Politecnico di Torino

Master's Degree in Data Science and Engineering

Academic Year 2024/2025

Graduation Session October 2025

AI Agents Leveraging RAG and MCP for Insurance Knowledge Management and Enterprise Workflow Automation

Case Studies at Reale Mutua Assicurazioni

Supervisor

Prof. Luca CAGLIERO

Company Supervisor

Dott. Felipe OPERTI

Candidate

Daniele ERCOLE

Abstract

This thesis explores how advanced Artificial Intelligence (AI) techniques can improve knowledge management in large organizations, with the use of Retrieval-Augmented Generation (RAG) and the Model Context Protocol (MCP). The work was carried out at Reale Mutua Assicurazioni, within the company's Data Science Center of Excellence, and focused on two experimental projects. The first, the POD Assistant, is a system of AI agents that helps project teams manage documentation and interact with platforms such as Azure DevOps. The second, the CGA Copilot, is a chatbot that makes it easier to interpret and use long and complex insurance documents.

By combining multi-agent collaboration, MCP to connect different tools in a scalable way and hybrid RAG pipelines for more accurate retrieval, the research shows that these solutions can deliver more precise, contextualized, and transparent answers than traditional approaches.

The findings demonstrate how, with the use of RAG and MCP, advanced AI can be adapted to real corporate needs, balancing innovation and efficiency, while opening new opportunities for agent-based applications in the insurance sector.

Acknowledgements

Vorrei esprimere la mia gratitudine a tutte le persone che, in modi diversi, hanno reso possibile questo percorso di studi e la realizzazione di questa tesi.

Desidero ringraziare innanzitutto il Prof. Luca Cagliero per la disponibilità dimostrata durante il percorso di tesi e per i preziosi consigli che hanno guidato il mio lavoro.

Un grazie speciale al mio tutor aziendale Dott. Felipe Operti per il supporto costante e i preziosi consigli che, durante il tirocinio in Reale Mutua, mi hanno aiutato ad affrontare nuove sfide e a crescere professionalmente. Un ringraziamento va anche a tutto il team DSCoE per l'accoglienza, la disponibilità e la collaborazione, che hanno reso questa esperienza davvero significativa e arricchente.

Alla mia famiglia, a mia mamma, mio papà e mio fratello, va il mio più grande grazie per il sostegno incondizionato, la fiducia e l'incoraggiamento che mi hanno sempre accompagnato, permettendomi di raggiungere questo traguardo.

Un grazie enorme va alla mia seconda famiglia, i miei amici Ale, Andre, Anna, Ema, Enea, Gaia e Nico; la vostra amicizia ha reso questi anni indimenticabili e ogni momento insieme è stato davvero prezioso. Un ringraziamento speciale anche ai miei coinquilini, con cui ho condiviso quotidianità e avventure universitarie, e che hanno avuto la pazienza di sopportarmi in tutti questi anni.

Un grazie sincero va anche a tutti i miei compagni dell'università, con cui ho condiviso i momenti più belli al PoliTo, dai progetti realizzati insieme, alle ore di studio, fino alle risate e al sostegno reciproco. Tutto ciò ha reso questi anni unici.

Infine, un sentito grazie a tutti coloro che, direttamente o indirettamente, hanno contribuito alla realizzazione di questa tesi.

A tutti voi, grazie.

Table of Contents

Chapter 1

Introduction

In recent years artificial intelligence (AI) has experienced a rapid evolution, driven by the development of large language models (LLM) and the spread of increasingly advanced machine learning techniques. This growth has made it possible to face with innovative approaches tasks that range from natural language processing to automatic content generation, up to the building of complex conversational systems. In particular, the focus has recently been on AI agents, software entities able not only to understand and process information, but also to act autonomously in an operational context, interacting with tools, data and external services to pursue specific goals.

According to the most recent industry analyses, the adoption of generative AI is already wide and is quickly evolving towards more autonomous and "agentic" forms. In particular, the McKinsey Global Survey reports that a relevant share of organizations declare a regular use of generative AI: more than 70% of companies are now using gen AI in at least one business function [1], and highlights how companies are redesigning workflows and roles to exploit these tools at the operational level. McKinsey also identifies agentic AI as the next phase of generative AI adoption: *"for high-impact processes, such as end-to-end customer resolution, adaptive supply chain orchestration, or complex decision-making, custom-built agents deeply aligned with company logic, data flows, and value creation levers will be necessary"* [2].

Gartner analysts also confirm this transition stating that agentic AI *"has the potential to significantly empower workers"* and *"will change decision-making and improve situational awareness in organizations through quicker data analysis and prediction intelligence"* [3].

They further observe that *"conversational platforms are evolving into intelligent agent systems capable of not just text generation, but also decision-making and orchestrated automation of complex workflows."* [3].

In the business field, these technologies offer concrete opportunities to optimize internal processes, automate repetitive activities and improve the quality and

timeliness of decisions. However, the integration of AI systems in real production contexts brings significant challenges: it is necessary to ensure reliability, security, regulatory compliance and a constant adaptation to business needs and to the specific features of company data.

The work described in this thesis is placed in this scenario, and was carried out within the Data Science Center of Excellence (DSCoE) of Reale Mutua Assicurazioni, a structure dedicated to the development of data-driven solutions to support different company functions. The activity was carried out in the context of two different projects but with the same goal of exploiting advanced artificial intelligence techniques and multi-agent architectures to improve access to information and operational efficiency.

The first project, *POD Assistant*, supports Product Oriented Delivery (POD) teams working with Agile Scrum. These teams face recurring challenges in managing documentation and workflows, including the creation of standardized project files (e.g., One Pager, Model Card, EU AI Act Risk Categorization) and using Azure DevOps for sprint planning and backlog management. While necessary, these tasks are time-consuming, repetitive, and prone to inconsistencies.

The proposed solution is an AI assistant built on a multi-agent architecture coordinated through LangGraph. This framework enables the construction of a graph-based structure that can substantially enhance the efficiency of execution flows, where each node or branch corresponds to a distinct functionality managed by a dedicated specialized agent. In this PoC three main agents were developed:

- An agent for QA on internal documentation, enabling users to query the Scrum guide and the company's operational model.
- An agent for automated documentation generation, producing mandatory project files consistently and efficiently.
- An agent for Azure DevOps integration, capable of retrieving backlog items, creating new user stories from natural language input, and interacting with repositories via the MCP protocol.

The assistant is accessible through a lightweight Gradio-based web interface, facilitating experimentation and demonstration. While limited to a PoC, the project shows the potential of agentic AI to improve team efficiency, reduce redundant work, and accelerate onboarding processes.

The second project (CGA Copilot) instead, focuses on General Conditions of Insurance (in Italian Condizioni Generali di Assicurazione or CGA), that are long and complex documents often exceeding one hundred pages and written in dense technical-legal language. These documents are critical both for insurance agents, who must interpret them quickly, and for customers, who require clear

and reliable explanations. Traditional approaches are insufficient, as generic LLMs tend to hallucinate or misinterpret specialized content, and classical Retrieval-Augmented Generation (RAG) systems are not accurate enough to handle such complex documents.

To address these limitations, this work introduces a RAG chatbot integrated with Azure AI Search and orchestrated through the LangGraph framework. LangGraph is used to enhance the retrieval process by performing query rewriting and issuing multiple refined queries, which increases the likelihood of retrieving all relevant information.

The system indexes CGA documents using vector embeddings and supports hybrid retrieval, combining vector search and semantic search with reranking to improve precision by reordering candidate passages.

The retrieved content is then injected into GPT-based models (via Azure OpenAI), which generate precise, contextualized, and verifiable answers. This approach ensures that responses remain consistent with official sources while improving both usability and trust.

From a methodological perspective, this thesis explores the integration of LangGraph to enable modular and scalable workflow orchestration, the adoption of the Model Context Protocol as a standardized interface for connecting AI agents with corporate tools and external services, thereby reducing integration complexity and improving interoperability, and the deployment of Azure OpenAI models from the GPT-4 family in combination with Azure AI Search to build hybrid retrieval pipelines.

The results obtained from the two proof-of-concept implementations demonstrate several key contributions.

First, multi-agent systems offer flexibility, scalability, and maintainability, allowing the modular evolution of AI assistants without the need to redesign the entire architecture.

Second, the use of MCP enables seamless integration within corporate environments, paving the way for reusable, plug-and-play AI components.

Third, hybrid retrieval pipelines significantly improve the accuracy and explainability of RAG-based solutions, reducing hallucinations and increasing user trust with respect to traditional RAG systems.

Finally, the practical applications developed in this work show how AI agents can be aligned with business constraints, regulatory frameworks, and user needs, ensuring that technological innovation remains both effective and sustainable.

1.1 The company context

Before going deeper into the technologies and methodologies adopted, it is important to first introduce the organizational context in which these innovations are applied. This is particularly relevant as the work presented in this thesis was carried out within a company environment, providing a concrete setting in which the described approaches have been developed and tested.

This thesis was carried out at Reale Mutua Assicurazioni, the largest Italian insurance company in mutual form. It is the parent company of Reale Group, a group active in the insurance, banking and real estate sectors with a consolidated presence in Italy, Spain, Chile and Greece. The Group operates through several subsidiaries, providing services in the insurance field (non-life and life), financial services and real estate. The headquarters is located in Turin, and it counts on a dense network of more than 1100 agencies across the country, with a total workforce of more than 4400 employees. Overall, the Group protects more than 5 million Members/Policyholders and records a premium income of over 6.3 billion euros [4].

In recent years, Reale Mutua has started a structured path of digital innovation, investing significantly in new technologies with the goal of optimizing internal processes and improving the quality of services offered to customers. This strategy has included the adoption of solutions based on artificial intelligence, advanced data analysis and automation, with the intention of promoting a concrete and sustainable digital transformation. In this context, a few years ago, within the Innovation area, the Data Science Center of Excellence (DSCoE) was created. The DSCoE is a transversal structure of the Group, designed to enhance the potential of information assets through an integrated and multidisciplinary approach. It brings together skills in statistics, mathematics, computer science and artificial intelligence, with the aim of supporting the different company areas in the realization of initiatives based on the analysis and advanced use of data.

The activity of the DSCoE focuses on forecasting situations and behaviors, on automatic inference of models and correlations, on text and image analysis, as well as on the development of conversational and decision support systems. The goal is to facilitate a synergic operational model, in which the competences of the different functions, both business and technological, can integrate and reinforce each other. In this way, the Group can strengthen its innovation capacity, leveraging the strategic use of internal and external data.

In the following chapters the technological and organizational context of the projects will be presented, together with the description of the developed solutions, the adopted methodologies and the future perspectives. The goal is to show how advanced AI techniques can be adapted to concrete needs of a complex organization, keeping a balance between technological innovation, operational constraints and quality requirements.

Chapter 2

AI Agents

After describing the company context and the operational needs that characterize modern organizations, it is natural to introduce the concept of AI agent. AI agents represent one of the most significant evolutions in the field of intelligent automation, able to support complex decision-making processes, manage heterogeneous information and interact with external tools and systems in an autonomous way.

This chapter aims to explore what an AI agent is, what its fundamental characteristics are and the key functionalities that distinguish it from traditional language models, such as Large Language Models. It will also analyze the different levels of complexity, from single agents to multi-agent systems, highlighting the advantages, the potential and the challenges of these technologies. The goal is to provide a clear understanding of how AI agents can be integrated in business contexts to optimize operations, reduce the cognitive load of users and support more efficient decision-making processes.

The definition and the general characteristics of agents were studied through the information provided by Google Cloud [5] and IBM [6], while the details on the Google ReAct framework were taken from [7]. The concepts related to multi-agent systems come from [8], while the Human-in-the-Loop (HITL) approach was described thanks to [9]. The paragraphs about risks and benefits of AI agents are based on [10].

