



Universidad Politécnica
de Madrid

**Escuela Técnica Superior de
Ingenieros Informáticos**



Master in Data Science

Master Thesis

PhronesisAI: GenAI Learning Platform

Author: Aaron Moyano Alcacer
Tutor: Alejandro Rodríguez González

Madrid, June, 2025

This Master Thesis has been deposited in ETSI Informáticos de la Universidad Politécnica de Madrid.

Master Thesis

Master in Data Science

Title: PhronesisAI: GenAI Learning Platform
June, 2025

Author: Aaron Moyano Alcacer

Supervisor: Alejandro Rodríguez González

Lenguajes y Sistemas Informáticos e Ingeniería del Software

ETSI Informáticos

Universidad Politécnica de Madrid

Summary

El objetivo principal de este Trabajo Fin de Máster es explorar y poner en práctica arquitecturas modernas de agentes de Inteligencia Artificial para crear una plataforma educativa interactiva basada en Large Language Model (LLMs). Este trabajo se apoya en el uso, cada vez mayor, de la Inteligencia Artificial en el ámbito educativo, con el fin de ofrecer experiencias de aprendizaje más autónomas y personalizadas.

La investigación empieza con un repaso a la historia de la Inteligencia Artificial, desde sus inicios con el enfoque simbólico y los sistemas expertos, hasta llegar al aprendizaje profundo que domina hoy en día. Se revisan momentos clave como la creación del perceptrón, el desarrollo del algoritmo de retropropagación y la aparición de las arquitecturas basadas en transformadores, que han dado lugar a modelos como GPT o LLaMA. Estos modelos han mostrado una gran capacidad para entender y generar lenguaje natural, aunque también tienen limitaciones importantes en cuanto a fiabilidad, transparencia y precisión de los datos que ofrecen.

Para hacer frente a estas limitaciones, el trabajo se centra en el concepto de agentes de Inteligencia Artificial como sistemas autónomos que pueden percibir, razonar y actuar. Se estudian varios tipos de agentes, junto con aspectos técnicos importantes como la ingeniería de prompts, los ciclos de razonamiento, la gestión de memoria y las estrategias de arquitecturas. También se destaca el uso del enfoque Retrieval-Augmented Generation, que ayuda a mejorar la precisión de los LLMs al integrar fuentes externas de información en tiempo real.

También se analizan dos arquitecturas modernas para coordinar agentes: AutoGen, desarrollada por Microsoft, que permite la comunicación entre varios agentes a través de conversaciones estructuradas; y LangGraph, un framework basado en LangChain que organiza el trabajo de los agentes en forma de grafos de ejecución dinámicos. Estas arquitecturas ofrecen una base modular, flexible y escalable para implementar agentes educativos con inteligencia artificial. Su utilidad se estudia en relación con la personalización de los contenidos y la conexión con bases de conocimiento específicas del área de estudio.

Este trabajo ofrece una base teórica y práctica para crear herramientas educativas apoyadas en inteligencia artificial. Al combinar el potencial generativo de los LLMs con la autonomía organizada de los agentes inteligentes, y gracias a arquitecturas que permiten su coordinación, se plantea un enfoque con potencial para impulsar el desarrollo de nuevas plataformas educativas.

Abstract

The main goal of this Master's Thesis is to explore and apply advanced architectures of intelligent agents to develop an interactive educational platform powered by Large Language Models (LLMs). This work builds on the increasing use of Artificial Intelligence in educational technologies and applies cutting-edge Artificial Intelligence approaches to support autonomous and personalized learning experiences.

The research starts with a historical overview of Artificial Intelligence, tracing its evolution from symbolic approaches and expert systems to the current dominance of deep learning. It highlights key milestones such as the development of the perceptron, the rise of backpropagation, and the introduction of transformer-based architectures that led to modern LLMs like GPT, and LLaMA. While these models have shown impressive abilities in understanding and generating natural language, they also face significant challenges regarding reliability, transparency, and factual accuracy.

To address these limitations, the thesis explores the concept of AI agents as autonomous systems capable of perceiving, making decisions, and taking action. It examines different types of agents, as well as key technical components such as prompt engineering, reasoning loops, memory management, and planning strategies. Particular focus is given to the Retrieval-Augmented Generation (RAG) paradigm, which improves the performance and factual reliability of LLMs by integrating external knowledge sources in real time.

Furthermore, the study reviews two frameworks for agent orchestration: AutoGen, developed by Microsoft, which enables multi-agent communication through structured conversational protocols; and LangGraph, a LangChain-based framework that structures agent workflows as dynamic execution graphs. These architectures offer a modular, flexible, and scalable foundation for deploying AI-powered educational agents. Their usefulness is analyzed in terms of pedagogical interaction, personalized content delivery, and integration with domain-specific knowledge bases.

Finally, this thesis lays a solid theoretical and practical foundation for the development of AI-powered educational tools. By combining the generative capabilities of LLMs with the structured autonomy of intelligent agents, enabled by orchestrated architectures, this work proposes a framework with the potential to support the creation of next-generation educational platforms.

Acknowledgements

I would like to begin by expressing my deepest gratitude to all the professors who have been part of my education at the Universidad Politécnica de Madrid. Their dedication and teachings have been essential to my development as a future data scientist.

I would like to make a special mention of my Master's Thesis advisor, Alejandro Rodríguez González, whose persistent support was fundamental during the most challenging moments of this project.

I cannot overlook the immense support I have received from my family, who deserve special recognition for their steadfast encouragement. Their dedication, sacrifice, and unconditional love have been the driving force that motivated me to keep moving forward. I am deeply thankful for the freedom they have given me to make my own decisions and face life in my own way. Thank you to Silvia, Antonio, and Katya for always being there, for believing in me, and for encouraging me to pursue my dreams.

I would like to express my deepest gratitude to my grandmother, Natividad Trigo, for everything she has done for me. For being like a second mother, for her unwavering support, and for always standing by me. I know she is proud of my achievements, and I hope one day to be able to repay everything she has done for our family.

I would also like to express my heartfelt appreciation to Alejandro, David, Iván, Marcos, Miguel, and Sergio. It is difficult to express just how much I value your friendship. Together, we have laughed, cried, overcome obstacles, and shared unforgettable moments that will forever remain in my memory, profoundly enriching my life. Without the constant support we have given each other as a group, continuing in this vocation as a data scientist would have been a much greater challenge. I am truly grateful to have your presence and support, and for that reason, you deserve your place in these acknowledgements.

He died doing what he wanted, no matter what, right? I bet he was happy.
— Kentaro Miura, **Berserk**

Contents

1 Introduction	1
1.1 Motivation and Objectives	3
1.1.1 Motivation	3
1.1.2 Objectives Definition	3
2 State of the Art	5
2.1 Historical Evolution of Artificial Intelligence	6
2.2 The Emergence of Multi-Agent Systems in AI	8
2.3 GenAI: LLMs, Agents, and Useful Systems	9
2.3.1 Large Language Models (LLMs)	9
2.3.2 AI Agents	10
2.3.3 Retrieval-Augmented Generation (RAG)	13
2.4 Generative AI Agent Architectures in Education	15
2.4.1 Homework and Assignment Assistance	17
2.5 Agent Architectures: AutoGen and LangGraph	18
2.5.1 Introduction and Motivation for AutoGen	19
2.5.2 Architectural Overview of AutoGen	20
2.5.3 Conversable and Customisable Agents	21
2.5.3.1 Multi-Agent Conversations and Coordination	22
2.5.3.2 Tool Integration and Action Execution	22
2.5.3.3 Execution Model and Observability	23
2.5.4 Key Differences Between AutoGen version 0.2 and version 0.4	23
2.5.4.1 Architectural Evolution	23
2.5.4.2 Performance and Scalability	23
2.5.5 Evaluation of AutoGen in Multi-Agent Scenarios	24
2.5.5.1 AutoGen Examples	25
2.5.5.2 Use Cases and Multi Agent Applications	26
2.5.6 LangGraph: Origin and Structure	27
2.5.6.1 LangGraph Examples	29
2.5.7 Comparison: AutoGen vs LangGraph	29
3 Methodology and Application Development	30
3.1 Technologies Used	30
3.1.1 Python as the Core Programming Language	31
3.1.2 Streamlit for Building the User Interface	31
3.1.3 LM Studio as the LLM Backend	32
3.1.4 PyCharm as the Integrated Development Environment	34
3.1.5 System Overview and Execution Flow	34

CONTENTS

3.2	Project Structure	35
3.3	Frontend Development	37
3.3.1	Interface Design	37
3.3.2	Name and Branding	38
3.3.3	Implemented Pages and Their Functionalities	39
3.4	Architecture of the GenAI Agents	44
3.4.1	Multiagent Design Implementation	44
3.4.2	Agent Workflow and Communication Flow	45
3.4.3	Integration with the User Interface	46
4	Results and System Outputs	49
4.1	Theoretical Explanation Page	49
4.2	Practical Help Page	52
4.2.1	Calculus Usage	52
4.2.2	Database Usage	54
4.3	GenAI Quiz Page	55
4.4	RAG Assistant Page	57
4.5	Generative Chat Page	58
5	Conclusions, Limitations, and Future Work	61
5.1	Conclusions	61
5.2	Future Work	62
	Annex	64

List of Figures

2.1	John McCarthy [3]	6
2.2	Organization of a Perceptron [7]	7
2.3	Agent customization and communication patterns [13]	9
2.4	AI Agent Architecture [18]	11
2.5	Khanmigo Website Overview [22]	13
2.6	RAG Flow [24]	14
2.7	GenAI as Tutor [27]	16
2.8	Performance comparison across different methods in complex problem-solving tasks [13].	21
2.9	Agent Customization in Autogen [34]	22
2.10	Magentic Benchmark [42]	25
2.11	Magentic Example performing Complex Task [44]	27
3.1	Python Logo [46]	31
3.2	Streamlit Logo [47]	32
3.3	LM Studio Logo [48]	32
3.4	LM Studio Interface [48]	33
3.5	PyCharm Logo [49]	34
3.6	Project High-Level Flow	34
3.7	Project Structure Tree	35
3.8	Phronesis AI Logo	38
3.9	Phronesis AI Introduction Page - Requesting Data	39
3.10	Introduction Page	39
3.11	Theoretical Page	40
3.12	Practical Page	40
3.13	Quiz Page	41
3.14	RAG Page	41
3.15	Conversation Page	42
3.16	Documentation Page	42
3.17	Profile Page	43
4.1	Streamlit Page Question 1	50
4.2	Streamlit Page Solution 1	50
4.3	Streamlit Page Solution 1 Terminal	51
4.4	Streamlit Page Solution 1 LM Studio Log	51
4.5	Streamlit Page Solution 2	52
4.6	Streamlit Page Solution 2 Terminal	53
4.7	Streamlit Page Solution 2 LM Studio Log	53
4.8	Streamlit Page Solution 3	54

LIST OF FIGURES

4.9 Streamlit Page Solution 3 Terminal	54
4.10 Streamlit Page Solution 3 LM Studio Log	55
4.11 Streamlit Page Solution 4	56
4.12 Streamlit Page Solution 4	56
4.13 Streamlit Page Solution 4	57
4.14 Streamlit Page Solution 5	57
4.15 Streamlit Page Solution 5 Terminal	58
4.16 Streamlit Page Solution 6 Part 1	58
4.17 Streamlit Page Solution 6 Part 2	59
4.18 Streamlit Page Solution 6 Part 3	60

List of Listings

1	Example configuration in <code>common_constants.py</code>	64
2	Prompt definition for <code>QuestionRewriterAgent</code>	65
3	Agent instantiation for <code>QuestionRewriterAgent</code>	66
4	<code>create_assistant_agent</code> function definition	66
5	<code>PythonCodeExecutor</code> agent instantiation	66
6	Retriever agent instantiation	67
7	<code>create_retrieve_user_proxy_agent</code> function definition	68
8	Creating a group chat with two agents	68
9	Initial setup and page loading	69
10	Main pages in the frontend configuration	69
11	Introduction page logic	70
12	Constructor of the <code>Profile</code> class	70
13	Loading the user profile	70
14	Displaying the user profile	71
15	Displaying quiz performance overview	71
16	Tab definition in the AI Tutor page	71
17	LLM explanation interaction	72
18	Asynchronous explanation chat using subject-specific agents	72
19	Subject and difficulty selection interface	73
20	Starting the correction phase	73
21	Updating grade files after correction	74
22	File uploader for subject-specific document	74
23	Initialization of agent and session state	74
24	Reset conversation state	74
25	Handling chat input and responses	75
26	Displaying the TFM documentation	75

Chapter 1

Introduction

Education has been a constant throughout human history, serving as the basis upon which civilizations have advanced, cultures have been preserved, and knowledge has been transmitted across generations. From oral storytelling to printed books, and now to digital platforms, the methods and tools for teaching and learning have evolved together with technological progress. In the present era, we are experiencing one of the most significant educational transformations to date, powered by the rapid advancement of Artificial Intelligence (AI).

Artificial Intelligence, particularly in its generative form, is reshaping the way we access, process, and interact with information. Generative AI systems are capable of producing coherent text, images, code, and even audio, and have opened new possibilities for how knowledge is created and delivered. Among the most impactful developments are Large Language Models (LLMs), such as OpenAI's ChatGPT, which have demonstrated the ability to participate in natural dialogue, solve complex problems, and generate educational content designed to the needs of individual learners.

Generative AI (GenAI) has the potential to become an ally for both students and educators. In traditional settings, topics that are abstract, technical, or cognitively demanding, such as advanced mathematics, theoretical physics, or computer programming, can be particularly challenging to teach and to understand. AI-powered tools can help overpass this gap by offering personalized explanations, dynamic examples, and real-time feedback. These systems can rephrase difficult concepts, simulate real-world applications, or generate interactive exercises, thereby transforming passive learning into a more active, guided experience.

For teachers, GenAI provides support in curriculum design, assessment creation, and content differentiation. It can assist in building custom lesson plans, generating quizzes aligned with specific learning outcomes, or even proposing scaffolding strategies for students who need additional reinforcement. Rather than replacing educators, these technologies act as amplifiers of pedagogical capacity, freeing up time and enabling deeper, more meaningful interactions between instructors and learners.

The impact of tools like ChatGPT extends beyond individual classrooms. Educational institutions and platforms around the world are exploring the integration of AI to create adaptive learning environments that respond in real time to students' needs. Furthermore, as students increasingly rely on digital tools to supplement their studies, the importance of embedding AI responsibly within educational ecosystems becomes critical. The objective is not only to enhance learning efficiency but also to ensure that these systems support cognitive development, curiosity, and critical thinking.

This Master's thesis explores the role of intelligent agents, autonomous AI systems designed to act, reason, and interact, in the context of education. Powered by LLMs and augmented by external knowledge retrieval mechanisms, these agents can perform a wide range of educational tasks: answering questions, guiding through problem-solving steps, simulating expert behavior, and more. Specifically, we examine how frameworks like AutoGen and LangGraph can be leveraged to construct multi-agent ecosystems capable of delivering interactive, modular, and scalable educational experiences.

In an era where technological literacy and adaptability are essential, the integration of generative AI into educational platforms offers a promising direction. By combining pedagogical principles with the capabilities of AI, we can create learning environments that are not only effective and accessible but also aligned with the demands of contemporary society. This project lays the theoretical and architectural foundations for such environments, demonstrating how next-generation AI agents can support both teaching and learning in meaningful, personalized, and innovative ways.

1.1 Motivation and Objectives

1.1.1 Motivation

The motivation behind this Master's Thesis (TFM) appears from the growing need to understand and effectively manage GenAI, particularly in the context of its rapid integration into modern society and, more specifically, into educational environments. The recent emergence of advanced language models, such as ChatGPT, has revolutionized the way students and educators learn and educate.

As AI becomes increasingly embedded in educational workflows, it is essential to explore not only its technical foundations but also the pedagogical, ethical, and moral dimensions of its use. In academic settings, understanding the responsible and constructive application of GenAI is crucial, not only to uphold the values and standards of higher education institutions, but also to foster personal and intellectual growth among students. Properly guiding the use of such tools is essential for cultivating a learning environment where AI serves as a partner in learning rather than a shortcut to circumvent effort or critical thinking.

This project is motivated by the opportunity to bridge the gap between the knowledge of AI and its practical implementation in educational contexts. It seeks to provide a comprehensive understanding of how generative models and agent-based systems can be designed and deployed to enhance learning, support teaching, and address complex academic topics in a more interactive and accessible manner.

1.1.2 Objectives Definition

This Master's Thesis aims to conduct an in-depth investigation into the current landscape of Artificial Intelligence, with a special focus on the integration of LLMs and GenAI agents within educational applications. The main objective is to explore, evaluate, and propose a framework that allows the development of interactive web-based educational platforms powered by advanced AI agents.

To achieve this goal, the following key research areas will be addressed:

- An introduction to Artificial Intelligence, covering its historical development and major paradigm shifts, from symbolic systems to neural networks and deep learning.
- A comprehensive analysis of Large Language Models (LLMs), including their architecture, training processes, and capabilities.
- An in-depth study of intelligent agents: their roles, types, configuration mechanisms, and their interaction models.
- A detailed explanation of the Retrieval-Augmented Generation (RAG) innovative approach, emphasizing its advantages in improving factual accuracy and enabling grounded responses in LLMs.
- An exploration of two cutting-edge agent orchestration frameworks: AutoGen and LangGraph, analyzing their structure, functionality, practical applications, and potential benefits in the context of AI-powered educational systems.

1.1. Motivation and Objectives

In order to fulfill the overarching objective of this project, the following sub-objectives have been defined to guide the research process:

- Conducting an initial review to become familiar with the core concepts and establish a strong theoretical foundation.
- Performing a detailed evaluation of the most recent frameworks and tools for multi-agent orchestration in conjunction with LLMs.
- Comparing different AI models in terms of performance, coherence, and vulnerability to hallucinations, with a focus on their applicability to educational contexts.
- Developing a prototype web application that utilizes agent-based architectures to tackle real academic challenges from university-level subjects, based on the insights and data obtained throughout the research.

Chapter 2

State of the Art

This section will present an analysis of the evolution of Artificial Intelligence, leading up to its application in the field of education. Although various introductory aspects are addressed in Chapter 1, this chapter emphasizes the study of Generative AI within educational settings. Specifically, it highlights the use of framework-based architectures, without exploring the full range of agent configurations.

To this end, the chapter is organized in three main sections. First, a historical overview of Artificial Intelligence is presented, covering its origins, core paradigms, and major breakthroughs, including symbolic systems, connectionism, deep learning, and neuro-symbolic approaches.

The second section examines the rise of multi-agent systems, emphasizing their recent integration with LLMs and their potential to coordinate tasks through natural language interactions. Particular attention is given to the benefits of modularity, role specialization, and conversational coordination in complex problem-solving environments.

The final section discusses the current state of Generative AI, with a specific focus on agent-based orchestration frameworks such as AutoGen and LangGraph, and their implications for educational applications. This includes their use in personalized learning, content generation, and intelligent tutoring systems.

Together, these sections establish a foundation for understanding how the evolution of AI has enabled new forms of interactive, autonomous, and adaptive educational technologies.

2.1 Historical Evolution of Artificial Intelligence

Artificial Intelligence (AI) has experienced several significant transformations since its conceptual and technical origins in the mid-20th century. Although the field was officially named during the seminal Dartmouth Workshop held in 1956, where the term *Artificial Intelligence* was first introduced by the computer scientist John McCarthy [1], its roots can be traced back to earlier work in connectionism.

In particular, the foundational model proposed by Warren McCulloch and Walter Pitts in 1943 introduced a logical calculus for neural activity, laying the groundwork for what would later become known as the connectionist paradigm [2]. This model represented one of the earliest formal efforts to simulate cognitive processes through computational mechanisms inspired by the brain

In addition to the formal neuron introduced by McCulloch and Pitts, other foundational concepts significantly influenced the early development of AI. Kenneth Craik's interpretation of human cognition in terms of symbolic, modular representations and rule-based reasoning provided the groundwork for what would later be called the symbolic paradigm.

In parallel, the cybernetic school led by researchers like Rosenblueth, Wiener, and Bigelow introduced the situated paradigm, emphasizing feedback systems as a model for intelligent behavior. Additionally, Alan Turing's 1950 proposal of an operational test for machine intelligence, now known as the Turing Test, further solidified the theoretical basis for symbolic AI [2].

During its initial phases, AI research primarily focused on what is now termed symbolic Artificial Intelligence. This approach was based on the manipulation of symbols and heavily depended on explicitly defined rules and formal logical systems to replicate aspects of human reasoning.



Figure 2.1: John McCarthy [3]

Between the 1960s to the 1980s, researchers who advocated for symbolic approaches, commonly referred to as symbolists [4], made significant advancements by developing expert systems. These systems encoded human knowledge using structured rule-based mechanisms, often formulated as if-then statements.

State of the Art

A prominent example from this period was the MYCIN system, which gained recognition for its success in medical diagnosis. MYCIN provided recommendations based on input symptoms and laboratory test results [1]. Although these systems demonstrated high effectiveness in narrow and specialized areas, they encountered notable limitations when applied to broader and more complex problem domains.

