly.

The second bot-facing endpoint is used for document enqueueing. When a bot collects an actual file, it sends the subject name, document name, and file to `POST /bots/callDocEnqueuer`.

The backend authenticates the bot and then calls the same `doc_enqueuer` service used by manual uploads. This means that bot-collected files and manually uploaded files enter the same parsing and indexing pipeline.

This design keeps bots from directly writing to the database or vector database. Bots collect data, but the backend decides how that data is inserted, updated, deleted, and queued. This improves reliability because the same validation and consistency rules are used for automated and manual input.

6.6.11 Ingestion Worker Integration

The ingestion worker is part of the backend implementation, but it does not run as a normal API endpoint. It is a separate script that processes the document queue. This is necessary because document processing can be heavier than normal API operations. A document may need parsing, OCR, chunking, embedding, Qdrant insertion, and old-version cleanup. These tasks should not block the user upload request.

The ingestion worker uses the same SQLAlchemy session factory as the rest of the backend, but it creates and manages its own session because it runs outside the FastAPI dependency system. It also uses a lock file to avoid multiple worker instances processing the queue at the same time. If a lock file already exists and is still recent, the worker exits. If the lock file is stale, it removes it and continues. This protects the queue from being processed twice by overlapping worker runs.

When the worker starts, it selects the next queue item. The selection logic prioritizes items with the status `waiting_to_be_indexed` before items with the status `waiting_to_be_parsed`. This is useful because if a document has already been parsed, it can move directly to indexing without waiting behind newly uploaded raw files. If no queue item exists, the worker logs that there is nothing to process and exits.

If the selected queue item is waiting to be parsed, the worker calls the parsing job. The parsing job reads the source file from the queue path and chooses the correct parser based on the file extension. The backend supports modern document formats such as PDF, DOCX, PPTX, XLSX, CSV, and several text-based formats. Legacy Office formats such as DOC, PPT, and XLS are rejected because they are less reliable to parse in this setup. For PDFs, OCR can also be forced, which is useful when files contain scanned content.

After parsing, the extracted text is saved as a text file in the `waiting_to_be_indexed` directory. The queue item path is updated to point to this text file, and the queue item status is changed to `waiting_to_be_indexed`. The original uploaded file is then removed. This creates a two-step pipeline: first convert the document into text, then index the text.

If the selected queue item is waiting to be indexed, the worker reads the extracted text file and splits it into chunks. The default chunking process uses a chunk size and overlap so that long documents are divided into smaller pieces while keeping some continuity between neighboring chunks. If no chunks are created, the worker raises an error because there would be nothing to index.

Before indexing, the worker may add title information to the chunks for some subject types. This is done for secretary-managed content, bot-managed general information, and bot-managed announcements. For announcements, the title is extracted from the subject name after the first dash, because

announcement subjects include an id at the beginning. This title prefix gives the retrieved chunk more context later, especially when the body text alone may not clearly show what the information refers to.

The worker then creates metadata for each chunk. This metadata includes the subject name, document name, file hash, chunk index, total chunk count, and document key. It generates embeddings for the chunks and stores them in Qdrant together with the metadata. After indexing succeeds, the worker activates the new document hash in MySQL and deletes the previous vector version if there was one. This is how the platform keeps only the current version of a document active for retrieval.

Finally, the worker deletes the queue item and removes the intermediate text file. If the processing succeeds, it commits the transaction. If an exception occurs, it rolls back the transaction and logs the failure. The lock file is removed in the final step so that a future worker run can process the queue again.

This worker design connects the relational queue with the vector database. MySQL tracks what needs to be processed and what document version is active. The worker performs the expensive processing. Qdrant stores the final vector data. FastAPI connects these pieces together by providing the enqueueing logic and the shared service layer used by the worker.

6.6.12 User Service and Shared User Data

The user service contains simpler backend operations that are needed by multiple frontend features. These endpoints are available to authenticated students, teachers, and secretaries. They do not modify sensitive data. Instead, they return shared information needed by the user-facing pages.

The `GET /users/courses` endpoint returns the available courses. It reads subjects with the type `Course` from the relational database and returns their names ordered alphabetically. This data can be used by features that need course selection, such as AInsights. The frontend does not need to know how courses are stored internally. It only receives the list of course names from the backend.

The `GET /users/models/active` endpoint returns the active LLMs. It queries the `llm_models` table and returns only records where the active flag is enabled. This supports the AI features without exposing inactive model records to normal user selection. The detailed management of model records is handled by the administration API, but the user service provides the smaller read-only view needed by the rest of the application.

Although this service is smaller than the chat or administration services, it is still useful because it avoids duplicating common queries in multiple parts of the backend. It also keeps the frontend dependent on clear API responses instead of directly depending on the database structure.

6.6.13 Conversation Management and AthenaAI Chat

The chat API is responsible for AthenaAI and conversation history. The main generation endpoint is `POST /chat`, which is used when a user sends a message to AthenaAI. Conversation management is handled through a separate group of chat endpoints, as shown in Table 6.2. These endpoints allow the user to create, list, open, pin, unpin, and delete conversations. This makes AthenaAI more than a simple one-message chatbot because users can keep and continue previous conversations.

When a conversation is created through `POST /chat/conversations`, the backend generates a title using the selected model. It sends the initial query to the generation service with instructions to return a concise title. If no useful title is generated, it falls back to `New chat`. The conversation is then stored with a unique id, the user id, title, and update timestamp.

Conversation ownership is checked in the backend. When a user opens, modifies, or deletes a conversation through the conversation endpoints, the backend checks that the conversation belongs to the authenticated user. This is important because a conversation id alone should not be enough to access someone else's messages. The helper function that loads a conversation filters by both conversation id and user id.

Messages are stored separately from the conversation record. User messages are stored with the role `user`, and assistant responses are stored with the role `agent`. When a conversation is loaded through `GET /chat/conversations/{conversation_id}`, the messages are sorted by creation time so the frontend can display them in the correct order. The conversation record also stores metadata such as title, pinned status, update time, and summary information.

The chat generation flow starts after the user sends a message to `POST /chat`. The backend first authorizes the user and resolves the conversation. Then it saves the user message and commits it before generation starts. This early commit is useful because the conversation state is saved even though the response will be streamed over time.

After the user message is saved, the backend loads the conversation history. It removes the newest user message from the history list because that message is passed separately as the latest question. It also loads the stored conversation summary if one exists. This allows the prompt to include both a compact summary of older messages and the recent conversation history.

If RAG is enabled, the chat service calls the RAG retrieval flow described earlier. It retrieves relevant chunks, builds a RAG prompt, and includes the retrieved context together with the conversation summary, recent history, and latest question. If RAG is not enabled, it builds a normal chat prompt that includes the conversation summary, recent history, and latest question, but without retrieved document context. In both cases, the final prompt is sent to the shared LLM generation service.

The response from `POST /chat` is streamed back to the frontend using SSE. At the start of the stream, the backend sends a metadata event with information such as whether RAG is enabled, the value of `k`, the source chunk ids, the conversation id, and the title. Then it streams token events as generated text arrives. If RAG was used and sources exist, it sends a sources event after generation. Finally, it sends a done event.

When generation completes normally, the backend saves the assistant response in the database and commits it. Then it checks whether the conversation summary should be updated. The summary update is not performed after every message. The backend keeps a fixed number of recent messages directly and summarizes older messages only when enough unsummarized messages have accumulated. This prevents the summary from being regenerated too often and keeps the prompt size more manageable for long conversations.

This design gives AthenaAI memory within each conversation without sending the full conversation forever. Recent messages are kept in detail, while older messages are compressed into a summary. The result is a practical balance between context and prompt size. It is not perfect memory, but it is

enough to keep longer conversations usable without making every request too large.

6.6.14 AInsights and Career Mentor Tools

The tools API contains two different AI-based features: AInsights and Career Mentor filtering. Both use the shared generation service, but they are designed for different purposes. AInsights is handled through `POST /tools/templates`, while Career Mentor filtering is handled through `POST /tools/filter`. AInsights is RAG-based and uses course material from the knowledge base. Career Mentor filtering is not RAG-based and works on the job data provided in the request.

The `POST /tools/templates` endpoint receives a course name, a template name, an optional custom prompt, and an optional model identifier. The backend first authorizes the user. Then it retrieves relevant chunks for the selected course by using the template name and custom prompt as the retrieval query. The retrieval is restricted to the selected course and active document hashes, so the generated material is based only on the relevant course context.

After retrieval, the backend normalizes the retrieval hits and extracts the chunk texts. It then builds a template prompt that includes the system instructions, the retrieved context, the course name, the selected template type, and any additional user instructions. The prompt asks the model to use only the provided context and to generate a clear, ready-to-use output appropriate for the requested template.

The AInsights response is streamed using the same general SSE style as chat. The backend first sends metadata with the selected course, template type, selected number of chunks, and source information. Then it streams generated token events. If sources exist, it sends source metadata before the final done event. This allows the frontend to show the generated result progressively while still keeping information about the retrieved material.

Career Mentor filtering uses a different flow. The `POST /tools/filter` endpoint receives the user's filtering prompt and a list of jobs to filter. The backend builds a prompt that includes the user's preferences and the available jobs. Each job is represented with fields such as id, name, and description. The model is instructed to return only a valid JSON array of job ids, ordered by relevance.

After generation, the backend collects the response text and tries to decode it as JSON. If the decoded result is a list, it returns that list as the response. If the model returns invalid JSON or a non-list value, the backend returns an empty list. This makes the feature safer because the frontend expects structured data, not a normal natural language answer.

This distinction between the two tools is important. AInsights uses RAG because it needs course-specific knowledge from the vector database. Career Mentor filtering does not use Qdrant because the relevant job data is already included in the request. Both features still share the same LLM routing layer, which keeps the backend implementation more consistent.

Overall, the tools API shows how the same backend foundation can support different AI workflows. One workflow retrieves knowledge from the indexed course material and generates study-related output. The other workflow uses the model as a reasoning/filtering component over structured job data. In both cases, FastAPI handles authorization, prompt preparation, model usage, and response formatting.

6.7 Bot Development

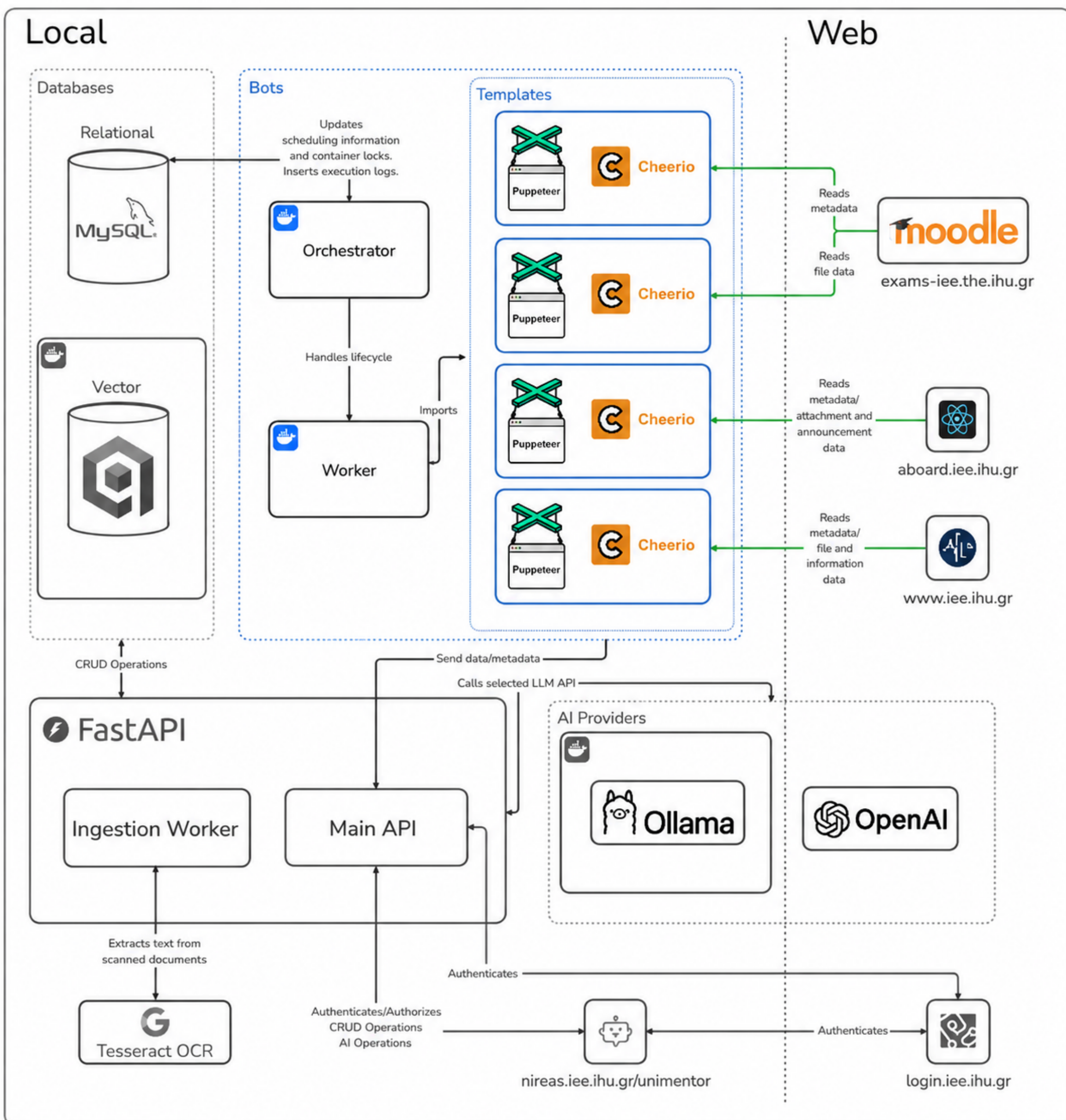

Figure 6.7: General architecture of UniMentor with emphasis on the Bots layer.

The bot subsystem is the part of UniMentor that allows the platform to collect university information automatically from external sources. In the rest of the system, most data is inserted either through the admin panel or through backend ingestion processes. However, some university information is not always available as a clean file that an administrator can upload manually. Course information may exist inside Moodle, announcements may exist in Aboard, and general department information may

exist on public web pages. Because of this, UniMentor needed a mechanism that could visit these sources, read the available data, and send it to the backend for storage and indexing.

The bot architecture was designed as a separate automation layer around the main platform. The bots are not part of the normal user-facing flow. A student does not interact directly with them, and they do not answer chat questions by themselves. Their purpose is to keep the knowledge base updated, minimizing the need for frequent manual monitoring. RAG-based systems depend heavily on the quality and freshness of the indexed material. If the stored documents are outdated, missing, or incomplete, the chatbot may retrieve weak context and produce less useful answers. This weak point makes the bot layer almost as important as the retrieval and generation parts of the platform.

The architecture separates the bot subsystem from the main backend. The main FastAPI backend remains responsible for database operations, document ingestion, parsing, chunking, embedding generation, and vector indexing. The bots only collect information from external sources and call specific backend endpoints. This separation is important because it keeps all sensitive data operations inside the backend. A bot does not directly write to the relational database or the vector database. Instead, it sends data to the backend, and the backend decides how to validate, store, and process it.

The bot subsystem consists of three main parts. The first part is the orchestrator. The orchestrator is responsible for deciding when bots should run and for managing their execution. The second part is the worker. A worker is a temporary Docker container that runs one bot execution. The third part is the template layer. Templates contain the source-specific scraping logic, such as Moodle course synchronization, Moodle document extraction, Aboard announcements synchronization, and department website information extraction.

This separation makes the implementation easier to understand and extend. The orchestrator does not need to know how Moodle pages are structured. The worker does not need to know the scheduling rules of the platform. The templates do not need to know how Docker containers are created or cleaned up. Each part has a specific role. This is useful since browser automation can easily become complex if scheduling, scraping, error handling, logging, and cleanup are mixed together in one script. It also makes the system more flexible, because support for more websites can be added later by creating new templates, without changing the whole bot execution mechanism.

6.7.1 Orchestrator

The orchestrator is the control component of the bot system. Its role is to manage bot execution, not to scrape the external websites itself. It communicates with the relational database to find which bots are ready to run, communicates with Docker to create worker containers, injects the required instructions into each worker, starts the container, monitors its state, and cleans it up after the execution finishes.

When the orchestrator starts, it initializes the main process lifecycle. The entry point creates a logger and then starts the process initialization component. During initialization, the orchestrator prepares the database manager, the Docker manager, the container state manager, the orchestration component, the cleanup component, the reconciliation component, the memory monitor, and the shutdown handler. This structure is necessary since the orchestrator is designed as a long-running service and not as a simple script that only starts one bot and exits.

The database manager initializes a MySQL connection pool. It reads its configuration from environ-

ment variables, including the database socket, username, password, database name, pool size, idle timeout, connection timeout, and query timeout. This approach makes deployment independent of the underlying system, as it only depends on having the correct environment variables set. Using a connection pool also improves stability, since existing connections can be reused rather than establishing new ones for each scheduling check.

The Docker manager is responsible for communication with Docker. The orchestrator uses it to list, create, start, and remove containers, but also to read logs, listen to Docker events, and create the worker network if needed. This means that Docker is not controlled manually for every bot execution. Instead, the orchestrator treats Docker as an execution platform and manages worker containers programmatically.

Before the orchestrator begins its normal scheduling loop, it performs health checks. These checks confirm that the database and Docker connections are available. This is necessary because the orchestrator depends on both. If the database is unavailable, it cannot know which bots should run. If Docker is unavailable, it cannot create workers. In these cases, continuing the process would not be useful, so the orchestrator logs the error and triggers a graceful shutdown.

The main scheduling logic is handled by the orchestration component. The orchestrator periodically calls the stored procedure `run_bots_scheduler`. This procedure receives the maximum number of allowed workers as an argument and returns the bots that are ready to run. The returned data includes the bot id, container name, session token, selected template, course, and parameters. This means that the database remains the source of truth for bot scheduling. The orchestrator does not calculate the full scheduling logic in JavaScript. Instead, it asks the database which bots should be executed at that moment.

This design fits the rest of UniMentor because bot schedules are stored in the relational database. The database already contains bot records with fields such as the course, template, schedule pattern, execution interval, scheduled day, scheduled time, active flag, start and end time, next execution time, execution time, and assigned container. The orchestrator uses this information indirectly through the scheduler procedure. This keeps the scheduling state persistent. If the orchestrator restarts, the information is still available in the database.

After the scheduler returns one or more bots, the orchestrator starts a worker container for each selected execution. However, it does not start unlimited workers. The `WORKER_N` setting controls how many workers can run at the same time. This is necessary since browser automation can consume significant CPU and memory. A worker may launch Chromium, log in to a remote platform, scan pages, read HTML, download documents, and upload files to the backend. If many workers start at once, the server can become unstable. By limiting concurrent workers, UniMentor keeps the automation process controlled.

Before creating a worker, the orchestrator checks if a safe container with the same name already exists. It lists containers with the label `role=orchestrator_worker_container`. This label acts as a simple boundary between bot workers and unrelated Docker containers. It limits the orchestrator to managing only the containers that belong to the bot system, ensuring that other containers running on the server are not accidentally removed or restarted.

If the container does not already exist, the orchestrator makes sure that the worker network has been created. The workers are attached to a Docker bridge network called `orchestrator_worker_`

`network`. This gives the workers the network access they need for their task, while still keeping its setup explicit. The orchestrator creates it automatically if it does not exist, helping the system recover from a missing network without manual setup.

The worker container is created from the image `unimentor/bots/orchestrator_worker:latest`. The orchestrator passes several environment variables to it. These include the selected template, the course, the UniMentor token, Moodle credentials, Aboard credentials, and browser-related paths such as `HOME`, `XDG_CACHE_HOME`, and `XDG_CONFIG_HOME`. These paths point to writable directories under `/tmp/job`, because the worker container uses a restricted filesystem and Chromium still needs places to store temporary profile and cache data.

The orchestrator also injects a file called `instructions.json` into the worker container. This file is placed under `/tmp/job/parameters`. It contains the bot parameters, restart instructions, and previous logs if the worker is being restarted after a failure. This approach was better than passing all the data only through environment variables. Environment variables are useful for simple values, but the instructions can become larger and more structured, especially when they include previous logs and restart steps. For this reason, passing them as a JSON file was more suitable while also simplifying their handling inside the worker.

The worker container is created with resource limits. The orchestrator sets the memory limit, memory swap limit, CPU limit, shared memory size, process limit, and file descriptor limit. It also sets a stop timeout, so that Docker gives the worker a small grace period before forcefully stopping it. These settings matter because the worker runs browser automation, which is less predictable than normal API code. A page may hang, a download may become slow, or Chromium may use more memory than expected. Resource limits reduce the chance that one failed worker affects the whole server.

