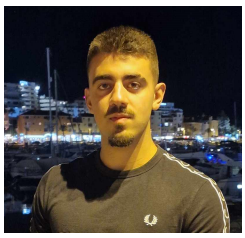
Agents and Multiagent Systems

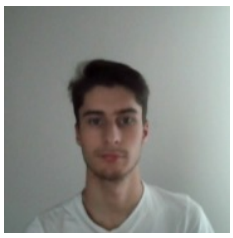
MentorAI

MEI - 2024/2025

João Barroso
PG57554



Lucas Oliveira
PG57886



Maurício Pereira
PG55984

Rafael Gomes
PG56000

Group 5

University of Minho

Contents

List of Figures

1 Introduction

This report is part of the practical project for the **Agents and Multi-Agent Systems** course, integrated in the Master's in Computer Engineering at the University of Minho.

The project focuses on the design and implementation of a conversational multi-agent system capable of assisting students in learning History through natural language interaction. The system leverages intelligent agents, each with specific responsibilities, cooperating in real time to simulate the behaviour of an empathetic and knowledgeable History teacher.

This educational assistant, named **MentorIA**, incorporates natural language understanding, emotional state detection, memory of past interactions, feedback collection and a dynamic adaptation to the educational cycle (1^{st}, 2^{nd} or 3^{rd} cycle of basic education in Portugal). The system integrates with a Large Language Model (LLM), combining it with information retrieval from a structured document base via Retrieval-Augmented Generation (RAG).

All agents were implemented using the **Smart Python Agent Development Environment (SPADE)** and coordinated via a message-passing architecture. This report outlines the domain analysis, the agent-based architecture, interaction flows, technical implementation details, and potential areas for future improvement.

2 Domain Analysis and Objectives

The primary objective of this project is to design and implement a multi-agent system that emulates the behaviour of an intelligent and empathetic History teacher, capable of interacting with students through natural language in real time.

Within this domain, the system supports students from different educational levels—namely the 1^{st}, 2^{nd}, and 3^{rd} cycles of basic education in Portugal—by interpreting their questions and delivering contextualised and pedagogically appropriate answers. To accomplish this, the system integrates multiple autonomous agents, each responsible for specific functionalities: emotional state detection, memory management, response generation, educational cycle classification, summarisation of previous interactions, and feedback collection.

The domain centres around conversational education, enriched by historical knowledge dynamically retrieved through Retrieval-Augmented Generation (RAG). The system ensures that answers are tailored not only to the content of the question but also to the emotional state of the student and their educational stage.

This architecture simulates the workflow of a real-life teacher, providing a coherent and structured learning experience that fosters historical understanding. Furthermore, the system is capable of maintaining conversational memory, encouraging personalised learning, and collecting user feedback to improve future interactions. The ultimate goal is to provide an engaging, adaptive, and intelligent virtual tutor that supports History education in a scalable and emotionally aware manner.

3 Project Structure

As illustrated in Figure 1, the project follows a modular and well-organized architecture that separates concerns between backend logic, data storage, semantic indexing, configuration, and user interface.

The `BE` directory encapsulates all backend logic related to the multi-agent system. It contains two core folders:

- `agents/` – where the SPADE agents are implemented;
- `behaviours/` – containing the behaviours that define each agent's specific functionalities.

The `classes/` folder defines serializable Python classes used in message exchange between agents, such as user inputs and LLM responses.

The `db/` folder manages all persistence logic and database interaction. It includes:

- `sql/` – with raw SQL scripts to create and populate tables;
- `models.py` – which defines the database models;
- `memory_repository.py` – the repository responsible for saving and updating memory entries;
- `db.py` and `setup_tables.py` – utilities for session and schema setup.

The `docs/` folder stores historical content for each educational cycle in JSON format (`1ciclo.json`, `2ciclo.json`, `3ciclo.json`), along with a subfolder `faiss/` where FAISS indexes are stored after embedding generation.

The `scripts/` directory contains the script `generate_embeddings.py`, which transforms the historical content into semantic vectors used during the Retrieval-Augmented Generation (RAG) process.

At the root level, the project includes:

- `main.py` – the application's entry point, responsible for launching the agents and the FastAPI server;
- `Dockerfile` and `docker-compose.yml` – for containerized deployment;
- `.env` – for managing API keys and environment variables;
- `requirements.txt` – for managing Python dependencies;
- `README.md` – with documentation for local deployment and usage.

Finally, the `FE/` directory hosts the frontend application responsible for user interaction via text or voice, and for receiving real-time responses from the backend via Server-Sent Events (SSE).

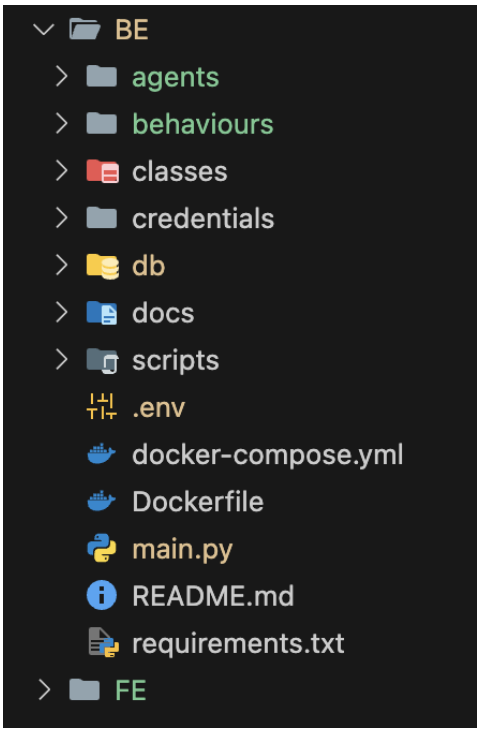


Figure 1: Project Structure

4 Architecture

In this section, a description of the developed multi-agent system will be presented, accompanied by diagrams to provide a better understanding of its architecture and operation. The architecture modeling was based on the previously described problem, attempting to conceptualize all the necessary components for the system's construction. Throughout this section, the interactions between the agents that comprise the system will also be analyzed, ensuring its integrity.