Finally, the contents related to AI agent frameworks were taken from [11] and [12], while the details on the different frameworks and operational platforms were extracted from CrewAI [13], LangChain [14], LangGraph [15], fast-agent [16] and LlamaIndex [17] [18]. These sources provided fundamental information to describe the frameworks, functionalities and operational modes of the AI agents presented in the chapter.

2.1 Introduction to AI Agents

An AI agent is a software program that uses artificial intelligence techniques to perform tasks and achieve goals defined by the user in an autonomous way. These agents are able to reason, plan, store information (both short-term and long-term), adapt to changes in the environment and, in many cases, act without direct human intervention. At the base of an AI agent there is usually an LLM, which can be considered the *cognitive engine* of the agent: it is the component responsible for understanding, reasoning and language generation. However, an AI agent is different from a simple LLM because it has the ability to interact with external resources, access updated knowledge sources and use tools to perform actions in the real world. In this sense, an AI agent represents a natural evolution of LLMs, extending their functionalities beyond simple text generation to include the execution of operations, the interaction with APIs, software systems, physical devices and much more. For example, while a traditional LLM cannot autonomously book a hotel room or turn on the heating in a room, an AI agent properly configured with the necessary tools is perfectly able to do so.

The language model, inside an agent, plays the role of *thinking mind*: it analyzes the user input, breaks down the problem into operational steps, selects the most suitable tools among those available and uses them to reach the defined goal. AI agents can also process multimodal information, meaning data coming from heterogeneous sources such as text, images, audio, video, source code and more, like the most advanced contemporary artificial intelligence models.

A particularly relevant aspect is the ability of collaboration between agents. It is possible to design systems made of multiple agents, each specialized in a specific task. In these cases, the agents can communicate with each other, share information and coordinate through a supervisor agent, thus building modular and scalable architectures able to face complex tasks in a more efficient and customized way.

2.2 Key functionalities of an AI agent

As highlighted before, the functionalities that an AI agent can have are numerous and complex. However, two of them are particularly central for the functioning of these systems: reasoning and action. These two components were systematized in the ReAct framework, proposed by Google in 2022, which integrates the ability of *Reasoning* with that of *Acting*.

The work on the ReAct framework showed that the combination of these two operational modes allows to obtain better performances compared to paradigms based only on reasoning or on action. According to the authors, by combining the

recent progress in reasoning of language models with the possibility of performing concrete actions, it is possible to effectively deal with a wide range of cognitive and decision-making tasks. In practice, ReAct allows language models to generate both textual reasoning traces (useful to maintain and update a coherent and informed internal state) and textual actions that produce an effect on the external environment. The actions generate observable feedback, which allows the system to update dynamically, while the reasoning traces serve to rationally guide the decision-making process without directly altering the environment.

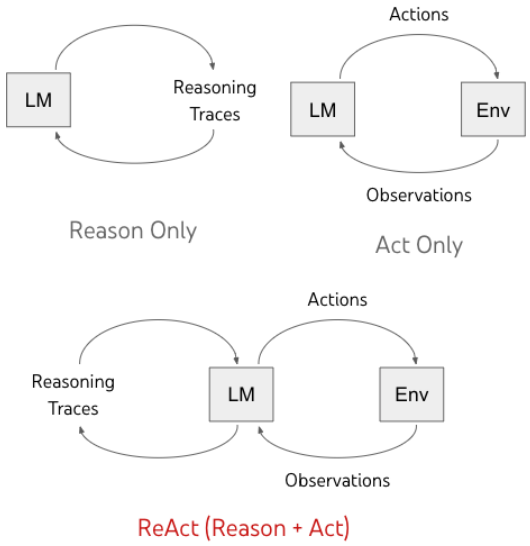

Figure 2.1: Google's ReAct architecture combines reasoning and actions in language models for improved task solving and interaction. Image taken from [7].

Reasoning is the fundamental cognitive function of the agent: it implies the ability to analyze data, identify patterns, make inferences, take decisions based on context and solve problems in an autonomous way. It is, in essence, the *intelligent* dimension of the agent, which allows it to plan strategies and choose the most appropriate options to reach a goal.

Action, on the other hand, is the operational component of the agent: it allows to execute the decisions taken during the reasoning phase, concretely interacting with the external environment. Without this function, the agent would remain confined to a purely theoretical sphere, unable to affect the world around it.

Beyond reasoning and action, other functionalities have been progressively developed and integrated, expanding the potential of AI agents. Among them there is observation, the ability to perceive and collect information about the external

environment through natural language processing, computer vision, sensors or other perceptive channels. It is an essential prerequisite for an intelligent interaction with the context.

Another functionality is collaboration: AI agents must be able to interact with other agents or with humans, coordinating complex activities and sharing information in real time to pursue common goals, even in dynamic and uncertain environments.

Agents are also able to plan and elaborate structured strategies that include sequences of actions and sub-goals necessary to reach a final result. Good planning allows the agent to optimize its behavior and choose the most effective, fast or precise paths depending on the assigned task.

Finally, there is self-improvement, that is the ability of the agent to learn from experience, adapt to changes and progressively improve its performance thanks to the received feedback. This characteristic is typical of the most advanced agents and requires the implementation of sophisticated machine learning techniques, together with optimization algorithms, which add further levels of architectural and computational complexity.

2.3 How AI agents work

After outlining a general overview of the characteristics and potential of AI agents, we can now go deeper into their internal functioning and understand how they concretely differ from a traditional LLM.

Unlike classical LLMs, which generate answers only based on the knowledge acquired during the training phase and which, for this reason, are limited by static and potentially outdated knowledge, an AI agent is designed to interact dynamically with the external environment. This is made possible through the integration with external tools, which the agent can autonomously activate to obtain updated information, perform specific operations or break down complex problems into sub-tasks to be solved sequentially.

2.3.1 Definition of the role and behavior of the agent

An AI agent is built by giving it a well-defined role, which includes not only the description of its *personality* or its task, but also the communication style, the response modes, the accessible tools and the purpose for which it was designed. This preliminary definition works as an operational context, providing the agent with a sort of functional self-awareness that allows it to maintain behavioral coherence throughout the interaction with the user.

A fundamental element that increases the effectiveness of agents is the integration of memory, both short-term and long-term. Thanks to this feature, the agent is

able to contextualize conversations, remember information that emerged in previous interactions and continue the dialogue without requiring the user to repeat the context.

2.3.2 The importance of tools

What really distinguishes an AI agent from a conventional language model is access to external tools, which may consist of specialized functions or external resources. The availability and intelligent use of these tools extend the agent's capabilities beyond the limits of natural language alone, allowing it to access real-time data, manipulate complex information, or perform concrete actions in the external world.

For the agent to use these tools effectively, it is essential to provide a clear and detailed description of the functionality of each tool, including the required parameters, usage contexts, and expected outputs. This enables the agent not only to autonomously decide which tool to activate depending on the situation, but also to do so without requiring explicit intervention from the user, significantly increasing the level of automation and adaptability of the system.

To better understand the operational differences between a traditional LLM and an AI agent equipped with external tools, let's consider a particularly meaningful illustrative example: planning an international trip with short notice.

Suppose a user needs to organize a trip and has to find affordable flights, book accommodation, check the weather forecast, and verify entry requirements for the destination country. In such a scenario, a traditional LLM, without access to external resources, could only provide generic information: suggestions on how to find cheap flights, recommendations on common tourist destinations, or explanations of general visa rules. However, it would not be able to search flight engines in real time, check up-to-date weather conditions, access official sources for travel requirements, or carry out direct bookings.

On the other hand, an advanced AI agent integrated with external tools could automate the entire organizational process. For instance, it could query a flight search engine via API to obtain real-time options for the desired destination. At the same time, it could access a weather service to assess the expected weather conditions for the chosen dates, helping the user find the best balance between low cost and favorable climate.

Once the most suitable solutions have been identified, the agent could present the user with a selection of dates and flights. Upon confirmation, it would proceed with the booking through an integrated payment system. Afterwards, by accessing hotel APIs, the agent could suggest accommodations matching the user's budget and preferences, also completing the booking once confirmed.

Finally, the agent could generate a personalized itinerary including all relevant information about flights, accommodations, travel documentation, and suggestions

for activities or places to visit. This itinerary could be delivered as a structured document, built according to explicit or implicit preferences expressed during the interaction.

This example clearly highlights the potential of AI agents with access to external tools. Unlike traditional models, they do not simply generate text, but can interact with real systems, access up-to-date data, perform complex actions, and deliver complete operational solutions. In this way, the agent is not just an informational assistant but becomes a true decision-making and operational assistant, capable of supporting the user in complex tasks while significantly reducing their cognitive and operational workload.

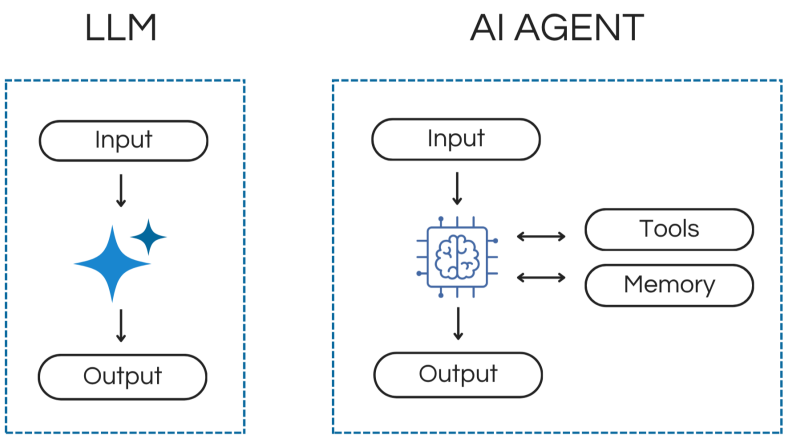

Figure 2.2: Illustration of the difference between a Large Language Model (LLM), focused on generating and understanding text, and an AI Agent, which extends beyond an LLM by incorporating memory, reasoning, tools, and actions within an environment.

2.3.3 Feedback and Human-in-the-Loop

To improve the quality and accuracy of an AI agent's responses, it is possible to integrate feedback mechanisms, which may come from two main sources: another AI agent or directly from user interaction, following the paradigm known as human-in-the-loop (HITL).

In the first case, a secondary agent can act as a validator of the responses generated by another agent. This approach helps reduce the need for direct user interaction, ensuring faster task execution. However, such autonomy also carries risks: the absence of human oversight can lead to the acceptance and propagation of errors, especially in domains where high accuracy is required or where the agent's actions may have critical consequences. The adoption of automated feedback between agents should therefore be carefully assessed, particularly in sensitive

contexts. Despite these risks, integrating feedback mechanisms among agents remains a key element in automatically enhancing decision-making and reasoning. In many cases, an iterative logic between multiple agents is implemented, where responses are reviewed and refined cyclically until a satisfactory result is achieved. This approach is especially useful when human intervention is not strictly necessary, enabling high operational efficiency.

The human-in-the-loop approach, on the other hand, is based on active collaboration between the system and the human user, and represents a widely adopted strategy in machine learning and AI applications in general. In this setup, the user provides explicit feedback on the outputs produced by the agent, helping it improve future responses and directly contributing to the system's learning and adaptation process.

The HITL paradigm is particularly relevant in contexts where decisions require a high degree of subjective judgment or a deep understanding of the context, that are elements that may escape a purely algorithmic system. Embedding the human into the decision-making loop increases the agent's reliability, mitigates error risks, and helps manage ambiguous or unforeseen cases. In this sense, HITL represents a fundamental element for developing AI systems that are more robust, ethical, and user-centered.