The orchestrator also restricts the worker container through Docker security options. The worker root filesystem is read-only. All Linux capabilities are dropped. The `no-new-privileges` option is enabled. The container only receives writable volumes for specific paths, such as downloads, parameters, browser profile, cache, crash dumps, and temporary files. This design does not rely on the worker behaving perfectly. It assumes that a worker may fail or behave unexpectedly, so the container itself is restricted.

Another part of the orchestrator design is the locking system. The container state manager allows containers to be locked by orchestration, cleanup, or reconciliation. This prevents different parts of the orchestrator from acting on the same container at the same time. For example, the orchestration component may try to start a container, while the reconciliation component may detect that the same container is missing or stuck. Without locks, these actions could conflict. The locking mechanism makes the lifecycle safer and reduces race conditions.

The orchestrator also uses a simple backpressure mechanism during container creation. Container creation is locked so that only one creation process happens at a time. If another container needs to be started while creation is already in progress, the request is placed in a queue. When the current creation finishes, the next request is processed. This avoids uncontrolled parallel creation of containers, which could create unnecessary load on the Docker daemon and the server.

The orchestrator does not only start workers, but also includes a reconciliation mechanism. Reconciliation compares the database state with the real Docker state. It reads bots that have an assigned container from the database and also lists Docker containers with the worker label. Then it checks

whether the expected containers actually exist and whether their Docker state is valid.

If a bot has a container assigned in the database but the container is missing from Docker, reconciliation can start it again. Containers stuck in the `created` state can be removed and recreated. Those running longer than the configured maximum runtime are stopped and sent to cleanup with a timeout reason. Exited containers trigger cleanup, while dead containers are removed and restarted. This was necessary for a long-running automation service such as the orchestrator. Since it cannot be assumed that every container will follow the ideal lifecycle, or that the orchestrator itself will always be running, reconciliation exists as a fallback to prevent deadlocks and safely restore the last healthy state after a system restart.

Cleanup is responsible for what happens after a worker finishes. The cleanup component listens to Docker `die` events for containers with the worker label. When a worker exits, cleanup reads the container logs, separates stdout and stderr, extracts timestamps, parses structured JSON log messages, and builds a clean execution log. These logs are then stored in the database through the `bots_logs` table. After that, the bot cleanup procedure is called, and the container is removed.

The cleanup logic distinguishes between successful, failed, partially successful, and restartable executions. If the worker exits with code `0`, the execution is stored as successful. If it exits with code `1`, cleanup checks the last structured error. If the error is restartable, the orchestrator starts the container again and passes the previous logs and restart instructions. If the error is not restartable, the execution is stored as failed. If some steps were already completed before the failure, the status can become partial success. If the worker was killed because of memory usage or another abrupt failure, cleanup can also restart it with a suitable reason.

This restart behavior is useful for document extraction. A Moodle course may contain many files. If the worker downloads and sends several files successfully but fails later, it would be wasteful to start from the beginning. Instead, the worker logs the completed steps, and those steps can be passed back as restart instructions. The next execution can skip files that were already processed. This makes the automation more resilient and reduces duplicate work.

The orchestrator also includes a memory monitor and shutdown handling. The memory monitor observes process memory usage and can trigger a shutdown if needed. The shutdown handler is responsible for graceful termination. This matters because the orchestrator is a long-running process. It should not leave containers unmanaged when the process is stopped. The shutdown flow gives the system a chance to stop scheduling, clean internal timers, and exit in a controlled way.

Overall, the orchestrator is the part that gives the bot subsystem reliability. It schedules bot executions, limits concurrency, creates restricted worker containers, injects job instructions, monitors Docker state, handles failed containers, reads logs, stores execution results, restarts workers when appropriate, and cleans resources after execution. It is the subsystem that makes the bots a managed automation system instead of a group of independent scripts.

6.7.2 Worker

The worker is the component that actually runs a bot execution. Each worker runs inside its own Docker container. The container is temporary and is created for one bot job. It receives the selected template, course, authentication token, credentials, parameters, and restart information from the or-

chestrator. Then it executes the selected template, writes structured logs, and exits. After that, the orchestrator reads the logs and removes the container.

The worker starts from its main file. First, it loads the job configuration from `/tmp/job/parameters/instructions.json`. This file includes the template parameters, restart instructions, and old logs. Then it reads environment variables such as the template name, course, and token, as well as the Moodle and Aboard usernames and passwords. These values are combined into a single configuration object that is passed to the selected template.

After reading the job configuration, the worker reads the template manifest. The manifest maps template names to JavaScript files. The current implementation includes four templates: `course_info_sync`, `announcements_info_sync`, `general_info_sync`, and `document_extraction_bot_template`. Each template has its own script file. For example, course information synchronization maps to `courseInfo.js`, announcements synchronization maps to `announcementsInfo.js`, general information synchronization maps to `generalInfo.js`, and Moodle document extraction maps to `documentExtraction.js`.

The worker does not hardcode the logic of every bot. Instead, it imports the correct template dynamically based on the manifest. It checks that the selected template exists and that the script exports a `main()` function. Then it creates a template logger and runs the template's `main()` function. This makes the worker generic. A new bot template can be added later by creating a new script and registering it in the manifest, without changing the main worker lifecycle.

The worker also manages the browser instance. It uses a `WorkerBrowser` class that initializes Chromium through Puppeteer. The implementation uses `puppeteer-extra` together with the stealth plugin. The browser runs in headless mode and uses the Chrome executable installed inside the worker image. The launch arguments disable unnecessary browser features, such as default apps, sync, extensions, translation features, GPU usage, and background networking. This keeps the browser more suitable for automation inside a container.

The browser uses controlled writable directories. The user data directory is placed under `/tmp/job/profile`. The disk cache is placed under `/tmp/job/cache`. Crash dumps are placed under `/tmp/job/crashes`. This is necessary because the worker container has a read-only root filesystem. Chromium still needs writable paths to function correctly, so these paths are explicitly provided through Docker volumes, as mentioned in Section 6.7.1.

The worker does not always use the browser for every step. In some templates, the browser is needed mainly for login and navigation. After authentication is completed, normal HTTP requests can be used for API calls or file downloads. This is visible in the announcements template. The worker opens the Aboard platform in the browser, performs the login flow, and captures the authentication token from a network response. After that, it can use `fetch` requests with the token to retrieve announcement data and attachments. This combination is practical because browser automation is useful for login flows, while direct HTTP requests are faster and simpler for structured data retrieval.

The worker writes structured logs instead of plain text messages only. There are two main log categories. The first category is normal template logs, written as `worker_logs`. These logs describe what the bot is doing, such as logging in, scanning pages, finding documents, sending data, or skipping a file. The second category is `worker_logs_to_orchestrator`. These logs are used by the orchestrator to understand lifecycle events, errors, restart information, and completed steps.

The worker writes a start message when a new execution begins. If the worker is restarted with old logs, it prints the old logs again and then writes a restart message. At the end of a successful execution, it writes an end message. These messages help the cleanup component understand whether the worker started correctly, restarted, completed normally, or failed before reaching the end.

Errors are represented through the `WorkerError` class. A worker error contains a message, a reason, a restart flag, and optional metadata. This is useful for error handling. A timeout may be restartable. A missing template is not restartable. A failed file download may allow partial progress. By attaching metadata and restart information to the error, the worker gives the orchestrator enough information to decide the next step.

The worker also handles termination signals and unexpected exceptions. If it receives a termination signal, it attempts to destroy the browser and writes a structured error. If an uncaught exception happens, the worker wraps it in a `WorkerError`, logs it, and exits with a failure code. This does not prevent every possible crash, but it makes many failure cases visible. Instead of a silent failure, the orchestrator receives structured information that can be stored in the bot logs.

This design keeps the worker focused. It does not decide when bots should run. It does not decide how to store logs in the database. It does not parse or index documents itself. Its responsibility is to load one job, execute one template, use the browser or HTTP requests when needed, send data to the backend, write logs, and exit with a meaningful status.

6.7.3 Templates

The template layer contains the actual source-specific bot logic. Each template is responsible for a different kind of external information. This is where the system knows how to interact with Moodle, Aboard, or the department website. The common worker lifecycle stays the same, but the template determines what will be collected and how it will be sent to UniMentor.

The first template is `course_info_sync`. This template collects course structure and document metadata from Moodle. It logs in to Moodle using the configured Moodle credentials and navigates through the course categories. The goal of this template is not to download all course documents. Its main purpose is to synchronize the platform with the available courses, teachers, and document names.

The course information template scans the Moodle category structure and identifies semester sections. For each semester, it reads the courses that belong to it. The subject name is constructed using the semester and the course title, so the platform can distinguish courses more clearly. The template also visits the course information area to collect teacher names. This information is then prepared in a structure that the backend can use to update subjects, teachers, and documents.

The template also scans available document references. It checks Moodle resources, file manager areas, and folder activities. It reads file names, filters them based on supported extensions, cleans invalid filename characters, and handles duplicate names. Duplicate handling is important since different Moodle areas may contain files with the same visible name. If those names are sent to the backend without normalization, they can create conflicts or confusion.

After collecting the course information, the template sends the result to the backend endpoint `/bots/updateSubjectInfo`. This endpoint receives the subject structure, teachers, and document

names. The backend can then update the relational database and create or update missing document records. This means that the course information template prepares the platform for later ingestion, but it does not perform the full document download and indexing process by itself.

The second template is `document_extraction_bot_template`. This template is responsible for downloading actual Moodle files and sending them to the backend. It receives the selected course from the worker environment. The course name includes information that allows the template to identify the correct semester and course title. The template then logs in to Moodle, navigates to the correct semester, finds the correct course, and enters the course page.

Some Moodle courses may require self-enrollment. The template includes logic for this case. If self-enrollment is available without a password, the bot attempts to enroll automatically. If a password is required, the bot does not try to bypass it. It logs the situation instead. This is important because the bot should automate normal access, not force access to restricted material.

After entering the course, the document extraction template scans the available files. It collects files from normal resources, file manager components, and folder activities. For each file URL, it may perform a HEAD request to read the Content-Disposition header and determine the actual filename. This is needed because the visible Moodle link is not always the final clean filename. The template then checks whether the extension is supported and normalizes the name.

The document extraction template can also receive a `documentsToSkip` parameter. This allows the backend or bot configuration to exclude specific files from processing. The template also uses restart information to avoid repeating completed work. When a file is processed, the template logs a step. If the worker fails later and is restarted, these steps can be used to skip files that were already handled. This is one of the most important practical parts of the implementation because course pages may contain many documents, and failures can occur midway through the process.

When a file is ready to be sent to UniMentor, the template downloads it and creates a `FormData` request. It sends the file to the backend endpoint `/bots/callDocEnqueuer`. The backend then takes over the ingestion process. It stores the file, creates a queue entry, parses the content, chunks it, creates embeddings, and indexes the chunks in the vector database. This keeps the bot simple, only collecting and forwarding the document. The backend remains responsible for how documents are processed.

The third template is `announcements_info_sync`. This template collects announcements from Aboard. The template first opens the Aboard platform in the browser and performs login with the configured credentials. During login, it waits for the authentication response and extracts the access token. This token is then used to make direct API requests for announcements and attachments.

The announcements template requests announcements updated within a recent period. For each announcement, it constructs a subject using the announcement id and title. This is useful because announcements are not the same as normal courses. They are individual information items, and their titles may not be unique on their own. Including the id in the subject name helps keep them distinct.

For announcement bodies, the template creates a text document. The body content may contain HTML, so the template uses Cheerio to parse it and extract readable text. This matters for RAG since raw HTML is not useful as indexed content. The chatbot needs clean text that represents the actual announcement message. The generated body text is then sent as an `info.txt` document to

the backend.

The announcements template also handles attachments. It checks each attachment URL, filters by allowed file extensions, normalizes filenames, and handles duplicates. Attachments are downloaded and sent to `/bots/callDocEnqueuer`, just like Moodle documents. Before sending documents, the template also calls `/bots/updateSubjectInfo` so the backend knows which announcement subjects and documents should exist.

The fourth template is `general_info_sync`. This template collects public information from the department website. It differs from the Moodle and Aboard templates because it does not primarily rely on a browser login flow. Instead, it uses HTTP requests and Cheerio to fetch and parse HTML pages. It starts on the main department website and follows internal links that are useful for the knowledge base.

The general information template avoids irrelevant or noisy links. For example, it excludes links that belong to categories such as news or events when they are not useful for the intended general information collection. It also avoids language switch links, bookmark links, and category links that would create duplicate or low-value pages. This filtering is necessary for a public website, as it may contain many links that do not need to be included in RAG material.

For each selected page, the template extracts the main content area. It removes elements that are not useful for text retrieval, such as base64 profile images and some public profile sections. If a page contains links to allowed file types, those files are treated as documents. The results are grouped by page or subject, and the template sends the subject information and document list to the backend. Then it sends the actual text or files to the document enqueuer endpoint.

Although the templates target different sources, they follow the same basic pattern. First, they access an external source. Second, they collect and clean the information. Third, they normalize names and filter unsupported files. Fourth, they update the backend with the available subject and document structure. Finally, they send actual content to the backend for ingestion. This common pattern makes the system consistent. It also makes it easier to add more templates later.

The templates use a shared idea of supported file types. These include common university document formats such as PDF, DOC, DOCX, PPT, PPTX, XLS, XLSX, CSV, TXT, and Markdown. They also include programming and structured text formats such as JSON, JavaScript, TypeScript, Python, Java, C, C++, PHP, HTML, CSS, YAML, XML, and SQL. These file types complement UniMentor by supporting course materials that include programming examples, datasets, configuration files, and source code, as well as lecture notes.

The template implementation also shows why the bot subsystem was not implemented as a simple crawler. Each source has different behavior. Moodle needs login, course navigation, document discovery, and sometimes self-enrollment. Aboard needs browser-based authentication and then API requests. The department website needs HTML crawling and filtering. A single generic crawler would not be enough. The template-based design allowed each source to have its own logic while still using the same worker and orchestrator infrastructure.

6.7.4 Security Concerns and Design Choices

The bot subsystem introduces security concerns that differ from those of the normal backend API. A normal API endpoint receives requests and responds to them. A bot worker actively visits external websites, logs in with credentials, runs a browser, downloads files, and sends data back to the platform. This creates risks related to resource usage, external content, credentials, file handling, and process isolation.

The first security concern is isolation. Workers are not executed inside the main FastAPI backend process. They are executed inside temporary Docker containers. This means that if a worker crashes, hangs, or uses too many resources, the backend does not crash with it. The backend can continue serving users and processing normal requests. The worker is treated as a disposable execution unit that can be removed and recreated.

The second concern is filesystem access. Browser automation needs writable paths, but the worker should not be able to write everywhere. For this reason, the worker container uses a read-only root filesystem. Only specific paths are writable, such as downloads, parameters, browser profile, cache, crash dumps, and temporary files. This limits the damage that a faulty template or browser process can cause inside the container.

The third concern is container privileges. The worker is created with all Linux capabilities dropped and `no-new-privileges` enabled. This reduces what the process can do inside the container. Even if the browser or a template behaves unexpectedly, the container has fewer privileges than a normal, unrestricted process. This matters because the workers interact with external web content, which should not be trusted completely.

The fourth concern is resource control. Each worker has limits on memory, memory swap, CPU, processes, shared memory, and file descriptors. These limits protect the server from heavy or broken automation jobs. A Moodle page may load slowly, a browser may consume more memory, or a loop may cause repeated requests. The worker limits reduce the likelihood that a single job affects the entire UniMentor deployment.

The fifth concern is execution time. A worker should not run forever. The reconciliation component checks running containers and compares their creation time with the maximum allowed runtime. If a worker exceeds that runtime, it is stopped and cleaned up with a timeout reason. This prevents stuck browser sessions from staying alive indefinitely.

Another important issue is Chromium sandboxing. The browser is launched with `--no-sandbox` and `--disable-setuid-sandbox`. This is often used in containerized Puppeteer environments because the browser sandbox can be difficult to use inside Docker. However, it also means that the browser's own sandbox is not the main protection mechanism. In this design, the protection is shifted to the Docker layer. The worker container is temporary, restricted, read-only, capability-limited, resource-limited, and isolated from the main backend process. This does not remove all risk, but it reduces the impact of a browser-level problem.

Credentials are another security concern. Moodle and Aboard credentials are not written directly inside the template code. They are passed through environment variables by the orchestrator. The UniMentor token is also passed to the worker and used when calling backend bot endpoints. This keeps secrets outside the source files and makes deployment configuration easier to control. It also

means that templates can be reused without hardcoded credentials.

The backend is still the authority for internal data operations. Workers do not directly modify the MySQL database or the vector database. They call backend endpoints such as `/bots/updateSubjectInfo` and `/bots/callDocEnqueuer`. The backend can authenticate these requests and apply its own validation before changing the platform state. This design reduces the risk of giving scraping scripts direct access to internal storage.

The design also protects stability through cleanup and reconciliation. Worker containers are removed after execution. Logs are read and stored, and then the container is cleaned. If a container is missing, stuck, exited, dead, or running too long, reconciliation corrects the state. This prevents the server from slowly filling with abandoned containers and temporary data. It also makes bot execution easier to monitor because each run produces a stored log.

Structured logging is also part of the security and reliability design. The worker writes JSON logs, and the orchestrator parses them. This makes failures easier to understand. For example, an administrator can see whether the bot failed during login, page scanning, file download, backend upload, or cleanup. This is better than storing only raw console text because the logs can include status, reason, metadata, timestamps, and restart information.

Partial success handling is another important design choice. In a long document extraction process, some files may be processed before a failure occurs. The system records completed steps and can restart without repeating everything. This avoids unnecessary duplicate downloads and backend calls. It also makes failures less harmful because progress is not always lost.

Overall, the security approach is based on separation and control. The orchestrator controls when workers are created. Docker controls the worker environment. The worker controls one template execution. The templates collect data but do not directly change internal databases. The backend remains responsible for validation and ingestion. Cleanup and reconciliation keep the runtime state clean. This makes the bot subsystem safer and more suitable for a university server, where stability, controlled resource usage, and maintainability are important.

With this design, UniMentor can automatically collect information from Moodle, Aboard, and the department website without turning the main backend into a web scraper. The bot subsystem works as a managed automation layer. It extends the platform's ability to stay up to date while keeping scraping, browser execution, scheduling, and failure handling separate from the core RAG and user-facing services.

6.8 Frontend Development

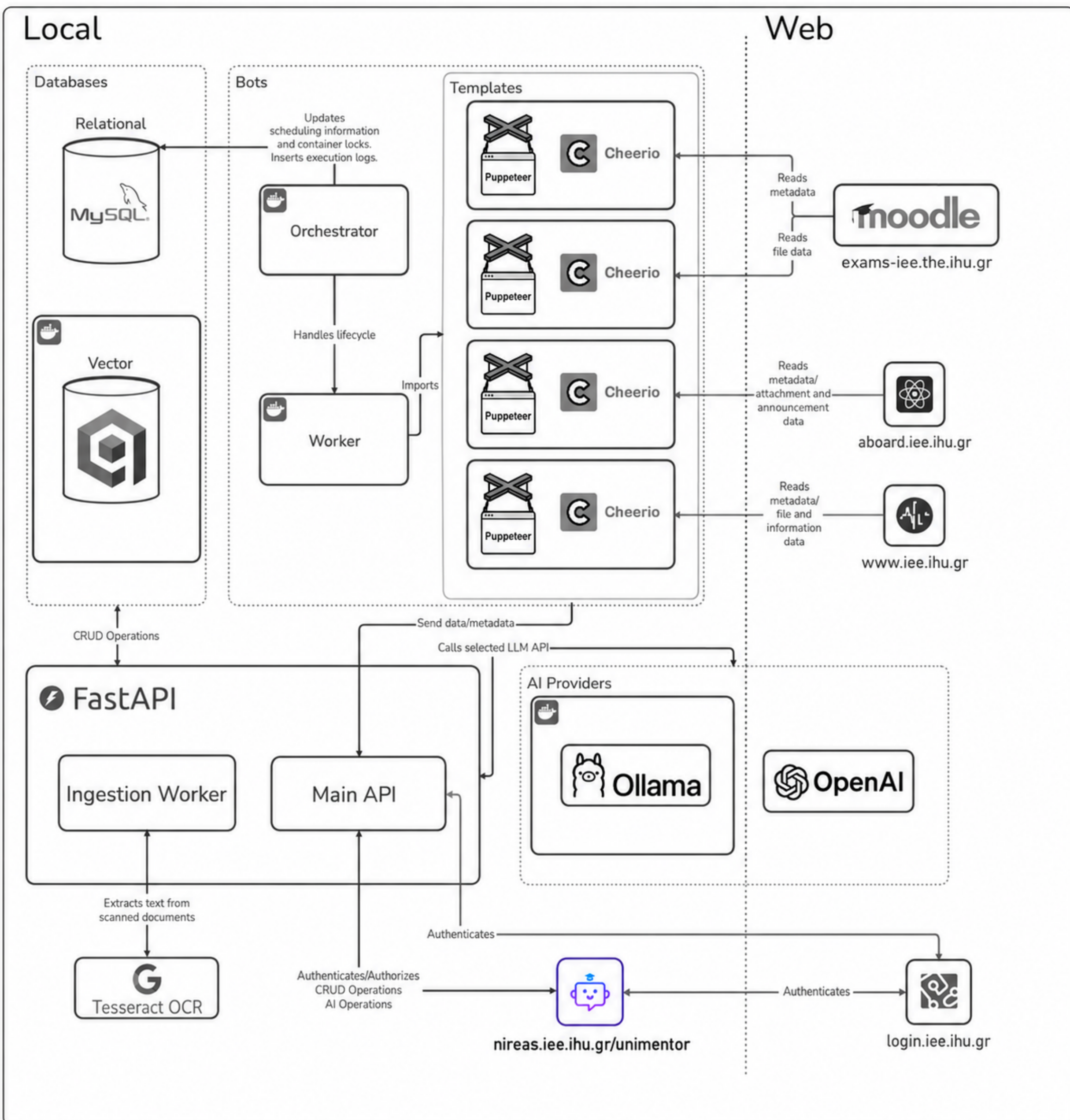

Figure 6.8: General architecture of UniMentor with emphasis on the FrontEnd layer.