Symbolic systems worked under the assumption that intelligent behavior could arise from a sufficiently extensive and well-organized collection of rules and facts. This approach reached its peak during the 1980s, when expert systems gained significant traction and were successfully applied across various industrial and commercial sectors. Despite these successes, symbolic AI encountered serious limitations, particularly in terms of scalability and flexibility. A key drawback was its inability to adapt or generalize beyond the predefined rule sets, as these systems could not learn directly from raw data inputs..

As problem domains grew in complexity, symbolic approaches became increasingly difficult to maintain and extend. These challenges led to a decline in research funding and public interest in Artificial Intelligence, resulting in a series of downturns known as AI winters [5].

At the same time, researchers began to explore a fundamentally different approach to Artificial Intelligence that diverged from the rule-based symbolic paradigm. This alternative direction was led by the connectionist community, which drew inspiration from the biological structure and functioning of the human brain.

Early efforts in this area led to the development of artificial neural networks capable of learning patterns and making predictions based on empirical data. One of the earliest and most influential models was the Perceptron, created by Frank Rosenblatt in 1958 [6]. The Perceptron, as seen in Figure 2.2, was designed to perform learning tasks by adjusting its internal parameters in response to input examples.

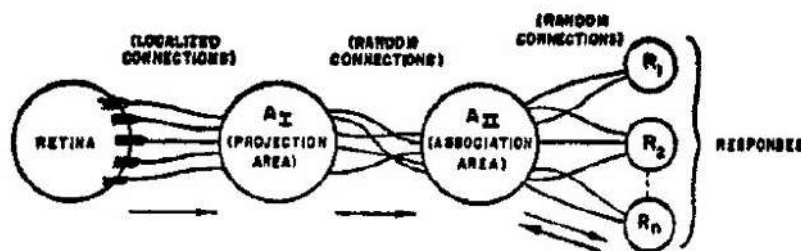


Figure 2.2: Organization of a Perceptron [7]

However, the progress of connectionist models was significantly delayed following an analysis published in 1969 by Marvin Minsky and Seymour Papert [8]. Their work rigorously demonstrated the limitations of single-layer Perceptrons, particularly their inability to solve certain classes of problems, such as the exclusive or (XOR) function. This critique led to a sharp decline in interest and funding for neural network research in the years that followed.

2.2. The Emergence of Multi-Agent Systems in AI

By the 1990s, the development of improved learning algorithms, especially backpropagation, along with greater computational power, led to a renewed interest in neural networks. This revival eventually triggered the deep learning revolution of the 2010s. Deep neural networks, composed of multiple layers of interconnected units, began to outperform traditional methods in a wide variety of tasks [9].

A major milestone came in 2012 when a deep neural network architecture called AlexNet significantly outperformed other models in the ImageNet Large Scale Visual Recognition Challenge, cutting the error rate of the second-best approach by half. This achievement, enabled by advancements in GPU computing and the availability of large-scale datasets, marked the beginning of a new era where data-driven deep learning approaches became the dominant paradigm in both AI research and applications [10].

By the end of the 2010s, deep learning systems had already surpassed human-level performance in areas such as speech recognition, image classification, and strategic games like Go, clearly illustrating the power of learning from large amounts of data [11].

Today's GenAI represents a convergence between symbolic reasoning and the statistical learning capabilities of deep neural networks. Some modern systems have started to adopt neuro-symbolic approaches, which aim to integrate the structured logic of symbolic AI with the flexibility and learning capacity of neural models. Although these developments are promising, there are still significant challenges in the field, particularly concerning transparency, bias, and the long-term goal of achieving artificial general intelligence.

The progression from rule-based expert systems to deep neural networks, and now to large-scale generative models, illustrates a clear and continuous evolution toward more adaptive and powerful forms of Artificial Intelligence.

2.2 The Emergence of Multi-Agent Systems in AI

A major recent advancement in Artificial Intelligence architecture is the transition from using single, monolithic models to adopting collaborative multi-agent systems [12]. In traditional settings, an AI application would typically rely on a single core model, such as an individual neural network, to perform a specific task. However, as AI systems are increasingly expected to manage complex and open-ended problems, researchers have identified clear benefits in using multiple AI agents that are capable of interacting, collaborating, and specializing in different subtasks.

This approach builds on earlier work in distributed Artificial Intelligence and multi-agent systems, which has now been considerably enhanced by the power of Large Language Models. The emergence of LLMs has made it possible for agents to communicate through natural language, enabling them to engage in meaningful dialogue to coordinate actions and solve problems effectively.

Instead of depending on one model to carry out all components of a task, this approach assigns specific responsibilities to separate agents, such as planning, solving, evaluating, or providing feedback. This results in a more modular, flexible, and adaptive system architecture.

Recent research [13] suggests that multi-agent configurations can surpass the performance of single-model systems in certain scenarios by leveraging a diversity of perspectives and cooperative problem-solving strategies. For example, agents can participate in brainstorming sessions or structured discussions, promoting divergent thinking and reducing cognitive blind spots. One agent might act as a fact checker or reasoning validator, reviewing the output of another agent, while another could serve as an ethical filter to ensure content safety and alignment with human values.

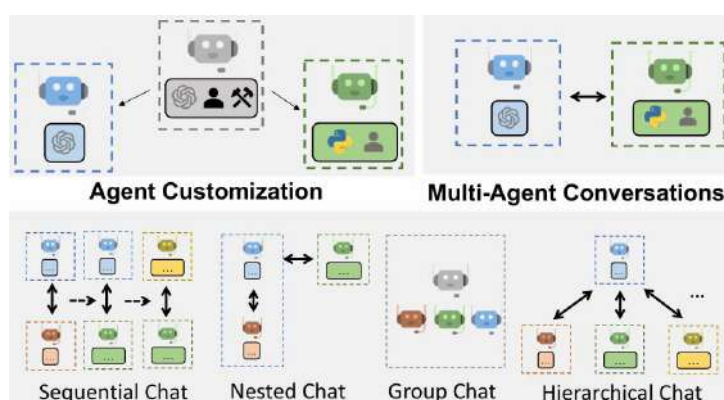


Figure 2.3: Agent customization and communication patterns [13]

2.3 GenAI: LLMs, Agents, and Useful Systems

The recent emergence of Generative Artificial Intelligence (GenAI) has revolutionized the field of intelligent systems by enabling machines to autonomously generate coherent, context-aware content across various modalities. This section examines the core components that underpin modern GenAI systems, beginning with the architecture and capabilities of Large Language Models (LLMs) and then moving toward multi-agent frameworks and real-world applications.

2.3.1 Large Language Models (LLMs)

Large language models (LLMs) are deep neural network architectures distinguished by their extremely large number of parameters, which often range from hundreds of millions to several billions. These models are typically built on the transformer architecture. They are trained on extensive textual corpora using self-supervised learning objectives, such as predicting the next word in a sentence. This training process allows them to acquire complex linguistic patterns and accumulate a broad base of factual knowledge from data.

2.3. GenAI: LLMs, Agents, and Useful Systems

After the pre-training phase, LLMs can be adapted or fine-tuned for specific applications or conversational formats. This adaptation process commonly includes techniques such as reinforcement learning from human feedback, which helps shape the model's behavior to follow instructions or interact more effectively in dialogue-based settings.

Due to their scalable architecture and exposure to large, diverse datasets, modern models like GPT-4 developed by OpenAI [14] and LLaMA developed by Meta [15] exhibit remarkable capabilities in understanding context and generating human-like language.

These models consistently achieve strong performance across various natural language processing tasks, including text generation, summarization, comprehension, and question answering. One of the key strengths of Large Language Models (LLMs) is their generalization ability and few-shot learning performance. For instance, GPT-3 has demonstrated the capability to perform entirely new tasks with only a few examples. This versatility has made LLMs highly valuable across multiple domains, including education.

In educational settings, LLMs have demonstrated potential as tools that support both students and educators. Initial implementations suggest that these models can be employed to automatically generate instructional content, such as quiz questions or explanatory hints, as well as to provide formative feedback on open-ended student responses.

For example, one study [16] employed an LLM to automate question generation within an educational context and found that the model was capable of producing a wide variety of diverse and pedagogically meaningful questions. Additionally, systems powered by LLMs have been explored for tasks such as student knowledge tracing and cognitive tutoring.

A major advantage of LLMs is their ability to generalize knowledge across different domains. With appropriate prompting or fine-tuning, these models can adapt to various subjects, including mathematics and science, making them well-suited for personalized learning applications.

2.3.2 AI Agents

AI agents built around LLMs are autonomous systems that utilize the language model as a central processing unit to interpret instructions, make decisions, and execute actions aimed at achieving specific goals [17].

In contrast to basic conversational bots that merely generate responses to user inputs, these agents are enhanced with additional functionalities such as tool integration, memory systems, and planning capabilities.

State of the Art

These components allow the agent to interact with its environment and engage in complex, multi-step reasoning, as seen in Figure 2.8.

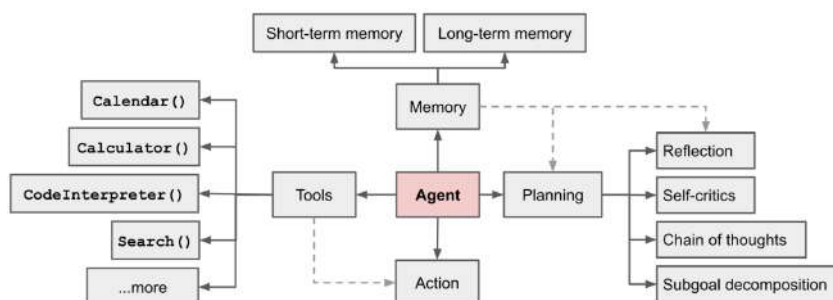


Figure 2.4: AI Agent Architecture [18]

In practical applications, an AI agent receives an input, which may come in the form of a user question or environmental data. The agent then employs the language model to determine an appropriate course of action and responds accordingly. This response could involve producing an answer, invoking an external tool or application programming interface, asking clarifying questions, or continuing with further steps.

This creates a continuous feedback loop in which the agent autonomously reasons and acts [18]. The ability to make independent decisions and execute actions enables such agents to handle complex tasks, such as diagnosing technical issues or guiding a student through a multi-step learning process, with minimal human intervention.

To operate effectively, language model-based agents typically include three essential components in addition to the core language model:

- **Planning:** The agent relies on the large language model for reasoning and determining its next actions. Contemporary language models are capable of breaking down complex tasks into manageable subcomponents through techniques such as chain of thought prompting. They can also reflect on their progress and revise their strategies accordingly. This planning functionality serves as the central mechanism for problem-solving and goal-oriented behavior within the agent [19].
- **Tool Utilization:** A language model-based agent enhances its capabilities by interacting with external tools or application programming interfaces, including databases, web search engines, calculators, or environments for executing code.

By generating properly formatted outputs or invoking specific interfaces, the agent can access information or perform operations that extend beyond the inherent capabilities of the language model. Effective agents are able to decide when and how to use these tools as part of their overall strategy [19].

2.3. GenAI: LLMs, Agents, and Useful Systems

- **Memory:** To maintain coherent and context-aware behavior, the agent incorporates memory mechanisms that store both short-term and long-term information. Short-term memory allows the agent to track recent dialogue and intermediate reasoning steps, while long-term memory enables the retention of knowledge across multiple sessions. Technologies such as vector databases for semantic retrieval or knowledge graphs are commonly employed to support this functionality [20].

Prompt engineering plays a crucial role in configuring artificial intelligence agents. Developers design prompts that define the agent's role, objectives, and interaction style. For instance, an agent might receive a system prompt such as you are a friendly mathematics tutor who helps students solve problems step by step, along with a few example interactions to illustrate the expected behavior.

Well-crafted prompts help ensure that the language model driving the agent remains focused and generates outputs that are coherent and interpretable by the agent itself or by other collaborating agents [20]. To facilitate reuse and consistency, developers often employ prompt models or prompt templates that encapsulate these instructions. These templates can be applied across multiple agents assigned to similar roles.

Moreover, agents actively manage their context windows by trimming or summarizing previous dialogue as necessary to stay within the language model's token limitations. In cases where long-term continuity is required, agents may also access external memory sources, as described earlier. All of these strategies are part of a broader discipline known as prompt and context management, which is essential for maintaining consistent and effective agent behavior throughout extended interactions.

AI agents powered by LLMs are increasingly being investigated for their potential as intelligent tutors, teaching assistants, and tools for student support. A well-designed agent is capable of engaging in natural language dialogue with learners, responding to questions, offering hints, and adapting its feedback to the specific needs of each student over time. In this way, the agent functions as a personalized tutor, delivering guidance that evolves with the learner's progress [21].

One of the most important advantages of educational agents based on LLMs is their ability to adjust explanations according to different learning styles and individual pacing. These agents can provide tailored support at any time, offering instant clarification and dynamically adapting the level of detail or tone based on the learner's input.

The interactive nature of such systems also contributes to more engaging learning experiences. Agents can simulate conversations, present interactive examples, or role-play instructional scenarios to deepen comprehension. An additional advantage is scalability: a single agent-based system can support thousands of students simultaneously, making it especially valuable for delivering personalized education at scale.

Beyond tutoring, AI agents can also assist educators and educational content creators. For example, an agent may take on the role of a teaching assistant, supporting instructors by generating lesson plans, quizzes, or summaries of course material.

State of the Art

A notable example is Khanmigo [22], developed by Khan Academy as seen in Figure 2.5, serves as a virtual tutor for students and a teaching assistant for educators. It assists in managing administrative tasks while also engaging learners through interactive dialogue. Preliminary evaluations of Khanmigo suggest that it can facilitate Socratic-style conversations, lead classroom debates, and encourage critical thinking, all while reducing the time teachers spend on repetitive or routine duties.

These emerging applications highlight the transformative potential of artificial intelligence agents in educational contexts. By combining the conversational capabilities of LLMs with curriculum integration, assessment tools, and memory systems for tracking student progress, generative artificial intelligence agents can deliver highly personalized and interactive learning experiences. Simultaneously, they provide substantial support to educators by reducing workload and enhancing instructional efficiency.

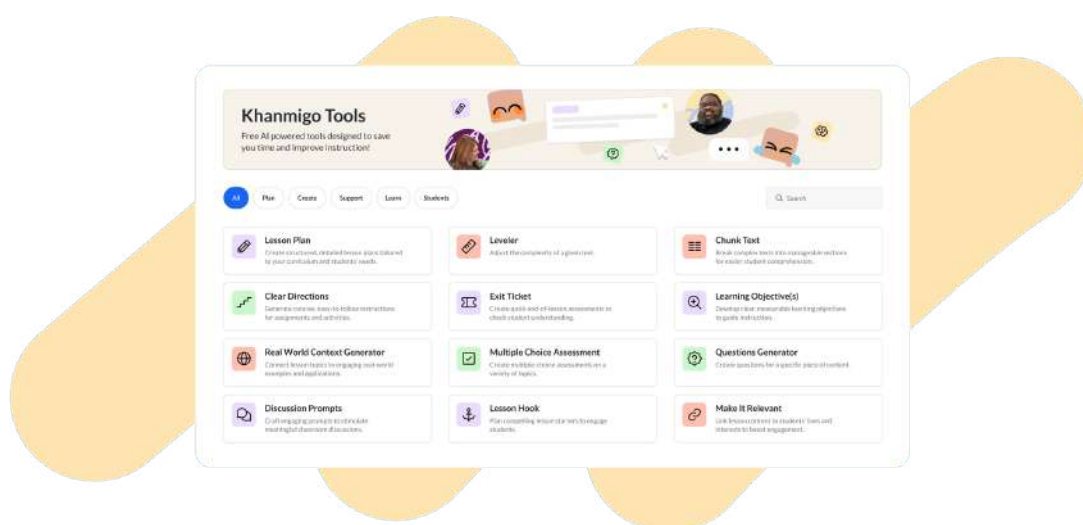


Figure 2.5: Khanmigo Website Overview [22]

2.3.3 Retrieval-Augmented Generation (RAG)

While LLMs offer remarkable capabilities, they also have inherent limitations. One key constraint is that their knowledge is limited to the data available during their training period, which results in knowledge cutoffs. Furthermore, these models may occasionally generate outputs that are factually incorrect or unsupported by evidence, a phenomenon commonly referred to as hallucinations [23].

Retrieval-Augmented Generation (RAG), as seen in Figure 2.6, is a technique designed to overcome these limitations by grounding the outputs of language models in external knowledge sources. This approach combines a language model with a retrieval mechanism capable of accessing relevant information from databases, document collections, or structured knowledge repositories.

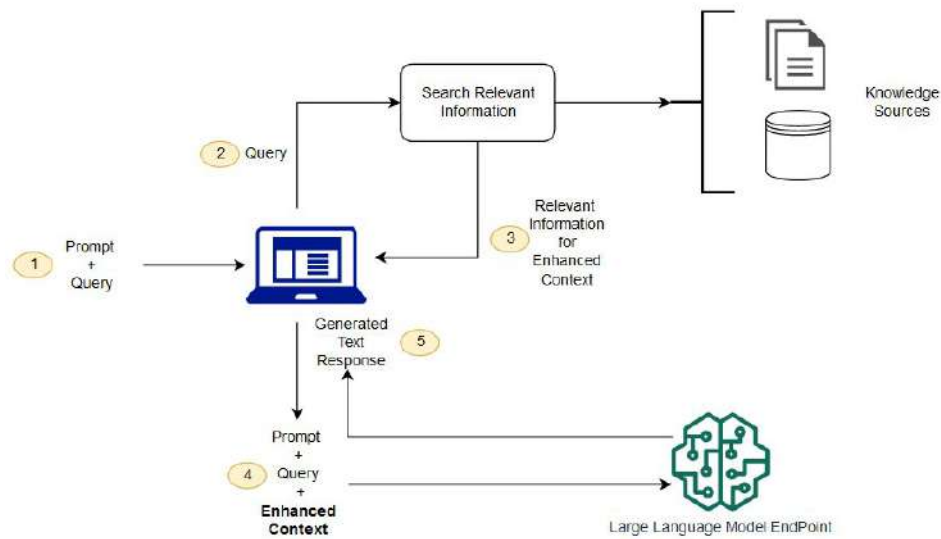


Figure 2.6: RAG Flow [24]

This enables the model to generate responses that are informed not only by its pre-trained internal representations but also by current and contextually appropriate information. A standard Retrieval-Augmented Generation pipeline for question answering typically involves the following three core steps [25]:

- **Indexing:** An external knowledge repository, such as textbooks, lecture notes, or encyclopedic sources, is pre-processed into a searchable index. The documents are segmented into smaller chunks and converted into vector embeddings, allowing for efficient semantic search. This index serves as a flexible and non-parametric form of memory that can be updated or expanded as needed.
- **Retrieval:** When a user submits a query or question, the system consults the index to retrieve the most semantically relevant text passages. In an educational context, for instance, a student query may result in the retrieval of the top-k most relevant excerpts from course materials or reference documents.
- **Generation:** The retrieved context, consisting of the selected documents or text fragments, is then passed to the language model along with the original query. The model generates a response that incorporates both its pre-trained knowledge and the supplementary information provided by the retrieved context. This process effectively augments the model's prompt with real-time, external information.

By combining retrieval with generation, the RAG approach ensures that the outputs produced by the language model are grounded in explicit evidence. This yields two major practical benefits:

- The model gains access to the most up-to-date and reliable facts, addressing the limitation of fixed knowledge due to training data cutoffs.
- The model's responses can be traced back to specific source documents, enabling users, such as students, to verify the accuracy and origin of the information provided.

These advantages are especially important in educational contexts. A generative artificial intelligence tutor should provide responses that are both accurate and verifiable. For instance, when responding to a complex question, the agent should be capable of referencing a specific textbook chapter or academic publication.

Retrieval-Augmented Generation supports this behavior by allowing the agent to retrieve relevant excerpts from authoritative documents and incorporate them into its responses, often including citations or direct quotations. This not only enhances the factual reliability of the output but also fosters user trust by making the origin of the information transparent [23].

Another major benefit of RAG is its ability to reduce hallucinations and biases, which are common issues in the outputs of LLMs. By grounding responses in retrieved facts, the model is less likely to rely solely on internal representations that may be incomplete or inaccurate. Studies have shown that retrieval-augmented systems generally produce answers that are more factual, precise, and contextually appropriate compared to language models operating without retrieval support [26].

Additionally, the use of external knowledge sources allows the system to be dynamically updated by incorporating new documents or datasets into the retrieval index. This approach eliminates the need to retrain the language model itself and provides a cost-effective way to keep the system aligned with the most current curricular materials or scientific developments. Such adaptability is particularly valuable in rapidly evolving subject areas or in contexts influenced by recent events.

For instance, if a course syllabus is revised or new research findings are published, a retrieval-augmented educational agent can immediately integrate this updated information through its retrieval component. In contrast, a standalone language model would continue relying on outdated training data until a new training cycle is performed.

Because of these advantages, RAG has become a foundational technique for context-aware question answering in education and other domains that require access to accurate and timely information. Many generative artificial intelligence-powered study tools and tutoring platforms use the RAG approach, leveraging domain-specific repositories such as course slides, lecture notes, and institutional resources to ground their responses. This ensures that the information provided to students is not only factually accurate but also closely aligned with the educational content they are expected to learn.

2.4 Generative AI Agent Architectures in Education

In educational environments, GenAI Agents offer considerable potential for personalized tutoring, automated content creation, and interactive simulations of instructional scenarios. One of the most impactful applications of generative artificial intelligence in education is its use as a personal tutor. Tutoring systems powered by artificial intelligence can deliver on-demand, individualized instruction that is tailored to the learner's pace, needs, and level of comprehension.