4.1 System Architecture

The developed system adopts a distributed architecture based on autonomous agents that communicate asynchronously to simulate the behaviour of a conversational History teacher. As shown in Figure 2, each agent plays a specialised role, and all interactions are coordinated dynamically through message exchanges, following the SPADE protocol.

The architecture begins with the **UserInterfaceAgent**, which receives user input (text or transcribed voice) and sends it to the **DialogueManagerAgent**. This central agent orchestrates the rest of the workflow by forwarding the input to the **EmotionAgent** (for emotional state detection) and the **CycleClassifierAgent** (to determine the educational level of the question).

Once the emotion and cycle are identified, the DialogueManager triggers two operations in parallel: saving the input to memory via the **MemoryAgent**, and requesting a memory summary from the same agent, which is then processed by the **SummarizerAgent**. With all context gathered (user message, emotion, cycle, and summary), the DialogueManagerAgent constructs the final prompt and sends it to the **LLMAgent**, responsible for interacting with the Gemini

LLM.

Optionally, the LLMAgent enriches the prompt using Retrieval-Augmented Generation (RAG), retrieving relevant historical documents previously embedded by semantic indexing. The generated response is then sent back to the DialogueManagerAgent, which delivers it to the UserInterfaceAgent and stores the reply in memory.

Finally, the **FeedbackAgent** is periodically triggered to ask the user if the response was helpful, allowing for real-time improvement and user engagement.

This architecture allows for continuous memory tracking, emotional sensitivity, content adaptation based on educational cycle, and real-time intelligent generation of responses. Thanks to asynchronous communication and autonomous behaviour encapsulated in each agent, the system is scalable, flexible, and pedagogically effective.

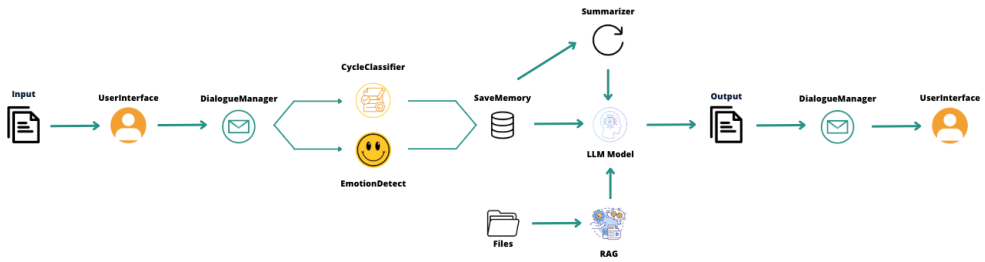


Figure 2: System Architecture

4.1.1 Agents

As mentioned earlier, the **HistIA** system is composed of eight agents, each with a specific responsibility that contributes to the overall goal of delivering pedagogically appropriate, emotionally aware, and contextually relevant answers to History students.

1. **UserInterfaceAgent** acts as the bridge between the frontend and the multi-agent backend. It receives user messages—whether written or transcribed from speech—and forwards them to the DialogueManagerAgent. It also handles the real-time delivery of responses back to the user via Server-Sent Events (SSE). This agent is stateless and operates as the representative of the user in the backend.

2. **DialogueManagerAgent** serves as the central orchestrator of the system. Upon receiving a message from the UserInterfaceAgent, it triggers parallel analysis tasks: sending the message to the EmotionAgent and to the CycleClassifierAgent. After classification, it stores the message in memory and requests a summary of past interactions. Once all data is collected (emotion, memory, and educational cycle), it composes and sends a structured prompt to the LLMAgent. Finally, it returns the response to the user and updates the memory.

   Attributes:

   - `current_user_message` – temporary state holding the user's latest message;
   - `current_memory_id` – tracks the message ID for later updates.

3. **EmotionAgent** is responsible for detecting the emotional tone of the user's message. Using the Google Cloud Speech or text APIs, it classifies emotions such as "happy", "sad", or "confused". This emotional context helps tailor the tone of the LLM response to be more empathetic and appropriate.

4. **CycleClassifierAgent** receives the user's question and determines which educational cycle it pertains to: 1st, 2nd, or 3rd cycle of Portuguese basic education. It uses the Gemini model to perform this classification via a zero-shot prompt, and returns the result to the DialogueManager.

5. **MemoryAgent** manages long-term conversational memory. It stores each user message and system response in a PostgreSQL database. It is also responsible for fetching full memory threads, saving feedback, and updating existing records with cycle classification, emotions, and final LLM outputs.

6. **SummarizerAgent** receives the structured message history from the MemoryAgent and reorganises it into a more readable and coherent format. This reformatted history is included in the prompt sent to the LLM, making it easier for the model to understand the conversation context without altering the content or introducing abstraction.

7. **LLMAgent** is responsible for generating the final response using the Gemini 2.0 model. It builds a complete prompt including user question, emotion, memory summary, and optionally context retrieved via RAG (Retrieval-Augmented Generation). It then returns a well-structured and pedagogically appropriate answer to the DialogueManager.

8. **FeedbackAgent** is triggered every few interactions (e.g., every 4 messages) to collect feedback from the user. It asks whether the response was useful, and optionally collects a suggestion for improvement. This feedback is sent back to the MemoryAgent for storage.

4.1.2 Behaviours

Each agent in the system operates based on one or more behaviours, which define their autonomous and reactive capabilities. The following behaviours were developed and are grouped below according to the agent that executes them:

- **CycleClassifierAgent**
  - `ClassifyCycleBehaviour` – Receives a user question and uses a zero-shot prompt with Gemini to classify it into the 1st, 2nd, or 3rd educational cycle. The detected cycle is returned to the DialogueManager.
- **DialogueManagerAgent**
  - `ProcessDialogueBehaviour` – Centralises the logic for coordinating the dialogue. Routes messages to the EmotionAgent and CycleClassifierAgent, manages memory interaction, forwards prompts to the LLMAgent, and handles the reception of the final response.
- **EmotionAgent**
  - `AnalyseEmotionBehaviour` – Processes the user's message to determine emotional tone using Google's emotion detection service. Returns the emotion to the DialogueManager.
- **FeedbackAgent**

  - `AskFeedbackBehaviour` – Decides when to ask the user for feedback and sends a message asking if the response was useful.
  - `ReceiveFeedbackBehaviour` – Receives the user's feedback (positive/negative) and optional suggestions, and forwards this information to the MemoryAgent.