6.8.1 Frontend Role in the System Architecture

The frontend is the part of UniMentor that the user interacts with directly. As shown in Figure 6.8, it is placed on the web side of the system and is served through the UniMentor web path. Its role is to provide the interface for the main platform features, while the backend remains responsible for

authentication, database access, RAG retrieval, AI provider communication, document processing, and bot-related operations.

The frontend does not communicate directly with MySQL, Qdrant, Ollama, OpenAI, Tesseract OCR, or the bot subsystem. All these operations are accessed through the FastAPI backend. This separation is important because the frontend should not contain sensitive logic or credentials, and it should not decide whether a user is allowed to perform a protected action. Instead, it sends requests to the backend and presents the result to the user.

The frontend was developed with React, TypeScript, and Vite. React was used for building the user interface as reusable components, TypeScript was used to make the data structures and service calls more predictable, and Vite was used as the frontend build tool. The project also uses Tailwind CSS and shadcn/Radix-style UI components for styling and interface elements. This makes it easier to build a consistent interface across the different pages of the platform.

The deployment configuration was also adapted to the university server path. The Vite configuration uses `base: "/unimentor/"` and the React router uses `basename="/unimentor"`. This was necessary because the application is not served from the root of the domain, but from the `/unimentor` path. Without this configuration, page routing and static assets could break after deployment.

6.8.2 Project Structure and Routing

The frontend project is organized into pages, components, services, shared libraries, and assets. The main page files are placed under `src/pages`. These include the Home page, the AthenaAI Chat page, the AInsights page, the Career Mentor page, and the Admin Panel page. Reusable UI elements are placed under `src/components`, while API-related logic is placed under `src/services` and `src/lib`.

The main application routing is defined in `App.tsx`. The application uses React Router and defines routes for the main pages of UniMentor. The root route opens the Home page, `/chat` opens AthenaAI, `/templates` opens AInsights, `/career` opens Career Mentor, and `/admin` opens the Admin Panel. This keeps the navigation clear and separates the major platform features into their own pages.

The `/admin` route is protected on the frontend side. It is wrapped with a `ProtectedRoute` component that only allows users with the teacher or secretary role to access the Admin Panel. If another role tries to open this route, the user is redirected back to the Home page. This does not replace backend authorization, but it improves the user flow and prevents users from seeing pages that are not intended for their role.

6.8.3 Authentication Context and Protected Routes

Authentication is handled globally through the `AuthProvider`. When the application starts, the provider calls the authentication service and checks whether the user has a valid session or whether the login callback contains an authentication code. If the backend returns a valid role, the role is stored in the authentication context. If the backend returns a redirect marker, the frontend redirects the user

to the university login page.

The authentication context stores the current role, the loading state, and possible authentication errors. This information is used by the rest of the frontend. For example, while the authentication check is still running, the application shows a loading screen. If authentication fails, it shows an error screen with a refresh button. If authentication succeeds, the normal application routes are shown.

A small but useful detail is that the frontend removes query parameters from the URL after authentication is handled. This keeps the visible URL cleaner after the login callback. Without this, the browser could continue showing the authentication code or error parameters in the address bar even after the user has entered the application.

The stored role is also used for conditional interface behavior. The Home page shows the Admin Panel button only for teachers and secretaries. The Admin route also checks the role through the protected route component. Inside the Admin Panel, the role affects which parts of the interface are available. For example, the Bots tab is available to teachers, while secretary users focus on secretary-managed knowledge base content.

6.8.4 API Communication Layer

The frontend communicates with the backend through a shared API layer. The base API URL is read from `VITE_API_URL`, and the helper function `apiUrl()` is used to build complete backend paths. This makes the API base configurable between local development and deployment without changing every request in the code.

The shared API helper provides functions for normal JSON requests, such as `getJSON()`, `postJSON()`, `patchJSON()`, and `delJSON()`. These helpers attach `credentials: "include"` to the requests. This is important because the backend authentication uses an HTTP-only session cookie. The frontend does not manually store or attach the session token. Instead, the browser sends the cookie with each request.

Error handling is also centralized in the API helper. If the backend returns an error response, the helper tries to read the error message from the JSON body. If the backend returns an authentication redirect marker, the frontend redirects the browser to the login page. This avoids repeating the same error-handling logic in every page.

The service files then build on top of this shared API layer. For example, `auth.ts` handles authentication checks, `user.ts` loads courses and active models, `chat.ts` handles chat and conversations, `tools.ts` handles AInsights and Career Mentor filtering, and `admin.ts` handles the Admin Panel operations. This keeps most raw endpoint paths outside the page components and makes the frontend easier to maintain.

File uploads are handled slightly differently because they require `FormData`. In the administration service, document uploads are sent to the backend using a multipart request with the subject name, document name, and selected file. This is used when a teacher or secretary manually uploads a file for processing.

6.8.5 Streaming Response Handling

Streaming is an important part of the frontend because both AthenaAI and AInsights receive generated text gradually from the backend. The frontend uses a shared streaming helper called `streamText()`. This helper sends a POST request with `Accept: text/event-stream` and then reads the response body as a stream.

The stream helper decodes the response using a `TextDecoder`, keeps an internal buffer, splits the incoming content into Server-Sent Event frames, and reads the event type and data from each frame. It handles token-like events such as `token`, `delta`, and `message`. It also handles metadata, completion, error, and heartbeat events. This matches the streaming behavior implemented in the FastAPI backend.

The main benefit of this helper is that the pages do not need to parse SSE manually. The Chat page and the Templates page call the stream helper and provide callback functions for received text chunks, metadata, and completion. This keeps streaming behavior consistent across different AI features and avoids duplicate parsing logic.

In the Chat page, streaming is also combined with a small typewriter effect. Instead of showing every received chunk instantly in a rough way, the page queues the received text and displays it more smoothly. The user can also cancel an active stream. This is useful because local models or long answers may take more time, so the interface should remain responsive while generation is happening.

6.8.6 Reusable UI Components and Styling

The frontend uses Tailwind CSS for styling and a set of reusable UI components based on shadcn/Radix-style components. These include buttons, cards, inputs, labels, select boxes, dialogs, tables, badges, tabs, switches, checkboxes, avatars, scroll areas, tooltips, calendars, and popovers. Using reusable components helps keep the interface consistent across pages.

The visual style is based on rounded panels, cards, shadows, badges, dialogs, and transparent or glass-like sections. This style appears in the Home page cards, Chat layout, AInsights panel, Career Mentor cards, and Admin Panel. The goal is not only to make the interface look better, but also to make the different tools feel like parts of the same platform.

A special reusable component is the chat message bubble. Assistant messages are rendered with Markdown support, including lists, tables, inline code, and code blocks. Code blocks are displayed with syntax highlighting and a copy button. This is useful because AthenaAI and AInsights may return formatted explanations, programming examples, study material, or tables. User messages and AI messages also use different layouts and avatars, which makes the chat easier to read.

6.8.7 Home Page and Navigation

The Home page is the entry point of UniMentor. It presents the main platform tools as large visual cards. These cards represent AthenaAI Chat, AInsights, and Career Mentor. Each card uses a local image asset and leads to the corresponding route when it is active. This gives the user a simple starting point instead of showing all platform functionality at once.

The Home page also uses the authentication role to show the Admin Panel button only to teachers and secretaries. This keeps the student interface simpler, while still allowing administrative users to reach the management area quickly. Career Mentor is also marked visually as inactive or coming soon on the Home page, even though the Career page implementation exists. This reflects the current state of the feature as a demonstration or partial feature rather than a fully connected job portal.

The navigation is handled with React Router. The user can move from the Home page to Chat, AInsights, Career Mentor, or the Admin Panel. Since the router is configured with the `/unimentor` base path, these routes work correctly inside the deployed university server path.

6.8.8 AthenaAI Chat Interface

The AthenaAI Chat page is one of the most complete frontend features. It supports conversation history, active conversation selection, new conversation creation, message sending, pinned conversations, conversation deletion, RAG mode selection, model selection, streaming response display, and stream cancellation.

When the page loads, it requests the active LLMs and the user's existing conversations from the backend. The active models are used in the model selector, while the conversations are shown in the sidebar. If the user opens an existing conversation, the frontend requests the full conversation and displays its messages in order.

When the user sends a message without an active conversation, the frontend first creates a new conversation. After that, it sends the actual chat request to the backend. The request includes the user query, the conversation id, the selected model id, and whether RAG is enabled. The backend then streams the answer, and the frontend appends the received text to the current assistant message.

The chat interface also includes useful controls around conversation management. Users can start a new chat, open a previous chat, pin or unpin conversations, and delete conversations. Pinned conversations are displayed separately from normal recent conversations, which makes important chats easier to find later.

The message rendering is designed for AI output. The assistant can return normal text, Markdown, tables, code snippets, and structured answers. The frontend renders these outputs in a readable way using the message bubble component. This is important because UniMentor is not only a casual chatbot. It may produce explanations, course summaries, programming answers, or study-related material.

6.8.9 AInsights Interface

The AInsights page is the frontend interface for generating course-based study material. The user selects a course, a template type, and an active model. The page also provides an optional custom prompt field, which allows the user to add extra instructions to the generation request.

When the page loads, it requests the available courses and active LLMs from the backend. The course list is used in the course selector, while the model list is used in the model selector. This means that the page does not hardcode available courses or models. It depends on the backend and database state.

When the user starts generation, the page sends the selected course, template type, optional custom prompt, and selected model id to the backend. The backend performs course-restricted RAG retrieval and streams the generated result. The frontend receives the stream and appends the generated chunks to the response area.

The generated output is displayed with Markdown support, which makes it suitable for summaries, quizzes, study plans, outlines, and other structured study material. The page also supports PDF export using `jsPDF`. This allows the user to download the generated material instead of only reading it inside the browser.

6.8.10 Career Mentor Interface

The Career Mentor page provides a proof-of-concept interface for job listing and AI-assisted filtering. In the current implementation, the job data is stored locally in the frontend as mock data. This means that the page demonstrates the intended user experience, but it is not yet connected to a real job provider.

The page presents job cards with information such as title, company, type, location, salary, and description. The user can apply normal filters, such as job type, company, location, remote availability, and search text. Since the current data is local, these filters are handled directly in the frontend.

In a real production scenario, the job data would need to be fetched from a secure third-party API. This connection should be handled through the backend, not directly from the frontend, because API keys, access tokens, and provider-specific logic should remain on the server side.

The AI filtering option works on top of the available job list. The user writes a prompt describing their skills, experience, or preferences. The frontend sends this prompt, the available jobs, and the selected model to the backend tools API. The backend returns the ids of the most relevant jobs, and the frontend uses them to filter the displayed job cards.

This feature is not RAG-based because it does not retrieve knowledge from Qdrant. Instead, it asks the selected model to reason over the job data provided in the request. In the current version this data is mock data, while in a complete implementation it would come from a secure job listings API through the backend.

The page also includes job detail dialogs and an application dialog. The application portal is still presented as under construction, so the current Career Mentor implementation works mostly as a demo job hub with filtering and AI-assisted matching.

6.8.11 Admin Panel Interface

The Admin Panel is the largest frontend page because it includes several management areas. It is available only to teachers and secretaries through the protected route. The page is organized into tabs for Knowledge Base, Models, and Bots. The Bots tab is shown only to teachers, because bot configuration is not available to secretary users.

The Knowledge Base tab allows administrative users to view subjects, select a subject, view its documents, filter the lists, and upload files for document processing. Secretaries can also create and delete

secretary-managed subjects and documents. Teachers mainly work with their assigned courses and existing course documents. Document statuses such as `missing`, `processing`, and `processed` are shown with badges, so the user can understand the indexing state more easily.

When a file is uploaded from the Admin Panel, the frontend uses a dialog and sends the selected file through the administration service using `FormData`. After the upload, the backend places the file in the document queue. The frontend then refreshes the document list so that the updated processing state can be shown to the user.

The Models tab handles the LLM records. Teachers can create, edit, delete, activate, and deactivate models. The model form includes the display name, technical model name, provider, and active flag. The supported provider values are Ollama and OpenAI. Secretaries can view model information, but the management actions are limited according to their role.

The Bots tab is teacher-only. It allows teachers to view bot configurations for their courses, create new bot settings, update existing settings, delete bots, and view bot execution logs. The bot form includes scheduling-related fields, such as one-time or repeat-interval pattern, interval type, scheduled time, scheduled day, start time, end time, repeat yearly setting, active flag, and parameters. This connects the frontend with the bot scheduling logic that is handled in the backend and database.

Bot logs are also presented through the Admin Panel. The frontend can load logs for a selected bot and filter them by execution period and status. Log statuses are displayed with badges, and individual log entries can be inspected. This makes the bot subsystem easier to monitor from the user interface instead of requiring the user to inspect server logs manually.

Overall, the frontend implementation provides the user-facing layer of UniMentor and connects each page to the corresponding backend services. It keeps routing, authentication state, API requests, streaming behavior, reusable UI components, and page-specific logic separated as much as possible. This makes the interface easier to extend while still supporting the main platform features: AthenaAI, AInsights, Career Mentor, and the Admin Panel.

6.9 Source Code Availability

The source code of UniMentor is available in a public GitHub repository. The repository was created so that the implementation of the platform can be accessed, reviewed, installed, and extended outside the thesis document. It contains the complete project structure, including the frontend application, the backend API, the bot subsystem, the Docker configuration, the setup scripts, and the runtime commands that are used for local development and remote deployment.

The public repository is available at:

`https://github.com/it185206/Unimentor-Project-Public`

The repository is not only used as a place where the source code is stored. It also works as the practical setup guide for the platform. This is important because UniMentor is not a single standalone program. It is made from multiple connected parts, such as the frontend, the backend API, the vector database, the local AI model service, the document ingestion worker, and the bot orchestrator. For this reason, the correct setup of the environment is as important as the source code itself.

6.9.1 Repository Structure

The repository is organized based on the main components of UniMentor. The `frontend/` directory contains the React and Vite application that provides the user interface of the platform. This includes the pages for AthenaAI, AInsights, Career Mentor, and the Admin Panel. The `backend/api/` directory contains the FastAPI backend, which handles authentication, role-based access, chat requests, RAG retrieval, AI tools, administration operations, document queueing, and communication with AI providers.

The `backend/bots/` directory contains the bot subsystem. This includes the bot orchestrator and the logic required to run browser automation workers in a controlled way. The `scripts/` directory contains helper scripts for deployment and runtime operations. The root `docker-compose.yml` file is used for Docker services such as Qdrant and Ollama. The root `package.json` contains local development commands, while the root `Makefile` provides the main command entry point for remote server operation.

This structure follows the architecture of the system. Each important layer of UniMentor is kept in a separate part of the codebase, which makes the project easier to understand and maintain. The frontend does not contain backend logic, the backend does not contain bot worker code, and the bot subsystem is separated from the normal API service. This separation also makes it easier to update one part of the platform without changing the whole project.

6.9.2 Required Environment Files

Before UniMentor can run locally or remotely, the required environment files must be created and configured. These files are not committed as final ready-to-use configuration files, because their values depend on the machine, deployment domain, database settings, authentication settings, Docker configuration, and available local tools.

The main required environment files are:

- `backend/api/.env`
- `frontend/.env.production`
- `backend/bots/options.env`

These files are usually created from their example versions:

```
cp backend/api/.env.example backend/api/.env
cp frontend/.env.production.example frontend/.env.production
cp backend/bots/options.env.example backend/bots/options.env
```

6.9.3 Local Development Requirements

For local development, the developer must install the tools needed by the different parts of the platform. The frontend requires Node.js and npm. The backend requires Python and a Python virtual environment. Git is needed to clone and manage the repository. Docker is needed to run services such as Qdrant and Ollama. Tesseract OCR and Poppler are also required because UniMentor supports document ingestion from different file types, including scanned PDFs or PDF files that may need conversion during processing.

The main local requirements are:

- Node.js and npm
- Python 3
- Git
- Docker Desktop or Docker Engine
- Tesseract OCR
- Poppler

The installation can be checked with commands such as:

```
node -v
npm -v
python --version
docker --version
tesseract --version
pdfinfo -v
pdftoppm -v
```

6.9.4 Local Setup Steps

The local setup starts by cloning the repository from GitHub. After the repository has been downloaded, the developer must move into the project root. From there, the environment files must be created from their example versions and edited with the correct local values.

A typical local setup begins with:

```
git clone https://github.com/it185206/Unimentor-Project-Public
cd Unimentor-Project-Public
```

Then the required environment files are prepared:

```
cp backend/api/.env.example backend/api/.env
cp frontend/.env.production.example frontend/.env.production
cp backend/bots/options.env.example backend/bots/options.env
```

After the files are created, they must be edited. The backend file must point to the correct database, Ollama service, models, CORS origins, and any document-processing paths. The frontend file must point to the correct backend API URL. The bot options file must match the local Docker and database environment if the bot subsystem is going to be used locally.

The next step is to install the project dependencies. The repository provides root npm commands that help set up both the frontend and backend from the project root. The general setup command is:

```
npm install
npm run setup
```

The `npm run setup` command executes the setup steps for the frontend and backend. The frontend setup installs the npm dependencies inside the `frontend/` directory. The backend setup creates the Python virtual environment under `backend/api/.venv` and installs the backend requirements. This makes the setup easier because the developer does not need to manually enter each subfolder and execute every command separately.

After the dependencies are installed, the required Docker services can be started. Qdrant is needed for vector storage, and Ollama is needed for local AI-related models. The services can be started with:

```
docker compose up -d qdrant ollama
```

After starting the containers, their status can be checked with:

```
docker ps
```

If the required Ollama model is missing, the model pull script can be used:

```
bash ./scripts/pull-ollama-models.sh
```

This step is important because the RAG flow depends on embeddings, and the platform expects the required embedding model to be available. If the model is not available in Ollama, embedding generation will fail and retrieval will not work correctly.

When the services are ready, the frontend and backend can be started. The frontend can be started with:

```
npm run dev:frontend
```

The Vite frontend usually runs on:

```
http://localhost:5173
```

The backend can be started with:

```
npm run dev:backend
```

There is also a command for starting both frontend and backend together:

```
npm run dev
```

6.9.5 Remote Server Requirements

The remote server setup is used when UniMentor is deployed in a server environment. In this case, the platform is not only started as a development application. The frontend is built into static files, Apache serves the frontend, Apache proxies API requests to the backend, Docker runs the required supporting services, and cron jobs are used for backend recovery and document ingestion.

The server must have the required system dependencies installed. These include:

- Linux server access
- Python 3
- Node.js and npm
- Docker and Docker Compose
- Tesseract OCR
- Poppler
- Apache configured for frontend serving and API proxying
- Correct environment files for backend, frontend, and bots

The server dependencies can be checked with:

```
python3 --version
node -v
npm -v
docker --version
docker compose version
tesseract --version
pdfinfo -v
pdftoppm -v
```

6.9.6 Remote Deployment Steps

On the remote server, the project is managed mainly through the root `Makefile`. The Makefile was added to avoid running many independent commands manually. It groups the main operations into commands for deployment, startup, shutdown, status checking, and log inspection.