2.4. Generative AI Agent Architectures in Education

Initial research supports the effectiveness of these systems in real-world educational settings. A study supported by the World Bank in Edo State, Nigeria, implemented an after-school AI Tutor based on GPT4 to assist students in learning the English language [27].

The results were remarkable as seen in Figure 2.7. After only six weeks, students who participated in the program showed an average improvement of approximately 0.3 standard deviations, a gain that corresponds to nearly two years of expected learning progress. These students significantly outperformed their peers who did not have access to the artificial intelligence tutor, not only in English but also in other subjects assessed during end-of-term examinations.

These results offer compelling evidence that GenAI, when implemented thoughtfully and in collaboration with educators, can serve as an effective and scalable virtual tutor. Such systems not only provide academic support but also contribute to promoting educational equity. These findings suggest that the development of learning to learn skills through guided interactions with AI can produce broad and meaningful educational benefits.

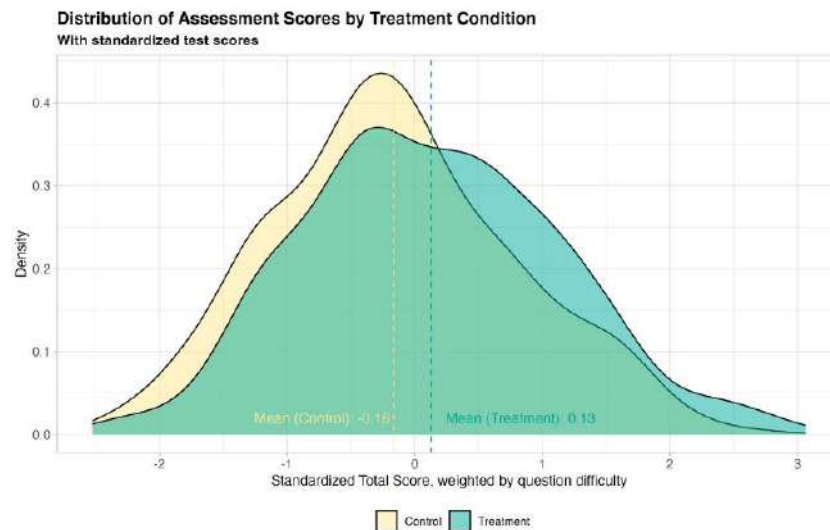


Figure 2.7: GenAI as Tutor [27]

Importantly, the AI tutor demonstrated benefits for a wide range of students, including those who had previously experienced learning difficulties. Moreover, the program contributed to reducing gender disparities, with girls showing particularly strong improvements in this pilot study.

The potential of GenAI in tutoring is not limited to large-scale implementations. Smaller studies have also demonstrated its effectiveness. For instance, researchers have explored the use of ChatGPT as a tutor in specialized academic domains. In one case study conducted in a university-level programming course, students used the system to debug code and understand algorithmic concepts. The findings showed that the artificial intelligence tutor successfully assisted learners in solving programming tasks and developing procedural knowledge [28].

GenAI tutors have demonstrated versatility across a wide range of academic subjects, from foundational reading skills at the elementary level to advanced topics in graduate-level engineering. One of their most notable strengths lies in their capacity for personalization. Because generative models can tailor explanations to align with a student's cognitive level and preferred learning modality, they are able to deliver individualized learning experiences at scale.

For example, an artificial intelligence tutor may recognize that a particular learner responds more effectively to visual information and begin integrating diagrams or images into its explanations. Conversely, for auditory learners, it may provide spoken responses or narrated summaries to enhance understanding.

Research in adaptive learning supported by Artificial Intelligence suggests that generative models can modify instructional content to accommodate various learning preferences, including audio narration, visual aids, and step-by-step textual explanations.

These systems can also dynamically adjust the pacing of instruction. Advanced students may receive more challenging content to accelerate their progress, while those needing additional support are guided through the material at a more gradual pace [29]. This level of adaptivity, once thought achievable only through individualized human tutoring, is now being scaled and delivered through GenAI technologies.

2.4.1 Homework and Assignment Assistance

Academic research is increasingly investigating the effectiveness of replacing traditional homework practices with artificial intelligence-assisted alternatives. A recent randomized controlled trial conducted at the secondary school level examined the impact of substituting conventional homework assignments with interactive tutoring sessions powered by GPT-4 for students learning English grammar [30].

In this experimental setup, the artificial intelligence system guided students through grammar exercises, provided immediate feedback, and posed follow-up questions, effectively simulating the experience of one-on-one tutoring during independent study. The results were promising.

Students in the AI-assisted group demonstrated significantly greater improvements in grammar proficiency compared to their peers who completed traditional homework tasks. Additionally, participants reported higher levels of engagement and motivation throughout the learning process.

Student satisfaction with the AI system was also notably high. Many students expressed a preference for continuing to use the homework assistant beyond the duration of the study [30]. These findings indicate that AI systems have the potential to enhance the effectiveness of homework by providing immediate feedback and adaptive questioning capabilities that are often challenging for human instructors to offer outside regular classroom hours.

2.5. Agent Architectures: AutoGen and LangGraph

Even for more open-ended assignments, such as essays or creative projects, generative artificial intelligence can provide valuable support. It can help students brainstorm ideas, outline responses, or clarify task requirements. This type of assistance can reduce frustration, which is a common cause of procrastination and disengagement. By offering initial structure or encouraging deeper reflection, generative tools can help students overcome writer's block and maintain motivation throughout the task.

Nevertheless, educators have expressed valid concerns regarding the ethical use of artificial intelligence in academic settings. While these tools can support learning and personal development, they must not be misused as a means to complete assignments without true understanding. In response to these concerns, some academic institutions have introduced honor code guidelines that clearly define acceptable uses of artificial intelligence in coursework.

These policies often require students to disclose and cite any assistance received from artificial intelligence systems. In addition, they typically prohibit the use of generative models to produce complete solutions without active student engagement or understanding [28].

GenAI also opens the door to entirely new forms of homework. Rather than assigning traditional take-home problem sets or essay questions, educators can now design assignments that involve direct interaction with artificial intelligence systems. In these tasks, students are expected to engage in meaningful dialogue with the model on a specific academic topic.

For instance, a history teacher might instruct students to interact with an AI System simulating George Washington and ask questions about his presidency, followed by a reflection on the insights gained through the exchange. Assignments of this nature were previously unattainable but are now possible with the capabilities of modern large language models. These tasks promote inquiry-based learning and require students to synthesize information by critically evaluating the responses provided by the model, interpreting them thoughtfully, and producing reflective outputs based on their understanding.

2.5 Agent Architectures: AutoGen and LangGraph

Building advanced generative artificial intelligence agents often requires the coordination tools, and interactions within a structured system. In response to this need, specialized agent architectures and development frameworks have recently been introduced to streamline the process.

Two notable examples are AutoGen [31], developed by Microsoft, and LangGraph [20], a framework within the LangChain ecosystem. Both are designed to support the development of complex multi-agent systems, although they adopt different design principles and implementation strategies.

2.5.1 Introduction and Motivation for AutoGen

AutoGen [31] is an open source framework developed by Microsoft that simplifies the creation of applications using LLMs through conversational multi agent systems. It was introduced in late 2023 in response to research showing that coordinating multiple cooperating agents can enhance divergent thinking in LLMs, improve factual accuracy and reasoning, and help control errors.

By breaking down complex problems into specialized tasks, each managed by different agents, developers can take advantage of the strengths of various models or tools, avoiding the limitations of any single agent. The framework offers a well-defined infrastructure in which agents can send and receive messages while also being fully customizable in terms of behavior, objectives, and operational roles.

AutoGen provides a flexible system for multi agent conversations. Its main goal is to offer a simple way to manage interactions between agents, so developers can set up cooperative behaviors without writing complex coordination code.

AutoGen has been widely used in areas like programming help and supply chain optimization. It even won the Best Paper award at the LLM Agents workshop at ICLR 2024, which shows its importance in the field of LLM based agent systems [32].

The central design objective of AutoGen is to offer a general-purpose, conversation-oriented infrastructure for coordinating agent behavior. Rather than requiring developers to script detailed sequences of operations, the framework allows them to define agents with distinct roles and capabilities, and then enable these agents to interact autonomously to achieve task objectives.

AutoGen manages the orchestration of agent interactions by enforcing conversation rules, which can specify, for example, which agent should initiate communication or when a session should end. These rules can be implemented using either code or natural language.

This design paradigm, known as conversation programming, simplifies the development of multi-agent systems by abstracting away much of the underlying complexity. Developers can decompose sophisticated workflows into a series of dialogues between specialized agents, each utilizing language model reasoning and tool integration as needed.

Additionally, the modularity of AutoGen allows agents to be reused across different contexts. For example, a MathSolver agent and a Critique agent may collaborate on mathematical problem solving in one scenario and be redeployed for code review tasks in another, demonstrating the flexibility and reusability of the framework.

AutoGen has been tested in a variety of application domains, including mathematics tutoring, code generation, and supply chain optimization. In the context of generative artificial intelligence systems for education, this architecture enables the creation of rich and interactive learning experiences based on dialogue.

2.5. Agent Architectures: AutoGen and LangGraph

For example, developers can construct a system involving a tutor agent and a simulated student agent engaged in a Socratic dialogue, or design a sequence of agents where one is responsible for generating a solution and another for verifying its correctness. This conversation-driven architecture supports dynamic and adaptive learning interactions that would be difficult to implement using a single, monolithic model.

By coordinating multiple specialized agents, AutoGen facilitates the design of flexible educational systems capable of responding to diverse learning needs and instructional goals.

2.5.2 Architectural Overview of AutoGen

AutoGen presents a modular architecture organised around conversational agents, structured conversations and integrated tools. Instead of implementing a hardcoded solution, developers using AutoGen define a group of interacting agents, each of which may be powered by different artificial intelligence models, integrated tools, or even human input.

These agents operate according to a predefined conversation protocol that regulates their interactions. Through this mechanism, agents autonomously exchange messages to collaboratively solve complex problems. The AutoGen framework has been effectively applied in a variety of domains, including mathematical problem solving, software engineering, and supply chain optimization [31].

The central principle of this approach is that breaking down a complex task into a dynamic dialogue among specialized agents offers more flexibility and efficiency than relying on a single, monolithic model that performs tasks in a linear sequence. Conversations between agents allow for modular system design and the distribution of responsibilities.

For instance, one agent may retrieve external information using an application programming interface or a third-party tool, while another interprets the result and draws inferences from it.

The transition toward generative agent frameworks has been driven by practical needs. Despite their impressive capabilities, large standalone models often encounter limitations, including factual inaccuracies, restricted access to real-time information, and difficulties in decomposing complex tasks.

State of the Art

By coordinating multiple agents, these challenges can be addressed more effectively. Tasks can be divided among specialized components, redundancy can be introduced to enhance reliability, and various forms of oversight can be implemented to ensure accurate and consistent performance.

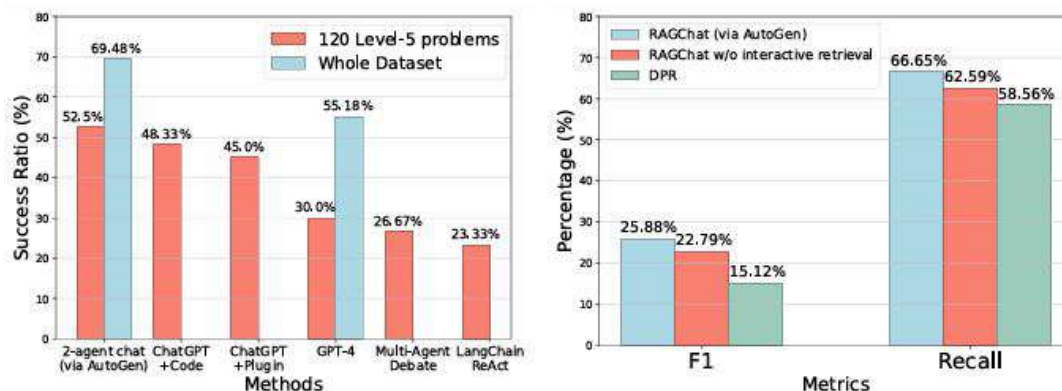


Figure 2.8: Performance comparison across different methods in complex problem-solving tasks [13].

This design naturally supports task decomposition. For example, a manager agent powered by a large language model may assign a specific step to a worker agent and then verify the result before continuing with the next stage [31].

This structure mirrors collaborative workflows in human teams, where tasks are delegated and results are reviewed for accuracy and completeness. With the rise of chat-optimized LLMs such as ChatGPT, communication between agents can now take place through natural language exchanges. This enables a more intuitive and flexible form of interaction compared to conventional software application interfaces, as agents simply exchange prompts and responses to coordinate their actions.

2.5.3 Conversable and Customisable Agents

Each agent is an entity capable of sending and receiving messages to other agents, that is, a participant in a multi user chat. Agents can represent LLMs, humans, software tools, or combinations of these. The framework provides generic base classes, such as `ConversableAgent`, and several predefined agents, of which two are especially important [33]:

- **AssistantAgent**: an autonomous AI assistant, usually powered by an LLM like GPT-4. It can understand incoming requests, generate natural language responses, and propose executable code, such as Python, or as we will do too in our use case, SQL queries with a database. It can run code snippets and read their outputs, making it useful for iterative debugging.
- **UserProxyAgent**: a proxy that acts on behalf of a human user. By default, it asks for human input each turn but can also run code or use tools automatically if instructed by another agent. Automatic execution is sandboxed, for example, using Docker, and can be disabled for safety. Optionally, an LLM can simulate the user.

2.5. Agent Architectures: AutoGen and LangGraph

Developers can create new types of agents or combine capabilities, making agents highly flexible, such as the ones mentioned in Figure 2.9.

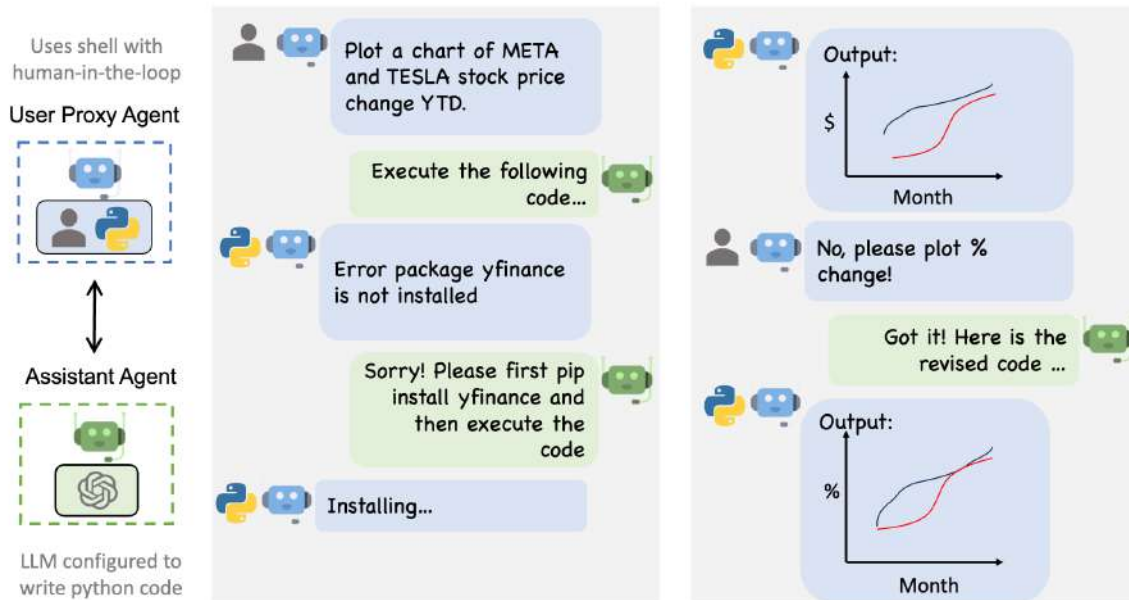


Figure 2.9: Agent Customization in AutoGen [34]

2.5.3.1 Multi-Agent Conversations and Coordination

Interactions in AutoGen are abstracted as structured conversations. Each agent determines its actions based on the messages it receives, and these actions are also communicated as messages directed to other agents. Beyond simple turn-taking dialogues, AutoGen supports flexible interaction patterns, including chained sequences, nested sub-conversations, group chats with multiple participants, and dynamic combinations of these approaches.

These patterns enable workflows such as a coordinator agent that divides a task into subtasks, assigns them to specialized agents, and then combines their results. Termination rules, such as fixed turn limits or a task complete signal, ensure that conversations do not enter infinite loops. We will discuss termination messages and the required architectures in detail later, when we explain how our example is set up. Conversations can operate autonomously or maintain a human in the loop through the UserProxyAgent.

2.5.3.2 Tool Integration and Action Execution

Agents in AutoGen can use external tools, execute code, or interact with environments [35]. For instance, an AssistantAgent can generate a code block, and then another AssistantAgent can detect and execute it, returning the output. AutoGen provides ready-to-use components such as the DockerCommandLineCodeExecutor and a collection of common tools. This design integrates both LLM and tool workflows within the conversational system.

2.5.3.3 Execution Model and Observability

Starting from AutoGen version 0.4, the framework adopts an asynchronous, event-driven actor model. Each agent processes events in a non-blocking manner, enabling parallel execution of independent subtasks. Higher-level orchestration can be achieved through a directed graph abstraction, also known as GraphFlow.

Observability is a built-in feature, allowing interactions to be traced using OpenTelemetry. Developers can monitor and inspect conversations in real time through AutoGen Studio [36].

2.5.4 Key Differences Between AutoGen version 0.2 and version 0.4

2.5.4.1 Architectural Evolution

Version 0.4, as described in the documentation [37], represents a complete redesign of the system. The most significant changes include the adoption of asynchronous, event-driven communication, replacing the previous sequential model used in version 0.2. This shift allows agents to interact simultaneously, making the system faster and more efficient. However, while these improvements greatly enhance performance and flexibility, they also increase the system's complexity. As a result, the learning curve for new developers may be steeper, especially for those unfamiliar with asynchronous programming or multi-agent architectures.

The architecture is now layered, with the Core package providing a basic framework for creating agents. Above this, the AgentChat package offers higher-level tools for managing conversations. The existing components, such as AssistantAgent, User-ProxyAgent, and Team, remain available but are now more robust.

Modularity has been improved through the new `autogen_ext` package, which provides a collection of interchangeable agents, model clients, tools, and executors. These components can be added or removed without affecting the core system.

The platform also supports distributed and multi-language environments. Agents can operate in separate processes or even on other platforms using remote communication technologies.

2.5.4.2 Performance and Scalability

The asynchronous engine in AutoGen version 0.4 significantly reduces the time required to complete tasks because agents can operate in parallel. It also supports scaling across multiple machines. Version 0.4 introduces AutoGen Bench [38], a suite of tools for measuring performance, enabling developers to monitor response times, completion steps, and resource usage.

2.5.5 Evaluation of AutoGen in Multi-Agent Scenarios

Evaluating multi-agent systems that use Large Language Models (LLMs) requires assessing both quality and coordination. Microsoft introduced AutoGen Bench with version 0.4 as a standard method for measuring these aspects. In technical reports, Magentic One was tested on three complex benchmarks known as GAIA, AssistantBench, and WebArena. These benchmarks assess abilities in planning, tool usage, and website interaction.

A single GPT-4 agent performed at approximately seven percent, sixteen percent, and fifteen percent on these benchmarks, while human performance was around ninety-two percent on GAIA and seventy-eight percent on WebArena. Magentic One significantly reduced this gap, achieving results that were among the best in the field.

GAIA (General AI Assistant) evaluates an agent across multiple dimensions, including logical reasoning, web navigation, API manipulation, and approximately 450 questions designed to simulate realistic assistant tasks. AssistantBench focuses on long-horizon web tasks, such as purchasing an item that meets specific criteria or gathering information from multiple pages, testing an agent's persistence and autonomy. WebArena is a simulated web-navigation environment where an agent must interact with pages, forms, and links to accomplish defined goals.

Beyond Magentic-One, the 2023–2025 literature introduced additional frameworks and benchmarks that offer useful comparisons with AutoGen, including MetaGPT [39], ChatDev [40], and CAMEL [41], demonstrate that AutoGen is distinguished by its flexibility, dynamic interaction patterns, support for human supervision, and integration with external tools.

The main metrics used to evaluate these systems include success rate, number of conversation turns, total completion time, and the ability to handle errors during execution. All of these metrics can be tracked using AutoGen's monitoring tools.

AutoGen Bench executes agents in controlled environments and measures their performance on specific tasks, enabling fair comparisons across approaches. It supports reproducible experiments, controlling LLM stochasticity, and component isolation to diagnose weaknesses. Researchers can add new evaluation tasks easily, fostering an ecosystem of benchmarks for agents.

State of the Art

The Magentic-One technical report considers three demanding benchmarks that approximate the workload of a general-purpose assistant: GAIA, AssistantBench, and WebArena. These tests cover multi-step scenarios that require planning, tool use, and handling information in diverse formats, closely mirroring real assistant tasks. The results are revealing: a single GPT-4 model acting alone achieved very low success rates, roughly 7%, 16%, and 15% on the three benchmarks, respectively, whereas human performance is far higher, about 92% on GAIA and 78% on WebArena [42].