- **LLMAgent**
  - `CallGeminiBehaviour` – Receives the composed prompt (including user input, memory, emotion, and optionally RAG context), sends it to the Gemini model, and returns the generated response to the DialogueManager.

- **MemoryAgent**
  - `SaveMemoryBehaviour` – Persists new user messages to the PostgreSQL database.
  - `FetchMemoryBehaviour` – Retrieves the full memory thread for the current user.
  - `UpdateMemoryBehaviour` – Updates an existing memory record with feedback, emotion, cycle, or the LLM response.

- **SummarizerAgent**
  - `SummarizeMemoryBehaviour` – Reorganises the memory history into a more structured and readable format for the LLM prompt. It does not perform abstraction, but improves coherence.

- **UserInterfaceAgent**
  - `ReceiveUserInputBehaviour` – Receives the user's initial question via FastAPI and sends it into the SPADE system.
  - `ReceiveLLMResponseBehaviour` – Listens for the response from the DialogueManager and streams it to the frontend using Server-Sent Events.

4.1.3 Classes

The system includes two main serializable classes that facilitate message exchange and structured data sharing between agents. These classes are stored in the `classes/` directory and are encoded and decoded using the `jsonpickle` library to allow transmission through SPADE messages.

- **UserMessage** – Represents a user input. It encapsulates the message text, the detected emotional state, and optionally other metadata such as timestamp or session identifiers. This class is used as a standard format when passing user input between agents like the EmotionAgent, CycleClassifierAgent, and DialogueManagerAgent.
- **LLMResponse** – Represents the response generated by the LLM. It contains a single field with the generated text, and may later be extended to support more metadata such as source citations (if RAG is enabled), confidence scores, or LLM model identifier.

4.2 Data Structure

The system uses a PostgreSQL database to store and manage long-term conversation memory. This memory is essential for maintaining context across user interactions, supporting both continuity and traceability of the dialogue.

The primary table is `memory_messages`, which stores one record per user interaction. This includes both the user's input and the system's response, along with relevant metadata. The structure of this table is defined and maintained using raw SQL scripts stored in `db/sql/`, and accessed programmatically through the SQLAlchemy-based `memory_repository.py`.

Each entry in the `memory_messages` table contains the following fields:

- `id` – A unique identifier for the memory record (primary key).
- `text` – The original message written by the user.
- `emotion` – The emotional tone detected by the EmotionAgent (e.g., "neutral", "confused", "happy").
- `response` – The response generated by the LLM and returned to the user.
- `cycle` – The educational level assigned to the question (e.g., "1º ciclo", "2º ciclo", "3º ciclo"), classified by the CycleClassifierAgent.
- `feedback` – A Boolean field indicating whether the user found the response useful.
- `user_suggestion` – An optional text field where users can leave suggestions for improving a response.
- `created_at` – A timestamp indicating when the interaction occurred.

All memory-related logic—such as saving, fetching, and updating records—is abstracted through a repository pattern, ensuring decoupling between data access and agent behaviours. This allows the MemoryAgent to persist and retrieve conversational context as needed, while maintaining clean separation of responsibilities.

4.3 Message Types

In our multi-agent system, communication between agents is performed through sending and receiving messages that follow a standardized structure, facilitating information flow and coordination among different components. To this end, we adopt the concept of **performative**, inspired by speech act theory, which represents the intention of the message and functions in our architecture analogously to HTTP `GET` and `POST` requests.

There are two main types of **performatives** used:

- **request**: This type of message is used to ask another agent to perform an action or provide data. It acts as an explicit request, similar to an HTTP `GET` or `POST` that expects a response or concrete action.
- **inform**: Indicates the communication of information or notification of a state or data. The sending agent informs the receiver about an event, result, or relevant data, without necessarily expecting a direct reply.

Besides the **performative**, each message includes important metadata for correct routing and processing:

- **source**: Identifies the agent or component that originated the message. This allows the receiving agent to know the origin of the information and apply appropriate logic in context.
- **purpose**: Defines the specific goal of the message within the conversational or operational flow, detailing the intention beyond the **performative**. For example, a `request` may have `purpose "emotion_request"` to request an emotional analysis, or an `inform` may have `purpose "save_memory"` to indicate that it is sending data to be stored.

This structure allows agents to know exactly what is expected of them and the context of the communication, facilitating modularity and scalability of the system. In the **DialogueManager** agent code, this logic is clear: upon receiving a message, the agent analyzes the combination of `performative`, `source`, and `purpose` to decide to which component to forward the message or which action to execute.

For example:

- When receiving a `request` message from the **UserInterfaceAgent**, the **DialogueManager** forwards the message to the **EmotionAgent** as an `inform`, requesting an emotional analysis of the user's message.
- When receiving an `inform` from the **CycleClassifierAgent**, it updates the internal state with the detected cycle.
- When receiving an `inform` from the **MemoryAgent** with different purposes (`full_memory`, `save_memory`, `update_memory`), it executes distinct actions such as forwarding to the **SummarizerAgent** or updating the memory state.
- The `purpose` field serves to distinguish, within the same `performative` type, different specific functionalities, making communication more semantic and clear.

Thus, the architecture based on **performatives**, **source**, and **purpose** ensures that messages are consistently interpreted, contributing to the coordinated and efficient behavior of the conversational assistant.

5 Case Study

Context and Problem

In the Portuguese educational system, two persistent challenges hinder effective learning outcomes: low student motivation and engagement, and a lack of real-time feedback and effective interaction during the learning process.

Motivation plays a critical role in student success. Many learners experience difficulty maintaining interest and active participation in their studies, especially when faced with abstract content or when learning autonomously. This disengagement often leads to poor academic performance and increased dropout rates.

Simultaneously, the absence of immediate, personalized feedback creates barriers for students trying to clarify doubts or reinforce knowledge. Traditional classroom settings and existing educational resources often fail to provide timely responses tailored to individual learning needs, limiting the potential for adaptive and effective learning.