The first step is to move to the project root on the server:

```
cd /var/www/html/unimentor/Unimentor-Project
```

The available Makefile commands can be checked with:

```
make help
```

Before deployment, the required environment files must exist:

```
backend/api/.env
frontend/.env.production
backend/bots/options.env
```

If they do not exist, they should be created from their example versions and edited with the correct server values:

```
cp backend/api/.env.example backend/api/.env
cp frontend/.env.production.example frontend/.env.production
cp backend/bots/options.env.example backend/bots/options.env

nano backend/api/.env
nano frontend/.env.production
nano backend/bots/options.env
```

After the environment is prepared, the deployment command can be executed:

```
make deploy
```

The deployment command prepares the platform for the server environment. It builds the bot images, checks that the required environment files exist, checks important dependencies, creates or uses the backend virtual environment, installs backend requirements, installs frontend dependencies, builds the frontend, and then restarts the platform. This command is useful after a new version of the code has been pulled from the repository.

After deployment, the platform can be started with:

```
make start
```

The start command performs several actions. It starts the bot orchestrator, starts the Docker services for Qdrant and Ollama, checks that the containers are running, checks or pulls the required Ollama models, starts the FastAPI backend on port `9000`, and registers the cron jobs for the backend watchdog and the ingestion worker. This means that `make start` starts the platform as a complete system, not only as a backend process.

The platform can be stopped with:

```
make stop
```

The stop command stops the bot orchestrator, removes the UniMentor cron jobs, stops the FastAPI backend, and stops the Qdrant and Ollama containers. The order is important. The cron jobs are removed before the backend is stopped, because otherwise the backend watchdog could detect that the backend is down and start it again, even though the platform was intentionally stopped.

For a clean restart, the recommended flow is:

```
make stop
make start
```

This is safer than restarting only one service, because UniMentor depends on several connected parts. A clean restart ensures that the backend, Docker services, bot orchestrator, and cron jobs are started again in the expected order.

6.9.7 Apache and Backend Runtime

In the remote setup, the frontend is built into static files and served by Apache. The backend runs locally on the server at:

```
127.0.0.1:9000
```

Public API requests are handled through Apache, which proxies them to the local backend service. This means that the FastAPI service does not need to be exposed directly to the public internet. The user accesses the platform through the UniMentor web path, while Apache serves the frontend and forwards API requests internally.

This setup also makes the frontend production configuration important. The frontend must call the public Apache-proxied API path, not the internal backend address directly. The backend, on the other hand, must allow the correct frontend origin through its CORS configuration. If either of these values is wrong, the frontend may load correctly, but API requests may fail.

6.9.8 Cron Jobs and Background Processing

The remote setup includes two important cron-based processes. The first one is the backend watchdog. This cron job calls the `start_backend_if_down.sh` script. Its purpose is to check whether the FastAPI backend is running. If the backend is already running on port `9000`, the script does nothing. If the backend is down, it starts it again using the Python virtual environment inside `backend/api/.venv`.

This mechanism is useful after a server restart or an unexpected backend failure. Without it, the backend could remain down until someone manually connects to the server and starts it again. With the watchdog, the backend has a simple recovery mechanism that keeps the API available when the platform is intended to be running.

The second cron-based process is the ingestion worker loop. This cron job calls the `run_ingestion_loop.sh` script. The ingestion worker is responsible for processing queued documents. When a document is uploaded from the Admin Panel or sent by a bot, it is not fully processed inside the upload request. Instead, it is placed in the queue. The ingestion worker later parses the file, extracts text, applies OCR if needed, chunks the content, creates embeddings, stores vectors in Qdrant, and updates the document status.

The loop script runs the ingestion worker multiple times per minute with short delays between executions. This makes document processing more responsive without making it part of the normal request-response flow. This is important because parsing, OCR, embedding, and indexing can be heavier operations. If they were executed directly during the upload request, the user interface could become slow or unstable.

The current cron jobs can be inspected manually with:

```
crontab -l
```

The stop command removes the UniMentor-related cron entries so that the backend watchdog and ingestion worker do not continue running while the platform is intentionally stopped.

6.9.9 Docker Services

UniMentor uses Docker for important supporting services. The main Docker services are Qdrant, Ollama, and the bot-related containers. Qdrant stores the vector embeddings that are used during retrieval. Ollama provides local AI-related model functionality, including the embedding model required by the RAG workflow. The bot orchestrator manages browser automation workers separately from the main backend API.

The Docker containers can be checked manually with:

```
docker ps
docker compose ps
```

The repository also provides Makefile commands for checking Docker status:

```
make docker-status
```

It is important that Qdrant and Ollama are running before the parts of the platform that depend on them are used. If Qdrant is down, retrieval from the vector database will fail. If Ollama is down or the required model is missing, local embedding or model-related operations may fail.

6.9.10 Logs and Maintenance Commands

The repository includes commands for checking the state of the platform and reading logs. This is necessary because UniMentor is made from several services, and a problem may appear in different places depending on which part failed. The most useful commands include:

```
make status
make docker-status
make logs-backend
make logs-ingestion-worker
make logs-orchestrator FOLLOW=true
```

The `make status` command shows the backend port and important Docker containers. The `make docker-status` command shows Docker Compose service status. The backend logs are useful when the API does not start or when an endpoint fails. The ingestion worker logs are useful when uploaded documents remain in processing or do not become indexed. The orchestrator logs are useful when bot executions fail or when worker containers do not behave as expected.

If a specific Makefile target for the backend watchdog logs is not available, the log can still be inspected directly with:

```
tail -f backend/api/backend_watchdog.log
```

Similarly, ingestion worker logs can also be inspected directly if needed:

```
tail -f backend/api/ingestion_worker.log
```

These commands make the platform easier to maintain on a server. Instead of guessing which part is failing, the maintainer can inspect the relevant logs and check the status of each service separately.

6.9.11 Common Issues and Troubleshooting

Some common issues are related to missing dependencies. If Tesseract is not installed or not available in the system path, OCR-related document processing will fail. If Poppler is missing, PDF-to-image conversion and some PDF parsing flows may fail. These issues are especially important when the uploaded document is a scanned PDF or when the parser needs to convert pages during extraction.

Other issues are related to wrong environment values. If the backend does not start, the backend log should be checked first. The backend `.env` file should also be checked, especially the database connection, backend port, Ollama URL, model names, and authentication-related values. It is also useful to check whether port `9000` is already in use.

If the frontend loads but cannot call the backend, the frontend API URL and the backend CORS settings should be checked. In local development, the frontend usually calls the local backend address. In production, it should call the Apache-proxied API URL. If the frontend points to the wrong URL, requests will fail even if both the frontend and backend are running.

If the ingestion worker is not processing documents, the cron job should be checked with `crontab -l`. The ingestion worker log should also be inspected. Possible causes include wrong file paths, missing environment values, parser errors, missing Tesseract or Poppler, unavailable Ollama embeddings, or Qdrant connection problems.

If Ollama does not have the required model, the model pull script should be executed. If Qdrant or Ollama is not running, Docker status should be checked with `docker ps`, `docker compose ps`, or `make docker-status`. These checks are important because the backend may be running, but RAG functionality can still fail if the supporting services are not available.

The repository therefore provides both the source code and the operational instructions needed to run the platform. It explains which files must be configured, which dependencies must be installed, how the local setup works, how the server setup works, which commands are used for deployment and runtime control, and how common issues can be checked. This makes the repository an important part of the final thesis implementation, because it allows UniMentor to be reproduced and maintained beyond the written description of the system.

Chapter 7

UniMentor Presentation

This chapter presents the final appearance of UniMentor through representative screenshots of the platform. In the previous chapter, the focus was on the design and implementation of the main system parts, such as the frontend, backend, databases, and AI-related components. Here, the focus moves to the actual interface as it is presented to the user.

The presentation begins from the Home page, which acts as the entry point of the platform, and then continues with the views available to the main user categories of UniMentor. More specifically, the following sections present the functionality available to the basic user, the secretary admin user, and the teacher admin user. In this way, the chapter shows not only the visual appearance of the platform, but also how the available functionality changes depending on the role of the user.

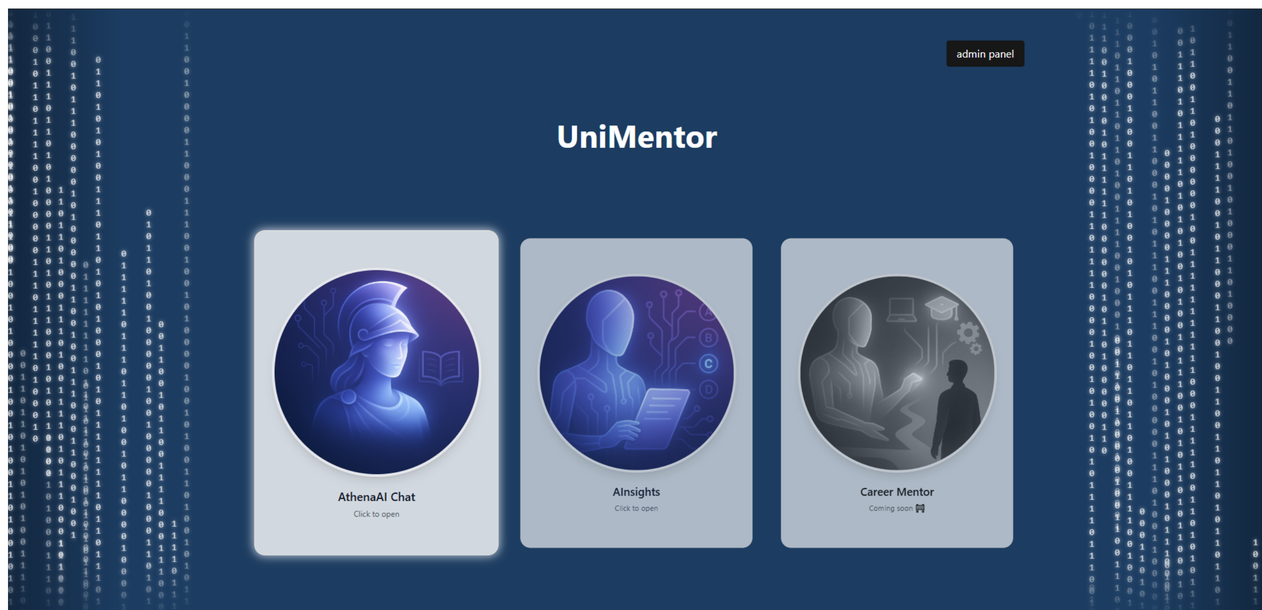

Figure 7.1: Home page of UniMentor.

The Home page is the main entry point of UniMentor and is designed to give the user direct access to the main features of the platform. As shown in Figure 7.1, the interface presents the available

tools through separate visual cards. These correspond to AthenaAI Chat, AInsights, and Career Mentor. This makes the initial navigation simple, since the user can immediately understand the core functionality of the platform and move directly to the desired feature.

This design was useful because UniMentor combines more than one type of functionality in the same environment. AthenaAI is focused on conversational interaction with the university knowledge base, AInsights is focused on generating structured course-related study material, and Career Mentor represents a job-related feature with AI-assisted filtering. Presenting these as separate cards on the Home page keeps the interface clear and avoids introducing unnecessary complexity at the start of the user experience.

The Home page also reflects the role-aware behavior of the platform. All authenticated users can access the main user-facing features, while administrative users are additionally given access to the Admin Panel through the button shown in the upper part of the page. This means that the Home page does not act only as a navigation screen, but also as a common starting point that adapts to the permissions of the currently logged-in user.

Another visible detail is that the Career Mentor tile is marked as coming soon. This is consistent with the current state of the feature, since its interface has been implemented as a proof of concept, but a complete production version would require connection to a secure third-party job listings API through the backend. In this way, the Home page also communicates which parts of the platform are fully available and which are still presented as partial or future functionality.

After the Home page, the following sections present the platform in more detail based on user role, starting from the basic user and then continuing with the secretary and teacher admin views.

7.1 Basic User

The basic user has access to the main user-facing features of UniMentor. These include AthenaAI Chat, AInsights, and Career Mentor. This role represents the normal platform user, such as a student, who uses the system mainly to ask questions, generate study material, and explore the proof-of-concept career feature. The basic user does not have access to the Admin Panel, since administrative actions are limited to secretary and teacher users.

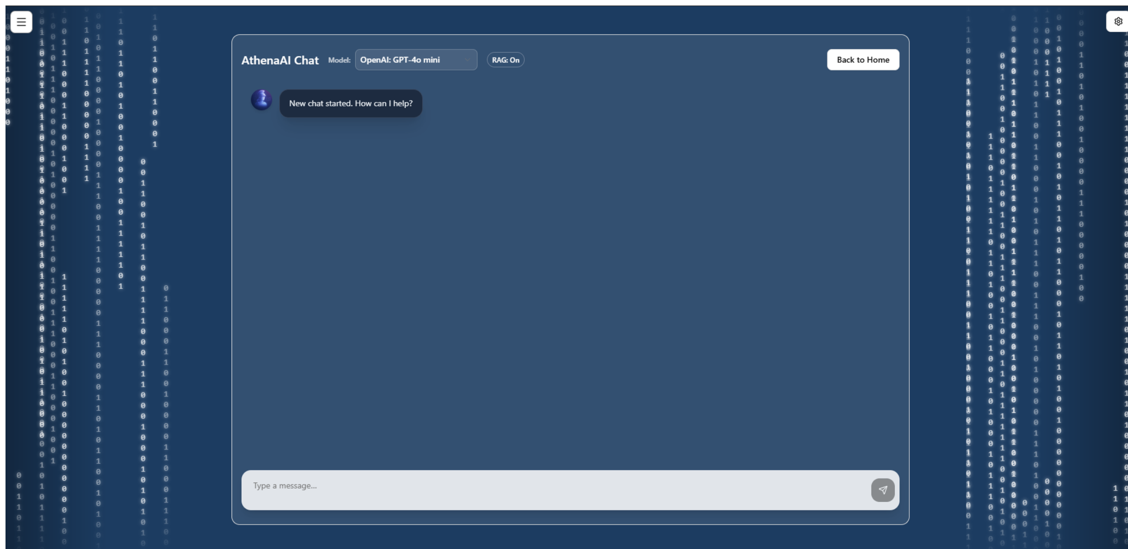

Figure 7.2: Initial AthenaAI Chat interface for a basic user.

AthenaAI Chat is the main conversational feature of UniMentor. As shown in Figure 7.2, the interface contains a model selector, a RAG status indicator, a message area, and an input field where the user can type a question. The user can also return to the Home page from the button placed at the top-right side of the chat container.

The model selector allows the user to choose one of the active LLMs that are available in the system. The RAG option shows whether the answer should be generated with retrieval from the university knowledge base or as a normal chat response. This gives the user more control over the way AthenaAI answers, while keeping the interface simple.

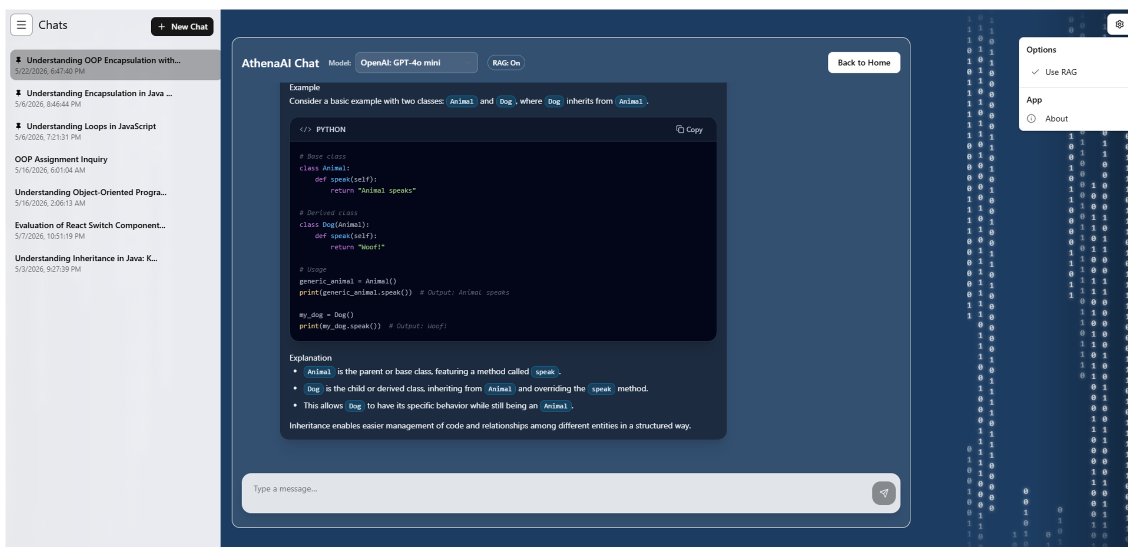

Figure 7.3: AthenaAI Chat with conversation history, options menu, and formatted answer.

Figure 7.3 shows AthenaAI after previous conversations have been created. The sidebar contains the user's chat history, including pinned conversations and older conversations. This allows the user to return to previous discussions instead of starting from zero every time. The options menu also gives access to settings such as enabling or disabling RAG.

The answer area supports formatted responses. In the example shown, AthenaAI returns an explanation that includes a code block, inline formatting, and structured text. This is important because the chatbot is not only used for simple text answers. It can also explain programming concepts, show examples, and present educational material in a readable way.

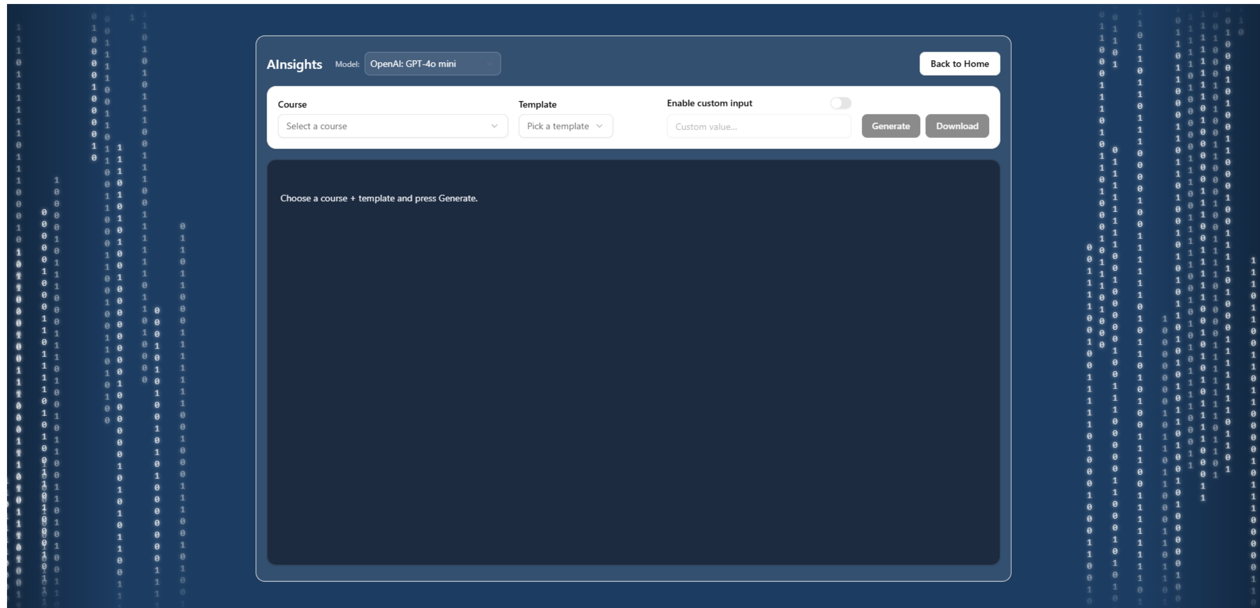

Figure 7.4: Initial AInsights interface for course-based material generation.

AInsights is the second main feature available to the basic user. Its purpose is to generate structured study material based on a selected course and template. As shown in Figure 7.4, the user can choose the model, course, and template type. There is also an optional custom input field that can be enabled when the user wants to provide extra instructions.

The page keeps the workflow simple. The user first selects a course, then selects the type of material they want, and finally presses the generate button. The result is displayed in the large response area below the form. The download button allows the generated result to be exported in PDF, which is useful when the user wants to keep the produced material for studying later.