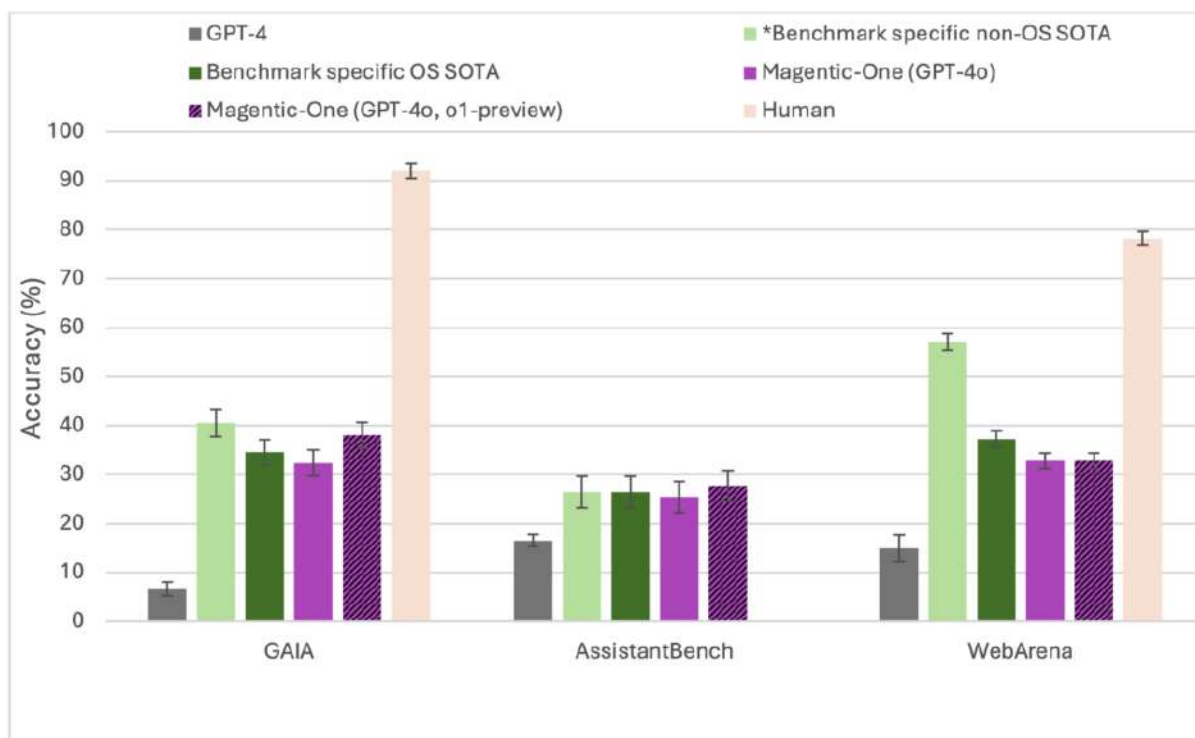


Figure 2.10: Magentic Benchmark [42]

This achievement demonstrates that collaboration among specialized agents can significantly close the gap between a single GPT-4 model and the leading systems, making these highly challenging tasks achievable.

2.5.5.1 AutoGen Examples

AutoGen supports a wide range of use cases, some of which are described below [43]:

- **Code Generation with Retrieval:** Using the specific agent, AutoGen enables code generation from docstrings or user instructions, optionally augmented with human feedback and retrieval from a knowledge base.
- **Multi-Agent Code QA:** A solution-verifier agent loop can be constructed where one agent generates a code solution and another evaluates correctness, mimicking peer-review or educational assessment flows.
- **Retrieval-Augmented Generation (RAG):** Group chats can be configured with retrieval components to enhance factual grounding, ideal for question answering or technical document synthesis.

2.5. Agent Architectures: AutoGen and LangGraph

- **Database-Backed Assistants:** AutoGen integrates with vector databases like Qdrant, MongoDB Atlas, Couchbase Capella, and PGVector for scalable and persistent knowledge retrieval.
- **Custom LLM Integration:** Through LangChain-based clients, AutoGen supports the use of Hugging Face-hosted models and locally served LLMs, offering flexibility in deployment and experimentation.
- **Domain-Specific Simulations:** Examples include supply-chain optimization scenarios where agents communicate over multiple steps to identify bottlenecks or propose logistics improvements.
- **Educational Scenarios:** A tutor-student dynamic can be designed where agents engage in Socratic dialogue, scaffold learning steps, and adapt to the learner's performance in real time.

These examples demonstrate the modular and extensible nature of AutoGen's multi-agent framework, showcasing its potential across educational, industrial, and research domains.

2.5.5.2 Use Cases and Multi Agent Applications

AutoGen has demonstrated its usefulness across various domains, ranging from mathematical problem-solving and code generation to complex question answering, supply chain optimization, online decision-making, and even interactive entertainment.

A notable example is Magentic One [44], a generalist multi-agent system that combines an Orchestrator with four specialized agents (WebSurfer, FileSurfer, Coder, and ComputerTerminal), as illustrated in Figure 2.11. This configuration enables Magentic One to handle open-ended tasks involving web navigation, file reading, code analysis, and execution.

Built entirely on AutoGen, Magentic One has achieved competitive results on complex multi-step benchmarks, showcasing the capabilities of coordinated agents that can perform actions rather than merely engaging in conversation.

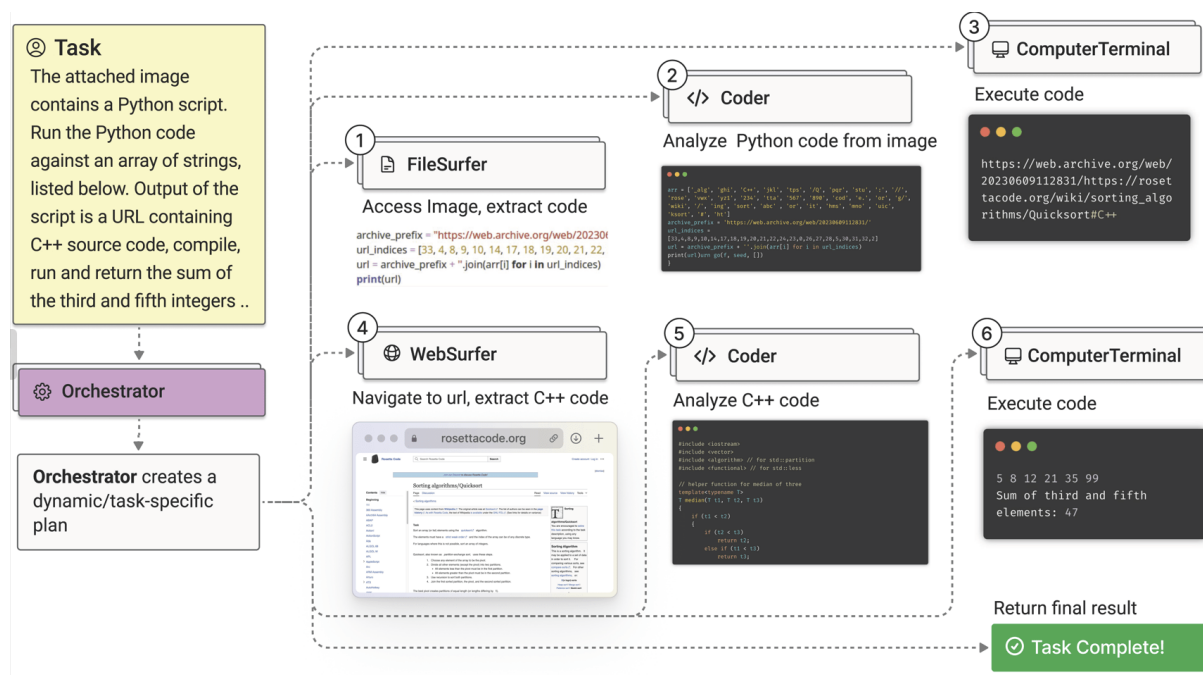


Figure 2.11: Magentic Example performing Complex Task [44]

Other use cases include programming assistants, where a code reviewer, a test generator, and a debugger collaborate to enhance code quality. Complex question-answering systems can distribute queries among agents specializing in information retrieval and synthesis, while role-playing characters can enrich interactive experiences in games. AutoGen’s flexible design and plugin ecosystem make it an excellent choice for developing enterprise AI copilots and autonomous digital assistants.

2.5.6 LangGraph: Origin and Structure

LangGraph [20] is an agent framework introduced as part of the LangChain ecosystem in 2023 and 2024. It is designed to support multi-agent workflows through a graph-based structure that makes agent coordination explicit and transparent.

In LangGraph, developers create a workflow graph where each node represents an independent agent with its own prompt configuration, language model, and associated tools. The edges between nodes define the direction of communication and the logic that governs transitions between agents.

This architecture models a multi-agent system as a type of state machine or workflow diagram, where each node represents a specialized agent state and the edges specify how and when transitions should occur. In contrast to the emergent, conversation-driven orchestration style of AutoGen, LangGraph adopts a more structured and rule-based approach.

2.5. Agent Architectures: AutoGen and LangGraph

Its primary design objective is to offer developers fine-grained control over the flow of agent interactions by explicitly encoding coordination logic within the graph structure.

For instance, developers can define a workflow where Agent A, acting as a tutor, consistently hands over control to Agent B, an explainer, after completing a specific task. Alternatively, they can include a router agent responsible for determining which among several sub-agents should handle a particular query. By expressing the system logic as a graph, developers can visualize, debug, and validate the coordination flow, making it especially suitable for educational applications.

LangGraph is particularly well suited for tasks that require structured workflows, predefined role assignment, and high reliability. The framework is tightly integrated with LangChain, enabling the use of its broad toolset, memory modules, and observability features.

This includes access to vector store memory, web search capabilities, calculator tools, and logging and monitoring functionalities via LangSmith. These integrations enhance the transparency and inspectability of agent behavior, supporting debugging and performance analysis through detailed tracing.

In contrast to AutoGen’s flexible and open-ended dialogue paradigm, LangGraph requires developers to explicitly define roles and communication paths in advance. This distinction between graph-based orchestration and conversation-based interaction is central to the design philosophy of LangGraph. While AutoGen views multi-agent collaboration as a dynamic exchange of messages, LangGraph approaches it as a directed execution graph. According to the LangChain team, this model is often more intuitive for developers building systems where control and predictability are essential.

In educational applications, LangGraph can be used to design hierarchical tutoring systems in which a top-level coordinator agent interprets a student’s request and delegates subtasks to specialized sub-agents. For instance, one agent may handle mathematical reasoning, another may focus on historical content, and others may address different domains.

The coordinator then aggregates the results and presents a coherent response to the learner. This hierarchical model maps naturally onto LangGraph’s structure, allowing queries to be routed and processed through a well-defined tree of agents.

2.5.6.1 LangGraph Examples

LangGraph offers a collection of tutorials demonstrating its applications in orchestrating complex workflows with language models. Below are some notable examples [45]:

- **Multi-Agent Collaboration:** This tutorial illustrates how to set up a system where multiple agents, each with specialized roles, collaborate to solve tasks. It showcases the coordination between agents for tasks like planning, execution, and verification.
- **Retrieval-Augmented Generation (RAG):** Demonstrates how to enhance language model responses by integrating external knowledge sources. This tutorial guides through setting up a RAG pipeline that retrieves relevant documents to inform the generation process.
- **Code Assistant:** Explores building an AI assistant capable of understanding and generating code snippets. The tutorial covers integrating language models with code execution environments to provide accurate and functional code suggestions.

2.5.7 Comparison: AutoGen vs LangGraph

Although AutoGen and LangGraph share the common goal of supporting the development of multi-agent AI systems, they are optimized for different use cases and modes of interaction. AutoGen's conversation-driven architecture provides a high degree of flexibility, making it particularly effective in scenarios where the problem-solving strategy is not fully defined in advance. For example, in open-ended educational settings, agents can collaboratively explore solution paths through dialogue, encouraging creativity and emergent behavior.

In contrast, LangGraph's graph-based orchestration emphasizes structure, control, and predictability. It is especially well suited for educational tasks that follow a clearly defined instructional sequence. Developers using LangGraph can explicitly script transitions between agents and enforce a logical instructional flow that aligns with specific learning objectives.

Notably, these two frameworks are not mutually exclusive but instead represent different points along a spectrum of agent coordination strategies. Both frameworks support human-in-the-loop interactions, allowing a teacher agent or educator to intervene or guide the process when necessary. They also include features for persistent memory and state tracking, which are essential for maintaining context across extended sessions.

By decomposing complex tasks into specialized roles and distributing responsibilities across multiple agents, both AutoGen and LangGraph promote modular system design and structured problem-solving. Research indicates that multi-agent configurations can outperform single-agent systems by enabling agents to verify and refine each other's outputs. This collaborative process leverages the strengths of individual agents to enhance factual accuracy, reasoning depth, and overall system robustness.

Chapter 3

Methodology and Application Development

In this chapter, we present the methodology adopted for the design and development of the application implemented using Streamlit. We begin by outlining the overall approach followed during the implementation phase, including the agent-oriented architecture that serves as the backbone of the system. This includes the coordination and interaction between the various agents involved, as well as the technologies employed to enable their integration.

Furthermore, we describe the key stages of development and how each component was structured to ensure modularity, usability, and performance. The chapter concludes with a reflection on the results obtained during testing and evaluation, along with a discussion of the overall user experience and impressions gathered throughout the development process.

3.1 Technologies Used

The development of the educational assistant application was guided by a clear set of principles centered on robustness, usability, modularity, and the ability to execute intelligent functionalities locally without reliance on cloud-based services. These principles informed every decision regarding the technological stack, architectural design, and integration strategy.

Particular emphasis was placed on selecting technologies that offered strong community support, comprehensive and up-to-date documentation, and proven compatibility with modern agent-based development paradigms.

It leverages Python for its expressiveness and flexibility, Streamlit for its ability to create responsive and intuitive user interfaces with minimal overhead, and LM Studio for enabling local execution of large language models with full control over privacy and performance. These core components were further supported by a suite of specialized libraries for data manipulation, visualization, API communication, and persistent storage.

By integrating these technologies into a cohesive whole, the application was able to deliver a seamless and responsive user experience while also supporting complex interactions between autonomous agents, each responsible for specific tasks such as theoretical explanation, code generation, and user progress analysis.

3.1.1 Python as the Core Programming Language

Python [46] served as the foundational programming language for the entire project. It was chosen not only for its widespread adoption in the fields of data science and artificial intelligence, but also due to its simplicity, readability, and extensive library ecosystem. The language's dynamic typing and high-level abstractions facilitated rapid prototyping, while its mature frameworks allowed for scalable production-level development.

Python enabled the integration of multiple components including user interaction logic, backend services, data preprocessing routines, and communication protocols. Its compatibility with packages like `pandas`, `numpy`, `streamlit`, and `requests` maintained a unified codebase without having to switch to other technologies for auxiliary tasks.



Figure 3.1: Python Logo [46]

3.1.2 Streamlit for Building the User Interface

The frontend of the application was developed using **Streamlit** [47], an open-source Python framework that simplifies the creation of interactive web interfaces. Unlike traditional web development stacks that require HTML, CSS, and JavaScript, Streamlit allows developers to build complete interfaces with pure Python code, greatly reducing development time and complexity.

Streamlit was particularly effective in enabling dynamic interactions with the underlying language model. Users could input questions, view contextual responses, and navigate through the application in an intuitive and responsive manner. Streamlit's built-in components such as `st.selectbox`, and `st.tabs`, were extensively utilized to deliver a clean, reactive, and engaging user experience.

Moreover, the modular nature of Streamlit's session state allowed for persistent interaction histories and user profile management across different views, facilitating a personalized learning experience tailored to each student's activity and progress.

One of the key reasons for selecting Streamlit over more traditional frontend frameworks such as React, Angular, or Vue.js was its significantly lower learning curve and seamless integration with the Python backend. Given the educational and experimental nature of this project, it was deemed counterproductive to allocate disproportionate development resources solely to the frontend layer.

Instead, development efforts were concentrated on optimizing core functionalities, agent orchestration, and the interaction between the user and the intelligent system. This pragmatic decision allowed for faster iteration, better alignment with the team's skill set, and greater focus on pedagogical effectiveness rather than visual complexity.

Nonetheless, careful attention was given to the design of the user interface, ensuring that visual consistency, usability, and responsiveness were maintained throughout the application. Several improvements and interface refinements were incorporated, and will be discussed in greater detail in subsequent sections.



Figure 3.2: Streamlit Logo [47]

3.1.3 LM Studio as the LLM Backend

For natural language understanding and generation, the application integrated **LM Studio** [48] as the backend engine for hosting large language models (LLMs) locally. LM Studio supports a wide variety of open-source models and enables their execution without requiring internet access, providing a secure, offline, and highly customizable environment for LLM-based applications.



Figure 3.3: LM Studio Logo [48]

Methodology and Application Development

In this project, the specific model deployed via LM Studio was **Nous Hermes 2 Mistral 7B**, a 7-billion-parameter instruction-tuned model that has demonstrated strong performance in tasks involving reasoning, conversation, and educational support. This model was selected due to its balance between performance, resource efficiency, and contextual understanding, making it particularly suitable for an AI-powered educational assistant.

Throughout the development process, several LLMs were tested and benchmarked. However, one of the most critical constraints encountered was related to hardware limitations. The machine used to develop and run the application was equipped with an NVIDIA RTX 2060 Super GPU. While sufficient for many medium-scale models, certain LLMs consumed excessive GPU memory, leading to performance bottlenecks or even failure to launch. The selection of Nous Hermes 2 Mistral 7B was ultimately driven by its ability to run smoothly on this hardware while maintaining high-quality output.

A fundamental design decision in this project was to avoid relying on cloud-based LLM providers such as OpenAI. While using an API key from OpenAI would have provided seamless access to powerful models like GPT-4 with minimal setup and without taxing local resources, the associated operational costs were considered prohibitive for a sustained development cycle. Since this project aimed to be cost-effective and reproducible, prioritizing local execution was a strategic choice.

LM Studio offered a robust alternative, enabling the application to run entirely offline, preserving user data privacy and ensuring that all computational resources and interactions remained under full local control. This approach not only eliminated ongoing API costs but also aligned with the broader goals of transparency, independence, and educational accessibility.

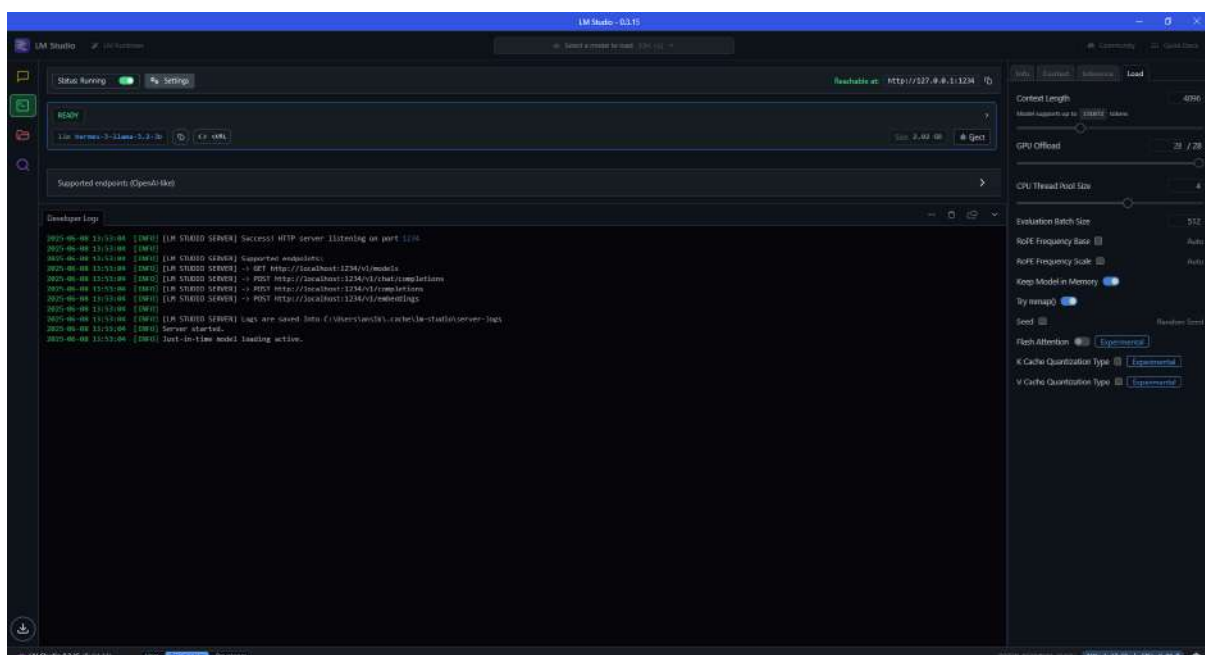


Figure 3.4: LM Studio Interface [48]

3.1.4 PyCharm as the Integrated Development Environment

To support structured and efficient software development, **PyCharm** [49] by JetBrains was employed as the Integrated Development Environment (IDE). PyCharm provides an advanced feature set including syntax highlighting, code analysis, version control integration, and an intelligent debugger.

The IDE was configured with project-specific virtual environments to manage dependencies and ensure compatibility across different modules. The integrated terminal and database tools also facilitated quick testing of data queries and API endpoints.