Solution: A Multi-Agent Conversational Educational Assistant

To overcome these challenges, we developed a virtual assistant based on a multi-agent architecture that offers personalized, empathetic, and interactive support. The system is designed to:

- Detect and respond to the emotional state of students, helping to maintain motivation through the **EmotionAgent**.
- Identify the student's current learning phase or cycle via the **CycleClassifierAgent**, allowing tailored educational content.
- Provide immediate and context-aware responses using a large language model managed by the **LLMAgent**.
- Maintain continuity and context across interactions through the **MemoryAgent**, enabling progressive and adaptive learning.
- Collect and integrate student feedback dynamically via the **FeedbackAgent**, continuously improving response quality.

Operational Workflow

When a student submits a question, the **UserInterfaceAgent** forwards it as a `request` to the **DialogueManager**, which orchestrates communication among agents. The **EmotionAgent** assesses the student's emotional tone to adjust responses empathetically. The **CycleClassifierAgent** classifies the learning stage to adapt the information provided.

The **MemoryAgent** retrieves prior interactions, ensuring that responses build on the student's learning history. The **LLMAgent** synthesizes these inputs to generate a timely, personalized answer delivered immediately to the student.

The system encourages engagement by inviting feedback, which is processed by the **FeedbackAgent** to refine future interactions and maintain high-quality support.

Impact and Benefits

This conversational assistant addresses core educational challenges by:

- Enhancing motivation through empathetic and emotionally aware dialogue.
- Increasing student engagement with personalized and adaptive support.
- Delivering real-time, interactive feedback that facilitates immediate doubt resolution.
- Promoting sustained learning via contextual memory and progressive assistance.

Through this case study, we demonstrate how a carefully designed multi-agent conversational system can significantly improve the learning experience, supporting students in overcoming motivational barriers and benefiting from timely, effective feedback.

6 Subjects and Educational Levels

The system currently supports three core school subjects: **Mathematics**, **Portuguese**, and **History**. For each subject, the information used by the large language model (*LLM*) during the *Retrieval-Augmented Generation* (*RAG*) process is stored in the ***docs/*** directory within the backend structure.

The educational content is organised according to the official Portuguese basic education cycles:

- **1st cycle** (grades 1–4)
- **2nd cycle** (grades 5–6)
- **3rd cycle** (grades 7–9)

Inside the ***docs/*** folder, structured data files are provided in *JSON* format, with one file per subject and educational cycle. Examples include:

- **Mathematics:** *math_1ciclo.json*, *math_2ciclo.json*, *math_3ciclo.json*
- **Portuguese:** *portuguese_1ciclo.json*, *portuguese_2ciclo.json*, *portuguese_3ciclo.json*
- **History:** *history_1ciclo.json*, *history_2ciclo.json*, *history_3ciclo.json*

This structure allows the system to retrieve content that is contextually aligned with the student's educational level. The **CycleClassifierAgent** is responsible for identifying the appropriate cycle based on the user's input, enabling accurate and targeted retrieval of supporting materials during the *RAG* process.

Importantly, this system is not designed as a traditional chatbot that provides brief answers to isolated questions. Instead, it functions as a virtual tutor. When a student asks a question, the system does not simply return the answer, it explains why that answer is correct, the reasoning steps involved, and any foundational knowledge needed to understand the concept fully. Much like a real teacher, it guides the student through the learning process, offering pedagogical explanations rather than just information.

For example, if a student asks for help with a math problem, the system will not only give the solution but also walk through the method step by step, clarifying concepts, justifying the steps, and encouraging understanding. This approach promotes deep learning, conceptual clarity, and critical thinking, aligning with the educational goals of the project.

7 Diagrams

7.1 Class Diagram

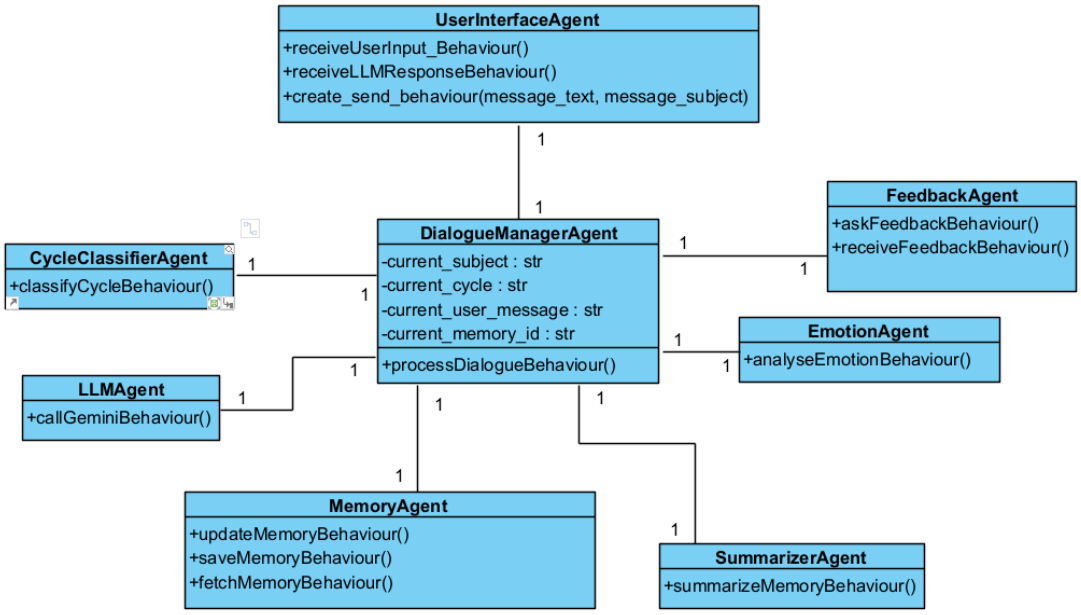


Figure 3: Class Diagram.