2.3.4 Benefits of AI Agents

The adoption and development of artificial intelligence agents offer numerous advantages, increasingly evident in light of the growing interest in intelligent automation and workflow optimization. Thanks to recent advances in generative AI, it is now possible to automate a wide range of tasks, including complex ones that previously required repetitive and often unstimulating human intervention.

AI agents enable objectives to be achieved more quickly, efficiently, and cost-effectively than traditional approaches, while also allowing for operational scalability that would be difficult to realize with conventional means. One of their defining aspects is their high level of autonomy, which reduces the need to provide detailed, continuous instructions. Agents can understand the tasks to be performed, plan the actions to be taken, and dynamically adapt to the operational context. Another significant advantage is their ability to deliver more accurate, contextualized, and personalized responses compared to traditional AI models. This is made possible by their integration with external tools, their ability to interact with real-time information sources, and, in some cases, their collaboration with other agents. Together, these features allow agents to display emergent behaviors that are not rigidly preprogrammed, thereby enhancing both the user experience and operational effectiveness.

2.3.5 Risks and limitations

Despite their numerous benefits, the introduction of AI agents also raises important issues related to safety, ethics, and social and economic impact. Precisely because of their ability to operate autonomously in the real world, and not just generate text, AI agents can, if not properly controlled, produce unintended or even harmful effects. A central risk concerns entrusting significant responsibilities to unsupervised agents. In the presence of programming errors, vulnerabilities in decision-making mechanisms, or uncontrolled access to external tools, agents could perform incorrect actions with potentially serious consequences. Unlike traditional LLMs, whose output requires human interpretation, an agent can autonomously activate systems or make automated decisions, increasing the scope of potential damage.

From an ethical perspective, the issue of transparency and accountability in automated decision-making processes emerges. It is essential to understand the criteria an agent uses to reach a certain decision, especially if that decision has a direct impact on people's lives. Furthermore, extensive automation can have negative effects on the labor market, particularly in sectors that are easily automated and have low decision-making value, contributing to forms of technological unemployment [19].

To mitigate these risks, specific measures must be adopted, including:

- Increasing the transparency of decision-making processes used by agents, also through explainable AI techniques;
- Integrating human-in-the-loop approaches, especially in high-impact contexts, to ensure human supervision of the agent's tools and actions;
- Establishing clear ethical and regulatory guidelines to guarantee user rights protection, privacy safeguarding, and compliance with current regulations;
- Complying with the principles and provisions of the new European AI Act, which defines specific obligations for the development and adoption of high-risk artificial intelligence systems.

2.4 Multi-agent systems

AI agents can be classified according to different criteria, including their functionalities, the roles they play within a system, or the number of agents involved in an application. A fundamental distinction concerns precisely the number of agents: single-agent vs multi-agent systems.

In the case of a single agent, the architecture consists of one agent operating autonomously to achieve a specific goal. Generally, this agent is supported by a

Large Language Model and can access various external tools to perform actions or retrieve information. Although this approach can be effective in well-defined scenarios, it has some limitations in terms of scalability and specialization.

In a multi-agent system, multiple agents collaborate or compete to achieve a common objective. Each agent is designed to perform a specific task or assume a defined role within the system. Multi-agent systems can consist of:

- Heterogeneous agents, each with different skills and responsibilities, suitable for addressing problems that require distinct specializations;
- Homogeneous agents, which operate in parallel with different strategies to solve the same problem, creating a competitive dynamic aimed at optimizing the outcome.

A key feature of multi-agent systems is the communication and cooperation between agents, which can be organized in more or less structured workflows. Some examples include:

- Cascade workflows, where each agent works in sequence on the output produced by the previous one;
- Decentralized architectures, in which agents have greater autonomy and are free to communicate, coordinate, and divide work dynamically.

2.4.1 Advantages of multi-agent systems

One of the main advantages of multi-agent systems is design flexibility. In a single-agent system, the agent must manage a wide range of tools and decide autonomously when and how to use them. This complexity makes it difficult to modify or extend the system's behavior without compromising the overall balance.

In contrast, in a multi-agent system, each agent is specialized in a well-defined subset of tasks. As a result, evolving or modifying the system becomes simpler and more modular: it is possible to add, remove, or replace individual agents without having to reconfigure the entire architecture. This not only promotes greater maintainability but also more effective scalability, as the computational load and problem complexity can be distributed among different agents.

Furthermore, specialization and division of labor among agents reduce the cognitive load on each agent, improving overall performance. A single agent, in fact, must understand and manage very different contexts simultaneously, which can be limiting compared to a system where each agent operates within a well-defined domain.

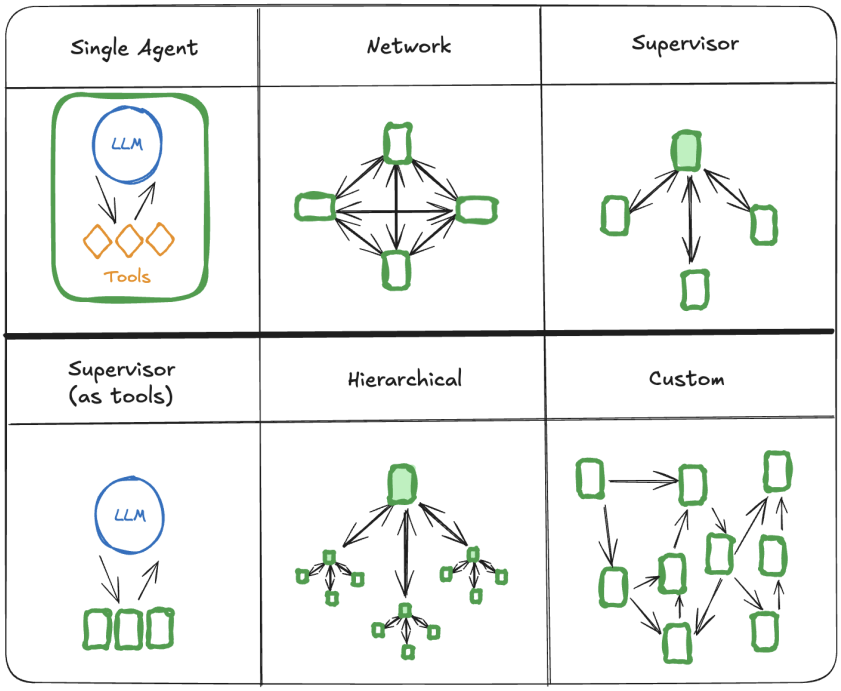

Figure 2.3: Examples of multi-agent architecture. Image taken from [20].

2.5 Introduction to agentic frameworks

An agentic framework is a software platform designed to simplify and speed up the creation of autonomous artificial intelligence agents. It represents the fastest and most scalable way to develop AI agents, especially in business contexts, where writing code from scratch in languages like Python or JavaScript would require high technical effort and longer development times.

The main goal of an agentic framework is to provide a ready-to-use infrastructure that supports the entire lifecycle of the agent, from design and development, through implementation, to management, maintenance, and monitoring. These tools allow developers to focus on the agent's logic and desired behaviors without having to deal with low-level technical details.

These frameworks provide ready-made modules to handle fundamental aspects such as the agent's memory and internal state, the invocation of external tools through APIs or local functions, and communication with other agents or human users. They also offer execution environments that allow the agent to operate autonomously, following predefined objectives and adapting to its context.

Recently, frameworks have started to support communication protocols such as the Model Context Protocol, which is designed to facilitate interaction between

agents in distributed environments (further details on this topic in Chapter 4).

In summary, an agentic framework enables the construction of intelligent systems in a more efficient and structured way, reducing technical complexity while promoting the adoption of AI agents as powerful and accessible tools.

2.6 Factors to consider when choosing an AI agent framework

The widespread adoption of agentic frameworks, driven by their ability to simplify, accelerate, and scale the development of autonomous systems, has made careful evaluation essential during the selection process. Choosing the framework best suited to one's needs requires a thorough analysis of the application context and the specific characteristics of the project.

The first element to consider is certainly the level of complexity of the framework relative to the tasks the agent will perform. It is important to clearly define which activities are to be automated, whether they require a single agent or a full multi-agent architecture, and the degree of user interaction needed to ensure reliability and control. For example, in some cases a single agent with limited functions may be sufficient, while in others it may be crucial to set up a cooperative ecosystem composed of multiple agents that exchange information and collaborate to achieve more complex objectives.

Another crucial factor concerns the level of control desired over the agent's behavior. Some frameworks, such as CrewAI, offer a highly intuitive high-level approach, allowing agents to be configured quickly with just a few lines of code, making them ideal for those who want results in a short time without managing too many implementation details. Other frameworks, like LangGraph, are more flexible but also more complex: they require greater effort during development but allow precise definition of the workflow followed by the agent, providing more granular control over each stage of the process.

Alongside these considerations, other fundamental technical aspects must not be overlooked, such as data privacy and security, interoperability with existing technological infrastructure, performance in terms of latency and responsiveness, and finally scalability, meaning the ability to expand the system efficiently as needs grow. Choosing an agentic framework is not just about finding the most powerful or widely used tool, but identifying the one that best fits the specific use case, the environment in which the agent will operate, and the system's ultimate objectives.

2.7 Most popular AI agent frameworks

Although agentic AI is still in an early stage of development, the current landscape already offers numerous frameworks, continuously evolving to keep up with the rapid technological advancements in the field. Many of these tools differ in complexity, flexibility, level of abstraction, and application areas, making a thorough study necessary to identify the solution best suited to a specific project.

2.7.1 CrewAI

Among the most well-known and widely adopted frameworks is CrewAI, an open-source platform that has quickly gained popularity for its extreme ease of use. This framework is particularly suitable for those who want to create a multi-agent system quickly, even without deep technical experience, thanks to its high-level approach, making it ideal for rapid prototyping and no-code projects.

CrewAI's architecture is based on the concept of a "crew," meaning a group of agents assigned specific roles, each with its own identity defined through natural language. For each agent, it is possible to describe background, objectives, and main functions, while tasks are formulated in textual form along with the expected output. Once the agents and their tasks are defined, the collaboration process must be configured, which can take a sequential form (where tasks are executed in order) or a hierarchical form, with a lead agent responsible for monitoring the workflow and assigning activities to the others.

This approach makes CrewAI extremely accessible and quick to implement, but at the same time involves a significant trade-off: the simplicity and speed of development come at the cost of a lower level of detailed control over the execution of the process and the internal logic of the entire agentic system. It is therefore a good choice for those who prioritize initial efficiency and usability, but it may not be ideal in contexts where more controlled behavior or greater customization of interactions between agents is required.

2.7.2 LangChain and LangGraph

Within the agentic ecosystem, LangChain is one of the most established open-source frameworks for developing applications based on large language models. It was designed to simplify the creation of chatbots, conversational systems, and AI agents through a highly flexible modular architecture. Each module encapsulates a specific concept or step in the interaction process with an LLM, allowing developers to link these functional blocks into logical chains to build complex applications. The framework's name derives directly from this mechanism.

LangChain can also be used to create AI agents with relatively simple behaviors, but to orchestrate more complex workflows and multi-agent systems, LangGraph was introduced as an extension of the same ecosystem, designed to model complex workflows in a more structured way. LangGraph is based on a graph architecture, where actions to be performed are represented as nodes, connected by edges that define possible transitions from one activity to another. The graph maintains an internal state that updates dynamically each time an action is executed, thus tracking the entire evolution of the system.

This structure allows the design of highly sophisticated workflows, which are not limited to a simple linear sequence but can include cyclic logic, conditional branches, and non-deterministic paths. Furthermore, one of the most interesting aspects of LangGraph is the possibility of introducing a human supervision phase, pausing automatic execution to collect user feedback before resuming processing from the point it was suspended.