Figure 3.5: PyCharm Logo [49]

3.1.5 System Overview and Execution Flow

The application follows a modular and efficient pipeline that connects the development environment, user interface, backend logic, and local language model through a structured flow of information.

The backend was developed in **Python** using the **PyCharm** environment, where all agent logic, API endpoints, and orchestration scripts were implemented. The frontend was built with **Streamlit**, which captures user input and sends it to the backend through local API requests.

Once received, the request is forwarded to the language model running locally on **LM Studio**, which processes the prompt using the appropriate architecture of agents. The response is then interpreted and returned to the user interface, where it is displayed in real time. Depending on the use case, the system may also activate specific tools, execute code, analyze user performance, or store results for later use.

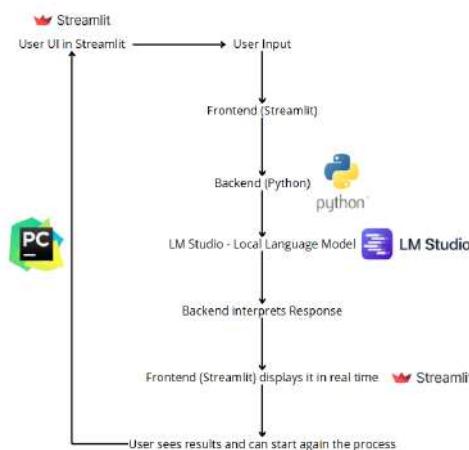


Figure 3.6: Project High-Level Flow

3.2 Project Structure

The project follows a modular and organized file structure designed to separate responsibilities between frontend components, backend logic, and configuration resources. This structure enhances maintainability, reusability, and readability, which are critical in the context of agent-based systems and local LLM integration.

The figure below illustrates the directory structure of the project:

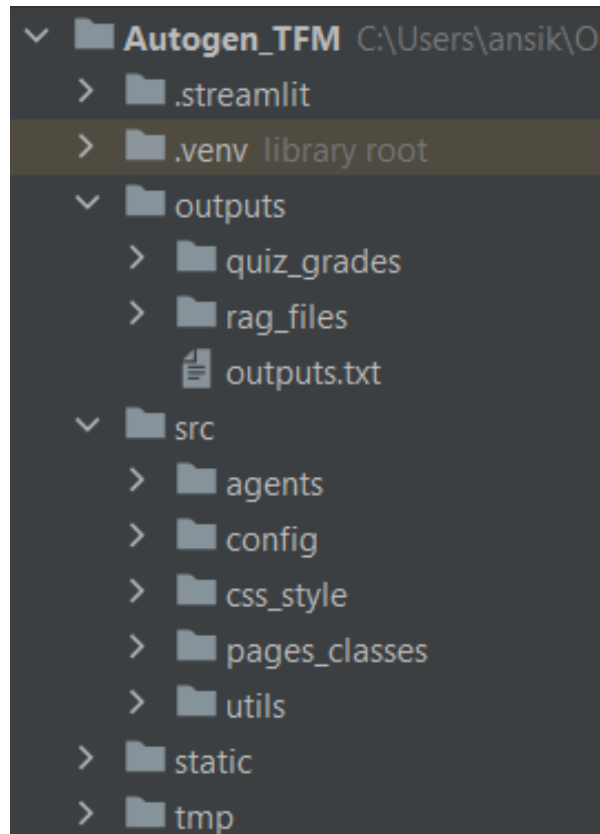


Figure 3.7: Project Structure Tree

The following is a detailed description of the main folders and files:

- **/streamlit:** Contains Streamlit-specific configuration files, such as `config.toml` and `secrets.toml`, which define layout parameters, page settings, and any necessary credentials or tokens used during development.
- **/src:** This folder contains all backend logic and agent-related processing. It also includes Streamlit-related logic required to display and render the application interface. It is divided into the following modules:
 - **/agents:** Contains all logic related to agent creation and instantiation. This module defines functions that return specific instances of AutoGen agents, tailored to the roles and configurations needed throughout the application.

- **/config**: As its name suggests, this folder centralizes all configuration elements, including architecture definitions and Streamlit settings.
 - * **agent_configs** and **architecture_configs**: Contain Python scripts with the exact parameter definitions used in AutoGen-based agent architectures. These files define how agents communicate, their tool usage policies, and memory behavior. These configurations are referenced in later sections.
 - * **streamlit_config/**: Includes Python modules that define helper functions for Streamlit, manage constants, and dictate frontend behavior. Files such as `frontend.py`, and `common_constants.py`, belong to this category.
- **/css_style**: Stores all custom CSS files used to style the application interface. Examples include `fonts.css`, `logo.css`, and `backgrounds.css`, which collectively contribute to the visual identity of the application.
- **/pages_classes**: Contains modular definitions of each Streamlit page. Each file (e.g., `introduction.py`, `performance.py`, `documentation.py`) corresponds to a user-facing page within the application. Some files focus on results and analytics, such as `performance_practical.py`, while others offer navigation and profile views.
- **/utils**: Utility scripts such as `loaders.py`, which are responsible for data preparation, loading from databases, or general support tasks.
- **/static**: Contains assets such as:
 - **backgrounds/**, **fonts/**, and **logos/**: Dedicated folders for static resources used across the frontend for styling and branding purposes.
- **/outputs**: This folder stores dynamic content generated during app usage, including:
 - **quiz_grades/**: Contains user quiz results or evaluation outcomes.
 - **rag_files/**: Intermediate or final files generated during Retrieval-Augmented Generation processes.
 - `outputs.txt`: A summary log of output history or system responses.
- **/tmp**: Temporary working directory used to store runtime data, session files, or volatile artifacts. Includes:
 - **/db**: Directory used to store or reference the database used for agent-driven SQL analysis.
- **app.py**: The entry point of the Streamlit application. It initializes page routing, sets the session state, and launches the user interface.
- **chinook.db**: Test SQLite database used for validating SQL agent responses.
- **TFM_AARON_MOYANO.pdf**: Local copy of the compiled TFM document for reference during development.
- **requirements.txt**: List of Python packages required to run the application in any environment.

Methodology and Application Development

- **README.md:** Main project documentation file with instructions, purpose, and usage guidelines.

This structure ensures a clean separation of frontend, backend, configuration, and experimental modules. It supports fast iteration, debugging, and modular growth of the application while aligning with best practices in both academic and production-grade software development.

3.3 Frontend Development

This section presents the design decisions, interface implementation, and user-facing functionalities integrated into the application. By using Streamlit, the system benefits from an easy and fast way to develop interactive interfaces that seamlessly integrate with the agents and architectural components. This compatibility allows for rapid experimentation and deployment of intelligent workflows without requiring complex frontend engineering.

Accordingly, this section highlights the key aspects of the frontend layer, including the reasoning behind the application's name and identity, visual design decisions, and the functionality offered through each implemented page.

3.3.1 Interface Design

Unlike traditional frontend frameworks that require knowledge of JavaScript, HTML, and CSS, Streamlit allows developers to build interactive user interfaces directly in Python, enabling rapid prototyping and a strong focus on backend and agent integration.

The following design decisions were implemented:

- **Typography:** Typography was selected deliberately, understanding its psychological impact on user perception. A classic yet elegant font was chosen to make the interface visually pleasing and readable, enhancing the user's comfort and trust while interacting with the platform.
- **Tabs and Layout:** The layout was structured using tabbed interfaces and modular pages, offering a clean and logical separation of content. From a technical perspective, Streamlit's native support for components such as `st.tabs`, `st.sidebar`, and expandable containers were used to create fluid navigation. Code fragments used to build these layouts are included in the appendix.
- **Clean and Professional Aesthetic:** Emojis were intentionally minimized to preserve an academic tone. Their use was limited to the `README.md` file, where visual guidance may be helpful. When emojis were used, only the professional-style icons from Google Fonts were considered appropriate for this setting.

3.3.2 Name and Branding

The application is named **PhronesisAI**, a term rooted in Aristotelian philosophy meaning "practical wisdom". This concept was deliberately chosen to reflect the dual objective of the platform: to provide intelligent assistance not only to students, through theoretical explanations, practical help, and interactive assessments, but also to educators, by enabling insight into performance metrics and fostering more effective learning processes. The name encapsulates the essence of applied knowledge powered by artificial intelligence.

Branding efforts were consciously minimalistic to preserve focus on functionality and clarity. However, several key visual identity decisions were implemented to ensure the application maintained a professional and consistent aesthetic:

- **Logo:** The logo used is a simplified adaptation of the official emblem of the *Universidad Politécnica de Madrid* (UPM). It was integrated into the sidebar and header components, establishing a direct institutional identity. No elaborate or animated graphics were included, in line with the academic tone of the tool.
- **Color Scheme:** A neutral and elegant color palette was selected, primarily based on shades of blue and gray. These tones were used to highlight section headers, delineate content blocks, and draw subtle attention to user actions without overwhelming the visual space. Color use was always subordinated to readability and user comfort.
- **UI Identity:** The user interface identity was carefully maintained across all pages using custom CSS files. These ensured consistency in spacing, button styles, font sizing, and background tones. The absence of emojis across the interface, except for a minimal presence in the `README.md`, reinforced the formal character of the tool. When used, emojis were selected from the Google Fonts emoji set to preserve a polished appearance.
- **Typography and Layout Integration:** The overall branding also extends to typographic decisions and layout logic, where spacing, alignment, and interactive components, were structured with clarity and accessibility in mind. The platform's visual flow was built to support extended reading, exploration of agent output, and analysis of personalized feedback without inducing cognitive fatigue.



Figure 3.8: Phronesis AI Logo

- **Theoretical Explanation:** Allows users to ask theoretical questions about specific subjects. The LLM provides academic-style answers based on context.



Figure 3.11: Theoretical Page

- **Practical Explanation:** Similar to the theoretical page, but focused on practical tasks such as coding, solving exercises, or explaining hands-on procedures.



Figure 3.12: Practical Page

Methodology and Application Development

- **GenAI Quiz:** A page dedicated to testing knowledge via quizzes generated by the agent. It tracks correct and incorrect answers and gives performance feedback at the end of each session.



Figure 3.13: Quiz Page

- **RAG Page:** Enables the user to upload academic or technical PDFs. The assistant uses RAG to answer questions using the uploaded documents as context.



Figure 3.14: RAG Page

- **Orchestrator:** Displays a continuous conversation with an agent, following an interactive chat format.



Figure 3.15: Conversation Page

- **Documentation Page:** Provides access to the TFM content, explaining the methodology, tools used, and references. Acts as an internal guide or integrated appendix.



Figure 3.16: Documentation Page

- **Profile Page:** A personalized area showing user data such as quiz history, progress metrics, and AI-generated feedback.



Figure 3.17: Profile Page

The application follows a dual level navigation strategy, combining sidebar based page switching with tab based content separation within pages.

All primary pages are accessed through the Streamlit sidebar. This sidebar acts as the main control hub, enabling users to jump between sections without reloading the application state. Each menu item in the sidebar is linked to a modular Python class that renders the corresponding view when selected.

Within some of these main pages, especially the tutor section, additional subnavigation is implemented using `st.tabs`. This component allows the interface to display multiple views within a single page container, each dedicated to a specific functionality. For instance:

The **Theoretical Explanation**, **Practical Help**, and **Generative Quiz** sections are all grouped under a unified tutor interface, and users switch between them using tab controls rendered with `st.tabs`.

In addition to tab usage, various interaction mechanisms were implemented to ensure smooth task flow. These include dropdown menus for selecting subjects and difficulty levels, buttons for submitting user inputs or generating responses from the language model, and expandable containers for displaying generated content and feedback.

3.4 Architecture of the GenAI Agents

This section details the design and implementation of the intelligent agent system that powers the educational assistant. Each agent is assigned a specific role and communicates with other agents when collaboration is needed, allowing for a division of responsibilities and scalability across tasks.

The framework supports the use of custom tools, and agent orchestration mechanisms that are essential for building complex, goal oriented interactions within an educational environment.

All source code referenced in this section will be included in the Annex 5.2, where it is most appropriate to provide complete context for the implementation. This ensures that the explanation in the main body of the document remains focused while still offering full technical transparency.

3.4.1 Multiagent Design Implementation

Within the file `src/agents/genai_agent_setup.py`, the setup of all agents used in the application is defined. This configuration is specific to each agent and includes the agent name, prompt content, language model settings, whether code execution is enabled, whether retrieval augmented generation is used including the document path, embedding method, chunk size, and other agent specific parameters.

To avoid code repetition, each agent is instantiated through dedicated factory functions. These functions are defined in `src/agents/functions/agent_functions.py`. This file returns the correct instance of the AutoGen agent, such as `UserProxyAgent`, `AssistantAgent`, or `RetrieveUserProxyAgent`, according to the configuration provided.

Since the prompt and name used by each agent can be lengthy and specific to its role, it was considered best practice to move those settings to a separate configuration file for clarity and maintainability. For this reason, names and prompts are stored in `config/agent_configs/genai_agent_config.yaml`.

Regarding agent tools, all callable functions that agents may execute are defined in `src/agents/functions/ai_tutor_functions.py`. These are registered using the `register_function` method provided by the AutoGen framework and are bound to the corresponding agents during their initialization.

The main genAI agents integrated into the system include:

- **HumanInputAgent:** Simulates the user interface for testing purposes or routes user messages during live sessions.
- **QuestionRewriter:** Reformulates input questions for clarity and alignment with model capabilities.
- **CoherenceReviewer:** Analyzes answers for logical coherence and completeness based on educational criteria.

- **SubjectSpecificExplainers:** Includes dedicated agents for each subject, such as Linear Algebra, Calculus, Probability and Statistics, Python Programming, and Databases. These agents generate detailed explanations based on the selected topic.
- **QuestionRewriterCodingAgent:** Reformulates technical or coding related questions to optimize the output from coding agents.
- **SQLWriterAgent** and **SQLExecutorAgent:** Handle generation and execution of SQL queries. The executor agent has access to a SQL environment and is equipped with registered tool functions such as `execute_sql_query`.
- **CodingResultAgent:** Analyzes the outputs generated by the SQL executor or Python code writer to verify accuracy and present interpretations.
- **PythonCodeWriter** and **PythonCodeExecutor:** A pair of agents that generate and execute Python code in response to programming or mathematical prompts. The executor uses a local command line environment configured with a working directory and optional virtual environment context.
- **RetrieverAgent:** A retrieval user proxy agent configured to work with uploaded documents using a defined chunk size and embedding model. It fetches relevant content from local files to support context rich generation.
- **RetrieverExplainerAgent:** Receives the retrieved context and produces answers that combine both user query understanding and document content.

All agents are configured to handle termination messages, control reply loops, and manage contextual parameters through consistent configuration logic. The tools required by specific agents, such as SQL execution, are registered programmatically using AutoGen's `register_function` utility.

3.4.2 Agent Workflow and Communication Flow

The communication between agents is governed by architectural patterns. Different chat types are employed throughout the application depending on the complexity and nature of the task. The two principal architectures used are group based chats and sequential chats with carryover context.

The majority of multiagent workflows are implemented using the `GroupChat` architecture in combination with the `GroupChatManager`. In simple terms, a group chat consists of multiple agents who share a common message thread. Each agent subscribes to the conversation and is responsible for a specific role such as generating content, validating answers, or offering pedagogical feedback.

3.4. Architecture of the GenAI Agents

In this design pattern, agents take turns publishing messages in a sequential manner. At each step, only one agent is active. The order of turns is controlled by the `GroupChatManager`, which decides which agent should respond next based on a pre-defined policy. Common strategies include round robin selection or logic driven by a language model. This setup is particularly useful for decomposing complex tasks into smaller subtasks that are handled by specialized agents with well defined responsibilities.

In the context of this application, agents participating in a group chat must be configured in the correct order along with their roles and parameters. These configurations are stored in `config/architecture_configs/genai_architecture_configs.py`. Additionally, fixed values such as maximum number of turns or retry limits are defined in `src/config/common_constants.py`, ensuring consistency across all group chat workflows.

In contrast to group chat, the application also supports sequential chat architectures with carryover memory. This is used in the generative conversational chat interface, which simulates an ongoing dialogue.

In this pattern, a pair of agents engages in a conversation, and the result of that conversation is summarized and passed forward as contextual memory to the next pairwise interaction. This mechanism, called *carryover*, allows the chat to evolve over time while preserving the full context of previous interactions.

Each subsequent session receives the accumulated carryover and continues the conversation accordingly. This pattern is suitable for long form interactions or tasks where continuity and memory are essential, such as in tutoring dialogues or coaching sessions.

The logic that controls when and how each chat session is launched is implemented in `streamlit_config/streamlit_aux_genai.py`. This module provides the auxiliary functions that connect user inputs from Streamlit pages to the appropriate agents and architectures. Each function loads the required configuration, initializes the chat structure, and routes the interaction flow according to the design pattern in use.

3.4.3 Integration with the User Interface

Each agent or group of agents is closely integrated with a specific page in the Streamlit frontend. When a user interacts with a given tab or section, the backend dynamically loads the necessary architecture and agents based on the context and input parameters. The integration between interface and agent logic is managed by the auxiliary module `streamlit_aux_genai.py`, which contains the execution functions for each interaction pattern.

Methodology and Application Development

The following describes how agents are launched and operate within each tab of the application:

- **Theoretical Explanation Tab:** When the user selects a specific subject using a `st.selectbox`, they are prompted to write a question inside a `st.text_area`. Upon clicking the `Generate Explanation` button, a function is triggered. This function determines the appropriate group chat based on the selected subject and sends the question to the corresponding agent group. The response is rendered in real time in the main content area.
- **Practical Help Tab:** The practical help section works in a similar manner but is oriented toward executable content. Based on the subject selected by the user, a group chat is launched that includes agents for code generation and execution. For example, Python code produced by one agent is executed locally by another agent using the virtual environment configured within the `.venv`. If the selected subject is Databases, then an agent generates an SQL query and another executes it, using a registered tool linked to the SQL executor.
- **GenAI Quiz Tab:** This page includes two `st.selectbox` components that allow the user to choose both the subject and the difficulty level. Once selected, a set of nine questions is displayed sequentially. The student answers each question through a text area, and the response is evaluated using a group chat composed of validation agents. After processing, if the response contains specific keywords indicating completion, the result is saved. These performance metrics are later displayed in the Profile page, where the user can view statistics per subject including the number of correct and incorrect responses.
- **RAG Assistant Tab:** This page mirrors the behavior of the theoretical explanation tab but introduces a retrieval augmented mechanism. The user uploads one or more documents, and the system builds an embedding index. Depending on the selected subject, a specific group chat with a retrieval agent is launched, and the query is answered using context retrieved from the documents. The result is displayed in a similar format as in the theoretical section.
- **Generative Chat Tab:** This section implements a conversational agent with persistent context. The user interacts through a continuous input field, and the conversation flows naturally without resetting state. The architecture here is not based on group chats but rather on a sequential chat structure using carryover memory. The context is accumulated and passed along the session, allowing the assistant to retain previous exchanges and respond in a coherent and contextually aware manner.
- **TFM Viewer and Profile Tabs:** These pages do not directly launch agents. Instead, they render outputs previously generated by agent sessions. The Profile page visualizes user statistics and quiz performance, while the TFM Viewer page displays the content of this report within the same interface.

3.4. Architecture of the GenAI Agents

All pages operate in a one shot format, meaning that each user interaction triggers a fresh response from the corresponding architecture. The only exception is the Generative Chat Page, which maintains continuity through memory. This design ensures that most sessions remain stateless and lightweight, except where persistence is pedagogically beneficial.

In addition to providing a fully functional conversational experience, all chat interactions are executed using the locally hosted language model loaded through LM Studio. This ensures full control over data privacy and removes the need for external APIs.

Moreover, each chat session automatically logs the entire conversation, including all exchanged messages and contextual decisions made by the agents. Token usage and associated metadata are also recorded. This allows developers or researchers to simulate the impact of token costs, as would occur in commercial LLM deployments, and to evaluate the computational or economic efficiency of each exchange.

Chapter 4

Results and System Outputs

This chapter presents the outputs generated by each main section of the application, leveraging the intelligent agent architectures implemented through AutoGen and executed via LM Studio. For each page, we illustrate how user input is processed, how agents respond, and how the final interface and log outputs appear. These examples validate the correct functionality of each component and demonstrate the educational value of the assistant.

As previously mentioned, all source code and implementation details behind the agent logic, configuration, and orchestration will be documented in the Annex section to provide a complete technical reference without interrupting the results narrative.

4.1 Theoretical Explanation Page

For this example, we use a basic theoretical question from Linear Algebra:

"Define an eigenvalue and an eigenvector."

This question is entered into the interface via the Theoretical Explanation tab. Once the subject is selected and the prompt submitted, a group chat specific to Linear Algebra is activated. The corresponding agent generates an academic response based on few-shot prompting and subject-specific configuration.

4.1. Theoretical Explanation Page



Figure 4.1: Streamlit Page Question 1

The outputs as displayed both in the Streamlit frontend and logged in LM Studio. These include the generated explanation, token usage, and time required for the response.