7.2 Sequence Diagram

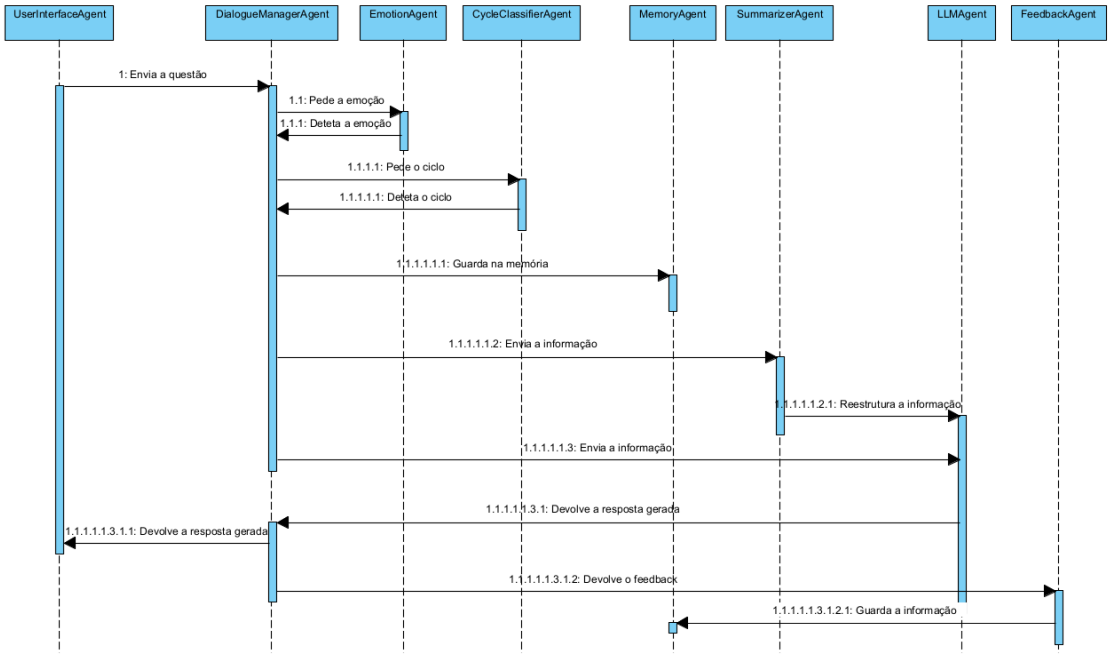


Figure 4: Sequence Diagram.

7.3 Communication Diagram

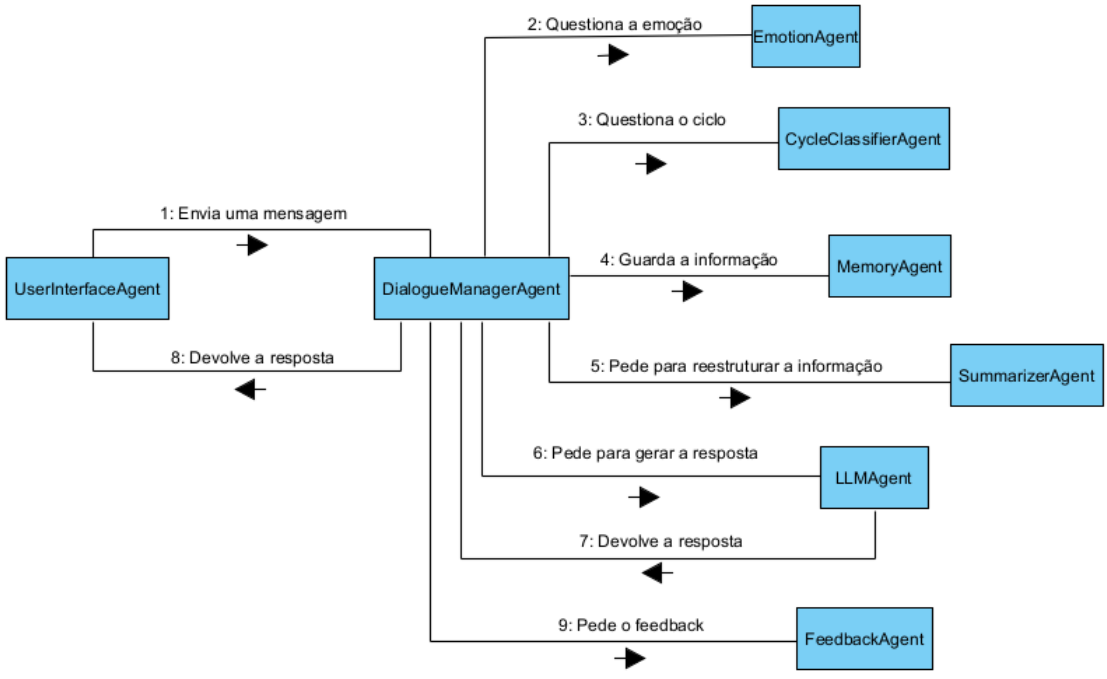


Figure 5: Communication Diagram.