LangGraph's main strength lies in its integration with LangChain, a widely used and supported framework. However, this synergy also introduces a certain level of complexity: the high degree of abstraction can be overwhelming, especially for leaner projects, and requires the developer to be familiar with the underlying concepts. Nevertheless, LangGraph stands out as a low-level solution that offers very fine control over defining agent behavior, making it particularly suitable in contexts where precision, traceability, and adaptability of the workflow are essential requirements.

2.7.3 LlamaIndex

Compared to the frameworks described previously, LlamaIndex stands out for its specific focus on facilitating the integration between language models and heterogeneous data sources. Born as an open-source project, LlamaIndex aims to provide a solution for orchestrating data and building AI applications based on LLMs that require contextual enrichment. Its main goal is to enable Retrieval-Augmented Generation, allowing models to query indexed content from structured or unstructured documents, databases, APIs, and other external sources using natural language.

The framework is ideally suited for scenarios where applications need to answer specific questions about local knowledge, such as chatbots trained on company documentation or technical archives. In this sense, LlamaIndex represents a highly powerful and versatile solution, especially when the challenge is not so much the agent's decision-making logic but its ability to access and understand complex contextual data.

Although it also supports the creation of agents capable of interacting with external tools through its own abstraction layer, the framework's core focus remains

on context enrichment. Not surprisingly, the definition provided by the development team describes it as "*the framework for Context-Augmented LLM Applications.*" For this reason, LlamaIndex may not be the most natural choice for those who want to develop agents with general operational capabilities, as it is better suited for projects where the distinctive element is the quality and extent of the contextual information.

2.7.4 Fast Agent

Among the most recent and innovative frameworks is Fast Agent, notable for being the first to fully and natively integrate the MCP protocol, including advanced features often unsupported by more popular solutions such as CrewAI, which currently lacks support for prompts and resources. Fast Agent allows full utilization of all protocol components, including resources, prompts, and the complete tool infrastructure, making it an attractive choice for those aiming to test or develop systems based on modern, interoperable standards.

The framework's name highlights one of its main strengths: rapid implementation. According to official documentation, it takes less than five minutes to configure a project and launch a functional agent starting from installation. This efficiency is achieved through a lightweight architecture and extensive use of declarative YAML configurations, significantly reducing the amount of code that needs to be written manually. Fast Agent is therefore well-suited for quickly building prototypes, such as for hackathons, educational environments, or experiments with MCP servers.

In addition to simplicity, another advantage is its advanced debugging and tracking tools, which allow detailed monitoring of all interactions between agents and the MCP backend, including tool calls and system decisions made during execution. This makes it particularly suitable for experimentation and testing phases.

However, due to its focus on speed and prototyping, Fast Agent may not be the optimal choice for developing complex products intended for long-term production. In such cases, frameworks with more structured architectures and features oriented toward scalability, security, and maintainability would likely be more appropriate.

Chapter 3

Introduction to LangGraph

Although the comparative analysis presented in the previous section cannot be considered exhaustive or free from limitations, it is clear that LangGraph emerges as the most suitable choice to meet the POD Assistant project requirements. As for the CGA Copilot, LlamaIndex might seem like the obvious choice, but since a traditional RAG system might not be sufficient, LangGraph was also chosen for this project to test a hybrid approach. Beyond the technical and architectural advantages offered by the framework, its adoption is further justified by the fact that it is already used within the company in products currently in production. This ensures full compatibility with the existing infrastructure, facilitating integration while reducing development time and costs.

This chapter will therefore examine the architecture of LangGraph, with particular focus on its core principles, internal functioning, and the reasons that make its adoption especially advantageous in the specific corporate context.

The following content has been extracted from the official documentation [15].

3.1 Graph

As introduced in the previous chapter, LangGraph's architecture is based on the concept of a graph, a structure that allows flexible and modular modeling of an AI agent's behavior. The graph is composed of three fundamental elements: state, nodes, and edges.

- **State**: this represents the current execution context and is shared across all graph components. Essentially, it is a data structure that can take any form but is typically defined as a `TypedDict` or a Pydantic `BaseModel`. All variables relevant to the agent's operation (such as the message history between user and assistant or intermediate data to pass between successive nodes) are declared within the state.

- **Nodes**: these constitute the computational core of the agent. Nodes are simple Python functions that receive the current state as input, perform a specific logic (such as invoking an LLM or transforming the state) and return the updated state. Due to their generic nature, nodes are extremely flexible and can implement virtually any operation.

- **Edges**: define transitions between nodes, determining which node should be executed next based on the updated state. Edges can be deterministic, representing a fixed transition, or conditional, acting as routers that dynamically select the next path depending on the state.

By integrating these three elements, it becomes possible to design sophisticated workflows that flexibly adapt to different application contexts while maintaining a high degree of customization. At the heart of this flexibility lies LangGraph's centralized state management, which allows nodes to coordinate seamlessly while preserving the simplicity of independent Python functions.

Building on this foundation, LangGraph adopts a message-based execution model. When a node completes its task, it sends one or more messages along its outgoing edges. Recipient nodes, upon receiving a message, are activated, execute their function, and then propagate new messages through their own edges. This mechanism unfolds in discrete phases, called super-steps, each representing a full iteration over the graph during which active nodes can operate in parallel.

The process begins with all nodes inactive. A node becomes active only when it receives a message, at which point it processes the state and produces a new output. After each super-step, nodes that have not received any messages remain inactive. The execution concludes once all nodes are inactive and no messages remain in transit, marking the natural termination of the workflow.

3.2 State

One of the first steps in defining a graph in LangGraph is configuring the state. The state represents the information shared among all nodes in the graph: it is both the input provided to each node and the result returned after node processing. Each transition in the graph therefore corresponds to an update of the state.

The state is typically defined as a typed structure, often using `TypedDict`, although a Pydantic `BaseModel` can also be used depending on the project requirements. Regardless of the form, it is crucial that all variables necessary for execution, ranging from messages exchanged between user and assistant to flags, intermediate results, and other control data, are explicitly defined.

A central aspect of state management concerns reducer functions. Reducers specify how the value of each state key should be updated when a node returns

a new state. Each key can have its own reducer function; if none is specified, a default behavior is applied, which simply overwrites the previous value with the new one.

In many cases, simple overwriting is limiting. A classic example is tracking messages in a conversation between a user and an assistant. Overwriting the last message each time would result in losing the conversation context. Instead, it is more useful to use a reducer that accumulates messages in a list, preserving the complete history of interactions. This behavior can be implemented, for instance, with an add-type reducer, which concatenates new elements with existing ones.

Modern LLM providers generally offer conversational interfaces that rely on an ordered list of messages. Specifically, LangChain's `ChatModel` accepts a sequence of Message objects as input, where each message can represent a user intervention (`HumanMessage`), a model response (`AIMessage`), or a system prompt (`SystemMessage`). Maintaining a coherent message history in the state is therefore not only useful but often essential for the correct operation of the agent.

3.3 Nodes

In LangGraph, nodes represent the fundamental processing units within the graph. As previously mentioned, a node is generally defined as a Python function that receives the current state of the graph as input. An optional configuration parameter can also be passed to the function, often used to specify information such as the thread ID or other useful metadata.

Nodes are added to the graph through an interface similar to NetworkX, allowing each node to be assigned a symbolic name. This name enables easy reference when defining the graph structure and its edges.

Alongside custom nodes, LangGraph provides two special built-in nodes: `START` and `END`.

The `START` node represents the entry point of the graph, i.e., the first node that receives user input. It does not perform any operations itself but acts as the initial trigger for graph execution.

The `END` node marks the termination of the workflow. Once execution reaches this node, the graph is considered complete, and its output is returned.

When constructing a graph, it is common to explicitly define an edge from `START` to an initial node (e.g., `node_a`), establishing that execution begins there. Similarly, connecting a final node (e.g., `node_z`) to `END` ensures that the process is considered complete after executing `node_z`.

This structure provides a clear entry and exit framework for the graph while allowing full flexibility for intermediate processing nodes and complex workflows.

3.4 Edges

In LangGraph, edges are the elements that connect nodes within the graph and define the transition logic from one node to another. They determine the execution path based on the current state and the graph structure. Deterministic edges establish a linear sequence, so once a node finishes execution, the next node in the sequence is executed directly. Conditional edges, on the other hand, introduce branching logic: a routing function evaluates the current state to decide which node to execute next, enabling alternative paths within the graph.

A special role is played by entry points, which connect the `START` node to the first actual nodes of the graph. Entry points can follow either a Direct transition, defining a fixed initial node, or a Conditional entry points, where a function evaluates the initial state or other contextual parameters to determine which node(s) should be activated first.

It is important to note that a single node can have multiple outgoing edges even if are not conditional. In this case, the destination nodes are executed in parallel in the next super-step. This mechanism enables the modeling of complex workflows and the simultaneous processing of multiple graph branches, enhancing both flexibility and execution efficiency.

3.5 Send and Command

So far we have seen how defining a graph requires, before execution, the explicit construction of the nodes and the edges that connect them. In addition, all nodes share the same State object, which represents the global state of the graph. However, there are situations where these assumptions are too rigid: for example, when the edges are not known in advance, or when it is necessary to maintain multiple versions of the state running simultaneously.

To support these scenarios, LangGraph introduces the `Send` object. This object can be returned by a conditional edge and allows sending a specific state to a specific node. It is a particularly useful tool when you want to implement a design pattern like Map-Reduce, where the same function is applied to different subsets of the state and then the results are aggregated. With `Send`, each branch of the graph can receive its own copy of the state, modified as needed, allowing parallel and independent execution.

Another very powerful construct offered by LangGraph is the `Command` object. Unlike `Send`, `Command` can be returned directly from a node function, allowing the combination of flow control and state updating in a single step. With `Command`, it is possible to dynamically decide which node to execute next and simultaneously modify the state that will be passed.

An even more interesting aspect is the ability of `Command` to navigate between subgraphs. When a graph contains one or more subgraphs (for example, to modularize the flow), it is possible, from a node in a subgraph, to direct execution to an external node in the parent graph. To do this, it is enough to return a `Command` that, in addition to specifying the name of the destination node, indicates that this node is in the parent graph using `Command.PARENT`.

Finally, `Command` is also essential for implementing human-in-the-loop flows, where the execution of the graph is paused to wait for human input. In these cases, the `interrupt()` function is used, which suspends the graph and preserves its state. When input is received from the user, execution can be resumed by invoking `Command(resume="User input")`, passing the received message directly to the next node.

Chapter 4

Model Context Protocol (MCP)

After exploring the concept of AI agents and the main frameworks that enable their development, it is useful to introduce the tools that allow these technologies to integrate effectively with the corporate ecosystem and with heterogeneous external resources. In this context, the Model Context Protocol (MCP) comes into play, a recent standard proposed by Anthropic to overcome one of the most relevant limitations of LLMs: the difficulty of communicating in a simple, secure, and scalable way with data sources, applications, and external services. MCP is therefore designed as an enabling element to make AI agents not only more powerful but also more flexible and interoperable, opening the way to more advanced and practical use cases.

The information presented in this chapter is mainly drawn from [21][22][23].

4.1 Introduction

The Model Context Protocol, commonly referred to as MCP, is an open and standardized framework introduced in November 2024 by Anthropic, an American startup active in the field of artificial intelligence and known for developing the Claude family of large language models.

MCP's primary goal is to simplify and enhance the way LLMs interact with external resources, such as data sources, tools, or services. The underlying idea of the protocol is to provide a universal interface that enables AI applications to access and communicate with heterogeneous systems, without requiring the integration of each external tool through a custom API.

One of the main challenges in building LLM-based systems today is their limited capacity to interface with third-party resources. This limitation stems from the

absence of a standardized communication protocol. Whenever a language model needs to connect with an external service, developers must create custom connectors for that specific case. This approach not only increases development complexity but also raises maintenance costs and makes the system architecture more fragile.