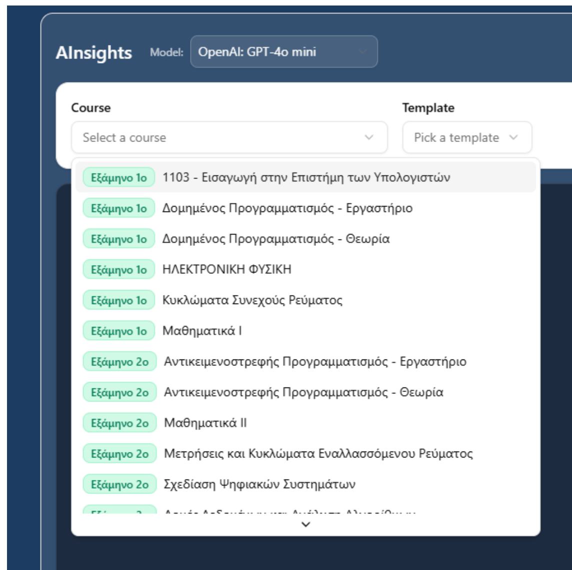

Figure 7.5: Course selection list in AInsights.

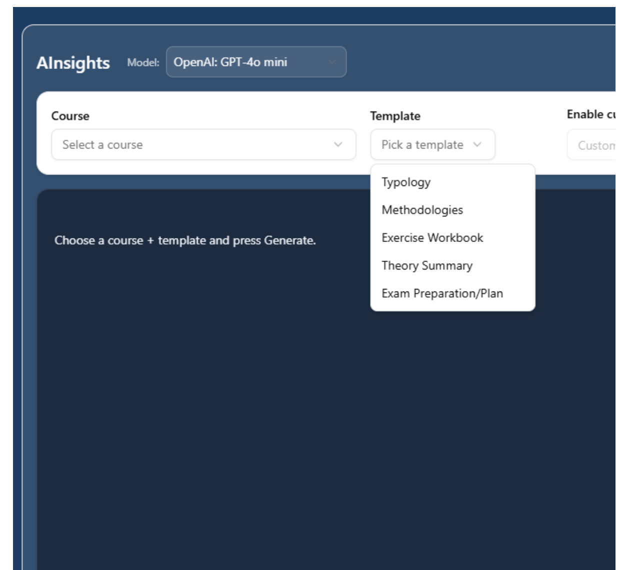

Figure 7.6: Template selection list in AInsights.

The course selector displays the available courses that the backend returns from the system database. As shown in Figure 7.5, the list includes course names and semester indicators. This helps the user find the correct course more easily, especially when the platform contains many subjects.

The template selector defines the type of output that AInsights should generate. As shown in Figure 7.6, the available templates include options such as typology, methodologies, exercise workbook, theory summary, and exam preparation plan. This makes the tool more guided than a normal chatbot, because the user does not need to manually describe the full structure of the requested material every time.

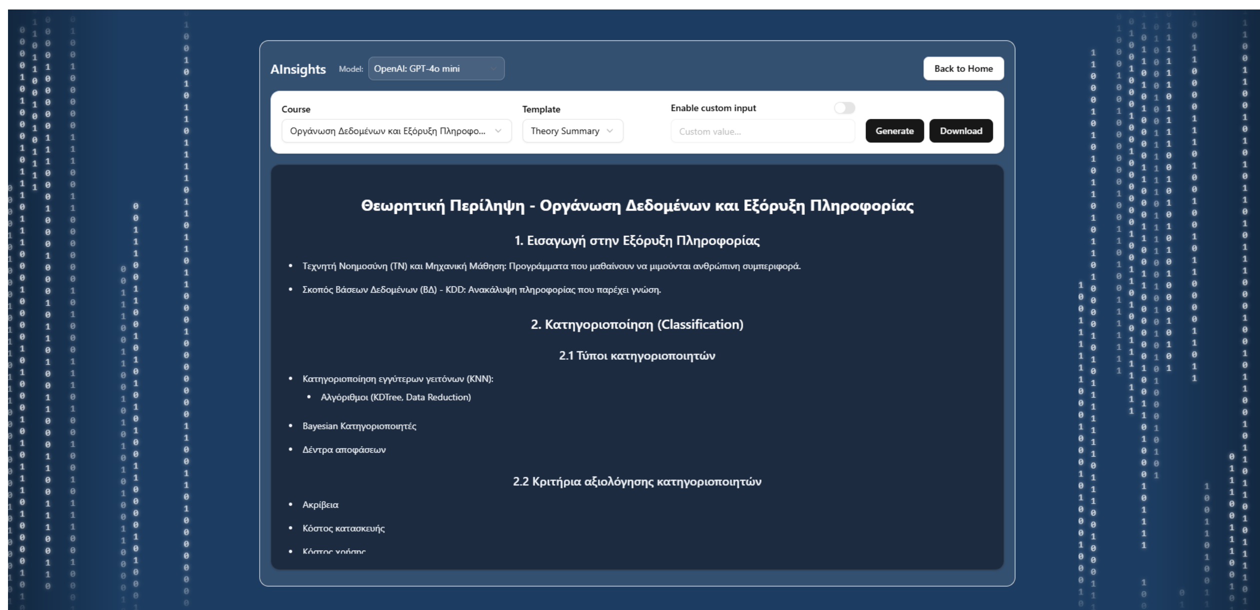

Figure 7.7: Generated AInsights response based on selected course and template.

Figure 7.7 shows an example of generated material. In this case, the user has selected a course and the theory summary template. The response is generated inside the same page and is presented in a structured format with headings and bullet points. This makes AInsights suitable for quick revision, summary creation, and exam preparation support.

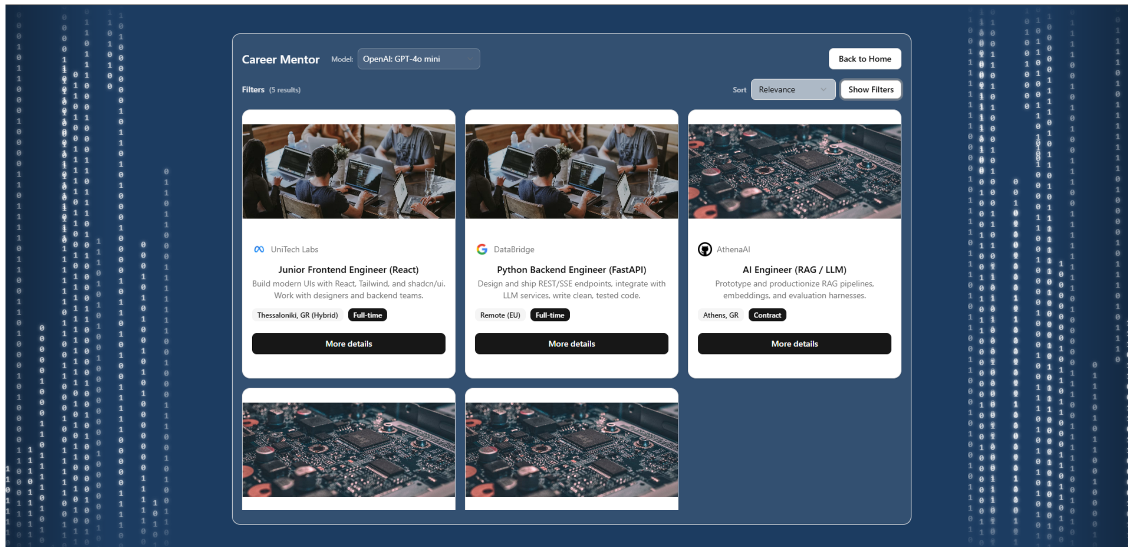

Figure 7.8: Career Mentor proof-of-concept job listing interface.

Career Mentor is also available from the basic user interface, but in the current version it is implemented as a proof of concept. As shown in Figure 7.8, the page presents job cards with information such as job title, company, location, job type, and a short description. The user can sort the results and open the filter panel to narrow down the displayed jobs.

The job data shown on this page is mock data stored locally in the frontend. This means that the page demonstrates the intended interface and interaction flow, but it is not yet connected to a real job provider. In a complete production scenario, the job data would need to be fetched through the backend from a secure third-party job listings API.

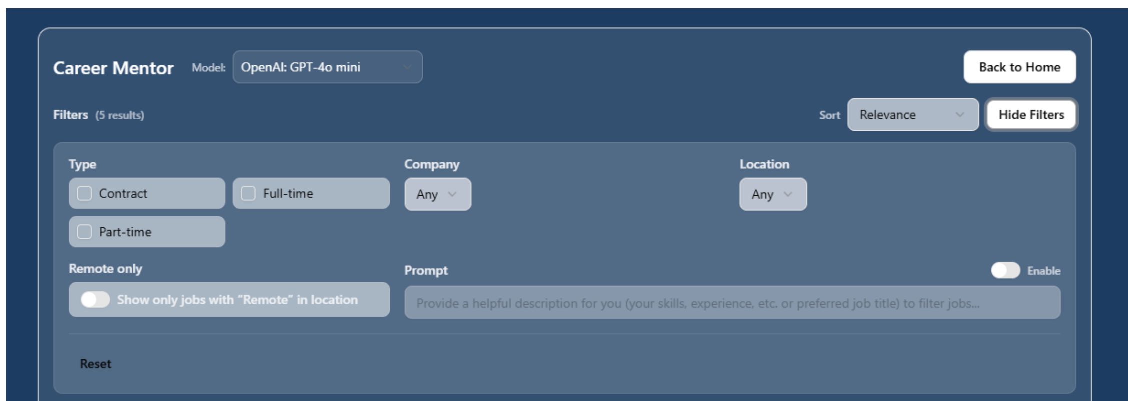

Figure 7.9: Career Mentor filter panel with normal and AI-assisted filtering options.

The filter panel allows the user to filter the job list by fields such as type, company, location, remote availability, and custom search criteria. It also includes a prompt field that can be enabled for AI-assisted filtering. With this option, the user can describe their skills or preferences, and the backend can return the most relevant job ids based on the available list.

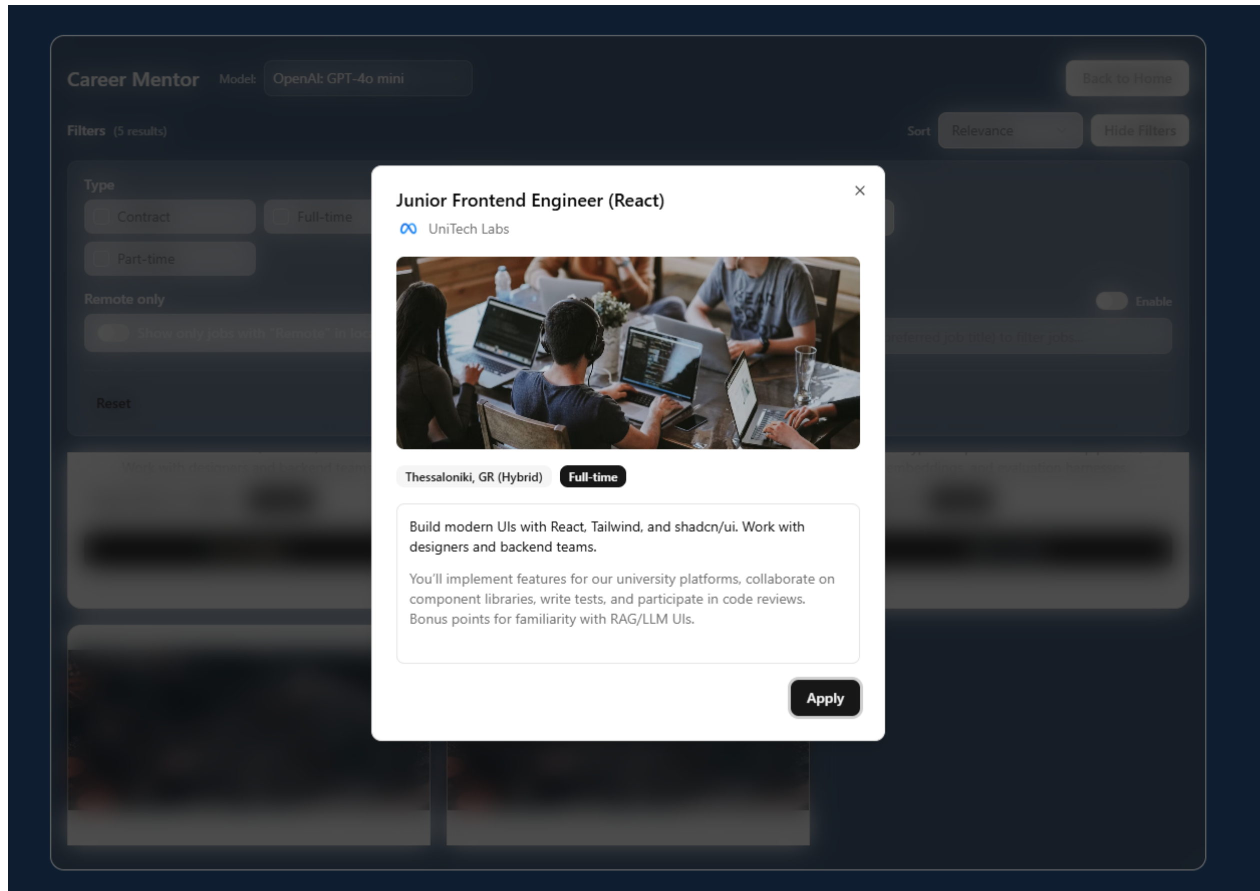

Figure 7.10: Career Mentor job details dialog.

When the user selects a job, a details dialog is displayed, as shown in Figure 7.10. The dialog presents the job title, company, image, location, employment type, description, and an apply button. This gives the feature a more complete user flow, even though the current implementation remains a proof of concept.

Overall, the basic user interface covers the main user-facing functionality of UniMentor. AthenaAI provides conversational support, AInsights generates structured study material from course data, and Career Mentor demonstrates how job-related AI filtering could be integrated in the future. These tools are separated into different pages, but they follow the same visual style and rely on the same backend for authentication, model selection, and AI-related requests.

7.2 Admin User - Secretary

The secretary admin user has access to all the main features that are available to a basic user, such as AthenaAI Chat, AInsights, and Career Mentor. In addition to these features, the secretary also has access to the Admin Panel. This role is mainly responsible for managing secretary-controlled knowledge base content, such as secretary-managed subjects and their documents.

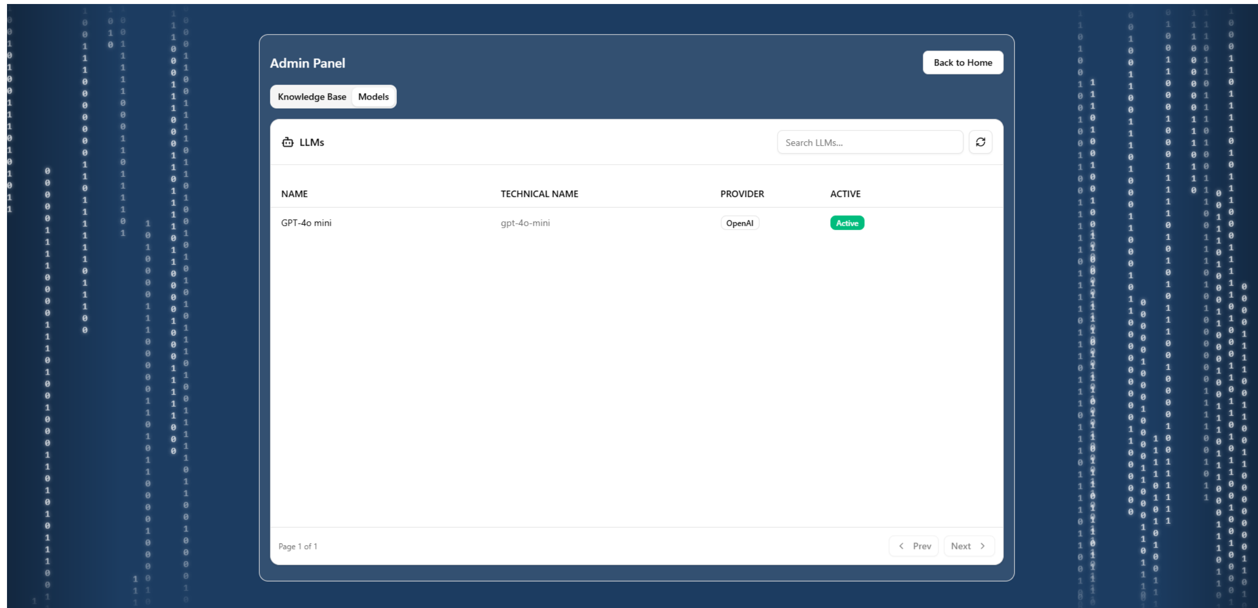

Figure 7.11: Secretary view of the Admin Panel models tab.

As shown in Figure 7.11, the secretary can access the Admin Panel and view the available LLMs through the Models tab. The table presents information such as the model name, technical name, provider, and active status. In this case, the secretary can view the model configuration, but model management actions are not available to this role. This is because model creation, update, and deletion are teacher-only operations in the current implementation.

The main administrative functionality for the secretary is found in the Knowledge Base tab. This part of the interface allows the secretary to manage secretary-controlled subjects and documents. These records are used by the backend as part of the knowledge base structure, while the actual uploaded files are later processed and indexed by the ingestion worker.

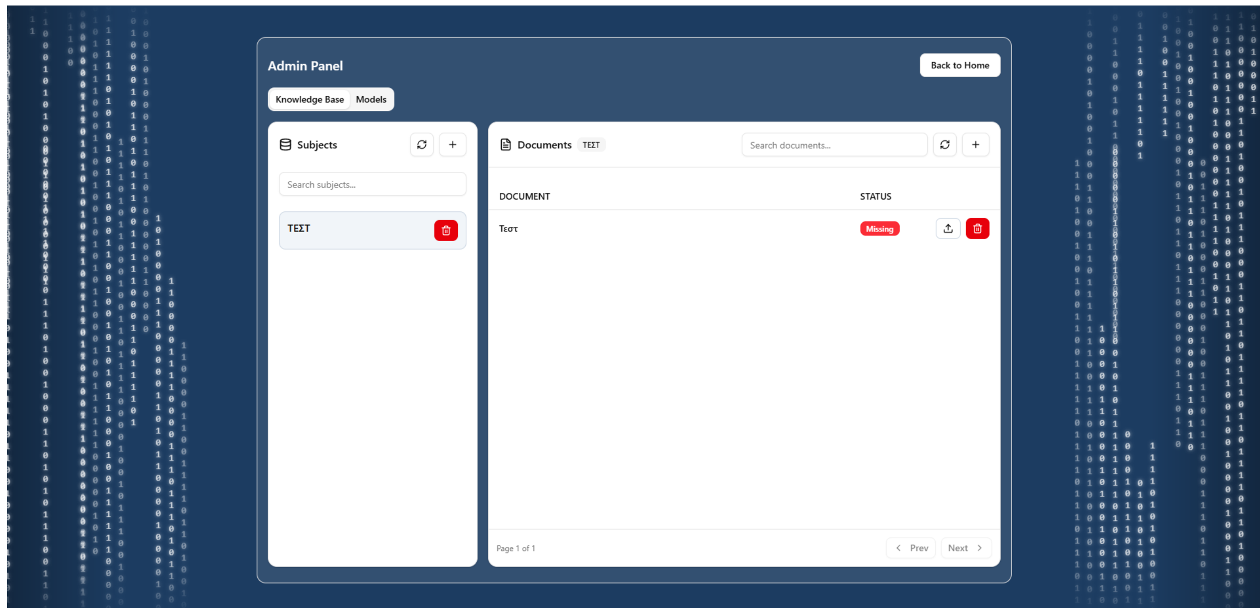

Figure 7.12: Secretary view of the Knowledge Base tab.

Figure 7.12 shows the Knowledge Base tab from the secretary view. The left panel contains the available subjects, while the right panel contains the documents that belong to the selected subject. This split layout makes it easier to first select a subject and then manage its related documents without leaving the same page.

In the example, the selected subject contains one document with the status `Missing`. This means that the document record exists in the system, but its actual file has not been uploaded and processed yet. The interface also includes buttons for refreshing the lists, creating new subjects or documents, uploading a file for a document, and deleting existing records.

Create subject
Create a new subject container for documents.
Subject name *
Enter subject name...
Cancel Create

Figure 7.13: Create subject dialog for secretary users.

Create document
Create a document entry before uploading its file.
Document name *
Enter document name...
Cancel Create

Figure 7.14: Create document dialog for secretary users.

The secretary can create new secretary-managed subjects through the dialog shown in Figure 7.13. The subject works as a container for related documents. This is useful for information that is not part of a normal course synchronization flow, but still needs to be available in the platform knowledge base.

After creating or selecting a subject, the secretary can also create document records, as shown in Figure 7.14. This step creates the document entry before the actual file is uploaded. Separating the document record from the file upload makes the process clearer, because the system can first know which document is expected and then track whether its file is missing, processing, or processed.