7.4 Activity Diagram

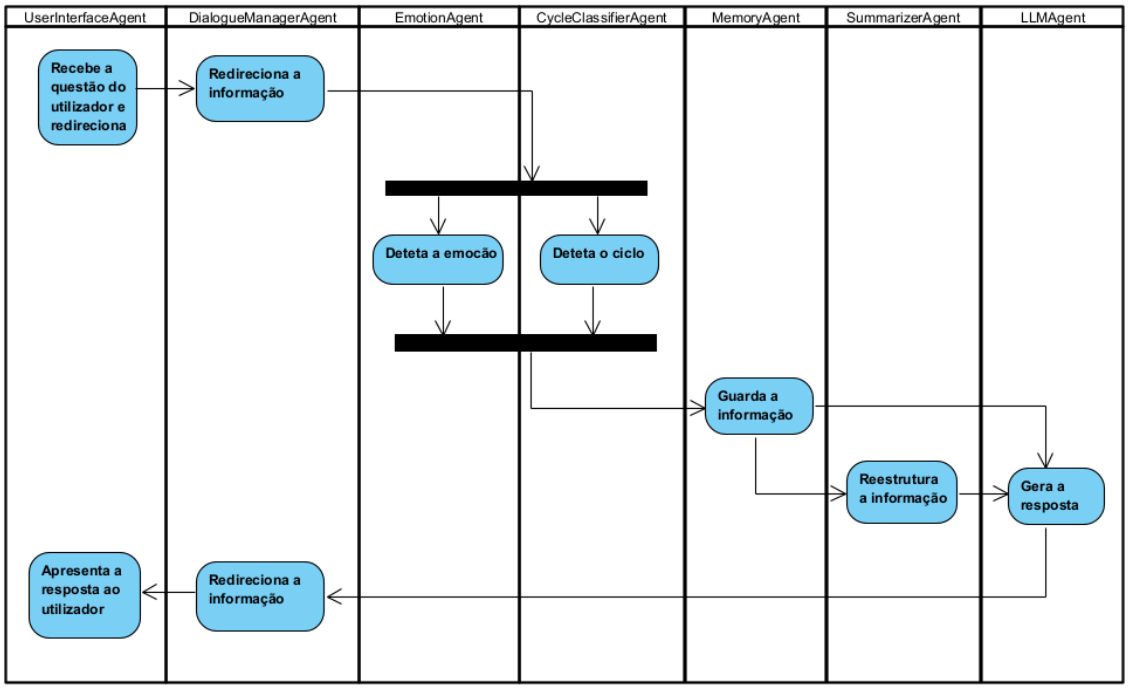


Figure 6: Activity Diagram.

8 Frontend Architecture and User Interaction Layer

The frontend of the **MentorAI** platform was developed using **React**, adopting a clean and modular design focused on user experience and pedagogical clarity. This interface enables students to interact naturally with the system, choosing their subject of study, engaging in contextual conversations, and receiving personalised educational support.

Communication between the frontend and backend is established using **Server-Sent Events (SSE)**, which allows the streaming of model-generated responses in real time. This creates a more natural, responsive, and human-like interaction flow.

Below, we present the main screens of the platform, highlighting their functional and visual characteristics.

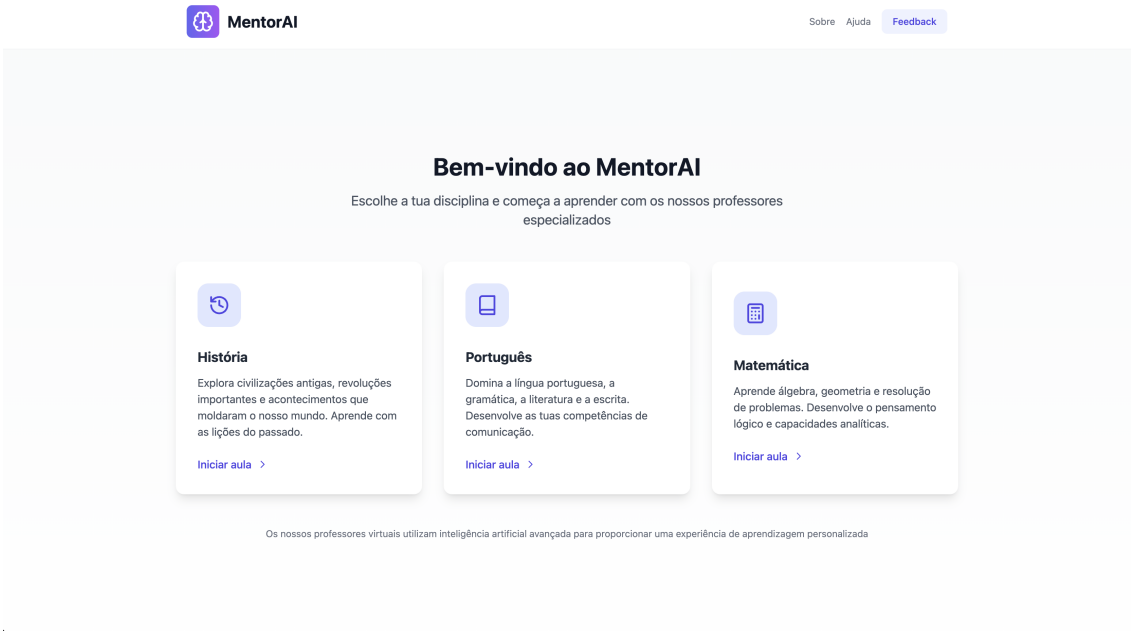


Figure 7: Welcome screen – Subject selection interface. Students choose the subject in which they want to receive assistance: History, Portuguese or Mathematics.

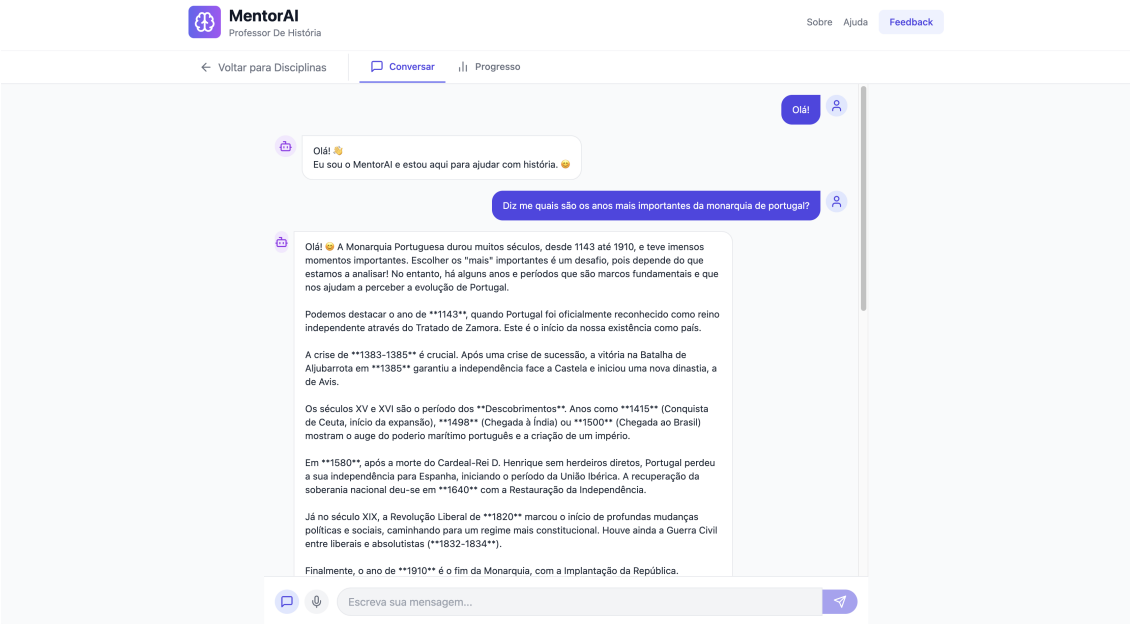


Figure 8: History conversation interface – The user starts a dialogue with the History teacher agent, who replies in a detailed and pedagogical way.