MCP was specifically designed to address this gap. By establishing a unified protocol for communication between LLMs and external resources, it seeks to streamline integration workflows, reduce technical overhead, and promote interoperability across different platforms and services. In this way, MCP aims to become a key standard for AI systems that operate in increasingly dynamic and interconnected environments.

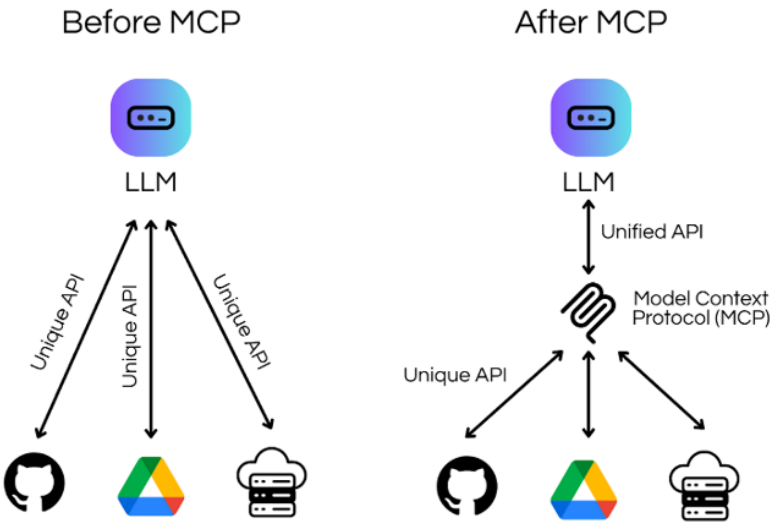

Figure 4.1: An example of how the MCP protocol can simplify the communication between the LLM and external services. Image adapted from [24].

4.2 How MCP works

To better understand the role and importance of the MCP protocol, an analogy from the official documentation is instructive. As stated: *"Just as USB-C provides a standardized way to connect electronic devices, MCP provides a standardized way to connect AI applications to external systems."* [22]

MCP can be thought, indeed, of as a USB-C port, offering a uniform interface to connect a wide range of devices and peripherals, each developed by different manufacturers for diverse purposes. Just as a USB-C port can transfer data,

transmit video, or deliver power, MCP provides a standardized interface that allows large language models to communicate seamlessly with heterogeneous external systems, regardless of their internal architecture or original design.

Building on this concept, MCP is implemented using a client-server architecture, enabling a single host application to establish and manage connections with multiple servers. Structurally, MCP consists of three main components: host, client, and server.

The host is the AI application that initiates and manages communication with external MCP servers. Since MCP was developed by Anthropic (the creators of Claude), Claude models natively support the protocol. For instance, Claude Desktop can act as a host, offering seamless integration and straightforward configuration for external servers. However, MCP is not limited to Claude models. Developers can also leverage integrated development environments (IDEs) such as Visual Studio Code to connect servers directly to Microsoft Copilot within the IDE or create custom host applications tailored to their specific requirements.

The client is a component embedded within the host. Its role is to establish one-to-one connections with individual MCP servers and manage interactions with them. Each client handles specific tasks or data flows according to the functionalities provided by the servers it is connected to.

Servers are external programs or services that expose data sources or operational capabilities via the standardized MCP interface. Anyone can develop these servers and make them publicly accessible, enabling other developers to integrate them into their AI systems. As protocol adoption grows, several platforms have started aggregating available MCP servers into searchable directories, akin to an app store, simplifying discovery and reuse. Some organizations, including GitHub, Microsoft, PayPal, AWS, and Alibaba, maintain official MCP servers to make certain platform functionalities accessible.

Communication between clients and servers is facilitated through the JSON-RPC 2.0 protocol, ensuring a consistent and efficient message exchange format. Each MCP server can expose three main types of elements: tools, resources, and prompts.

Resources enable servers to supply data and information that can be used to provide context for language models. These resources can encompass almost any type of data, including files, database records, API responses, real-time system data, logs, images, and more. Each resource is uniquely identified by a URI (Uniform Resource Identifier) and can contain either textual or binary content, following the format: `[protocol]://[host]/[path]`.

Resources are managed by the application, meaning the client decides when and how to access them. This design gives each client the flexibility to adopt its own strategy for resource management, based on the use case and internal application logic.

Prompts, in contrast, are user-controlled and act as reusable templates exposed by MCP servers for use by clients and end users. They are typically presented in the user interface, allowing humans to select and execute them. Prompts help guide specific workflows, support multi-step interactions, and enable structured tasks or standardized input suggestions for models.

Finally, tools represent operational capabilities that a server makes available to the client. Tools allow the language model to perform specific actions, ranging from simple calculations to complex interactions with APIs or external systems. Tools are controlled by the model, meaning the model can invoke them autonomously when needed, potentially requiring human approval for sensitive or high-risk operations. Each tool has a unique name and includes a detailed description of its purpose and context.

Unlike resources, tools can modify the external environment or system state. Therefore, it is crucial for the client to understand what a tool does and the consequences of its execution. From the client's perspective, tools can be treated as black boxes: they are invoked via a standard interface, while their internal workings remain hidden. Nevertheless, the operations performed can range from simple tasks to complex action chains.

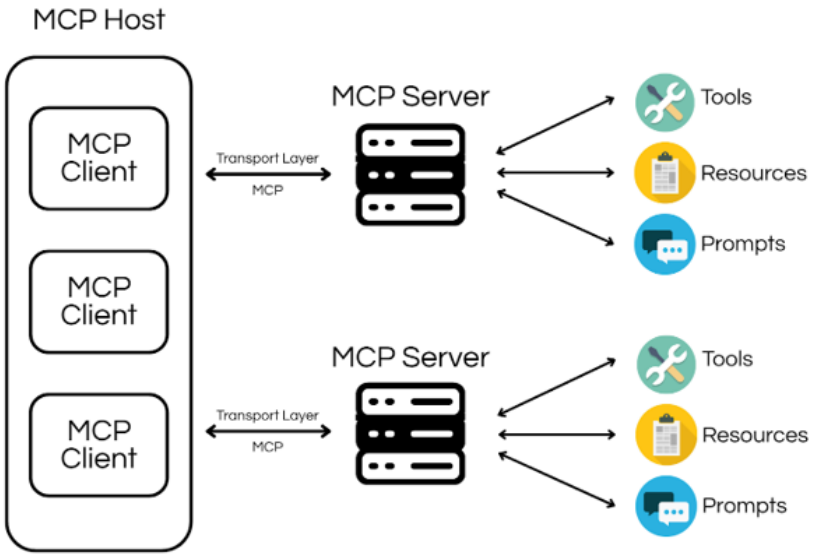

Figure 4.2: An example of architecture that implements the MCP protocol. Image adapted from [25].

4.3 MCP vs traditional APIs

The primary difference between MCP and traditional APIs lies in how easily and efficiently AI systems can connect to external tools.

Historically, integrating external services into an AI application via APIs required a lot of effort: developers had to write custom integration code, handle authentication, navigate often complex documentation, and track updates or changes to each service.

This approach not only extended development timelines but also made scaling harder, because each new tool needed its own dedicated integration, increasing the overall complexity of the project.

In contrast, MCP greatly reduces this integration burden. Instead of needing a separate connection for every external service, MCP offers a standardized and unified interface that allows AI models to interact with a wide range of tools and data sources with minimal setup. This design enables faster development cycles and a more maintainable codebase, especially as the number of integrated tools grows.

From a scalability perspective, MCP represents a real shift. MCP servers are designed to be plug-and-play: once a server is developed and made available, it can be reused in any AI application that supports the protocol. This reusability removes the need to duplicate integration work and supports a modular ecosystem of reusable services.

A particularly powerful feature of MCP is dynamic tool discovery. At runtime, MCP clients start a discovery phase in which they identify the tools and capabilities offered by connected servers. This allows clients to automatically recognize available tools as soon as a server is connected, without prior knowledge of its interface or functions. In practice, this means developers can expand an AI system's capabilities simply by connecting new servers, without writing or coding additional integrations.

By comparison, traditional APIs lack this flexibility: each new service requires manual registration, reading documentation, and a rigid, predefined setup of endpoints and expected responses.

MCP, instead, provides a more flexible and extensible interaction model, making it especially suitable for AI systems that need to dynamically engage with constantly evolving external environments and services.

Chapter 5

Retrieval Augmented Generation (RAG)

One of the main limitations of LLMs lies in their static nature. Once the training phase is complete, these models cannot update themselves with new information and remain limited to the dataset on which they were trained. This becomes particularly critical when the goal is to build systems capable of providing precise, up-to-date responses in a specific domain, such as a chatbot that must interact with users using company documents as the primary source of knowledge.

In this context, Retrieval-Augmented Generation (RAG) emerges as an effective solution. RAG combines the generative capacity of an LLM with a mechanism for retrieving information from external document databases, which can be constantly updated and specialized for the domain of interest. In this way, the model does not just "remember" what it learned during training, but can "consult" relevant and contextualized documents at the right moment, improving both the accuracy and reliability of the responses provided.

The information presented in this chapter is mainly drawn from [25].

5.1 How RAG works

The RAG process is typically composed of two main stages: retrieval and generation.

In the retrieval stage, when a user provides a query, the system searches one or more external data sources to find information that is relevant to that query. These sources can include public knowledge bases, proprietary internal documentation, or other collections of unstructured text. The documents are usually indexed using vector embeddings, which map their content into a high-dimensional space to enable semantic search.

When a query is submitted, it is converted into an embedding and compared with the stored embeddings in order to identify the most semantically similar documents. The retrieved documents are then ranked according to their relevance to the query, and the top results are selected as contextual evidence for the next stage.

During the generation stage, the retrieved information is provided as additional context to the generative model. The LLM uses this contextual input to produce an answer that is grounded in the retrieved content, rather than relying solely on the knowledge it acquired during pretraining. This allows the model to incorporate specific factual information into its response, reducing the risk of hallucinations and increasing the overall accuracy and reliability of the generated text. In some implementations, a post-processing step follows to refine the output, ensuring that it is coherent, grammatically correct, and aligned with user expectations.

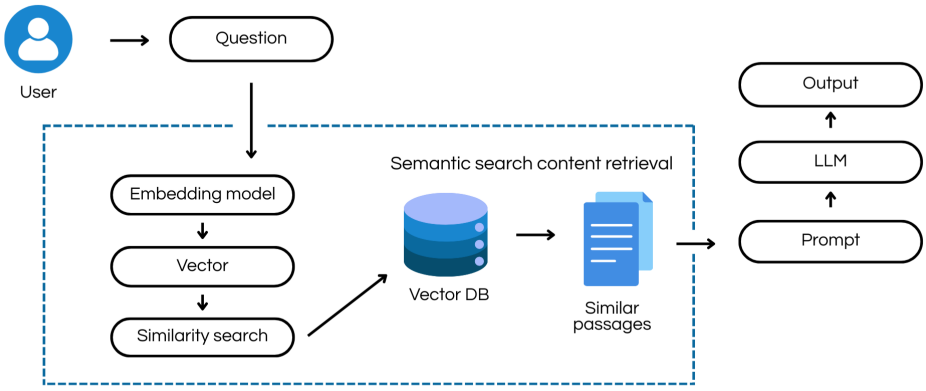

Figure 5.1: An example of RAG pipeline. Image adapted from [25].

By combining retrieval and generation, RAG offers several advantages over traditional language model architectures. It allows systems to use more recent and dynamic information than what is contained in the static training data of an LLM. This improves factual accuracy by grounding the model's responses in real data and can also lower costs compared to retraining or fine-tuning the model on new information.