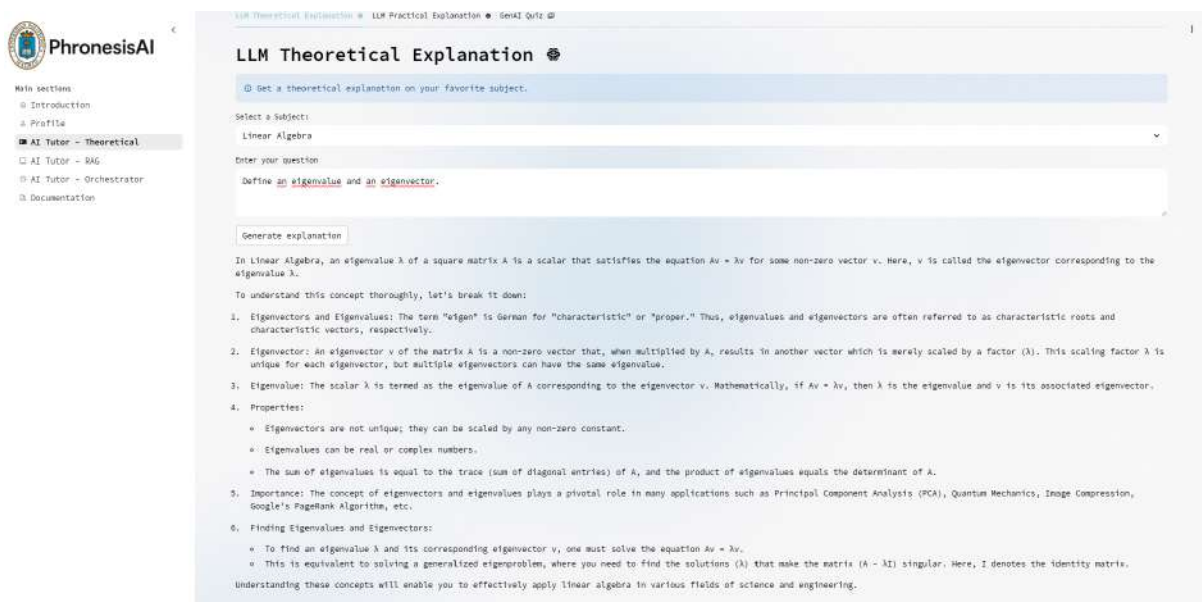


Figure 4.2: Streamlit Page Solution 1

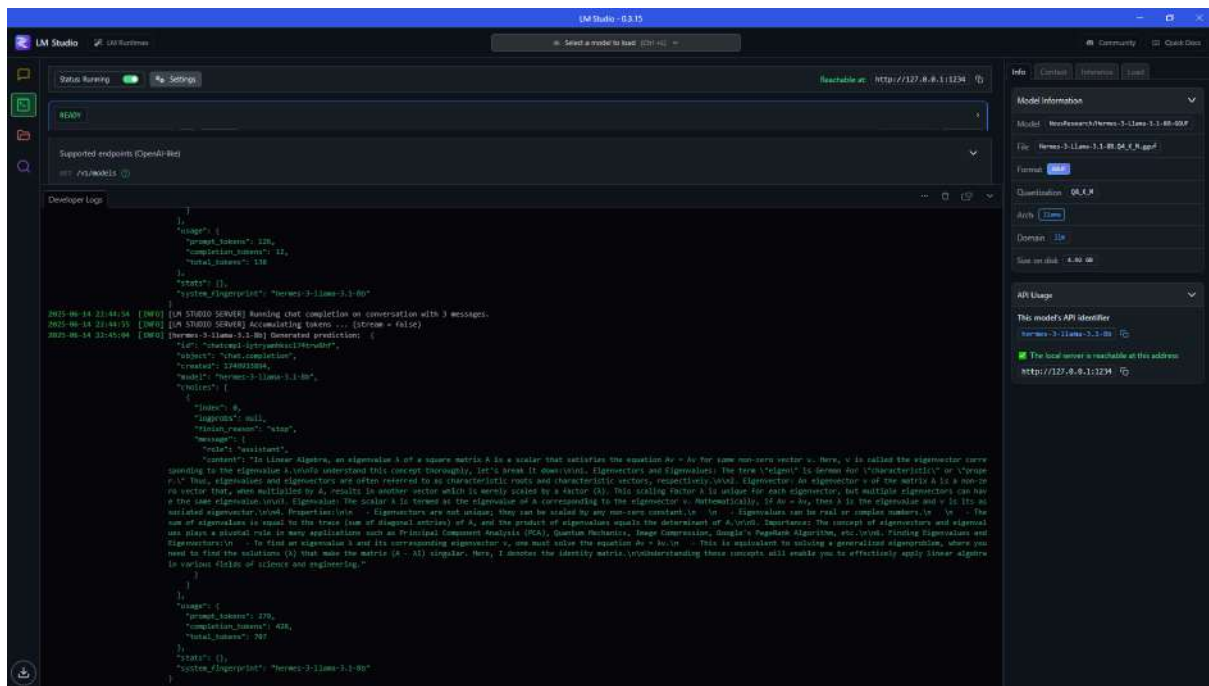
The question is first displayed along with a loading bar. Once the loading completes, it indicates that a solution has been generated, which is then rendered in the Streamlit interface.

Results and System Outputs

```
python/requirements.txt (in chat_session)
What are eigenvalues and eigenvectors of linear algebra?
.....
user @streamlit: LinearAlgebra/LinearAlgebra
[python.usr.11:56] 06-26 22:01:00 [!E] WARNING - Model Hermes-3-11mo-3.1-80 is not found. The cost will be 0. In your config.yaml, add field 'url_api' : [python.usr.per.3], completion_tokens_price_per_1k] for customized pricing.
LinearAlgebra/LinearAlgebra (in chat_session)
In Linear Algebra, an eigenvalue  $\lambda$  of a square matrix  $A$  is a scalar that satisfies the equation  $Ax = \lambda x$  for some non-zero vector  $x$ . Here,  $x$  is called the eigenvector corresponding to the eigenvalue  $\lambda$ .
To understand this concept thoroughly, let's break it down:
1. Eigenvectors and Eigenvalues: The term "eigen" is German for "characteristic" or "proper." Thus, eigenvalues and eigenvectors are often referred to as characteristic roots and characteristic vectors, respectively.
2. Eigenvectors: An eigenvector  $x$  of the matrix  $A$  is a non-zero vector that, when multiplied by  $A$ , results in another vector which is merely scaled by a factor  $\lambda$ . This scaling factor  $\lambda$  is unique for each eigenvector, but multiple eigenvectors can have the same eigenvalue.
3. Eigenvalues: The scalar  $\lambda$  is termed as the eigenvalue of  $A$  corresponding to the eigenvector  $x$ . Mathematically, if  $Ax = \lambda x$ , then  $\lambda$  is the eigenvalue and  $x$  is its associated eigenvector.
4. Properties:
   - Eigenvectors are not unique; they can be scaled by any non-zero constant.
   - Eigenvalues can be real or complex numbers.
5. Finding Eigenvalues and Eigenvectors:
   - To find an eigenvalue  $\lambda$  and its corresponding eigenvector  $x$ , one must solve the equation  $(A - \lambda I)x = 0$ .
   - This is equivalent to solving a generalized eigenvalue problem where one needs to find the solutions  $\lambda$  that make the matrix  $(A - \lambda I)$  singular. Here,  $I$  denotes the identity matrix.
Understanding these concepts will enable you to effectively apply linear algebra in various fields of science and engineering.
```

Figure 4.3: Streamlit Page Solution 1 Terminal

To verify that the output is as expected, both the terminal and LM Studio are reviewed.



The screenshot shows the LM Studio interface with a chat window on the left and a terminal window on the right. The chat window contains a user prompt: "What are eigenvalues and eigenvectors of linear algebra?". The terminal window displays the output of the Streamlit application, which includes a detailed explanation of eigenvalues and eigenvectors, their properties, and how to find them. The terminal output is identical to the content shown in Figure 4.3.

Figure 4.4: Streamlit Page Solution 1 LM Studio Log

The provided solution is correct in the context of Linear Algebra. Moreover, the output did not always appear this way by default. Significant prompt engineering was required to optimize the results, ensure proper understanding of the workflow, and guarantee that the agents behave as subject-matter experts in their assigned domain.

4.2 Practical Help Page

This page is designed to support hands-on learning by generating and executing code. When users describe a problem, a group of agents collaborates to generate a solution in either Python or SQL, depending on the context. A secondary agent is then responsible for executing the code and returning the full output.

4.2.1 Calculus Usage

For this example, we use a fundamental theoretical question from Calculus:

"Calculate the volume generated by rotating $f(x) = x^2$ around the x-axis from $x = 0$ to $x = 2$."

The outputs generated by the agents can be observed, where they proceed to write and execute the corresponding code. In the Streamlit interface, only the final result is displayed, no intermediate steps are shown.



Figure 4.5: Streamlit Page Solution 2

Therefore, screenshots of the terminal output and the LM Studio logs are provided to confirm that the generated Python code is correctly executed and that the expected solution is produced.

4.2.2 Database Usage

In this section, a question is posed regarding the `chinook.db` database.

For this example, we use the following question:

"List the names of 5 albums along with the name of the artist who created them."



Figure 4.8: Streamlit Page Solution 3

It can be observed how a query is generated, and the agent proceeds to execute it using the appropriate tool to retrieve the result from the database.

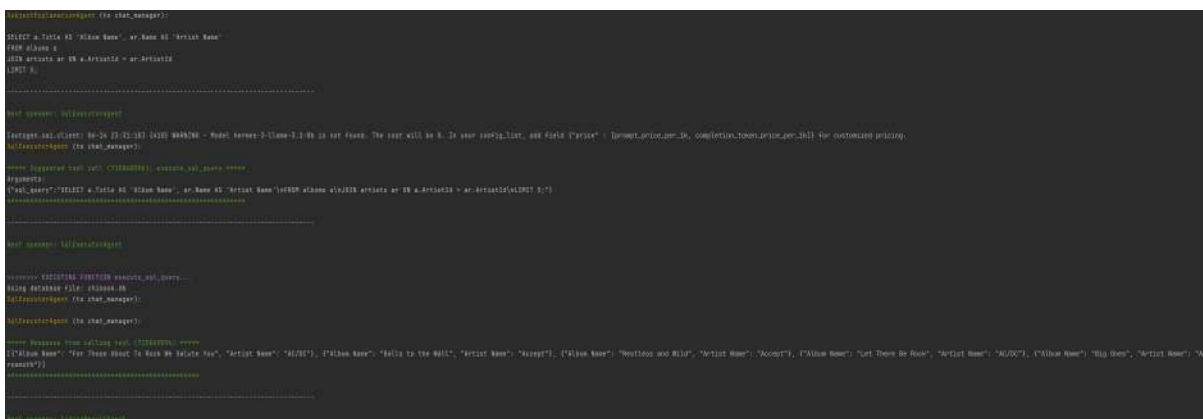


Figure 4.9: Streamlit Page Solution 3 Terminal

Results and System Outputs

This result is then collected, and a final solution is generated based on the query execution.

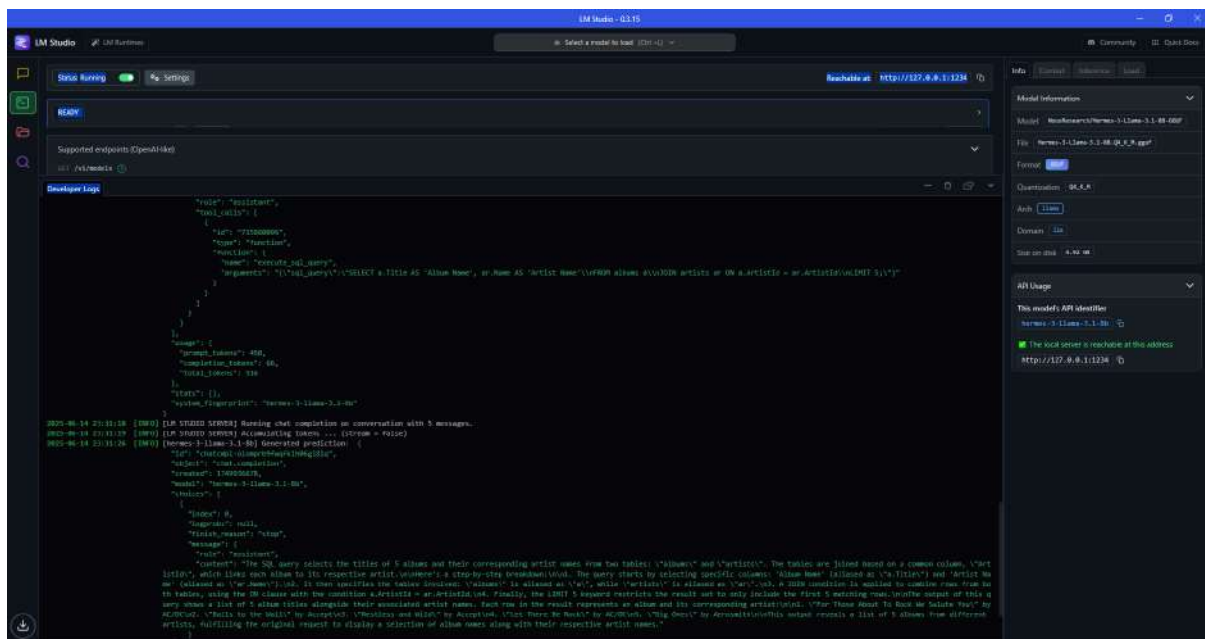


Figure 4.10: Streamlit Page Solution 3 LM Studio Log

4.3 GenAI Quiz Page

In this section, the selected subject is **Python Programming**, with an **easy** difficulty level. The chosen question is:

"What is Pandas in Python, and why is it popular for data science?"

As students, we will intentionally provide a partially incorrect answer in order to test the system's ability to identify mistakes and provide corrective feedback.

As we can see, the provided answer is not correct, and the system responds with a **No**, followed by the correct solution.

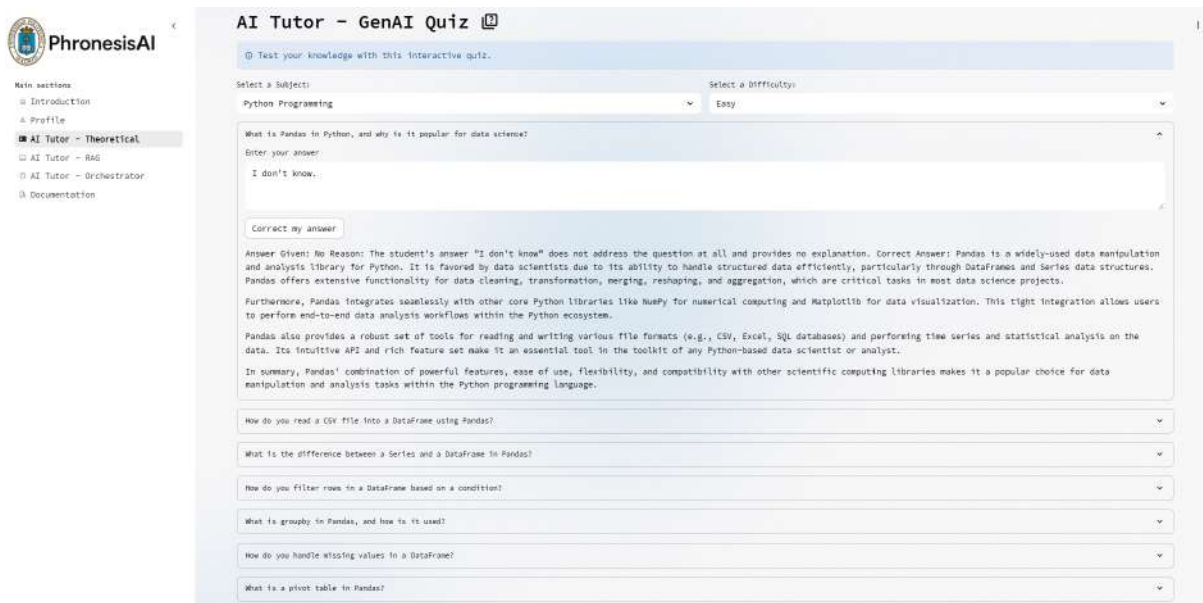


Figure 4.11: Streamlit Page Solution 4

After this interaction, we can observe in the following image that the incorrect answer has been registered in the performance tracking.

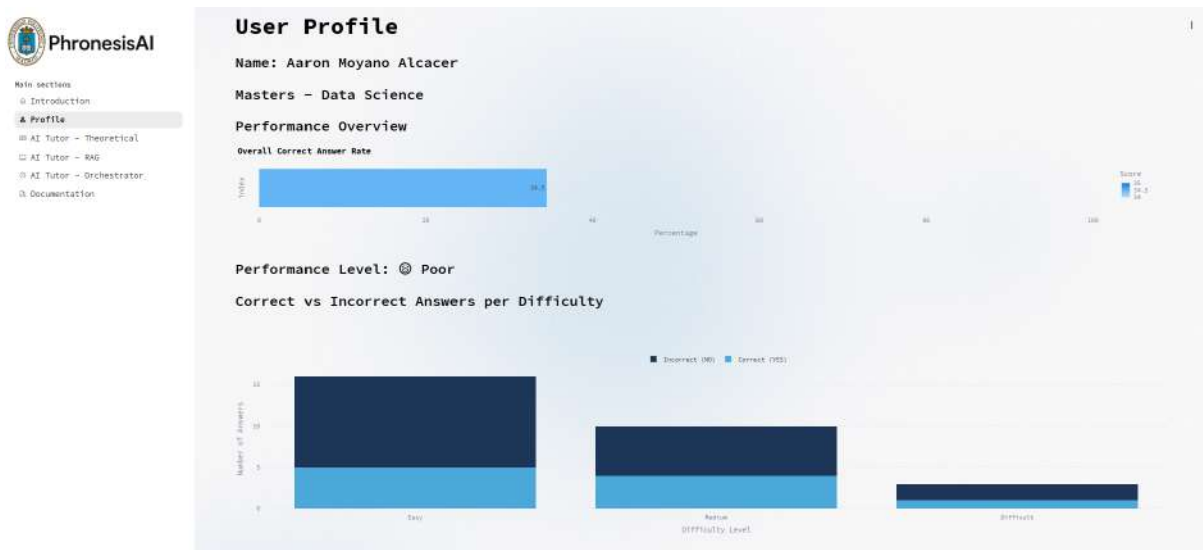


Figure 4.12: Streamlit Page Solution 4

Results and System Outputs

The **No** response is reflected under the subject *Python Programming*, updating the student's progress accordingly.



Figure 4.13: Streamlit Page Solution 4

4.4 RAG Assistant Page

In this section, a PDF file is uploaded to simulate lecture notes on Probability and Statistics. A question is then provided:

"Why are Kolmogorov's axioms necessary, and what would go wrong without them?"

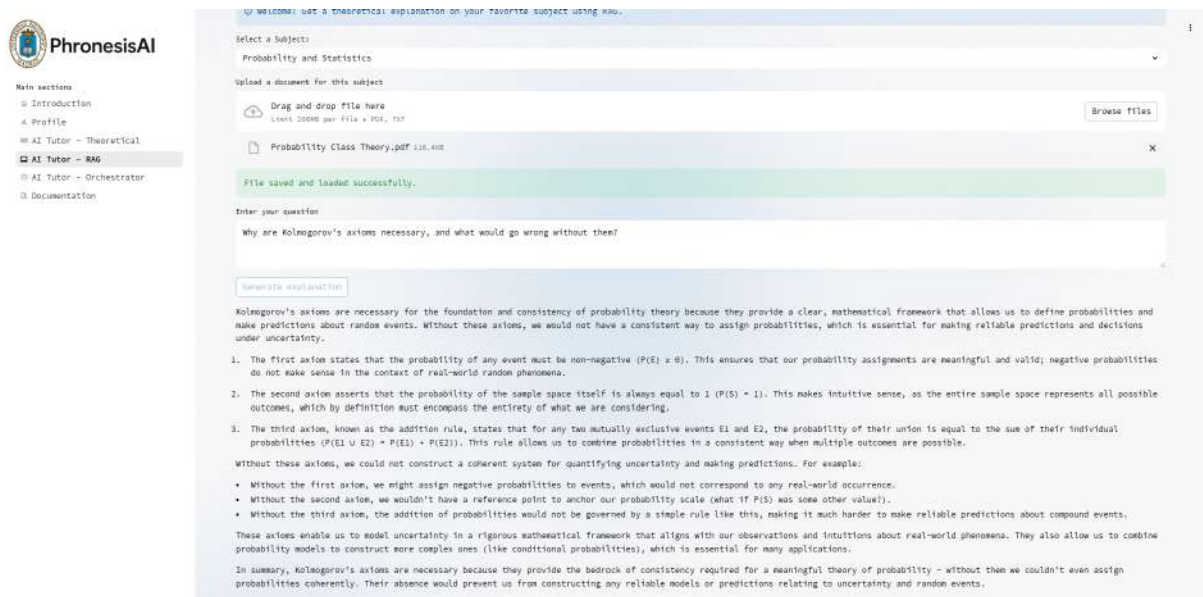


Figure 4.14: Streamlit Page Solution 5

As shown in the terminal, an embedding is created using chunk sizes based on the uploaded document. The relevant information is retrieved for the agent, which then generates the following solution.

```

2024-08-24 23:54:07,927 - whisper_agentchat.controller: Retrieving user_prompt_agent - INFO - Found 2 chunks
2024-08-24 23:54:09,982 - whisper_agentchat.controller: Retrieving user_prompt_agent - INFO - No context necessary is provided. Will use the VectorDB's embedding function to generate the context embedding.
Number of requested results 20 is greater than number of elements in index 1, setting n_results = 1
VectorDB returns DOC_IDS: ['589F0A6111']
Using contents of the embedding to construct:
Embedding: (0, ProbabilisticTheoryOfProbabilityAgent)
You're a retrieval-augmented chatbot. You answer user's questions based on user's ask knowledge and the context provided by the user.
If you can't answer the question with or without the current context, you should reply exactly "SORRY I DON'T KNOW".
You must give an answer as soon as possible.
User's question is: My are Kolmogorov's axioms necessary, and what would go wrong without them?
Context is: Comprehensive Theory of Probability
and
Statistics
1. Introduction
Probability and Statistics are core mathematical disciplines that provide the foundation for decision-making
under
uncertainty.
They
are
widely
applied

```

Figure 4.15: Streamlit Page Solution 5 Terminal

4.5 Generative Chat Page

The conversation begins with the conversation:

"Could you tell me: What is a vector space? Can you give an example?"