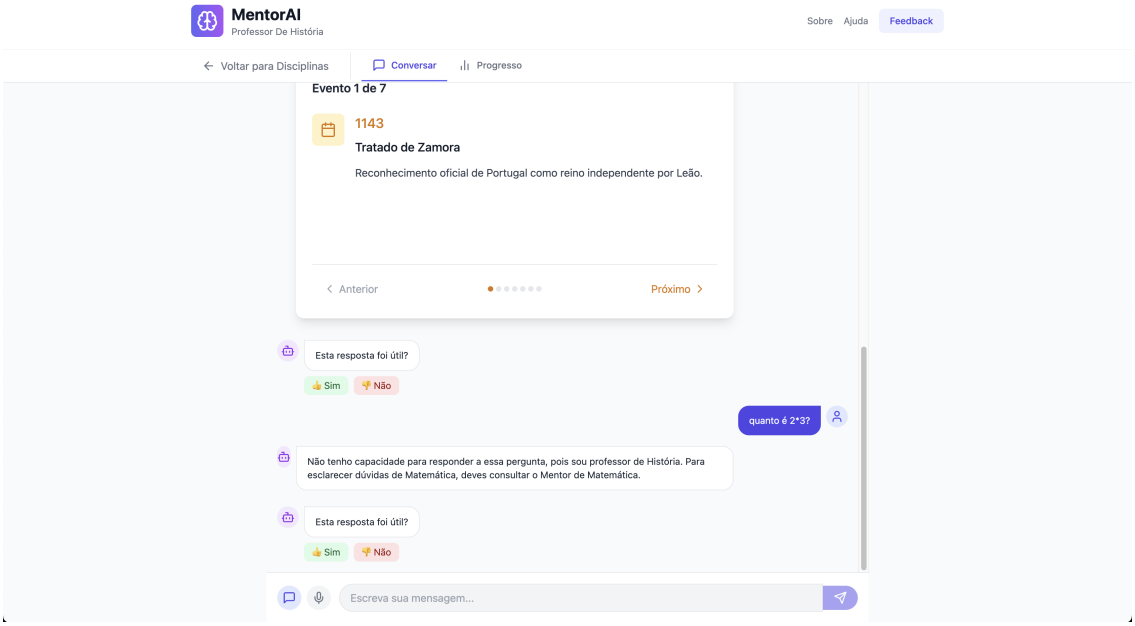


Figure 9: History assistant with feedback mechanism – The system explains historical events using a visual format and collects real-time user feedback.

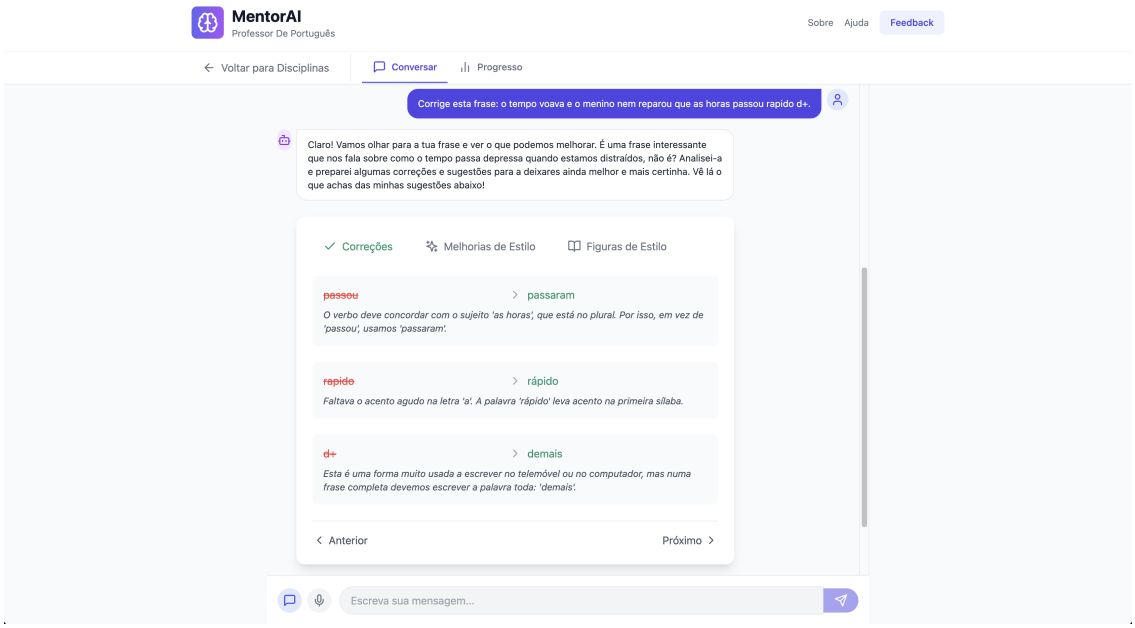


Figure 10: Portuguese assistant – The system analyses a sentence and provides grammar corrections, style improvements, and stylistic suggestions.