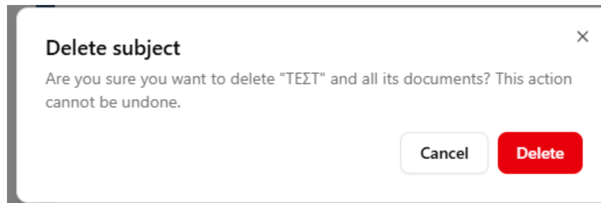

Figure 7.15: Delete subject confirmation dialog.

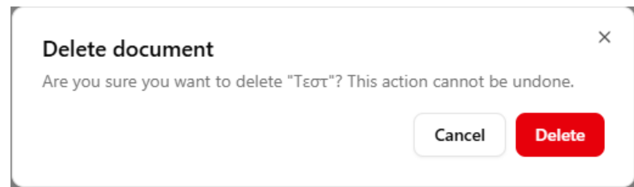

Figure 7.16: Delete document confirmation dialog.

Delete actions are protected with confirmation dialogs. Figure 7.15 shows the confirmation dialog for deleting a subject, while Figure 7.16 shows the confirmation dialog for deleting a document. This is important because these actions can also affect related data, such as document records and indexed vector chunks. The confirmation step helps avoid accidental deletion from the Admin Panel.

When a secretary deletes a subject, the backend also handles the related document data and attempts to remove the corresponding vectors from Qdrant. When a secretary deletes a document, the backend also removes the vector data that belongs to that document. In this way, the interface action is connected with the consistency logic described in the implementation chapter.

Figure 7.17: Upload document dialog for secretary users.

Figure 7.17 shows the upload dialog for a selected document. The subject and document name are already filled in, so the secretary only needs to choose the file that should be uploaded. After the upload, the backend does not immediately index the file inside the same request. Instead, it enqueues the document for parsing and indexing, and the ingestion worker handles the heavy processing later.

Overall, the secretary admin view focuses on managing secretary-controlled knowledge base content. The secretary can still use the platform as a normal user, but the Admin Panel adds the ability to create secretary-managed subjects, create document records, upload files, and delete content when needed. This role is therefore connected mostly with maintaining institutional information that should be available to UniMentor users through the RAG-based features.

7.3 Admin User - Teacher

The teacher admin user has access to all the main features that are available to a basic user, such as AthenaAI Chat, AInsights, and Career Mentor. In addition to these features, the teacher also has access to the Admin Panel. Compared to the secretary role, the teacher view focuses more on course-related knowledge base content, LLM management, and bot configuration.

The teacher can use the Knowledge Base tab to view the courses assigned to them and the documents that belong to each course. These course subjects are not manually created by the teacher through the interface, because they are mainly synchronized by the bot process from an external university system (in our case, Moodle). However, the teacher can still inspect the available course documents and upload files for documents that are missing or need to be processed.

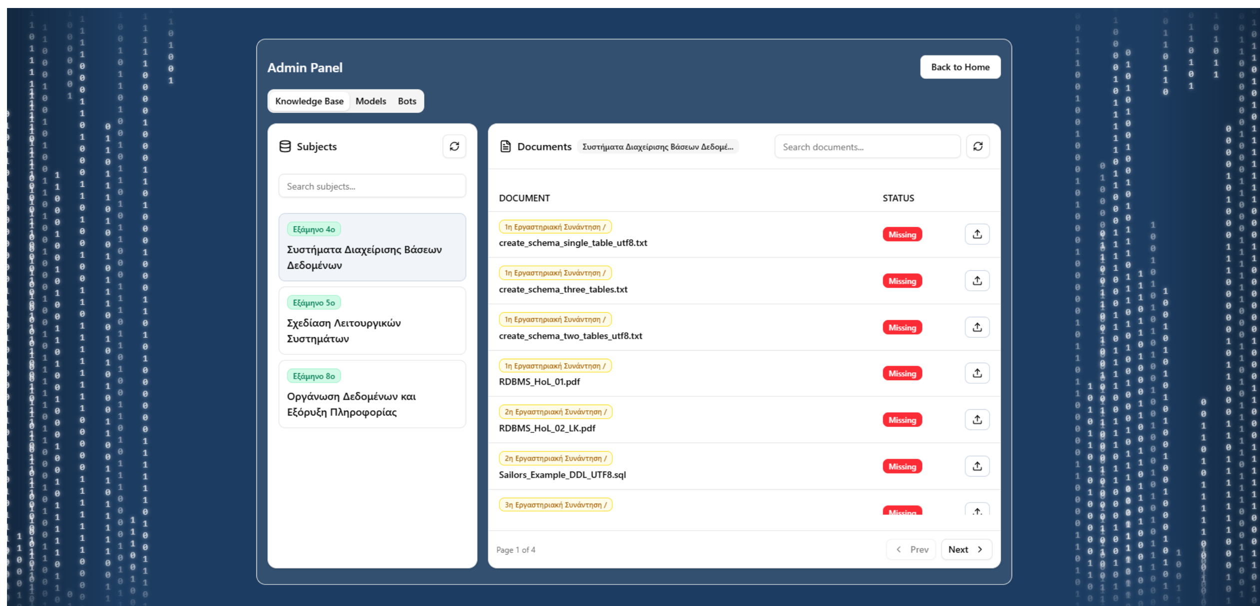

Figure 7.18: Teacher view of the Knowledge Base tab.

As shown in Figure 7.18, the Knowledge Base tab is split into two main panels. The left panel contains the teacher's assigned courses, while the right panel contains the documents of the selected course.

Each course is shown with its semester indicator, and each document has a status that shows whether it has already been processed or is still missing from the knowledge base.

The upload button allows the teacher to upload the actual file for an existing document record. This is useful because the course structure and document names may already exist in the system, but the file content still needs to be provided before it can be parsed, chunked, embedded, and indexed.

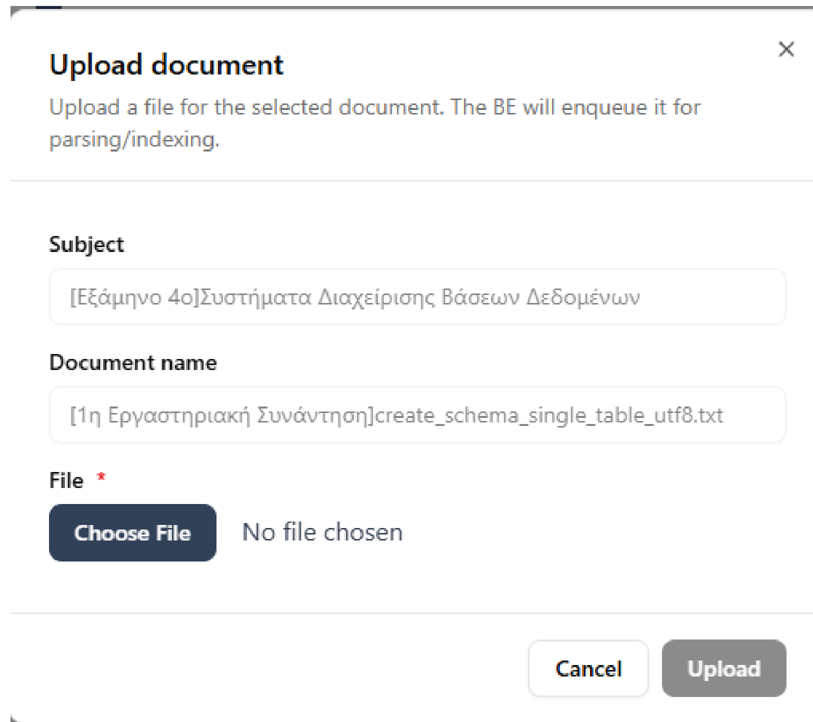

Figure 7.19: Upload document dialog in the teacher Knowledge Base view.

Figure 7.19 shows the upload dialog for a selected course document. The subject and document name are already filled in, so the teacher only needs to choose the file. After the upload, the backend places the document in the processing queue. This means that the file is not indexed directly from the frontend request, but is later handled by the ingestion worker.

The teacher role also has access to the Models tab. This part of the Admin Panel is used to manage the LLMs that can be selected from the user-facing tools. These models are used by AthenaAI Chat, AInsights, and Career Mentor, so managing them from the Admin Panel makes the AI behavior configurable without changing the code.

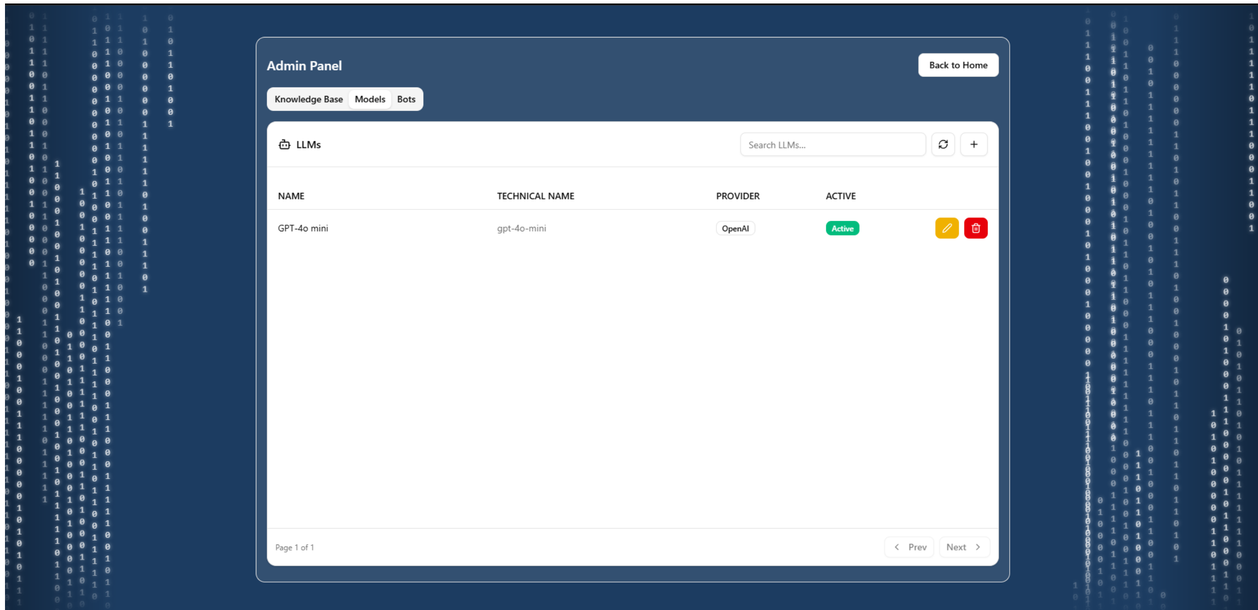

Figure 7.20: Teacher view of the Models tab.

As shown in Figure 7.20, the Models tab lists the registered LLMs with their display name, technical model name, provider, and active status. The teacher can search the list, refresh it, add a new model, edit an existing model, or delete one. This gives the teacher more control over which AI models are available on the platform.

Add LLM model

Register an LLM model that can be used in the app.

Name *

Enter display name for the model, e.g. GPT-4o Mini

Technical model name *

Enter technical model name used in prompts, e.g. gpt-4o-mini

Provider

OpenAI

Active

Cancel Create

Figure 7.21: Dialog for adding a new LLM.

Edit LLM model

Update the model properties or toggle its active status.

Name

GPT-4o mini

Technical model name

gpt-4o-mini

Provider

OpenAI

Active

Cancel Save changes

Figure 7.22: Dialog for editing an existing LLM.

The model creation and edit dialogs are shown in Figures 7.21 and 7.22. A model record contains a display name, a technical model name, the provider, and an active flag. The display name is what users see in the interface, while the technical model name is the value used by the backend when it calls the selected provider.

The active flag is also important because it controls whether the model is available in the user-facing tools. In this way, a model can remain registered in the system but be temporarily hidden from normal users. This is useful when testing a model or when a model should not be used anymore but the record should remain available for administration.

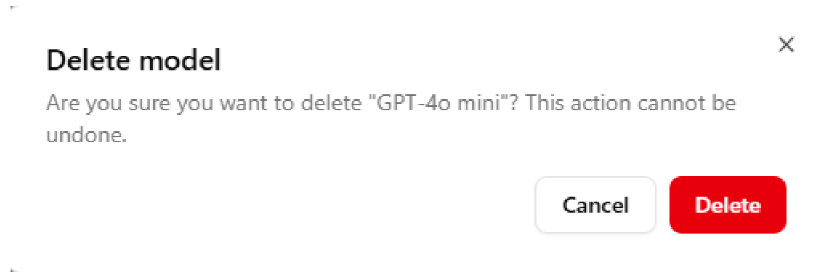

Figure 7.23: Delete confirmation dialog for an LLM.

Figure 7.23 shows the delete confirmation dialog for an LLM. The confirmation step is used to prevent accidental deletion, because removing a model affects the options available in the rest of the platform. This is especially important when the same model may be used by multiple tools.

The teacher role also has access to the Bots tab. This tab is not available to secretary users. It is used to configure automated bots that collect or update course-related information from an external university system (in our case, Moodle). The bot configuration does not execute the bot directly from the frontend. Instead, it stores the schedule and configuration that the backend and bot subsystem later use.

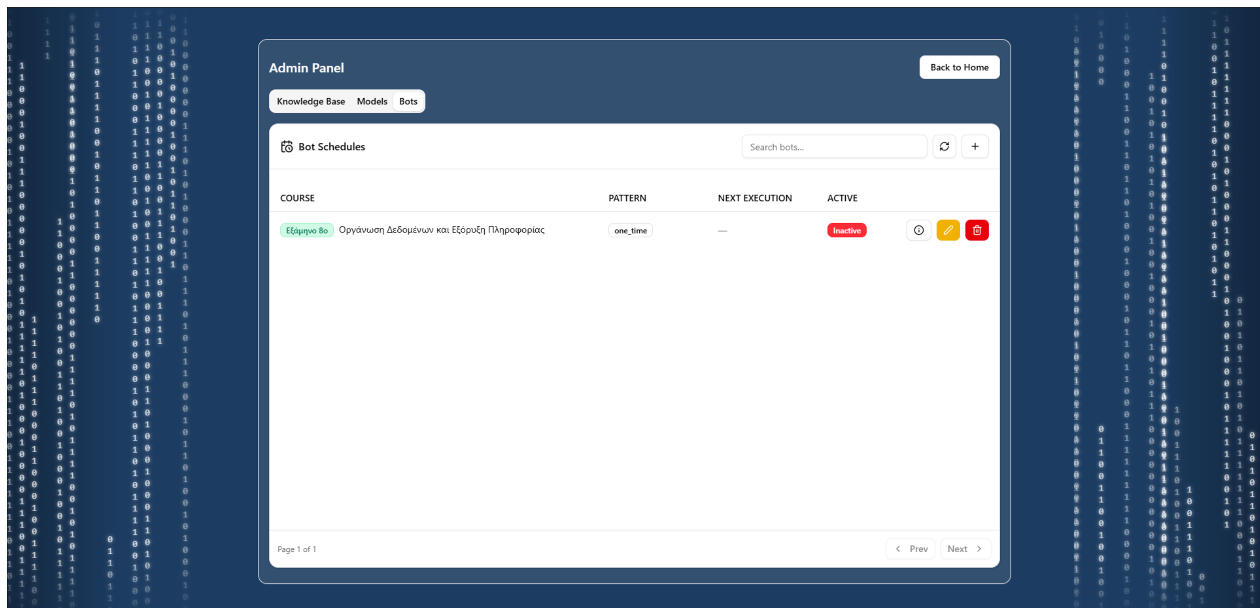

Figure 7.24: Teacher view of the Bots tab.

As shown in Figure 7.24, the Bots tab lists the configured bot schedules. Each row shows the course,

scheduling pattern, next execution information, and active status. The teacher can also view logs, edit a bot, or delete it. This makes the bot subsystem visible from the platform instead of leaving it only as a background server process.

Figure 7.25: Create bot dialog with one-time execution pattern.

Figure 7.26: Create bot dialog with repeat-interval pattern.

The create bot dialog supports different scheduling patterns. Figure 7.25 shows the one-time pattern, where the bot is configured to run once based on the selected scheduling values. Figure 7.26 shows the repeat-interval pattern, where the teacher can define an execution interval, interval type, start time, end time, and whether the schedule should repeat yearly.

The bot form also includes a parameters field in JSON format. This allows extra bot-specific configuration to be passed without changing the form structure every time a bot needs different parameters. The active flag controls whether the bot schedule is enabled or disabled.

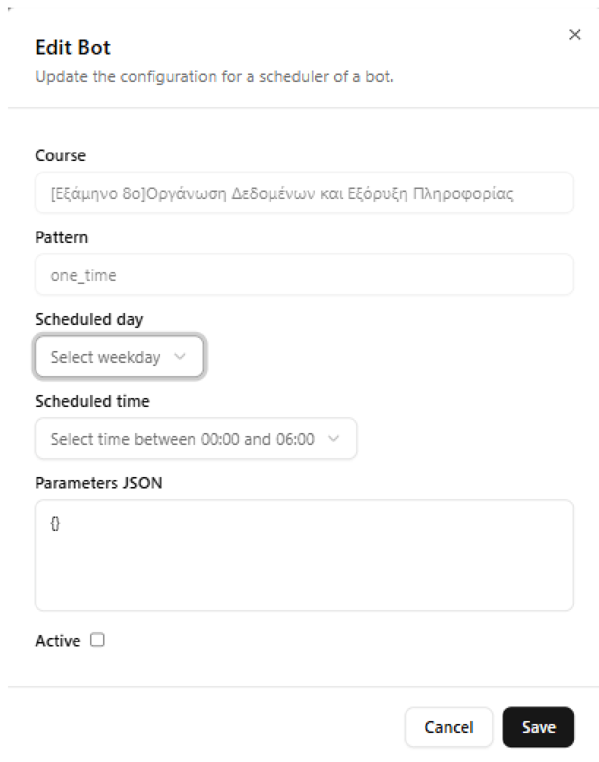

Figure 7.27: Edit bot dialog for an existing bot schedule.

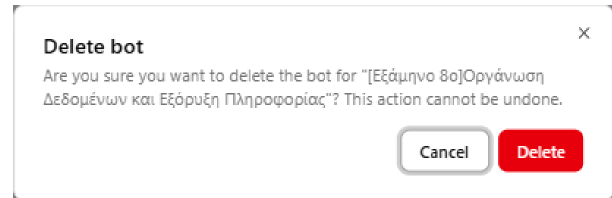

Figure 7.28: Delete confirmation dialog for a bot schedule.

Existing bot schedules can also be edited or deleted. The edit dialog allows the teacher to update the bot configuration, while the delete dialog confirms that the selected bot should be removed. These confirmation steps are useful because bot schedules may affect automatic data collection and document synchronization for course content.

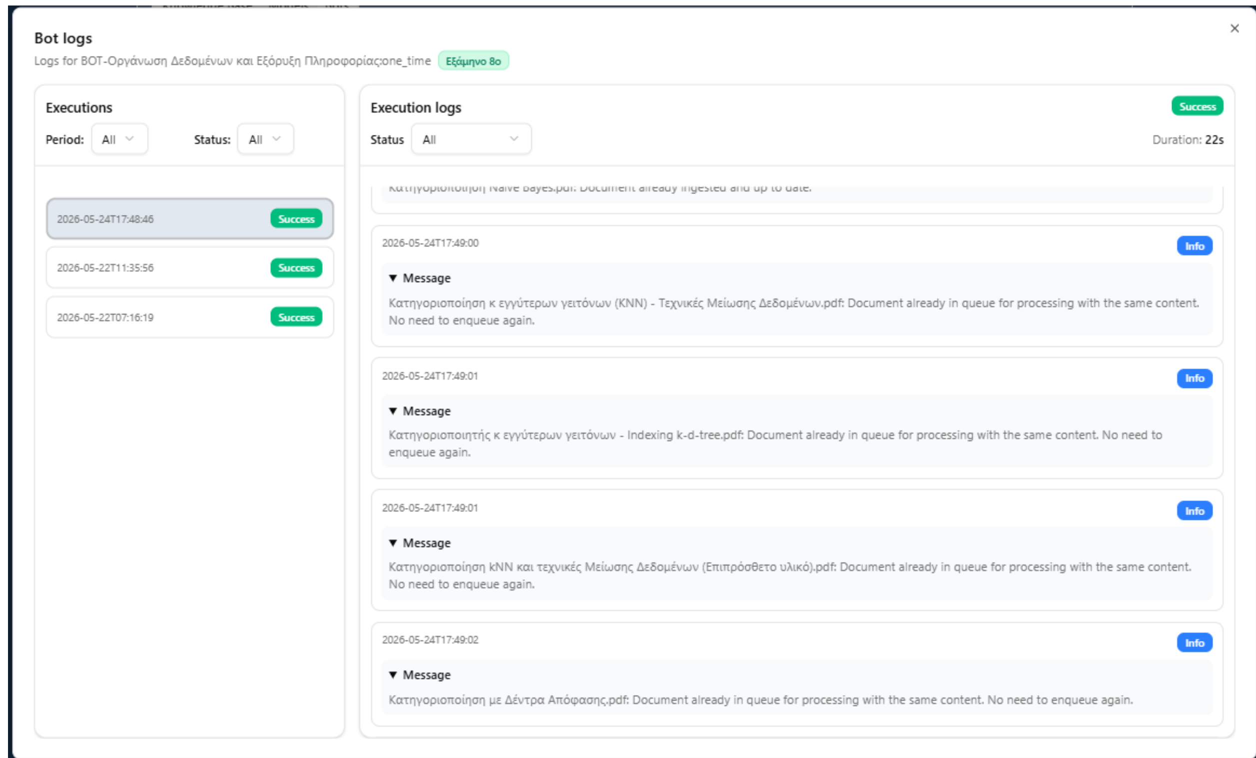

Figure 7.29: Bot logs dialog showing execution history and messages.