The tutor replies with a definition: a vector space is a mathematical structure consisting of a set of vectors equipped with two operations, vector addition and scalar multiplication. These operations must satisfy specific properties such as closure under addition, associativity, commutativity, identity, and inverses. The tutor uses the real number system \mathbb{R} as an initial example, noting how addition and scalar multiplication are defined.



Figure 4.16: Streamlit Page Solution 6 Part 1

Results and System Outputs

In the second exchange, we challenge part of the explanation, noting that real numbers are typically considered scalars in this context and asking whether a better example would be \mathbb{R}^n or polynomial spaces. The tutor agrees and clarifies that real numbers are indeed scalars and belong to a larger class called fields. It then provides two more appropriate examples of vector spaces:

- \mathbb{R}^n , the n -dimensional real coordinate space.
- The set of all polynomials with real coefficients.

Similarly, multiplying a real number r by a real number a results in the real number $c = r \cdot a$.

Let's denote this set of vectors as $V(\mathbb{R})$. The vector space properties include:

1. Closure under addition: If v and w are in $V(\mathbb{R})$, their sum $v + w$ is also in $V(\mathbb{R})$.
2. Associativity: For any constants a , b , and c , $(a + b) + c = a + (b + c)$.
3. Commutativity: For any vectors v and w in $V(\mathbb{R})$, $v + w = w + v$.
4. Identity element: For every vector v in $V(\mathbb{R})$, there exists an element 0 called the zero vector such that $v + 0 = v$.
5. Inverses: For each vector v in $V(\mathbb{R})$, there is another vector $-v$, called the additive inverse of v , such that $v + (-v) = 0$.

A more complex example is the set of all polynomials with real coefficients. Addition and scalar multiplication are performed as expected (e.g., $2x + 3 = 4x + 6$), and this set follows the same properties as $V(\mathbb{R})$.

Vector spaces find applications in various fields, including linear algebra, physics, and computer graphics. They provide a framework for describing and manipulating collections of objects (vectors) that can be combined or scaled to model complex phenomena.

Does this explanation clarify the concept of a vector space?

In your explanation, you said:

"If you have two numbers a and b , their sum is another real number $c = a + b$. Similarly, multiplying a real number r by a real number a results in the real number $c = r \cdot a$."

But aren't real numbers considered scalars, not vectors in this context? Shouldn't you instead refer to something like \mathbb{R}^n or polynomials as actual vector examples?

Vector spaces are mathematical structures consisting of a set V of elements called vectors, along with two essential operations: addition and scalar multiplication. These operations must satisfy specific properties for the set to qualify as a vector space.

Real numbers belong to a larger class called fields, which include addition, subtraction, and multiplication operations. Scalars are elements used in scalar multiplication, such as integers, fractions, or other fields like complex numbers.

Two examples of actual vector spaces include:

1. \mathbb{R}^n (n -dimensional real coordinate space): In this case, vectors represent points in an n -dimensional Cartesian plane. The operations are addition and scalar multiplication:
 - * $(a_1, a_2, \dots, a_n) + (b_1, b_2, \dots, b_n) = (a_1 + b_1, a_2 + b_2, \dots, a_n + b_n)$
 - * $r \cdot (a_1, a_2, \dots, a_n) = (ra_1, ra_2, \dots, ra_n)$
2. Polynomials: The set of all polynomials with real coefficients is also a vector space. The operations are addition and scalar multiplication:
 - * $(p(x) + q(x)) = p(x) + q(x)$
 - * $r \cdot p(x) = p(rx)$

Both examples satisfy the properties of a vector space, including closure under addition, associativity of addition, commutativity of addition, existence of zero vectors, and additive inverses.

Figure 4.17: Streamlit Page Solution 6 Part 2

In both cases, the tutor explains how the vector space operations apply and satisfy the required properties.

In the third part of the conversation, we point out a mistake in the notation used by the tutor:

$$“(p(x) + q(x)) = p(x) * q(x)”$$

Finally, we note that the use of multiplication is incorrect and that the correct form should involve addition. The tutor acknowledges the error and provides a refined explanation. It shows that adding two polynomials involves combining corresponding coefficients, and it illustrates this with a detailed example involving two specific polynomials and their derivatives.

PhronesisAI

Main sections

- Introduction
- Profile
- AI Tutor - Theoretical
- AI Tutor - RAG
- AI Tutor - Orchestrator**
- Documentation

* $(p(x) + q(x)) \cdot p(x) + q(x)$
 * $r = p(x) \cdot p(x)$

Both examples satisfy the properties of a vector space, including closure under addition, associativity of addition, commutativity of addition, existence of zero vectors, and additive inverses.

Vectors in \mathbb{R}^n represent points, while polynomials represent functions. Both are collections of numbers (scalars) that can be combined using vector operations.

Does this explanation clarify the distinction between scalars and vectors in the context of vector spaces?

You wrote:

" $(p(x) + q(x)) \cdot p(x) + q(x)$ "

But isn't that incorrect? Shouldn't the addition of two polynomials be written as: " $(p(x) + q(x)) + p(x) + q(x)$ " instead of using multiplication?

Could you clarify if this was a mistake or if there's a reason behind the use of \cdot instead of $+$?

Certainly! Here's a refined version of the explanation:

The correct expression for combining two polynomials $p(x)$ and $q(x)$ is:

" $(p(x) + q(x)) = p(x) + q(x)$ "

This means that you can directly add their coefficients term-by-term without using scalar multiplication.

Let's break it down:

- $p(x)$ represents a polynomial with coefficient 'a' and degree 'n'.
- $q(x)$ represents another polynomial with coefficient 'b' and degree 'n'.

The sum $(p(x) + q(x))$ would have the same leading term as $p(x)$, the same leading term as $q(x)$, but their respective coefficients are added together.

For instance:

- If $p(x) = 2x^3 + 3x^2$, then its derivative is $6x^2 + 6x$.
- If $q(x) = -x + 4$, then its derivative is -1 .

The sum of these polynomials is $(2x^3 + 3x^2) + (-x + 4) = 2x^3 + 3x^2 - x + 4$. The result is a new polynomial with the same leading term as each original polynomial, but their respective coefficients are added together.

This process of adding polynomials is fundamental in calculus and differential equations where derivatives and integrals are commonly used.

Does this explanation clarify the correct approach to combining polynomials?

Type a message...

Figure 4.18: Streamlit Page Solution 6 Part 3

Chapter 5

Conclusions, Limitations, and Future Work

This chapter summarizes the outcomes of the development, identifies the current limitations of the application, and proposes directions for future improvements.

5.1 Conclusions

Before concluding this project, it is important to briefly recap the main milestones covered throughout this Master's Thesis and the implementation methodology adopted. This summary enables a clear understanding of the process followed and highlights the contributions made, as well as the potential lines of work that could be pursued in the future.

The project has focused on the development of a local educational assistant powered by intelligent agents and large language models. The initial chapters introduced the motivation behind the system: to assist students and educators through an AI-powered platform capable of adapting to various academic needs such as theoretical explanations, code assistance, quiz evaluation, and document-based answers using retrieval-augmented generation.

From the beginning, the project's architecture was designed with modularity and maintainability in mind. The choice of Streamlit for the frontend enabled rapid prototyping and a clear user interface, while the integration with LM Studio allowed local deployment of LLMs, avoiding reliance on external APIs and ensuring full data privacy.

Throughout the implementation, the AutoGen framework played a key role in managing the logic of each agent, including user simulators, coding assistants, reviewers, and RAG components. Each agent was assigned a specific task within a group or sequential architecture, creating a flexible pipeline adaptable to multiple educational workflows.

The documentation chapter demonstrated that each part of the system operates correctly. The LM Studio logs show that the model responds accurately, with each page triggering the appropriate agents depending on the user's selection. From theoretical queries to executable code and adaptive quizzes, all functionalities were integrated successfully.

In conclusion, this work demonstrates the feasibility and educational value of combining modular intelligent agents with locally hosted LLMs. The results support the idea that such systems can effectively assist both learning and teaching tasks. Personally, this project has been a deeply enriching experience that has allowed me to grow both technically and academically. I strongly believe that platforms like this could play a valuable role in the future of personalized education, and I would be excited to continue developing and expanding this idea beyond the scope of this thesis.

5.2 Future Work

While the core objectives have been achieved, several directions remain open for future development:

- **Web deployment:** Migrating the application from a local Streamlit environment to a deployed platform with user authentication, persistent storage, and support for multi-user sessions.
- **Advanced agent collaboration:** Expanding the use of hierarchical group chats, conversational memory, and long-term agent planning to support more complex learning paths and interactive scenarios.
- **Instructor dashboards:** Implementing dedicated dashboards for educators to track student performance, export reports, and gain insights through aggregated analytics.
- **Adaptive learning algorithms:** Introducing mechanisms that dynamically adjust difficulty and explanation depth based on each user's history and performance over time.
- **Personalization for both students and instructors:** Enhancing customization so that professors can organize content not only by academic subject but also by specific topics within their own course. Students, on the other hand, would benefit from an improved document management system that supports local note uploads and task organization, making the assistant a more versatile and personalized companion in their academic journey. The core idea remains to support, complement, and simplify the adaptation to AI-assisted learning and teaching on both sides of the educational process.
- **Switching to hosted LLMs via API keys:** Transitioning from LM Studio to production-grade language model services such as OpenAI, allowing for scalable, stable, and monetized access through monthly billing, thereby removing local hardware dependencies.

Conclusions, Limitations, and Future Work

- **Backend API integration:** Separating the frontend and backend into independent services through the development of a dedicated backend API, which would improve modularity and deployment capabilities across cloud platforms.
- **Multilingual and multimodal interaction:** Extending language support beyond English and Spanish, and incorporating additional modalities such as voice-based interaction, image generation, or document parsing to enhance engagement and accessibility.
- **Enhanced security through input validation agents:** Adding a dedicated agent responsible for content moderation that detects and blocks queries unrelated to the academic domain. This would prevent inappropriate or off-topic questions from being processed, ensuring the assistant is used strictly within its educational scope.
- **Improved prompt engineering and hallucination control:** Refining the construction of system prompts and response templates to reduce the likelihood of hallucinated or imprecise outputs, thereby increasing trustworthiness and pedagogical reliability.

These improvements would allow the platform to evolve from a prototype into a comprehensive educational ecosystem, capable of adapting to a wider range of users, institutions, and learning contexts.

Annex

Autogen Agent's Functionality

Before diving into the integration of Autogen with Streamlit at the code level, it is necessary to describe the agents used in the system, how they are configured, and how their prompts are defined. Each agent is instantiated with a specific name, a system message that acts as a prompt to define its behavior, and a configuration that determines how it interacts with the underlying language model.

The agent setup is defined within the `genai_agent_setup.py` file, while the detailed explanation of each agent and its role has been provided in Section 3.4. Each agent is instantiated programmatically, and the corresponding functions and object instances are organized clearly in this module.

Prompt configurations for all agents are within the `genai_agent_config.yaml` file. This approach promotes modularity and allows prompt engineering to be performed independently of the core logic, enabling rapid experimentation and refinement of agent behavior.

In addition, commonly used constants, such as subject names, keys, and other configuration parameters, are defined in the `common_constants.py` file. This separation of concerns improves code maintainability and readability across the system.

It serves as the centralized location for system-wide configuration parameters. One of its core components is the definition of `llm_config`, which specifies the language model to be used by the agents. For instance, when working locally with LM Studio, the configuration includes the `base_url`, the model identifier, and a placeholder `api_key`. If an OpenAI model were to be used instead, the configuration would only require the API key and the model name, with no need for a base URL.

```
1 # Model configurations
2 self.llm_config = {
3     "config_list": [
4         {
5             "base_url": "http://127.0.0.1:1234/v1", # Ensure LM
6                 Studio is running
7             "model": "NousResearch/Hermes-3-Llama-3.1-8B-GGUF",
8             "api_key": "hermes-3-llama-3.1-8b"
9         }
10     ],
11 }
```

Conclusions, Limitations, and Future Work

```
12 # RAG-related constants
13 self.docs_path = str(Path(__file__).resolve().parent.parent.parent /
    "outputs" / "rag_files" / "user_uploaded.pdf")
14 self.chunk_token_size = 4000
15 self.embedding_model = 'all-mpnet-base-v2'
16
17 # Number of max rounds for the orchestrator group chat
18 self.max_round_orchestrator = 20
```

Listing 1: Example configuration in `common_constants.py`

This file also contains essential parameters for other modules, such as those used for RAG. Examples include the document path used for context retrieval `docs_path`, the chunk size in tokens for splitting documents `chunk_token_size`, and the embedding model employed for vector representation `all-mpnet-base-v2`.

Additionally, global interaction limits, such as the maximum number of message exchanges in orchestrator group chats `max_round_orchestrator`, are also defined here. This modular approach allows the configuration to be adjusted easily without modifying core logic, fostering maintainability and flexibility across the system.

Once the constants are defined, the prompts for each agent are specified through the YAML configuration file `genai_agent_config.yaml`. This file structures the behavior of agents by associating them with a unique name and a corresponding system message that serves as their prompt. This setup decouples prompt logic from implementation logic, ensuring modularity and simplifying prompt updates.

Each agent is defined under a relevant category, like `assistant_agents`, with entries that specify both the agent's identifier and its associated prompt. For example, the following configuration defines the `QuestionRewriterAgent`, whose purpose is to rephrase academic questions provided by the user. The system message defines the expected inputs and the behavior that the agent must follow.

```
1 assistant_agents:
2   - name: 'QuestionRewriterAgent'
3     system_message: "
4 You are the QuestionRewriterAgent, a specialized AI designed to
5 reformulate user-submitted academic questions.
6 You will receive two inputs:
7 - Subject: the academic topic of the question (e.g., Linear Algebra,
8   Calculus, Probability, Databases, Python Programming)
9 - Raw Question: a possibly vague or unstructured question written by
10 the user
11 Your sole task is to rewrite the question exactly as requested by
12 the user, for the next agent to explain it."
```

Listing 2: Prompt definition for `QuestionRewriterAgent`

With both the constants and prompt configurations defined, the actual agent instances are created within the `genai_agent_setup.py` module. This module acts as the centralized entry point where each agent is instantiated using the configurations previously discussed. The instantiation process leverages factory methods defined within a utility class to encapsulate the logic of agent creation.

For example, the `QuestionRewriterAgent` is created using the following function, which receives the agent's name, system message, language model configuration, termination condition, and code execution setup:

```

1 # 2. QuestionRewriter
2 self.question_rewriter = self.agent_functions.create_assistant_agent
  (
3     name=self.yaml_config['assistant_agents'][0]['name'],
4     system_message=self.yaml_config['assistant_agents'][0]['
      system_message'],
5     llm_config=self.common_constants_config.llm_config,
6     is_termination_msg=self.agent_functions.termination_msg,
7     code_execution_config=self.common_constants_config.
      code_execution_configuration,
8 )

```

Listing 3: Agent instantiation for `QuestionRewriterAgent`

The `create_assistant_agent` method wraps the creation of an `AssistantAgent` as follows:

```

1 def create_assistant_agent(self, name, system_message, llm_config,
  is_termination_msg, code_execution_config):
2     return AssistantAgent(
3         name=name,
4         system_message=system_message,
5         llm_config=llm_config,
6         is_termination_msg=is_termination_msg,
7         code_execution_config=code_execution_config,
8     )

```

Listing 4: `create_assistant_agent` function definition

Other agents, such as those capable of executing code, are instantiated similarly. For example, the following agent is designed to run Python code using a local command-line executor:

```

1 self.python_code_executor = self.agent_functions.
  create_assistant_agent(
2     name=self.yaml_config['assistant_agents'][14]['name'],
3     system_message=self.yaml_config['assistant_agents'][14]['
      system_message'],
4     llm_config=self.common_constants_config.llm_config,
5     is_termination_msg=self.agent_functions.termination_msg,
6     code_execution_config={
7         "executor": LocalCommandLineCodeExecutor(
8             work_dir=self.common_constants_config.output_dir_code,

```

Conclusions, Limitations, and Future Work

```
9         virtual_env_context=virtual_env_context
10     ),
11 },
12 )
```

Listing 5: PythonCodeExecutor agent instantiation

Similarly, retrieval-based agents are defined through the following function. These agents are responsible for fetching relevant documents given a user query:

```
1 self.retriever = self.agent_functions.
  create_retrieve_user_proxy_agent (
2     name=self.yaml_config['retriever_agents'][0]['name'],
3     system_message=self.yaml_config['retriever_agents'][0]['
  system_message'],
4     is_termination_msg=self.agent_functions.termination_msg,
5     human_input_mode=self.common_constants_config.
  human_input_mode_never,
6     max_consecutive_auto_reply=self.common_constants_config.
  max_consecutive_reply,
7     retrieve_config={
8         "task": self.common_constants_config.task,
9         "docs_path": self.common_constants_config.docs_path,
10        "chunk_token_size": self.common_constants_config.
  chunk_token_size,
11        "embedding_model": self.common_constants_config.
  embedding_model,
12        "collection_name": self.common_constants_config.
  collection_name,
13        "get_or_create": self.common_constants_config.get_or_create,
14    },
15    code_execution_config=self.common_constants_config.
  code_execution_configuration,
16 )
```

Listing 6: Retriever agent instantiation

The function that instantiates this retrieval agent is shown below:

```

1 def create_retrieve_user_proxy_agent(self, name, system_message,
   is_termination_msg, human_input_mode,
2
   max_consecutive_auto_reply,
   retrieve_config,
   code_execution_config):
3
   return RetrieveUserProxyAgent (
4     name=name,
5     system_message=system_message,
6     is_termination_msg=is_termination_msg,
7     human_input_mode=human_input_mode,
8     max_consecutive_auto_reply=max_consecutive_auto_reply,
9     retrieve_config=retrieve_config,
10    code_execution_config=code_execution_config,
11    )

```

Listing 7: create_retrieve_user_proxy_agent function definition

Once the individual agents are defined and instantiated with their respective roles and prompts, they can be grouped into group chat structures. This is done via an architecture where agents are organized based on communication rules, the number of allowed message rounds, and dialogue permissions.

For example, the following group chat involves a subject-specific workflow with two agents: a `QuestionRewriterAgent` and an `ExplainerAgent`. The structure indicates that each agent is allowed to communicate with the other across a fixed number of interaction rounds:

```

1 subject_groupchat = agent_setup.agent_functions.create_groupchat (
2     [question_rewriter, explainer_agent],
3     3, # Number of communication rounds
4     {
5         question_rewriter: [explainer_agent],
6         explainer_agent: [question_rewriter],
7     },
8     "allowed", # Interaction policy
9 )

```

Listing 8: Creating a group chat with two agents

In this example:

- The first parameter specifies the list of agent instances included in the group chat.
- The second parameter defines the number of dialogue rounds that will be executed before termination.
- The third parameter establishes a communication graph—a dictionary where each agent is mapped to the list of agents it can send messages to.
- The fourth parameter controls whether this message passing behavior is "allowed" or restricted under certain conditions.

Streamlit Code Functionality

The main components of the code for the main pages, how the application is executed, the agent architectures, and all constants and other options that can be edited within the application are shown in this section.

First of all, the application is executed using the terminal command `streamlit run app.py`. This file simply contains the Streamlit pages to be executed, along with the styles being used.

As shown in the following code:

```
1 # Apply custom styling to Streamlit components
2 apply_custom_styling()
3
4 st.session_state["pages"] = load_pages(st.session_state["config"].
    pages)
5
6 pg = st.navigation(st.session_state["pages"])
7 pg.run()
```

Listing 9: Initial setup and page loading

After this, the pages are mainly executed from within the `src/pages_classes/` directory. In order for these to be displayed, they must be included in the file named `frontend_config.json`, inside the dictionary that holds the main pages, such as:

```
1 "pages": {
2   "Main_sections": [
3     {
4       "import_path": "src.pages_classes.introduction",
5       "class": "Introduction",
6       "title": "Introduction",
7       "icon": ":material/home:"
8     },
```

Listing 10: Main pages in the frontend configuration

Once we understand how a page is included, the process is the same for the rest in case we want to add a new one. This configuration defines the title, class, and icon for the first page, namely `introduction.py`.