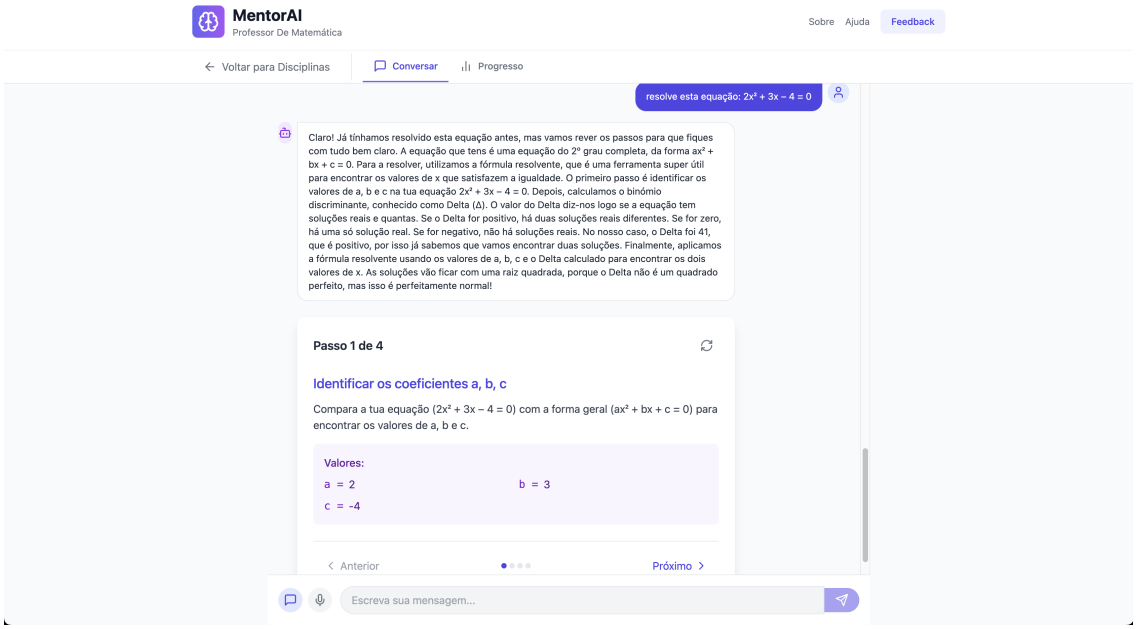


Figure 11: Mathematics assistant – The system guides the student step by step through solving a second-degree equation, fostering understanding of the underlying process.

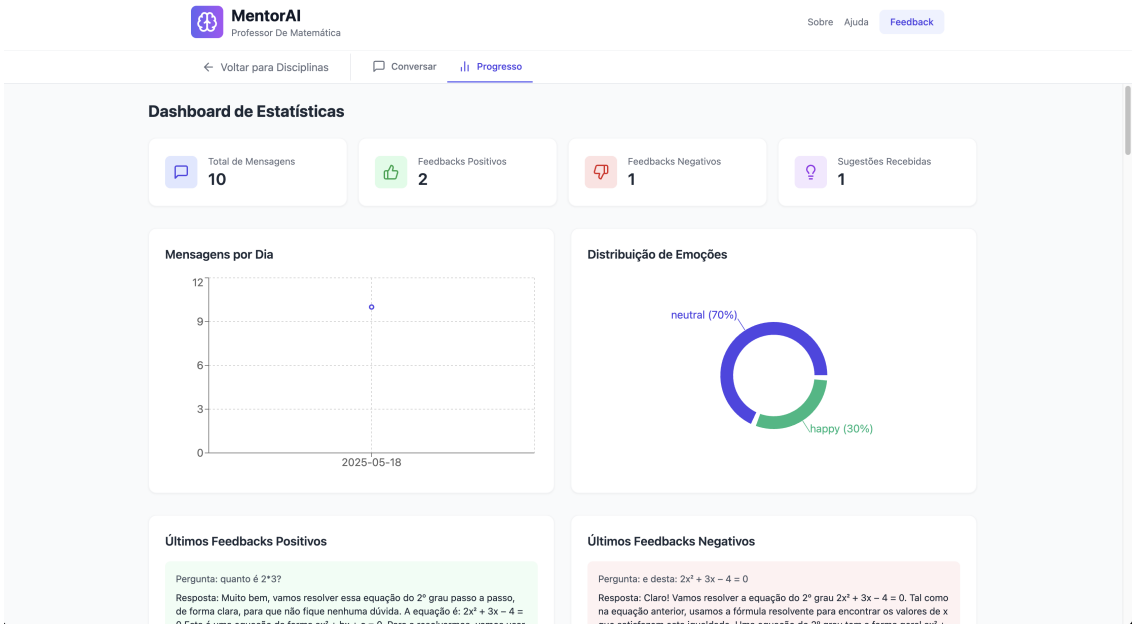


Figure 12: Progress Dashboard – Displays usage statistics, emotion distribution, and the most recent user feedback, supporting system improvement and monitoring.

9 Conclusions and Future Work

Conclusions

This project presented the development of an innovative educational platform, built upon a multi-agent architecture, designed to support students in their learning journey through personalised, interactive, and emotionally aware assistance. The proposed solution — named **MentorAI** — integrates natural language processing, emotional state detection, educational level adaptation, and conversational memory, offering a coherent and pedagogically oriented learning experience.

By combining autonomous agents with a large language model (LLM), the system is capable of engaging in meaningful and contextualised conversations with students across multiple subjects, namely History, Portuguese, and Mathematics. The modularity of the architecture and the incorporation of Retrieval-Augmented Generation (RAG) further enhance the system's ability to deliver accurate, relevant, and well-structured responses.

The use of Server-Sent Events (SSE) in the communication between frontend and backend allows real-time feedback and a fluid user experience, reinforcing the sense of a natural dialogue. This contributes to increased student engagement and motivation, addressing common challenges in educational environments.

Future Work

Although the system has demonstrated its potential in offering intelligent and adaptive support, several avenues for future improvement have been identified, particularly with a view to scaling and long-term deployment:

- **Feedback classification and tracking:** Future versions of the system should include a structured mechanism for automatically classifying user feedback into actionable categories or "tickets". These could include suggestions for improving the interface, identifying limitations in the current model responses, or proposing content enhancements.
- **Data layer restructuring:** The current data persistence model may be restructured into a fully relational database schema, optimised for performance, scalability, and data integrity. This would support more complex queries, analytics, and long-term growth of the system.
- **Scalability and deployment optimisation:** A major focus of future development lies in preparing the system to support simultaneous access by a large number of students. This includes container orchestration (e.g., with Kubernetes), load balancing, and asynchronous processing optimisations to ensure system responsiveness and reliability under high demand.

These improvements aim to elevate the platform from a proof-of-concept to a production-ready educational assistant capable of supporting real-world classroom environments and potentially national-level deployments.