These benefits are particularly relevant in applications based on company documents. RAG helps ensure that a chatbot's responses are consistent with official sources rather than based on unsupported generalisations, reducing the risk of hallucinations. It also allows the system to adapt to specific domains without retraining a model from scratch: it is enough to update or expand the collection of documents from which the system retrieves information. Finally, the ability to show the sources used to generate each answer increases transparency and builds

user trust, as it allows users to verify where the information comes from.

However, using RAG also brings some challenges. The quality of the responses depends heavily on the relevance and accuracy of the retrieved documents. If the retrieval phase returns irrelevant or incorrect content, the generated output will likely be flawed as well. Searching and ranking documents can also increase latency and computational costs, especially when working with large knowledge bases. Moreover, using external data requires careful attention to privacy, security, and access control, especially when handling proprietary or sensitive information. It is also important to keep the document base accurate and up to date to avoid spreading outdated or confidential content.

Overall, RAG is an effective way to improve the performance of LLMs by giving them access to external, continuously updated information while still using their ability to produce coherent and fluent text. This makes it possible to build systems that are more accurate, trustworthy, and flexible without the need to constantly retrain the underlying model.

In the context of a company chatbot based on documents, RAG is therefore a key element: it turns a general-purpose language model into a truly useful tool for consultation and support, combining the flexibility of natural language with the reliability of official sources.

Chapter 6

Integration of Azure OpenAI and Azure AI Search Services

To effectively leverage the potential of next-generation artificial intelligence models, it is necessary to have an infrastructure that is scalable, secure, and easily integrated into company systems. In this context, the Microsoft Azure platform represents a strategic solution, as it provides access to the GPT family models through Azure OpenAI, as well as the management of company data and knowledge through Azure AI Search, enabling RAG scenarios based on vector databases.

This chapter describes the integration architecture adopted by the company, illustrating the methods of accessing the models, the techniques used for security management, and the solutions implemented to ensure flexibility, operational efficiency, and the ability to intelligently leverage company information.

6.1 Access to LLMs with Azure OpenAI

Within the context of the project, Reale Mutua uses the Microsoft Azure platform as the reference cloud infrastructure for managing and deploying artificial intelligence models. In particular, the LLMs employed are customized and deployed versions of GPT-4o, GPT-4.1, GPT-5, GPT-5-mini, and others, made available through Azure OpenAI services.

These models are hosted on Azure and accessible via specific REST APIs, integrated into the company's cloud infrastructure. Access to the models is handled through a dedicated Python script, which serves as an interface to the Azure OpenAI services. The script uses the `httpx` library to make both synchronous and

asynchronous HTTP calls, and employs the `langchain_openai` library to manage interaction with the `AzureChatOpenAI` and `AzureOpenAIEmbeddings` models. This approach ensures a modular and easily extensible interface, allowing the configuration of model parameters such as deployment name, API version, generation temperature, and other operational settings.

Security and configurability of access are guaranteed through a `.env` file containing the necessary environment variables, including authentication API keys (`OPENAI_API_SUBSCRIPTION_KEY`), endpoints, and other critical parameters. The script loads these variables using the `dotenv` library, avoiding direct inclusion of sensitive information in the source code and following best security practices.

During execution, the script instantiates an HTTP client configured for HTTP/2 and carefully manages request headers, ensuring proper authentication with the Azure API Management service. Through the implementation of custom hooks on HTTP requests, it is possible to remove unnecessary authentication headers, ensuring clean communication that complies with Azure gateway requirements.

This software architecture allows sending text prompts to the LLM and receiving generated responses in real time, supporting the advanced dialogue functionalities required by the chatbot. Furthermore, adopting this system simplifies maintenance, allowing updates to the models or modifications to their configuration without major changes to the application code, simply by adjusting relevant parameters.

6.2 Integration with Azure AI Search for RAG

The retrieval and organization of knowledge within a RAG system do not depend only on the language models' ability to generate text, but mainly on how effectively documents are indexed, searched, and selected. In this context, Azure AI Search plays a central role because it combines traditional keyword-based search with advanced semantic and vector search mechanisms. The goal is to create a flow where a user question is associated with a set of relevant documents, which are then integrated into the prompt provided to the LLM, ensuring contextualized answers based on up-to-date knowledge.

At the core of the system is the indexing of documents in the form of embeddings, that is, high-dimensional vectors encoding the semantic meaning of the texts. In Azure AI Search, these embeddings are organized in structures optimized for nearest neighbor search, such as HNSW (Hierarchical Navigable Small World) algorithms. These structures allow the retrieval of the vectors closest to a query in sub-linear time, even when working with millions of documents. The notion of closeness is typically based on cosine similarity, which measures the angle between two vectors: the smaller the angle, the higher the semantic similarity between the query text and the stored document. This approach overcomes the limitations of simple keyword

searches, capturing conceptual relationships even without direct lexical matches.

Alongside vector search, Azure AI Search provides a semantic search mode, which uses pre-trained language models to interpret the meaning of queries and documents. Unlike pure vector similarity, semantic search produces a ranking of results based not only on numerical proximity in the embedding space but also on the ability to understand entities, synonyms, relationships, and context. For example, a query containing the term "health insurance policy" will be able to retrieve documents referring to "medical insurance," even if the two expressions never appear together in the corpus.

The quality of results can be further improved through semantic reranking, which involves applying a second model (often more sophisticated and computationally expensive) to the results already retrieved by an initial search. In practice, the query selects a subset of candidates, and the reranker reorders them by evaluating the consistency between question and document with stricter criteria. This process reduces the risk of including marginal or irrelevant information in the context provided to the LLM, improving the accuracy of the final answers.

Another level of sophistication comes from using hybrid search, which combines keyword search with vector search. This approach simultaneously leverages the precision of lexical queries and the ability of embeddings to capture semantics, making it particularly useful in domains like insurance, where specialized texts, technical terminology, and natural language coexist. The system also allows applying filters based on metadata, further contextualizing the results.

Chapter 7

POD Assistant

After introducing and describing the main technologies used throughout this work, we now move on to the analysis of the projects developed in the company during the internship. This chapter focuses on the first of these, the POD Assistant, a solution developed as a Proof of Concept (PoC) aimed at supporting the POD (Product Oriented Delivery) teams organized according to the company's operational model. The project was born from the need to simplify information management and improve the efficiency of daily workflows, providing team members with an intelligent chatbot capable of assisting with operational tasks. The chapter therefore illustrates the organizational context in which the initiative is placed, based on the Agile Scrum framework, and describes how the proposed solution integrates into the company model to promote speed, adaptability, and added value in the activities of the Data Science Center of Excellence (DSCoE).

7.1 Introduction

Within the DSCoE several specialized teams operate, each focused on specific thematic or technological areas. The organization of work within these teams follows the Agile Scrum framework, chosen to ensure effective, iterative, and flexible project management.

Scrum allows activities to be divided into short work cycles called *sprints*, at the end of which functional increments of the project are delivered. This approach encourages continuous improvement, rapid adaptability to changes, and strong alignment between business needs and technical development. The framework defines clear roles: the Scrum Master, who facilitates the process and removes operational obstacles; the Product Owner, who represents the business, sets priorities, and manages the backlog; and finally, the Development Team, composed of technical and analytical figures responsible for the actual implementation of

solutions.

This structure has been formalized within the company's operational model to ensure uniformity across teams, clarity in roles, and a rapid, coordinated response to the needs of the Group's various business areas.

In this operational model, teams are organized into PODs, which are designed to be autonomous, adaptable, and focused on delivering value quickly, sustainably, and efficiently.

In this context, the present project is placed, aiming to develop a PoC for a POD assistant chatbot, designed to support the team's operational activities, facilitate information management, and contribute to the overall efficiency of the workflow.

7.2 The problem

The main goal of the chatbot is to automate internal processes that, although essential for the proper functioning of the PODs, are often slow, repetitive, and not very productive. These activities, while necessary, take up valuable time that could be dedicated to developing AI solutions, slowing down the delivery cycle and affecting the overall efficiency of the team.

Among the most time-consuming tasks is the production of mandatory documentation required during the release of an AI model. Each new application must be accompanied by a set of standardized documents, including:

- The One Pager, a summary sheet that outlines the main characteristics of the model, its objectives, and information about the project contacts.
- The Model Card, which provides a detailed overview of the model, its performance, risks, and limitations, to promote transparency and informed adoption by stakeholders.
- The EU AI Act Risk Categorization, necessary to ensure compliance with the new European regulation on artificial intelligence. This document classifies the model's risk level and documents the mitigation measures.

Manually creating these files involves duplication of information, considerable time expenditure, and poor scalability: every modification must be updated in each document separately, with a risk of inconsistencies.

Another critical area is the management of operational activities through the Azure DevOps platform. Teams work according to two-week sprint cycles, in line with the Scrum framework. Each sprint involves a planning phase (Sprint Planning) in which the sprint goal, the list of tasks to be completed (formalized as Product Backlog Items, or PBIs), and the acceptance criteria to evaluate whether the goal has been achieved are defined.

Creating user stories, managing the backlog, and prioritizing PBIs are time-consuming and require careful attention. Tasks may have dependencies on each other or on external elements, and the fully manual management currently in place demands particular care during planning meetings or daily stand-ups.

Finally, another challenge concerns the consultation and understanding of the company's operational model and the Scrum framework, which are often complex and structured. Having a virtual assistant capable of answering questions, providing clarifications, or summarizing technical documentation can accelerate onboarding for new members, promote continuous learning, and support the team in daily activities.

The problems identified represent only a portion of the critical issues that emerged during the analysis phase. Discussions with DSCoE members revealed further opportunities for automation and simplification, which an AI assistant integrated into the POD's operational flows could enable, significantly contributing to time reduction and resource optimization.

7.3 The AI solution

The Proof of Concept led to the development of an AI chatbot organized according to a graph structure, built using the LangGraph framework. Each branch of the graph corresponds to a distinct functionality of the assistant, managed by specialized agents and orchestrated in a modular way to ensure scalability and maintainability.

Within this architecture, three AI agents were developed, each equipped with specific tools to respond to different types of requests.

The first agent is dedicated to interacting with technical and methodological documentation, allowing users to ask questions about the Scrum guide and the company's operational model and receive precise explanations, summaries, in-depth insights, and direct references to key concepts. The goal is to make often complex documentation accessible and usable.

The second agent focuses on automating the generation and updating of project documentation. Thanks to its ability to process both structured and unstructured input, the chatbot automatically produces documents such as the One Pager, the Model Card, and the EU AI Act Risk Categorization. These documents are essential for AI model release and feature recurring sections that the assistant fills consistently, reducing the risk of redundancy and the time the team spends on repetitive tasks.

Alongside these, a third agent is focused on interaction with the Azure DevOps (ADO) platform. Through the MCP protocol, it connects to company repositories to retrieve information about ongoing projects, list the Product Backlog Items

(PBIs) of a specific project, create new ones from natural language descriptions, and consult project wikis, thus providing direct support for development and management activities.

In addition to these advanced functionalities, the chatbot also handles simpler interactions, classified as small talk, where the involvement of agents with specific tools is not necessary and the direct use of an LLM is sufficient.

To make the assistant easily usable, a graphical interface was developed with Gradio, a lightweight and intuitive solution that allows interaction with the chatbot through a web environment accessible to everyone, facilitating experimentation and demonstration of its capabilities. The image below represents the high-level architecture of the chatbot's main components.