Figure 7.29 shows the bot logs dialog. The left side contains previous executions, while the right side contains the log messages of the selected execution. The interface also includes filters for execution period and status. This allows the teacher to inspect whether a bot ran successfully and to see messages related to document updates, skipped documents, or queueing decisions.

Overall, the teacher admin view gives teachers control over the parts of UniMentor that are connected to their courses and to the AI configuration of the platform. Teachers can use all normal user features, upload course documents, manage available LLMs, configure scheduled bots, and inspect bot logs. This makes the teacher role more technical than the basic user role, because it is responsible not only for using the platform, but also for helping maintain the course-related knowledge base and the automation layer.

Chapter 8

User Experience

8.1 Evaluation Methodology

After the implementation of UniMentor, a user experience evaluation was conducted in order to examine how users perceived the platform in terms of usability, ease of use, consistency, and overall interaction. UniMentor is not only a technical system but also a platform that must be used by students, teachers, and administrative users in a simple, understandable way. For that reason, the system's success does not depend only on the correctness of the backend, the retrieval pipeline, or the AI responses. It also depends on whether users can interact with the platform without confusion or unnecessary effort.

The evaluation focused on the general usability of UniMentor. The users interacted with the platform and then completed a questionnaire through Google Forms. The questionnaire was based on the System Usability Scale (SUS), which is a commonly used method for measuring the perceived usability of a system. SUS was selected because it is short, simple to distribute, and suitable for evaluating software systems without requiring a long or complex user study. This made it appropriate for UniMentor, as the purpose of the evaluation was to collect structured feedback from users after they had tested the platform's core features.

SUS was originally introduced by Brooke as a quick method for collecting a global usability score for interactive systems [46]. It consists of ten statements, answered on a five-point Likert scale. The questions alternate between positive and negative statements. This prevents the questionnaire from being only one-directional and encourages the participant to think about each statement separately. Later studies have also shown that SUS is widely used in practice and can provide reliable results for comparing and interpreting the usability of different systems [47]. For these reasons, it was considered a suitable method for evaluating UniMentor.

The questionnaire was answered by 16 participants. The answers were collected anonymously through Google Forms. Each participant rated the ten SUS statements from 1 to 5, where 1 represented strong disagreement and 5 represented strong agreement. The statements were adapted only in wording so that the term ”system” would refer specifically to UniMentor. The structure and scoring logic of SUS remained the same.

8.2 System Usability Scale

The System Usability Scale produces a final score from 0 to 100. However, this score should not be interpreted as a percentage of correctness. Instead, it is a usability score that represents the user's subjective perception of the system. A higher score indicates that users found the system easier, more consistent, and more comfortable to use.

The SUS score is calculated using a specific scoring method. For the positive statements, which are questions 1, 3, 5, 7, and 9, the contribution of each answer is calculated by subtracting 1 from the selected value. For the negative statements, which are questions 2, 4, 6, 8, and 10, the contribution is calculated by subtracting the selected value from 5. The total of these ten adjusted values is then multiplied by 2.5. This produces the final SUS score for each participant.

The formula used for the calculation was:

$$SUS = \left[\sum_{i \in \{1,3,5,7,9\}} (Q_i - 1) + \sum_{i \in \{2,4,6,8,10\}} (5 - Q_i) \right] \times 2.5$$

This scoring approach was useful because it converted the answers into a single value that could be compared across participants. At the same time, the individual answers were still examined in order to understand which parts of the experience were evaluated more positively or negatively.

SUS was chosen for this evaluation mainly for three reasons. First, it is simple and quick for users to complete. Since the evaluation was conducted after users interacted with the prototype, a longer questionnaire could reduce the number or quality of responses. Second, SUS gives a numerical score that can be easily interpreted and compared with existing usability benchmarks. Third, it has been used and examined in different types of software systems, which makes it more suitable than creating a completely custom questionnaire without a known scoring method [47], [48].

Although SUS is useful, it also has limitations. It gives a general view of perceived usability, but it does not explain every specific reason behind a user's opinion. For example, a user may give a lower score because of navigation, response time, visual layout, or because they were not familiar with AI-based systems. For this reason, the results were not treated as a complete usability study. Instead, they were used as an indicative evaluation of the platform's usability at this stage of development.

8.3 Questionnaire Structure

The questionnaire included the standard ten SUS statements, adapted to refer directly to UniMentor. The questions were the following:

1. I think that I would like to use UniMentor frequently.
2. I found UniMentor unnecessarily complex.
3. I thought UniMentor was easy to use.
4. I think that I would need the support of a technical person to be able to use UniMentor.

5. I found the various functions in UniMentor were well integrated.
6. I thought there was too much inconsistency in UniMentor.
7. I would imagine that most people would learn to use UniMentor very quickly.
8. I found UniMentor very cumbersome to use.
9. I felt very confident using UniMentor.
10. I needed to learn a lot of things before I could get going with UniMentor.

The positive statements assessed whether users wanted to use the platform again, found it easy, felt the functions were connected, could learn the system quickly, and felt confident while using it. The negative statements measured the opposite side of the experience, such as unnecessary complexity, need for technical support, inconsistency, difficulty of use, and the amount of learning needed before using the platform.

This structure was suitable for UniMentor, a platform that includes multiple connected parts. A user can interact with AthenaAI, use the AInsights tools, navigate through different pages, and receive AI-generated information based on university-related content. Because of this, the evaluation had to measure not only whether one page was understandable, but whether the platform as a whole felt usable and consistent.

The Google Forms graphs were used to visualize the distribution of responses for each question. These graphs help show whether the answers were concentrated around positive values or whether there were mixed opinions for specific statements. The final SUS score gives one overall value, but the graphs make it easier to understand the behavior of each question separately.

8.4 Question-Level Results

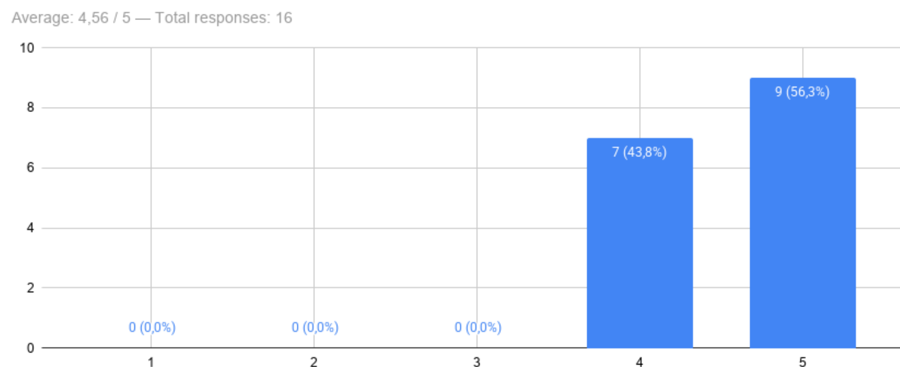

Figure 8.1: Responses to SUS Question 1: I think that I would like to use UniMentor frequently.

The first question examined whether users would like to use UniMentor frequently. As shown in Figure 8.1, most users gave positive answers to this statement. The average score for this question was

4.56 out of 5. This indicates that most participants saw UniMentor as a platform that they would be willing to use again. This is a positive result since the platform is designed to support students during repeated academic information searches and not only during a single interaction.

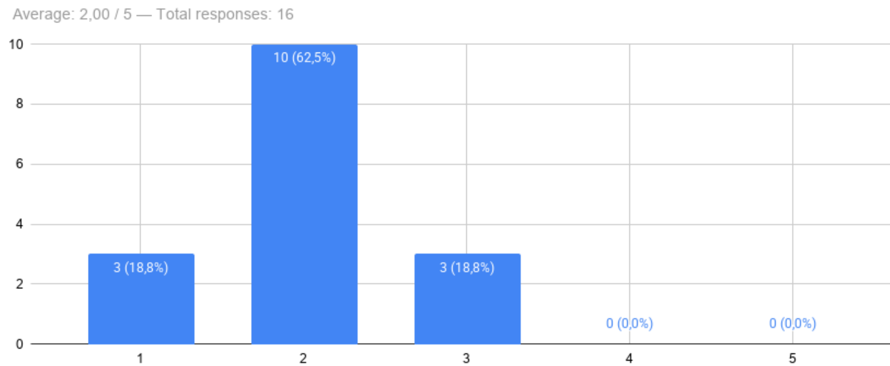

Figure 8.2: Responses to SUS Question 2: I found UniMentor unnecessarily complex.

Figure 8.2 shows that most participants disagreed with the statement that UniMentor was unnecessarily complex. The average score was 2.00. This suggests that the platform was generally perceived as straightforward, despite UniMentor's multiple features, such as AthenaAI, AInsights, and different navigation pages.

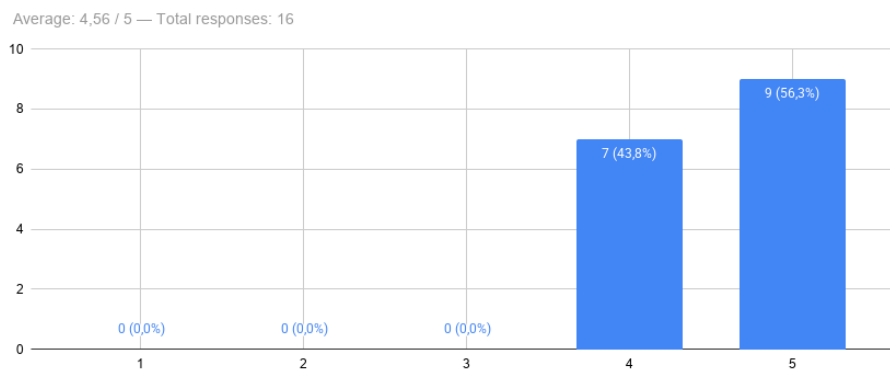

Figure 8.3: Responses to SUS Question 3: I thought UniMentor was easy to use.

As shown in Figure 8.3, the responses were strongly positive for UniMentor's ease of use. The average score was 4.56. This result supports the idea that the main interaction flow of UniMentor was understandable. Users could access the main features without needing to understand the technical details behind the system, such as retrieval, embeddings, vector search, or the AI model connection.

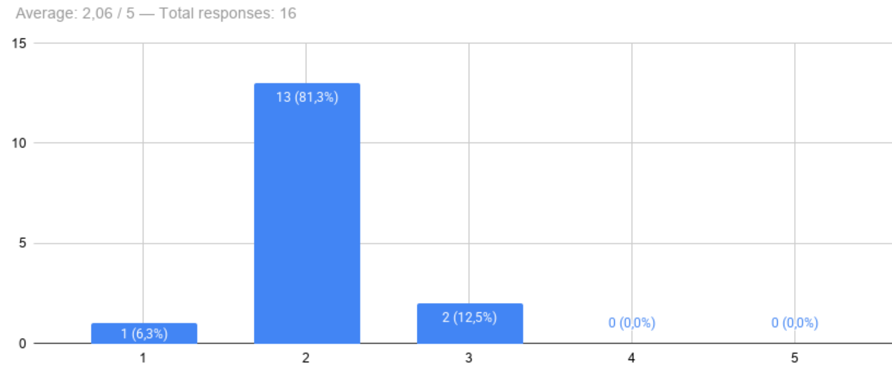

Figure 8.4: Responses to SUS Question 4: I think that I would need the support of a technical person to be able to use UniMentor.

Figure 8.4 shows that most participants did not feel that technical support would be necessary. The average score was 2.06. This is a useful result for UniMentor because the platform is intended for regular university users, not only technical users. Students should be able to ask questions, use AI tools, and navigate the platform without needing help from a developer or administrator.

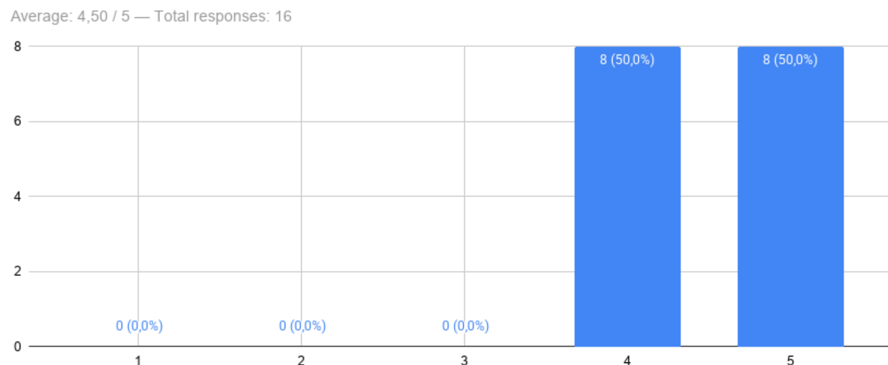

Figure 8.5: Responses to SUS Question 5: I found the various functions in UniMentor were well integrated.

Figure 8.5 shows that the participants generally agreed that UniMentor functions were well integrated. The average score was 4.50. This is important because UniMentor is not just a single-page chatbot. It combines conversational AI, RAG-based educational tools, and a wider platform structure. The result suggests that users perceived these parts as belonging to the same system rather than as disconnected tools.

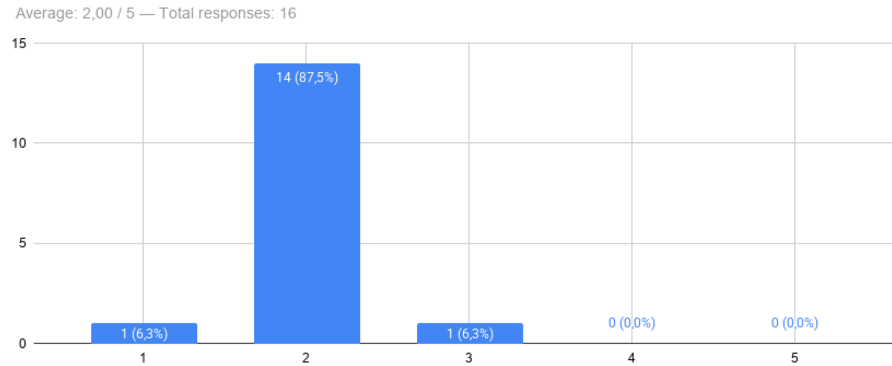

Figure 8.6: Responses to SUS Question 6: I thought there was too much inconsistency in UniMentor.

The sixth question examined whether users noticed too much inconsistency in the platform. As shown in Figure 8.6, the average score was 2.00. This result supports the design decision to keep a common visual style, similar page structure, and consistent interaction patterns across the main parts of the system.

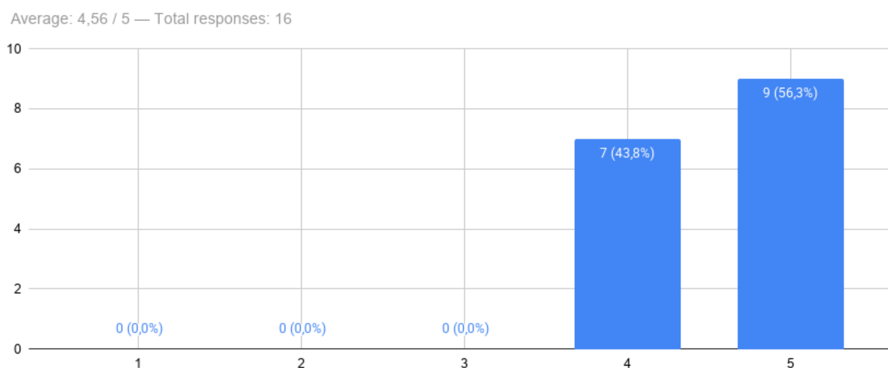

Figure 8.7: Responses to SUS Question 7: I would imagine that most people would learn to use UniMentor very quickly.

Figure 8.7 shows a positive response distribution for the seventh question, which examined whether users believed that most people would learn to use UniMentor quickly, with an average score of 4.56. This result indicates that UniMentor was considered easy to learn and is especially important for a university-oriented platform, since users may not use it every day at first. A platform with good learnability allows users to return after some time and still understand how to use its basic features.

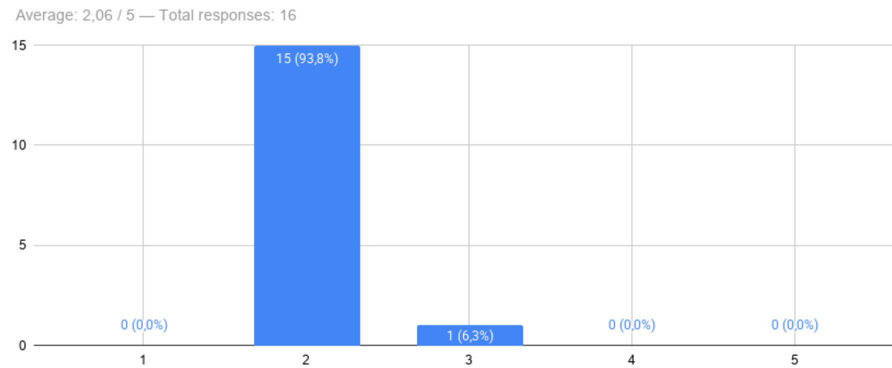

Figure 8.8: Responses to SUS Question 8: I found UniMentor very cumbersome to use.

The eighth question examined whether users found UniMentor cumbersome to use. As shown in Figure 8.8, the average score was 2.06. This shows that users generally did not find the system tiring or difficult to operate. This result is connected with the previous questions about ease of use and technical support. Together, these answers suggest that the basic user experience was simple enough for the participants.

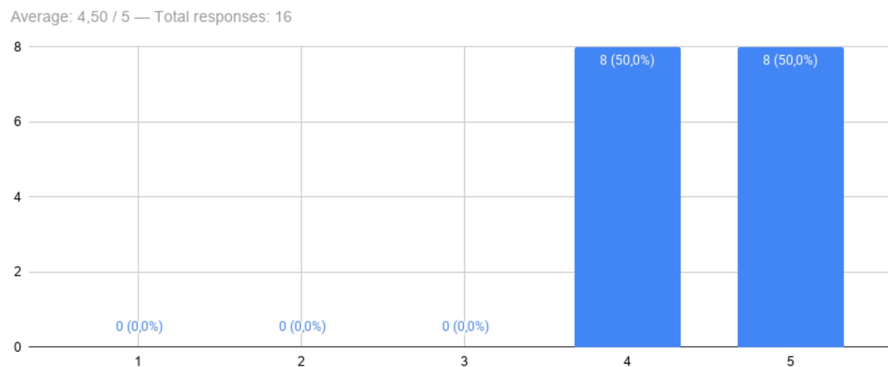

Figure 8.9: Responses to SUS Question 9: I felt very confident using UniMentor.

Figure 8.9 shows that users generally felt confident while using the platform. The average score was 4.50. This is a useful result because confidence affects whether users are willing to rely on a system, especially when it is based on AI interaction. If the interface feels unclear, users may hesitate to ask questions or use the generated results. In this case, the responses suggest that the platform gave users enough confidence during interaction.

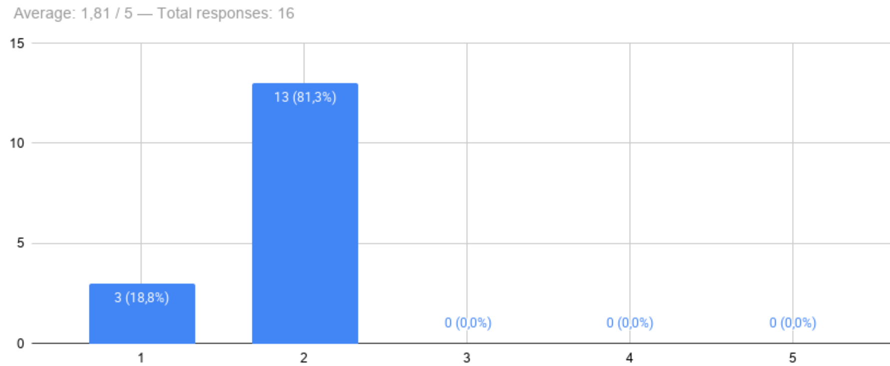

Figure 8.10: Responses to SUS Question 10: I needed to learn a lot of things before I could get going with UniMentor.