The main functionality involves checking whether there is an associated profile for the user. If such a file with the user data does not exist, the application will prompt the student to input it. If it does exist, the page will display the title and the project's README.md file.

```
1 class Introduction:
2     def __init__(self):
3         if not self.profile_exists():
4             self.request_user_profile()
5         else:
6             self.display_something()
7             self.show_readme()
```

Listing 11: Introduction page logic

After detailing the structure and inclusion of pages, the next step is to review the remaining pages and the high-level code associated with them. Specific attention will later be given to the agent architecture functions.

The profile page contains the following code:

```
1 class Profile:
2     def __init__(self):
3         self.display_profile()
4         self.display_performance_overview()
```

Listing 12: Constructor of the Profile class

This class initializes the profile view, displaying both the user's personal information and an overview of their performance based on quiz results.

The user's profile data is retrieved and shown using the following logic:

```
1 @staticmethod
2 def load_user_info():
3     if os.path.exists(PROFILE_FILE):
4         try:
5             with open(PROFILE_FILE, "r", encoding="utf-8") as f:
6                 lines = [line.strip() for line in f.readlines()]
7
8                 # ... More Code ...
9
10            except Exception as e:
11                st.error(f"Error_loading_profile:_{e}")
12    return None
```

Listing 13: Loading the user profile

Conclusions, Limitations, and Future Work

```
1 def display_profile(self):
2     st.title("User_Profile")
3     user_info = self.load_user_info()
4     if user_info:
5         st.write(f"###_Name:_{user_info['name']}")
6         st.write(f"###_{user_info['year']}_{user_info['degree']}")
7     else:
8         st.warning("No_user_profile_found._Please_complete_the_
                    introduction_form.")
```

Listing 14: Displaying the user profile

The method below generates a performance summary by reading the results of past quizzes and presenting the data visually through auxiliary functions:

```
1 def display_performance_overview(self):
2     st.subheader("Performance_Overview")
3
4     # ... More Code ...
5
6     if total_questions == 0:
7         st.warning("No_quiz_data_available_to_calculate_performance.
                    ")
8         return
9
10    accuracy = total_correct / total_questions
11    self.display_horizontal_bar(accuracy)
12    self.display_performance_badge(accuracy)
13    self.display_yes_no_chart(difficulty_summary)
14
15    difficulty_tabs = st.tabs(["Easy", "Medium", "Difficult"])
16    with difficulty_tabs[0]:
17        self.display_subject_histogram("easy")
18    with difficulty_tabs[1]:
19        self.display_subject_histogram("medium")
20    with difficulty_tabs[2]:
21        self.display_subject_histogram("difficult")
```

Listing 15: Displaying quiz performance overview

The file depends on the number of correct answers obtained in the GenAI quiz, which will be discussed in more detail later.

The AI Tutor page is structured into three main sections, accessible via tabs:

```
1 tabs = st.tabs([
2     "LLM_Theoretical_Explanation:_material/network_intelligence:",
3     "LLM_Practical_Explanation:_material/network_intelligence:",
4     "GenAI_Quiz:_material/quiz:"
5 ])
```

Listing 16: Tab definition in the AI Tutor page

Within the theoretical section, the following logic is implemented to handle user queries using an LLM-based architecture:

```

1 selected_subject = st.selectbox("Select_a_Subject:", subjects, key="
  explanation_subject")
2 formatted_subject = selected_subject.lower().replace("_", "-")
3
4 question = st.text_area("Enter_your_question")
5
6 ask_button = st.button("Generate_explanation")
7
8 if question and ask_button:
9
10     # More code ...
11
12     start_explanation_chat(message, formatted_subject)

```

Listing 17: LLM explanation interaction

The backend logic for this functionality resides in the file `streamlit_aux_genai.py`, with the key function `start_explanation_chat` defined as follows:

```

1 def start_explanation_chat(message, subject):
2
3     # DOCSTRING...
4
5     assistant_user_proxy = StreamlitUserProxyAgentWrite(
6         # More Code ...
7     )
8
9     subject_to_explainer = {
10         "linear_algebra": agent_setup.linear_algebra_explainer,
11         "calculus": agent_setup.calculus_explainer,
12         "probability_and_statistics": agent_setup.
13             probability_explainer,
14         "python_programming": agent_setup.python_explainer,
15         "databases": agent_setup.databases_explainer,
16     }
17
18     try:
19         explainer_agent = subject_to_explainer[subject]
20         subject_groupchat = create_subject_groupchat(explainer_agent
21             )
22     except KeyError:
23         raise ValueError(
24             f"Invalid_subject_{subject}_specified."
25             "Use_one_of:" + ",".join(subject_to_explainer.keys())
26             + "."
27         )
28
29     manager = agent_setup.agent_functions.create_manager(

```

Conclusions, Limitations, and Future Work

```
27     subject_groupchat,  
28     constants_config.llm_config  
29 )  
30  
31 explanation_executor = agent_setup.agent_functions.  
    create_society_of_mind(  
32     "SubjectExplanationAgent", manager, False  
33 )  
34  
35 async def start_chat():  
36     # Start chat ...  
37  
38 return asyncio.run(start_chat())
```

Listing 18: Asynchronous explanation chat using subject-specific agents

A custom subclass of Autogen's `UserProxyAgent`, `StreamlitUserProxyAgentWrite`, is used to manage asynchronous input from Streamlit interfaces. Once the user submits a question and selects a subject, the system launches a group chat involving the following agent roles:

- **Rewriting Agent:** Reformulates the question to improve clarity and context.
- **Subject Expert Agent:** Generates a subject-specific explanation aligned with the user's query.
- **Groupchat Manager:** Coordinates interactions and ensures consistency between the reformulated question and the final answer.

In the *AI Tutor Practical* case, the architecture follows the same group chat-based structure described previously, but this time leveraging agents capable of executing code. These agents are configured to support programming tasks and computational exercises.

In the *GenAI Quiz* case, once the user selects a subject and difficulty level, a new tab is displayed where the corresponding question is presented, and the correction process is initiated. This selection is implemented using Streamlit's column layout:

```
1 cols = st.columns(2)  
2  
3 with cols[0]:  
4     selected_subject = st.selectbox("Select_a_Subject:", subjects,  
    key="quiz_subject_tab")  
5  
6 with cols[1]:  
7     selected_difficulty = st.selectbox("Select_a_Difficulty:", ["  
    Easy", "Medium", "Difficult"], key="quiz_difficulty")
```

Listing 19: Subject and difficulty selection interface

After the user interacts with the quiz interface, the system initiates the correction phase by calling a dedicated function:

```
1 result = start_correction_chat(final_question, formatted_subject)
```

Listing 20: Starting the correction phase

Once the correction message is received—whether it confirms the user answered correctly or not—the following function is executed to update performance records:

```
1 update_quiz_grade_file(formatted_difficulty, formatted_subject,  
    user_messages)
```

Listing 21: Updating grade files after correction

This function analyzes the assistant’s feedback message and determines whether the answer was correct. Based on this outcome, it updates the grade tracking files associated with each subject and difficulty level. These same files are later used by the user profile to generate visual performance analytics, such as progress graphs and score summaries.

The *AI Tutor - RAG* module follows a similar structure to the previous examples but is tailored for RAG tasks. In this case, the system is driven by specific prompts designed to extract relevant information from user-uploaded documents. The input mechanism uses a file uploader component from Streamlit:

```
1 uploaded_file = st.file_uploader("Upload_a_document_for_this_subject  
    ", type=["pdf", "txt"])
```

Listing 22: File uploader for subject-specific document

Once a file is uploaded, the document is preprocessed and indexed using the parameters defined in the YAML configuration (e.g., chunk size, embedding model, and collection name). The uploaded content is then made available to a retrieval agent capable of accessing semantically similar chunks during the dialogue.

The *AI Tutor - Orchestrator* page provides an open-ended conversational interface that allows users to engage in dynamic discussions with their GenAI-powered tutor. This interface is designed using Streamlit and integrates agent memory through the use of session state variables.

The application layout includes a title and an informational message, followed by logic that initializes and manages a persistent agent instance:

```
1 st.title("AI_Tutor_-_Orchestrator")  
2 st.info(":material/info:_Have_a_conversation_with_your_GenAI_Tutor."  
    )  
3  
4 # Initialize the agent and message history  
5 if "agent" not in st.session_state:  
6     st.session_state["agent"] = Agent()  
7  
8 if "messages" not in st.session_state:  
9     st.session_state["messages"] = []
```

Listing 23: Initialization of agent and session state

A reset button is provided to clear the conversation history and restart the dialogue:

```
1 if st.button(":material/refresh:_Restart_Conversation"):  
2     st.session_state["agent"].clear_history()  
3     st.session_state["messages"] = []
```

Conclusions, Limitations, and Future Work

```
4 st.rerun()
```

Listing 24: Reset conversation state

The main chat interface displays previous messages stored in the session state and listens for new user input through the Streamlit chat input widget:

```
1 # Display chat history
2 for message in st.session_state["messages"]:
3     with st.chat_message(message["role"]):
4         st.markdown(message["content"])
5
6 # Process new user input
7 prompt = st.chat_input("Type a message...")
8 if prompt:
9     st.session_state["messages"].append({"role": "user", "content":
10         prompt})
11     with st.chat_message("user"):
12         st.markdown(prompt)
13
14     response = st.session_state["agent"].chat_sync(prompt)
15
16     st.session_state["messages"].append({"role": "assistant", "
17         content": response})
18     with st.chat_message("assistant"):
19         st.markdown(response)
```

Listing 25: Handling chat input and responses

Finally, regarding the display of the documentation, the process is straightforward: the relative path of the documentation is read and rendered at the highest possible resolution.

```
1 class Documentation:
2     def __init__(self):
3         self.display_title()
4         self.show_documentation()
5
6     @staticmethod
7     def display_title():
8         st.title("TFM_Aaron_Moyano_Alcacer_Documentation:_material/
9             menu_book:")
10
11     @staticmethod
12     def show_documentation():
13         pdf_viewer(Path(__file__).resolve().parent.parent.parent / "
14             TFM_AARON_MOYANO.pdf", 1920)
```

Listing 26: Displaying the TFM documentation

© Aaron Moyano Alcacer - 2025

Some rights reserved

This document is distributed under the Creative Commons "*Attribution-ShareAlike 4.0 International*" license available at: <https://creativecommons.org/licenses/by/4.0/>

Bibliography

- [1] E. Strickland, "The turbulent past and uncertain future of artificial intelligence," *IEEE Spectrum*, Sep. 2021. [Online]. Available: <https://spectrum.ieee.org/history-of-ai>.
- [2] J. M. Mira, "Symbols versus connections: 50 years of artificial intelligence," *Neurocomputing*, vol. 71, no. 4, pp. 671–680, 2008, ISSN: 0925-2312. DOI: 10.1016/j.neucom.2007.06.009. [Online]. Available: <https://www.sciencedirect.com/science/article/pii/S0925231207003451>.
- [3] News Text Area, *The incredible life of john mccarthy: Inventor of artificial intelligence*, <https://newstextarea.com/the-incredible-life-of-john-mccarthy-inventor-of-artificial-intelligence>, 2024.
- [4] A. Newell and H. A. Simon, "Computer science as empirical inquiry: Symbols and search," *Communications of the ACM*, vol. 19, no. 3, pp. 113–126, 1976. DOI: 10.1145/360018.360022. [Online]. Available: <https://dl.acm.org/doi/10.1145/360018.360022>.
- [5] M. Mitchell, "Why ai is harder than we think," *arXiv*, 2021. [Online]. Available: <https://arxiv.org/abs/2104.12871>.
- [6] F. Rosenblatt, "The perceptron: A probabilistic model for information storage and organization in the brain," *Psychological Review*, vol. 65, no. 6, pp. 386–408, Nov. 1958, ISSN: 0033-295X. DOI: 10.1037/h0042519.
- [7] F. Rosenblatt, "The perceptron: A probabilistic model for information storage and organization in the brain," *Psychological Review*, vol. 65, no. 6, pp. 386–408, 1958. DOI: 10.1037/h0042519. [Online]. Available: https://deeplearning.cs.cmu.edu/S24/document/readings/Rosenblatt_1959-09865-001.pdf.
- [8] M. Minsky and S. Papert, *Perceptrons: An Introduction to Computational Geometry*, Expanded. Cambridge, MA: MIT Press, 1988, ISBN: 978-0-262-63111-2. [Online]. Available: <https://direct.mit.edu/books/monograph/3132/PerceptronsAn-Introduction-to-Computational>.
- [9] P. J. Werbos, "Backpropagation through time: What it does and how to do it," *Proceedings of the IEEE*, vol. 78, no. 10, pp. 1550–1560, 1990. DOI: 10.1109/5.58337. [Online]. Available: https://www.researchgate.net/publication/2984354_Backpropagation_through_time_what_it_does_and_how_to_do_it.
- [10] A. Krizhevsky, I. Sutskever, and G. E. Hinton, "Imagenet classification with deep convolutional neural networks," in *Advances in Neural Information Processing Systems*, vol. 25, Curran Associates, Inc., 2012, pp. 1097–1105. [Online]. Available: https://www.researchgate.net/publication/267960550_ImageNet_Classification_with_Deep_Convolutional_Neural_Networks.

-
- [11] J. Clark, "Alphago wins match against top go player," *IEEE Spectrum*, 2016. [Online]. Available: <https://spectrum.ieee.org/alphago-wins-match-against-top-go-player>.
- [12] K.-T. Tran, D. Dao, M.-D. Nguyen, Q.-V. Pham, B. O'Sullivan, and H. D. Nguyen, "Multi-agent collaboration mechanisms: A survey of llms," *arXiv preprint arXiv:2501.06322*, 2025. [Online]. Available: <https://arxiv.org/abs/2501.06322>.
- [13] Q. Wu, G. Bansal, *et al.*, "Autogen: Enabling next-gen llm applications via multi-agent conversation," in *Proceedings of the Conference on Language Models (COLM)*, Aug. 2024. [Online]. Available: <https://www.microsoft.com/en-us/research/publication/autogen-enabling-next-gen-llm-applications-via-multi-agent-conversation-framework/>.
- [14] OpenAI, *Gpt-4 technical report*, <https://arxiv.org/abs/2303.08774>, 2023.
- [15] H. Touvron, T. Lavril, G. Izacard, *et al.*, *Llama: Open and efficient foundation language models*, <https://arxiv.org/abs/2302.13971>, 2023.
- [16] W. Xing, N. Nixon, S. Crossley, *et al.*, "The use of large language models in education," *International Journal of Artificial Intelligence in Education*, 2025. DOI: 10.1007/s40593-025-00457-x. [Online]. Available: <https://doi.org/10.1007/s40593-025-00457-x>.
- [17] Zep, *Introduction to ai agents*, <https://www.getzep.com/ai-agents/introduction-to-ai-agents>, 2023.
- [18] L. Weng, *Agents: An in-depth look*, <https://lilianweng.github.io/posts/2023-06-23-agent/>, 2023.
- [19] Prompting Guide, *Components of ai agents*, <https://www.promptingguide.ai/agents/components>, 2023.
- [20] LangChain, *Agentic concepts in langgraph*, https://langchain-ai.github.io/langgraph/concepts/agentic_concepts/, 2024.
- [21] S. Wang, T. Xu, H. Li, *et al.*, "Large language models for education: A survey and outlook," *arXiv preprint arXiv:2403.18105*, 2024. [Online]. Available: <https://arxiv.org/abs/2403.18105>.
- [22] Khan Academy, *Khanmigo: Your ai guide for learning*, <https://www.khanmigo.ai/>, 2025.
- [23] IBM Research, *What is retrieval-augmented generation (rag)?* <https://research.ibm.com/blog/retrieval-augmented-generation-RAG>, 2023.
- [24] Amazon Web Services, *What is retrieval-augmented generation (rag)?* <https://aws.amazon.com/what-is/retrieval-augmented-generation/>, 2024.
- [25] Y. Gao, Y. Xiong, X. Gao, *et al.*, "Retrieval-augmented generation for large language models: A survey," *arXiv preprint arXiv:2312.10997*, 2023. [Online]. Available: <https://arxiv.org/abs/2312.10997>.
- [26] P. Lewis, E. Perez, A. Piktus, *et al.*, "Retrieval-augmented generation for knowledge-intensive nlp tasks," in *Advances in Neural Information Processing Systems*, vol. 33, Curran Associates, Inc., 2020, pp. 9459–9474. [Online]. Available: <https://proceedings.neurips.cc/paper/2020/file/6b493230205f780e1bc26945df7481e5-Paper.pdf>.
- [27] World Bank, *From chalkboards to chatbots: Transforming learning in nigeria*, <https://blogs.worldbank.org/en/education/From-chalkboards-to-chatbots-Transforming-learning-in-Nigeria>, 2023.
- [28] J. Ryttilahti and E. Lökkila, "Chatgpt as tutor? a case study on competitive programming," in *Education and New Developments 2024 – Volume 2*, InScience

BIBLIOGRAPHY

- Press, 2024, pp. 91–95. DOI: 10.36315/2024v2end019. [Online]. Available: <https://doi.org/10.36315/2024v2end019>.
- [29] M. Guettala, *Generative artificial intelligence in education: Advancing adaptive and personalized learning*, https://www.researchgate.net/publication/383303500_Generative_Artificial_Intelligence_in_Education_Advancing_Adaptive_and_Personalized_Learning, 2023.
- [30] A. Vanzo, S. P. Chowdhury, and M. Sachan, “Gpt-4 as a homework tutor can improve student engagement and learning outcomes,” *arXiv preprint arXiv:2409.15981*, 2024. [Online]. Available: <https://arxiv.org/abs/2409.15981>.
- [31] Microsoft Research, *Autogen 0.2: An open-source programming framework for agentic ai*, <https://microsoft.github.io/autogen/0.2/>, 2023.
- [32] W. Wang, D. Han, X. Luo, and D. Li, “Addressing signal delay in deep reinforcement learning,” in *Proceedings of the 12th International Conference on Learning Representations (ICLR)*, Microsoft Research, May 2024.
- [33] Microsoft AutoGen Team, *Multi-agent conversation framework | autogen 0.2*, https://microsoft.github.io/autogen/0.2/docs/Use-Cases/agent_chat/, 2024.
- [34] Q. Wu, G. Bansal, J. Zhang, *et al.*, “Autogen: Enabling next-gen llm applications via multi-agent conversation,” in *Proceedings of the 12th International Conference on Learning Representations (ICLR)*, May 2024. [Online]. Available: <https://arxiv.org/abs/2308.08155>.
- [35] Microsoft AutoGen Team, *Tool use | autogen 0.2*, <https://microsoft.github.io/autogen/0.2/docs/tutorial/tool-use/>, 2024.
- [36] Microsoft AutoGen Team, *Autogen studio — autogen*, <https://microsoft.github.io/autogen/stable/user-guide/autogenstudio-user-guide/index.html>, 2025.
- [37] Microsoft AutoGen Team, *Autogen: A framework for building ai agents and applications*, <https://microsoft.github.io/autogen/stable/index.html>, 2025.
- [38] A. Fourney and Q. Wu, *Autogenbench – a tool for measuring and evaluating autogen agents*, <https://microsoft.github.io/autogen/0.2/blog/2024/01/25/AutoGenBench/>, Jan. 2024.
- [39] S. Hong, X. Zheng, J. Chen, *et al.*, “Metagpt: Meta programming for multi-agent collaborative framework,” *arXiv preprint arXiv:2308.00352*, 2023. [Online]. Available: <https://arxiv.org/abs/2308.00352>.
- [40] C. Qian, W. Liu, H. Liu, *et al.*, “Chatdev: Communicative agents for software development,” *arXiv preprint arXiv:2307.07924*, 2023. [Online]. Available: <https://arxiv.org/abs/2307.07924>.
- [41] G. Li, H. A. A. K. Hammoud, H. Itani, D. Khizbullin, and B. Ghanem, “Camel: Communicative agents for “mind” exploration of large language model society,” *arXiv preprint arXiv:2303.17760*, 2023. [Online]. Available: <https://arxiv.org/abs/2303.17760>.
- [42] A. Fourney, G. Bansal, H. Mozannar, V. Dibia, and S. Amershi, *Magentic-one: A generalist multi-agent system for solving complex tasks*, <https://www.microsoft.com/en-us/research/articles/magentic-one-a-generalist-multi-agent-system-for-solving-complex-tasks/>, Nov. 2024.
- [43] Microsoft Research, *Autogen 0.2 documentation: Notebooks*, <https://microsoft.github.io/autogen/0.2/docs/notebooks>, 2024.

- [44] Microsoft AutoGen Team, *Magentic-one — autogen*, <https://microsoft.github.io/autogen/dev/user-guide/agentchat-user-guide/magentic-one.html>, 2025.
- [45] LangChain, *Langgraph tutorials (documentation)*, <https://github.com/langchain-ai/langgraph/tree/main/docs/docs/tutorials>, 2024.
- [46] Python Software Foundation. “Python programming language – official website.” (2024), [Online]. Available: <https://www.python.org/>.
- [47] Streamlit Inc. “Streamlit — a faster way to build and share data apps.” (2025), [Online]. Available: <https://streamlit.io/>.
- [48] Element Labs, Inc. “Lm studio – discover, download, and run local llms.” (2025), [Online]. Available: <https://lmstudio.ai/>.
- [49] JetBrains s.r.o. “Pycharm – the python ide.” (2025), [Online]. Available: <https://www.jetbrains.com/pycharm/>.