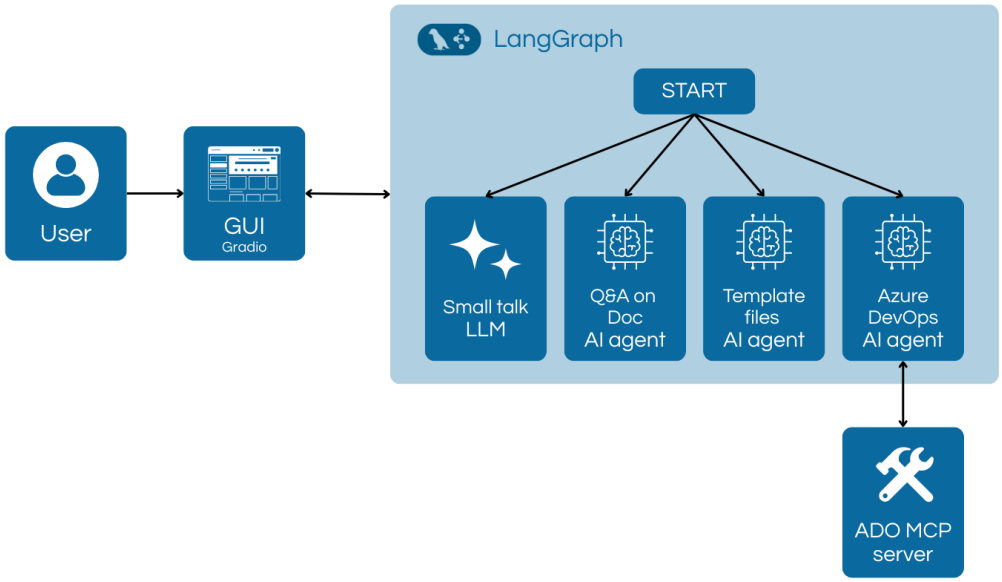

Figure 7.1: The high level architecture of the POD Assistant.

7.4 The AI agents

7.4.1 Q&A on documentation

The first AI agent presented is dedicated to handling questions related to the Scrum guide and the company's operational model.

Since this is a Proof of Concept and considering the limited size of the documents involved, it was decided not to implement a full RAG system, as this choice would have been excessive for the project's goals. However, if in the future the chatbot's scope were to be extended to other types of documentation or larger information volumes, the introduction of a RAG system would certainly be necessary to ensure scalability and efficiency.

For this version, a simpler but still effective solution was chosen: providing the LLM directly with the textual content of the document as context in the prompt, taking advantage of the small file sizes. To this end, the documents were converted into a suitable format, preferably text or Markdown, to maintain a format readable by the LLM. In the case of the Scrum guide, it was a simple mostly textual PDF. It was therefore sufficient to use Microsoft's Markitdown library [26] to convert the file to Markdown, obtaining a clean result that could be easily integrated into the prompt.

A different situation arose with the company's operational model, contained in a PowerPoint presentation. Although the Markitdown library supports Office file conversion, the result was unsatisfactory. The slide content, composed of charts, tables, and concise diagrams, became chaotic after conversion, risking confusing the LLM or leading to incorrect answers. To overcome this limitation, the slides were converted into images, and the LLM was asked to describe them, allowing extraction of as much information as possible from the visual content. This provided sufficiently rich and accurate context to reliably answer questions about the operational model.

At this stage, the agent's tools are limited to returning the textual content of relevant documents but are already designed to be replaced in the future with more advanced tools based on a vector database and semantic queries. This would allow moving from a simple static extraction to a full RAG system without modifying the overall graph structure or the agent's logic.

The agent is created using LangGraph's `create_react_agent` function, which allows building an agent capable of calling tools in a loop until a termination condition is met. Below is the agent definition:

```
1 agent = create_react_agent(
2     model=llm,
3     tools=[get_full_doc_scrum, get_full_doc_operative_model, get_datetime]
4 )
```

where *llm* is the LLM object obtained through the Azure OpenAI interface and *tools* is the list of functions available to the agent.

The list of the available tools is the following:

- **get_full_doc_scrum**: reads the Scrum documentation locally (in .txt format) and returns it as a string. To avoid reading the file on each call, a caching method is used: the content is loaded once and then stored in a variable.
- **get_full_doc_operative_model**: similar to the previous one, but for the company's operational model documentation.
- **get_datetime**: provides the agent with date and time information to contextualize the interaction.

In this way, the agent can autonomously decide which tools to invoke to answer the user's questions.

The choice of the correct tool largely depends on the docstring associated with each method, which precisely describes what the method does, which parameters it receives, and what it returns.

For example, the docstring for the `get_full_doc_scrum` method specifies in detail the document content, including descriptions of the Scrum framework, events, artifacts, values, and the tool's purpose.

In this way, a first AI agent was created that can answer user questions about the Scrum framework and the company's operational model. Thanks to the available tools and the agent's autonomous operation, it will select the most appropriate tools on its own and combine the obtained information to provide precise and coherent answers to the user.

7.4.2 Management of template files

The second AI agent is dedicated to automating the creation and updating of project-related documents, including the One Pager, Model Card, and EU AI Act Risk Categorization. For simplicity, these documents are collectively referred to as *template files*, as they represent standard models to be populated with project-specific information.

7.4.2.1 Integration with SharePoint and access issues

The initial goal was to operate directly on SharePoint, the Microsoft CMS platform adopted in the company, in order to leverage its document management capabilities. However, some challenges arose: there was no MCP server available to enable secure and complete integration of an AI agent with SharePoint, and although the

SharePoint API was accessible, it involved complex permission handling and raised security implications that required careful consideration.

Due to these limitations, it has been decided to replicate the SharePoint structure locally, postponing full integration and access management to a later stage of the project.

7.4.2.2 Issue of information duplication

Currently, project information is stored both in template files and in separate wiki documents, causing data duplication, increased complexity, and risk of inconsistencies.

To address this, the concept of a Single Point of Truth (SPoT) was proposed: a centralized source from which all information necessary to automatically generate project documents could be retrieved.

During the PoC, this SPoT was not yet formalized and so, in its absence, a local structure was defined as follows:

Local filesystem structure

```
SharePoint/
    Project1/
        wiki.md
    Project2/
        wiki.md
    Project3/
        wiki.md
    ...
```

Each wiki.md file collects all available information on the project, without strict formatting constraints. A template containing all information needed to populate the three template files was created, but additional data can also be included. The agent automatically selects only the relevant information for document generation.

This approach ensures that all information comes from a single source, maintaining consistency, and alerts the user if any information is missing to complete the files correctly. The template files are then generated by the agent directly from the wiki files of each project.

7.4.2.3 Format management

The three template files were originally created in different formats: the One Pager in PowerPoint (single slide), the Model Card in Markdown, and the EU AI Act Risk Categorization in Word. For a better integration with the LLM, all templates

were converted into the unified Markdown format. This format is easier to read and manipulate for an LLM, and the conversion was carried out using Microsoft's Markitdown library.

However, converting the One Pager back from Markdown to PowerPoint introduces formatting challenges, particularly regarding the placement of text boxes on the slide. Therefore, a manual review may be required to ensure the output remains consistent with the original template.

7.4.2.4 Tools integrated in the agent

Once the template files, their formats, and the methods for accessing the information to populate them were defined, the tools needed for the agent to work with these files were determined.

In this case too, to create the agent it is enough to use the `create_react_agent` method of LangGraph as follows:

```
agent = create_react_agent(
    model=llm,
    tools=[...]
)
```

The tools integrated in the agent in this case are:

- **list_files**: given the path of a folder, returns the list of files inside it. For security reasons, an upper limit has been set beyond which the agent cannot access the local filesystem, meaning the root directory is set as the SharePoint folder containing the project directories.
- **read_file_markdown**: given the path of a Markdown file, returns its content as a string.
- **write_file_markdown**: given the output path and the content in Markdown format, creates a Markdown file at the indicated path.
- **convert_file_to_markdown**: given the path of a file, returns its content in Markdown format as a string using the Microsoft Markitdown library. This library supports a wide range of formats, including: PDF, PowerPoint, Word, Excel, images (with EXIF metadata and OCR), audio (with EXIF metadata and voice transcription), HTML, text formats (CSV, JSON, XML), ZIP files (with content analysis), YouTube URLs, EPUB, and more.
- **convert_markdown_to_word**: converts a Markdown file to Word format using the pypandoc library [27], saving the result in the same folder.

- **convert_markdown_to_power_point**: converts a Markdown file to PowerPoint format, also using pypandoc, saving the result in the same folder.
- **setup_project_template_files**: given the path of a project folder, checks for the presence of all template files and their Markdown versions. If any are missing, it copies the missing files from the local template folder (which contains all the original template files), including the Markdown version if available. This tool is useful both to initialize a new project folder and to check the completeness of files without overwriting those already existing, creating only the missing ones.
- **get_datetime**: as in the other agents, provides the LLM with a time reference, useful when the user refers to time-related information.

Note: the template files in the local template folder have been modified by adding a placeholder `{{write_here}}` in the points where the agent must insert information. During testing, without these placeholders, it was difficult to indicate to the agent the exact insertion points, especially in cases where the files contained tables to fill. The manual introduction of placeholders has therefore greatly simplified the process, providing a clear and direct indication to the LLM of where to insert the data.

7.4.3 Azure DevOps

The last AI agent is dedicated to integration with the Azure DevOps platform. At the beginning of the exploration, problems similar to those encountered with SharePoint emerged, mainly related to API integration and access management. However, towards the end of June, Microsoft released the official MCP server for Azure DevOps in public preview [28]. It is important to note that there are many unofficial versions of the MCP server, even for Microsoft services such as DevOps and SharePoint. For security and reliability reasons, it was decided to use only official solutions, reducing the risk of unexpected behavior or incompatibilities.

The integration of the official MCP server, although still in preview, allowed progressive testing of its features and observation of improvements over time.

To enable interaction with the LLM, the MCP protocol was implemented, creating a custom client capable of communicating with the tools available on the server. Although conceptually simple, integrating third-party tools can introduce complexity, especially in ensuring a coherent connection with the agent's execution graph.

Following this implementation, an AI agent dedicated exclusively to interaction with Azure DevOps was developed, making all MCP tools available on the platform accessible to the user.

To create the MCP client, the following procedure was followed:

```
client = MultiServerMCPClient({
    "Azure DevOps": {
        "command": "npx",
        "args": ["-y", "@azure-devops/mcp", self.ado_org],
        "transport": "stdio",
    },
 })
```

Once the connection was established, the available tools were obtained using: `tools_mcp = await client.get_tools()`

Finally, as in the previous cases, the agent was created using LangGraph's `create_react_agent` method.

Thanks to this configuration, it is possible to access numerous ready-to-use tools, including:

- **wit_list_backlog_work_items**: retrieves the list of backlog items for a specific project, team, and backlog category.
- **wit_add_child_work_items**: creates one or more child work items of a certain type, associated with a parent ID.
- **core_list_projects**: lists the projects present in the Azure DevOps organization.
- **repo_list_repos_by_project**: retrieves the list of repositories for a specific project.
- **search_wiki**: performs searches within the wikis.

The DevOps MCP server is constantly evolving, with more and more tools being made available to developers who want to integrate these functionalities into their AI applications.

At the same time, some important safeguards have been introduced. For security reasons, deletion or irreversible modification of content (such as removing PBIs or structurally changing projects) is currently not allowed, in order to prevent potentially harmful behavior by the agent.

Access to these tools is also protected by an authentication mechanism. When the POD Assistant starts, a browser window opens, allowing the user to log in with their corporate credentials. Once the login is completed, the personal access token is acquired and sent to the MCP server, which can then automatically manage access to information in compliance with company policies.

This integration between authentication and permissions ensures that the system can tailor its responses. For instance, if the user asks "Which PBIs are assigned to me for project X?", the MCP server can retrieve the user's name from the credentials and return the correct information filtered according to specific permissions.

At the current stage, it is only possible to use the available tools individually and directly. However, as the MCP server becomes more stable, these foundations will enable the development of more complex workflows, supporting automatic processes such as guided generation of user stories, automatic classification and prioritization of PBIs, or the automation of repetitive tasks in project management.