As shown in Figure 8.10, the average score was 1.81. This was the lowest average among the negative statements, indicating that users did not feel that UniMentor required a lengthy learning process before they could start using it. This result is integral, as the platform is designed to provide quick access to academic information without introducing another complicated system that users must study before using.

8.5 Overall SUS Results

The final SUS results showed an average score of 81.88, a median of 82.50, and a standard deviation of 8.04, with the lowest score being 67.50 and the highest 95.00. The median was close to the average, which shows that the result was not strongly affected by extreme values. The standard deviation, calculated using the sample standard deviation because the respondents represent a sample of potential UniMentor users rather than the full user population, also indicates some variation among participants, but the scores were still generally close enough to support a consistent positive usability evaluation.

Metric	**Value**
Number of participants	16
Average SUS score	81.88
Median SUS score	82.50
Minimum SUS score	67.50
Maximum SUS score	95.00
Standard deviation	8.04

Table 8.1: Summary of SUS results for UniMentor

The average score of 81.88 can be considered a strong result for the current version of the platform. Previous work on SUS interpretation suggests that scores above the average benchmark indicate good perceived usability, while higher scores can be associated with more positive usability ratings [49]. In

this case, the result shows that the participants generally found UniMentor easy to use and suitable for its intended purpose.

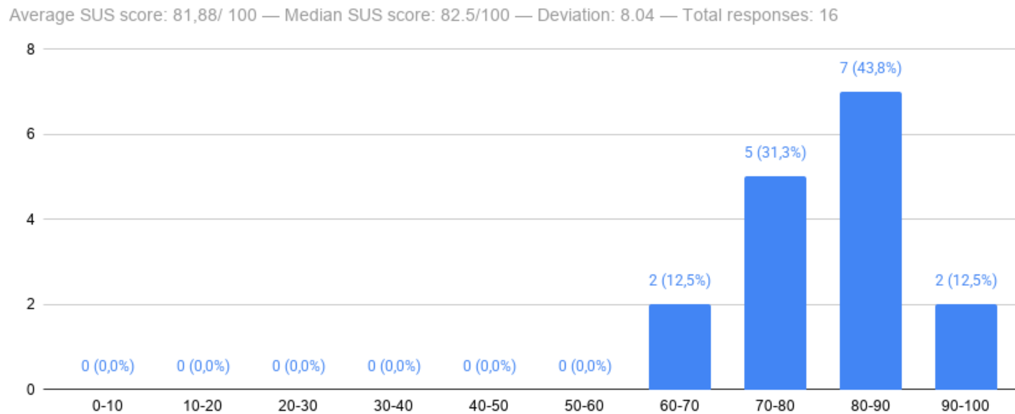

Figure 8.11: Distribution of final SUS scores for UniMentor participants.

Figure 8.11 shows the distribution of the final SUS scores. Most participants gave scores in the range between 80 and 90. This indicates that the majority of users had a clearly positive experience. A smaller number of responses were lower, with the minimum score being 67.50. This is still close to the commonly used average SUS reference point, but it also shows that not every user experienced the platform in the same way. This is expected because users may have different levels of familiarity with AI chatbots, educational platforms, or web-based tools.

Overall, the SUS evaluation shows that UniMentor was received positively by the participants. The average SUS score of 81.88 indicates that the platform has good usability at this stage. The results also show that the platform was easy to learn, not considered overly complex, and generally comfortable to use. This supports the main goal of UniMentor, which is to make university information and educational support easier to access through a simple AI-based platform.

Chapter 9

Future Extensions and Conclusions

9.1 Future Extensions

UniMentor was developed as a working platform, but there are still several directions that could improve it further. Some of these extensions are related to the knowledge gathering process, while others are connected to additional user-facing tools and wider AI provider support. These improvements would make the platform more complete and more useful in a real university environment.

One possible future extension is the creation of more bot templates. In the current version, the bot subsystem is already designed in a configurable way, with templates that can collect information from selected university sources. However, this idea could be expanded further. More templates could be added so that the platform can collect information from other external academic sources, not only from the sources that are already supported. For example, some teachers may maintain personal websites, laboratory pages, GitHub pages, or other course-related pages outside the main university systems. If these pages contain useful information for students, then a bot template could be created to collect this content and send it to the backend for processing and indexing.

This would make the knowledge base richer and more flexible. Instead of depending only on a small number of predefined sources, UniMentor could gradually support many different types of academic content. Of course, this should still be controlled. The goal is not to index random external information, but to allow authorized users to configure trusted sources that are useful for a course or department. In this way, the bot subsystem could become a more general mechanism for keeping the university knowledge base updated.

Another future extension is the completion of Career Mentor. In the current implementation, Career Mentor works as a proof of concept. It presents job cards, normal filters, and AI-assisted filtering, but the job data is stored locally as mock data. A complete version would need an additional communication layer with a real third-party job API. This layer should not be implemented directly in the frontend, because API keys, access tokens, request limits, and provider-specific logic should remain protected on the backend side.

In a complete implementation, the backend would act as middleware between UniMentor and the external job provider. It would call the third-party API, receive job listings, normalize the data into a format suitable for the frontend, and then return it to the user interface. This would also make it easier to change job providers in the future, because the frontend would not need to know the details

of each external API. After this integration, the AI filtering feature could work on real job data and become more useful for students who want to search for internships, junior positions, or jobs related to their studies.

A third possible extension is the addition of a new tile or application called Exams Simulator. This feature could work in a similar way to AInsights, because it would also use course selection, template-like generation, and AI support. However, instead of generating summaries or study plans, it would generate exam-style questions. These questions could be created in a specific JSON format, so that the frontend can render them as interactive quiz elements.

The Exams Simulator could support different question types. For example, it could generate multiple-choice questions, true-or-false questions, and short-answer questions. Multiple-choice and true-or-false questions could be evaluated automatically based on the generated correct answer. Short-answer questions could be evaluated by an AI model, which would compare the student's response with the expected answer and provide feedback. This would make the tool more interactive than a simple generated document, because the student would be able to practice and receive some form of evaluation.

This extension would be especially useful for exam preparation. A student could select a course, choose the type and difficulty of questions, and generate a small exam simulation based on the available course material. Since the questions would be generated using the course knowledge base, the tool could remain connected to the material that has actually been inserted into UniMentor. This would also make the feature more aligned with the rest of the platform, because it would reuse the existing RAG and AI generation logic instead of being a completely separate system.

Another future improvement is the addition of support for more AI providers. In the current version, the system supports local and external model usage through the provider layer, mainly with Ollama and OpenAI. This design already makes the platform more flexible than a system that depends on only one model or one provider. However, the same idea could be expanded further by adding support for more providers, such as other commercial APIs or other local inference solutions.

This would give administrators more options when choosing which models should be available in the platform. Some providers may be better for speed, others may be better for cost, and others may perform better in specific tasks such as summarization, reasoning, or multilingual answers. Supporting more providers would also reduce dependency on a single service. This is important because AI services change over time, and a university platform should be able to adapt without requiring major changes in the rest of the codebase.

Overall, these future extensions follow the same direction as the current system. They do not change the main idea of UniMentor, but they expand it. More bot templates would improve knowledge gathering. A complete Career Mentor integration would make the job-related feature more practical. An Exam Simulator would add a new study-support tool. Additional AI providers would make the platform more flexible and easier to adapt in the future.

9.2 Conclusions

This thesis presented the design and implementation of UniMentor, a university-oriented platform based on Artificial Intelligence and Retrieval-Augmented Generation. The main goal of the work

was to improve access to academic and administrative information by connecting a conversational AI assistant with controlled university knowledge sources. Instead of depending only on the general knowledge of a language model, UniMentor retrieves relevant content from the available knowledge base and uses it as context before generating an answer.

The motivation behind the platform came from a practical problem. University information may exist online, but it is often spread across different systems, course pages, announcements, documents, and administrative sources. This makes it harder for students to quickly find the information they need, especially when they do not know where to search or which source is the most updated. UniMentor addresses this problem by offering a more direct way to interact with university-related knowledge.

The final system combines several important parts. The backend handles authentication, authorization, RAG retrieval, AI model usage, streaming responses, document queueing, and administration operations. The relational database stores structured data such as users, subjects, documents, conversations, models, and bot configurations. The vector database stores embedded document chunks so that relevant information can be retrieved during question answering. The frontend provides the user interface for AthenaAI, AInsights, Career Mentor, and the Admin Panel.

Another important part of the implementation is the ingestion and automation flow. Documents can be uploaded manually or collected through bots, then parsed, split into chunks, embedded, and indexed. The ingestion worker handles the heavier processing outside the normal request-response flow, while the bot subsystem helps collect and update information from selected sources. This makes the knowledge base easier to maintain over time and reduces the need for repeated manual updates.

The platform also supports different user roles. Basic users can use AthenaAI, AInsights, and the proof-of-concept Career Mentor. Secretary users can manage secretary-controlled subjects and documents. Teacher users can upload course documents, manage LLM models, configure bots, and inspect bot logs. This role-based design was important because different users need different levels of access and control.

Overall, UniMentor shows that a RAG-based platform can be useful in a university environment. It does not replace official university systems, teachers, or secretariat staff, but it provides an additional access layer that helps users find and use information more easily. At the same time, it gives authorized users tools to manage the content that the assistant depends on.

The system still has limitations. The quality of the answers depends on the inserted documents, the retrieval results, the selected AI model, and the maintenance of the knowledge base. For this reason, UniMentor should be seen as a support tool and not as an official decision-making authority. Important academic or administrative actions should still be confirmed through official university channels.

In conclusion, UniMentor demonstrates a practical way of combining RAG, AI models, document processing, vector search, role-based administration, and automated knowledge gathering in one academic platform. With further extensions, such as more bot templates, real job API integration, an Exams Simulator, and more AI provider support, the platform could become an even more complete tool for academic assistance.

Bibliography

[1] E. Adamopoulou and L. Moussiades, "An overview of chatbot technology," in *Artificial Intelligence Applications and Innovations*, Springer, 2020, pp. 373–383. DOI: 10.1007/978-3-030-49186-4_31.

[2] S. Hussain, O. Ameri Sianaki, and N. Ababneh, "A survey on conversational agents/chatbots classification and design techniques," in *Web, Artificial Intelligence and Network Applications*, Springer, 2019, pp. 946–956. DOI: 10.1007/978-3-030-15035-8_93.

[3] P. Lewis et al., "Retrieval-augmented generation for knowledge-intensive nlp tasks," in *Advances in Neural Information Processing Systems*, vol. 33, 2020, pp. 9459–9474.

[4] Y. Gao et al., *Retrieval-augmented generation for large language models: A survey*, 2024. arXiv: 2312.10997 [cs.CL]. [Online]. Available: https://arxiv.org/abs/2312.10997.

[5] B. Abu Shawar and E. Atwell, "Chatbots: Are they really useful?" *Journal for Language Technology and Computational Linguistics*, vol. 22, pp. 29–49, 2007. DOI: 10.21248/jlcl.22.2007.88.

[6] J. Weizenbaum, "Eliza—a computer program for the study of natural language communication between man and machine," *Communications of the ACM*, vol. 9, pp. 36–45, 1966. DOI: 10.1145/365153.365168.

[7] I. V. Serban, R. Lowe, P. Henderson, L. Charlin, and J. Pineau, "A survey of available corpora for building data-driven dialogue systems," *arXiv preprint arXiv:1512.05742*, 2015. arXiv: 1512.05742 [cs.CL].

[8] O. Vinyals and Q. V. Le, "A neural conversational model," *arXiv preprint arXiv:1506.05869*, 2015. arXiv: 1506.05869 [cs.CL].

[9] I. V. Serban, A. Sordoni, Y. Bengio, A. Courville, and J. Pineau, "Building end-to-end dialogue systems using generative hierarchical neural network models," in *Proceedings of the Thirtieth AAAI Conference on Artificial Intelligence*, AAAI Press, 2016, pp. 3776–3783.

[10] T. B. Brown et al., "Language models are few-shot learners," in *Advances in Neural Information Processing Systems*, vol. 33, 2020, pp. 1877–1901.

[11] Z. Ji et al., "Survey of Hallucination in Natural Language Generation," *ACM Computing Surveys*, vol. 55, no. 12, pp. 1–38, 2023. DOI: 10.1145/3571730.

[12] N. Reimers and I. Gurevych, "Sentence-BERT: Sentence embeddings using Siamese BERT-networks," in *Proceedings of the 2019 Conference on Empirical Methods in Natural Language Processing and the 9th International Joint Conference on Natural Language Processing (EMNLP-IJCNLP)*, K. Inui, J. Jiang, V. Ng, and X. Wan, Eds., Hong Kong, China: Association for Computational Linguistics, Nov. 2019, pp. 3982–3992. DOI: 10.18653/v1/D19-1410. [Online]. Available: https://aclanthology.org/D19-1410/.

[13] J. J. Pan, J. Wang, and G. Li, *Survey of vector database management systems*, 2023. arXiv: 2310.14021 [cs.DB]. [Online]. Available: https://arxiv.org/abs/2310.14021.

[14] V. Karpukhin et al., "Dense passage retrieval for open-domain question answering," in *Proceedings of the 2020 Conference on Empirical Methods in Natural Language Processing (EMNLP)*, B. Webber, T. Cohn, Y. He, and Y. Liu, Eds., Online: Association for Computational Linguistics, Nov. 2020, pp. 6769–6781. DOI: 10.18653/v1/2020.emnlp-main.550. [Online]. Available: https://aclanthology.org/2020.emnlp-main.550/.

[15] J. Swacha and M. Gracel, "Retrieval-augmented generation (rag) chatbots for education: A survey of applications," *Applied Sciences*, vol. 15, no. 8, 2025, ISSN: 2076-3417. DOI: 10.3390/app15084234. [Online]. Available: https://www.mdpi.com/2076-3417/15/8/4234.

[16] Z. Li, Z. Wang, W. Wang, K. Hung, H. Xie, and F. L. Wang, "Retrieval-augmented generation for educational application: A systematic survey," *Computers and Education: Artificial Intelligence*, vol. 8, p. 100 417, 2025, ISSN: 2666-920X. DOI: https://doi.org/10.1016/j.caeai.2025.100417. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S2666920X25000578.

[17] C. Antico, S. Giordano, C. Koyuturk, and D. Ognibene, *Unimib assistant: Designing a student-friendly rag-based chatbot for all their needs*, 2024. arXiv: 2411.19554 [cs.HC]. [Online]. Available: https://arxiv.org/abs/2411.19554.

[18] D. Oreski and D. Vlahek, "Retrieval Augmented Generation in Large Language Models: Development of AI Chatbot for Student Support," in *Proceedings of the 15th International Conference on e-Learning 2024*, ser. CEUR Workshop Proceedings, vol. 3938, CEUR-WS, 2024.

[19] G. Lang and T. Gurpinar, "AI-Powered Learning Support: A Study of Retrieval-Augmented Generation (RAG) Chatbot Effectiveness in an Online Course," *Information Systems Education Journal*, vol. 23, no. 2, pp. 4–13, 2025. DOI: 10.62273/ZKLK5988.

[20] Y. Lian, *Machine assistant with reliable knowledge: Enhancing student learning via rag-based retrieval*, 2025. arXiv: 2506.23026 [cs.IR]. [Online]. Available: https://arxiv.org/abs/2506.23026.

[21] L. S. T. Nguyen and T. Quan, "Urag: Implementing a unified hybrid rag for precise answers in university admission chatbots – a case study at hcmut," *arXiv preprint arXiv:2501.16276*, 2025. arXiv: 2501.16276 [cs.CL].

[22] E. Matthes, *Python Crash Course: A Hands-On, Project-Based Introduction to Programming*, 3rd ed. No Starch Press, 2023, ISBN: 9781718502703.

[23] D. Flanagan, *JavaScript: The Definitive Guide*, 7th ed. O'Reilly Media, 2020, ISBN: 9781491952023.

[24] B. Cherny, *Programming TypeScript: Making Your JavaScript Applications Scale*. O'Reilly Media, 2019, ISBN: 9781492037651.

[25] M. Casciaro and L. Mammino, *Node.js Design Patterns: Level up your Node.js skills and design production-grade applications using proven techniques*, 4th ed. Packt Publishing, 2025, ISBN: 9781803238944.

[26] S. P. Kane and K. Matthias, *Docker: Up & Running: Shipping Reliable Containers in Production*, 3rd ed. O'Reilly Media, 2023, ISBN: 9781098131821.

[27] Docker Inc., *Docker compose documentation*, `https://docs.docker.com/compose/`, Accessed 2026-06-06, 2026.

[28] S. Smirnova and A. Tezuysal, *MySQL Cookbook: Solutions for Database Developers and Administrators*, 4th ed. O'Reilly Media, 2022, ISBN: 9781492093169.

[29] Qdrant, *Qdrant documentation*, `https://qdrant.tech/documentation/`, Accessed 2026-06-06, 2026.

[30] Ollama, *Ollama documentation*, `https://docs.ollama.com/`, Accessed 2026-06-06, 2026.

[31] OpenAI, *Openai api documentation*, `https://developers.openai.com/api/`, Accessed 2026-06-06, 2026.

[32] M. Aleksendrić, S. Batra, R. Palmer, and S. Ranjan, *Full stack FASTAPI, react, and mongodb: Fast-paced web app development with the Farm Stack*, 2nd ed. Packt Publishing Ltd, 2024.

[33] Pydantic, *Pydantic documentation*, `https://docs.pydantic.dev/`, Accessed 2026-06-06, 2026.

[34] J. Myers and R. Copeland, *Essential SQLAlchemy: Mapping Python to Databases*, 2nd ed. O'Reilly Media, 2015, ISBN: 9781491916469.

[35] pypdf Contributors, *Pypdf documentation*, `https://pypdf.readthedocs.io/`, Accessed 2026-06-06, 2026.

[36] python-docx Contributors, *Python-docx documentation*, `https://python-docx.readthedocs.io/`, Accessed 2026-06-06, 2026.

[37] python-pptx Contributors, *Python-pptx documentation*, `https://python-pptx.readthedocs.io/`, Accessed 2026-06-06, 2026.

[38] openpyxl Contributors, *Openpyxl documentation*, `https://openpyxl.readthedocs.io/`, Accessed 2026-06-06, 2026.

[39] R. Smith, "An overview of the tesseract ocr engine," in *Proceedings of the Ninth International Conference on Document Analysis and Recognition - Volume 02*, ser. ICDAR '07, USA: IEEE Computer Society, 2007, pp. 629–633, ISBN: 0769528228.

[40] Google Chrome Developers, *Puppeteer documentation*, `https://pptr.dev/`, Accessed 2026-06-06, 2026.

[41] Cheerio Contributors, *Cheerio documentation*, `https://cheerio.js.org/`, Accessed 2026-06-06, 2026.

[42] A. Banks and E. Porcello, *Learning react: Modern patterns for developing react apps*, 2nd ed. O'Reilly Media, 2020.

[43] Vite Contributors, *Vite documentation*, `https://vite.dev/`, Accessed 2026-06-06, 2026.

[44] Tailwind Labs, *Tailwind css documentation*, `https://tailwindcss.com/docs`, Accessed 2026-06-06, 2026.

[45] WorkOS, *Radix primitives documentation*, `https://www.radix-ui.com/primitives/docs`, Accessed 2026-06-06, 2026.

[46] J. Brooke, "Sus: A quick and dirty usability scale," *Usability Eval. Ind.*, vol. 189, Nov. 1995.

[47] Bangor, Aaron, P. Kortum, P. T., Miller, and J. T., "The system usability scale (sus): An empirical evaluation," *International Journal of Human-Computer Interaction*, vol. 24, pp. 574–, Aug. 2008. DOI: `10.1080/10447310802205776`.

[48] S. Borsci, S. Federici, and M. Lauriola, "On the dimensionality of the system usability scale: A test of alternative measurement models," *Cognitive Processing*, vol. 10, no. 3, pp. 193–197, Aug. 2009, ISSN: 1612-4790. DOI: `10.1007/s10339-009-0268-9`. [Online]. Available: `https://doi.org/10.1007/s10339-009-0268-9`.

[49] A. Bangor, P. Kortum, and J. Miller, "Determining what individual sus scores mean: Adding an adjective rating scale," *J. Usability Studies*, vol. 4, no. 3, pp. 114–123, May 2009.