7.5 The Graph

Once the AI agents were defined, the next step was to design an architecture capable of managing them efficiently, ensuring execution according to a clear and structured workflow. This architecture takes the form of a graph created with LangGraph, the framework introduced in Chatper 3, specifically designed to orchestrate multiple agents within modular and flexible structures.

This approach allowed the creation of a system able to coordinate different activities while maintaining strong operational consistency and facilitating potential future extensions of the project. The resulting graph is shown in the image below.

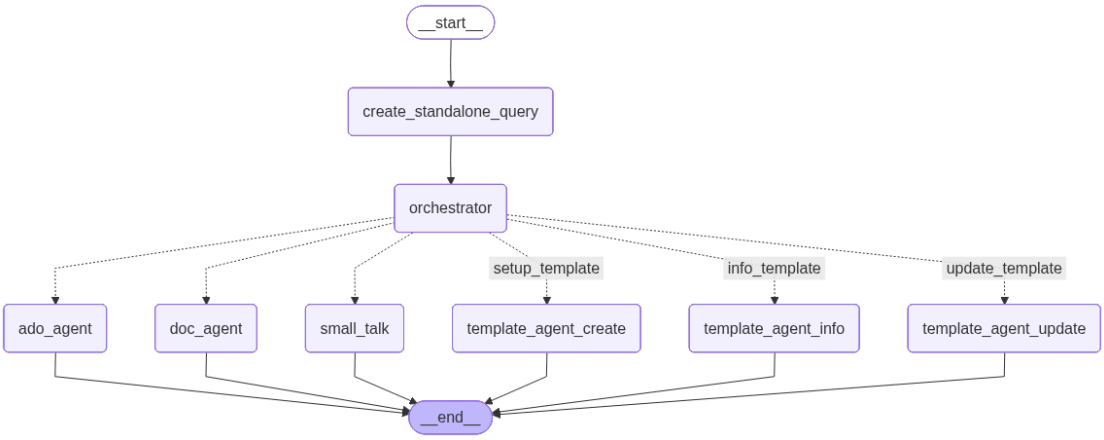

Figure 7.2: The LangGraph architecture of the POD Assistant.

The graph is executed every time the user sends a message, and its processing ends only when the end node is reached. From the image, two abstract nodes provided by the framework can be distinguished: the start node, which indicates the entry point and the beginning of processing, and the end node, which represents the graph's exit and the moment when the updated state is returned.

Since these two nodes were already discussed in the introductory section dedicated to LangGraph, the following focuses exclusively on the other nodes.

The next section explains in detail the creation of the graph, the implementation of each node and their functionalities.

7.5.1 The State

Before going into the details of the individual nodes of the graph, it is essential to define the state of the graph and the various fields that make up its class.

The state represents the set of information that is passed from node to node and updated at the end of each node's execution with the values produced as output. These values, by default, overwrite the corresponding previous fields of the state.

For example, if a node returns:

```
return {"standalone_query": query}
```

the `standalone_query` field of the state is updated with the value contained in query, while all other fields remain unchanged.

Below is the definition of the GraphState class, whose fields will be explained in more detail during the presentation of the nodes.

```
class GraphState(BaseModel):
    messages: List[BaseMessage]
    standalone_query: Optional[str] = ""
    user_intent: Optional[str] = None
    project_intent: Optional[str] = None
```

7.5.2 The Nodes

7.5.2.1 Standalone query

Immediately after the start node, there is the `create_standalone_query` node, whose task is to generate a standalone query, that is a self-sufficient query that can be understood by the LLM without directly relying on the entire conversation history. In other words, this query incorporates all the necessary context, reconstructed from previous interactions with the user.

The interface built with Gradio allows retrieving the entire list of messages exchanged between the user and the AI assistant and using it as context. If a new chat is created, the history is reset and Gradio sends to the backend only the messages present in that conversation. This way, it is possible to manage multiple independent chats, each with its own distinct context.

A recurring problem is that during the conversation, the user tends to make implicit references to previous messages without repeating all the information already provided (a natural behavior in human dialogue). If only the user's last question were sent to the LLM without context, the model would hardly be able to interpret it correctly.

To solve this problem, it is necessary to provide context to the LLM, which can be managed in different ways: pass the entire list of previous messages directly; generate a summary of the history when it becomes too long and provide it along with the latest interactions; or, as chosen in this project, use the history to build a single standalone query, able to clearly and completely express the user's request without requiring further external details.

For example, consider an interaction without using a standalone query:

User: Can you summarize the model card of project X?
Assistant: The model card of project X describes objectives, datasets used, performance metrics, and known model limitations. Here are some details: ...
User: And the risk categorization?
Assistant: I did not understand, can you specify which model or document you are referring to?

In this case, the system loses the thread of the conversation: it does not connect that "the risk categorization" refers to project X, and in particular to the EU AI Act Risk Categorization document.

With the introduction of the standalone query, the same exchange becomes:

User: Can you summarize the model card of project X?
Assistant: The model card of project X describes objectives, datasets used, performance metrics, and known model limitations. Here are some details: ...
User: And the risk categorization?
Standalone query generated: Summarize the EU AI Act Risk Categorization of project X.
Assistant: According to the EU AI Act Risk Categorization, project X falls into the high-risk category because it uses sensitive data and supports critical financial decisions. Here are some details: ...

Thanks to the standalone query, the LLM receives sufficient context to respond coherently and accurately, avoiding ambiguity. This node is therefore fundamental: its implementation affects all conversations, and prompt calibration is crucial to ensure reliable results.

From a technical point of view, the implementation of the node consists of a

simple call to the LLM with a targeted prompt that instructs the model to generate a self-sufficient query. The input is the list of messages returned by the Gradio frontend, which includes the current conversation history. Below is a snippet of the code:

```
standalone_query_prompt = ChatPromptTemplate.from_messages([
            ("system", prompts.STANDALONE_QUERY_PROMPT),
            ("human", "{chat_history}")
        ])

structured_llm = self.llm.with_structured_output(StandaloneQuery, method="function_calling")

chain = standalone_query_prompt | structured_llm

standalone_query = await chain.ainvoke({"chat_history": chat_history_text}, config=self.config)

return {"standalone_query": standalone_query.query}
```

7.5.2.2 Orchestrator

Immediately after the generation of the standalone query, the orchestrator node comes into play. Its tasks are to:

- classify the user's intent into one of the following categories:
  - **small_talk**: general conversation with the LLM;
  - **doc_agent**: questions related to documentation;
  - **ado_agent**: interactions related to DevOps and associated activities;
  - **setup_template**: creation of project files based on templates;
  - **update_template**: updating existing project files;
  - **info_template**: reading or requesting information about project files.
- extract the project information (if present), so it is already available for the following nodes.

Since the intents related to template files are three (setup_template, update_template, info_template), it was decided not to concentrate all the logic in a single node, avoiding the risk of ambiguity or unexpected behaviors by the agent. A modular solution was therefore adopted, creating three separate nodes, each dedicated to a single intent. In this way, even though the same agent responsible for managing template files is always called, each node operates with a specific system prompt, calibrated according to the user's goal.

Coming back to intent detection, extracting as much information as possible from the start makes it possible to obtain many useful pieces of information for managing the graph flow with a single call to the LLM.

An example of behavior can be the following:

User query: "Show me the model card of project BlueBot"
Detected intent: info_template
Detected project: BlueBot

so in this case the flow continues to the node specialized in reading template files.

If instead the user wants to start an operation with DevOps, the orchestrator classifies:

User query: "List all the workitems assigned to me"
Detected intent: ado_agent
Detected project: not specified

and the flow moves to the node that manages interactions with DevOps, ignoring the part related to projects.

Regarding the implementation of the node, to ensure safe and controlled classification, the orchestrator uses a structured output based on Pydantic. In particular, a class IntentOutput is defined that inherits from BaseModel and constrains the format of the LLM output. This approach avoids complex post-processing and reduces the risk of non-compliant responses.

Below is a code snippet:

```
1 structured_llm = self.llm.with_structured_output(IntentOutput, method="
      function_calling")
2
3 chain = intent_prompt | structured_llm
4
5 result = await chain.ainvoke({"input": state.standalone_query}, config=
      self.config)
```

The definition of the class IntentOutput used in the project is as follows:

```
class IntentOutput(BaseModel):
    intent: Literal["small_talk", "doc_agent", "ado_agent", "
    setup_template",
    "update_template", "info_template"] = Field(
        ...,
        description="The user's intent could be simply to chat with the
    LLM (small_talk), to interact with an agent for asking questions
    about the documentation (doc_agent), " \
        "to work with an agent to read or list project files based on a
    template (info_template), to work with an agent to create project
    files based on a template (setup_template), "\
        "to work with an agent update project files based on a template (
    update_template) or to interact with an agent specialized in DevOps
    and activity related to that (ado_agent).",

    )

    project: Literal["BlueBot", "ChatBotSitoRealeMutuaIta", "
    HDAssistantAI", "other"] = Field(
        ...,
        description="The name of the project directory the user is
    referring to"
    )
```

To ensure consistency in classification, the intent property of the IntentOutput class is defined using Pydantic's Literal type. This way, the LLM is constrained to return only one of the predefined values (small_talk, doc_agent, ado_agent, setup_template, update_template, info_template), preventing interpretation errors or ambiguity.

Thanks to this architecture, a single call to the LLM extracts both the intent category and, if available, the information about the relevant project. These data are essential for correctly routing the flow to the appropriate node.

Once the intent and any project information are determined, they are returned as output to the node and saved in the global state:

```
return {"user_intent": intent, "project_intent": project}
```

7.5.2.3 Small talk

Once the user's intent is determined, the graph branches into six distinct paths, each leading to a node dedicated to handling a specific intent.

The first branch concerns "small talk" interactions, meaning conversations that do not require the activation of specialized agents or the use of external tools. This category includes greetings, general questions about the assistant's features, comments or feedback from the user, and, more generally, any requests that are not directly related to projects, documentation, or DevOps tools.

Handling these interactions correctly is crucial both to ensure a smooth user experience and to clearly define the operational boundaries of the AI assistant. Since the assistant is designed to support the POD work, it is essential that it does not provide responses on topics outside the company context. For this reason, the system prompt of the agent includes clear constraints: the assistant can only answer questions related to projects, agile working methodology, and the internal organization of the team.

From an implementation point of view, the "small talk" node makes a direct call to the LLM, passing as input the standalone query generated in the previous node. The prompt not only includes the necessary context but also incorporates the constraints described above. The LLM output is then returned by the node and saved in the graph state to be sent to the user.

Below is the code snippet used to call the LLM and obtain the output of the "small talk" node:

```
prompt = ChatPromptTemplate.from_messages([
    ("system", prompts.SMALL_TALK_PROMPT),
    ("human", "{input}")
])

chain = prompt | self.llm

response = await chain.ainvoke({"input": state.standalone_query}, config
   =self.config)

return {"messages": [AIMessage(content=response.content)]}
```

As output, the node returns the response generated by the LLM as an AIMessage, which is forwarded to the frontend for display. At this stage, the messages list in the state is replaced with only the response produced by the LLM.

In the next interaction, the frontend will again send the entire conversation history, allowing the messages field in the state to be repopulated and used to create the new standalone query.

For a more detailed description of history management and interaction with the frontend, see the section dedicated to Gradio 7.6.

7.5.2.4 Q&A on documentation with doc_agent

The second branch of the graph is dedicated to handling questions related to documentation, such as the Scrum Guide or the company's operative model. Unlike the "small talk" node, in this case the LLM is not queried directly; instead, the specialized agent doc_agent is used, designed to handle these types of requests.

The node's role is therefore limited to building the prompt and invoking the agent, which has the necessary tools to access and use the documentation, supported

by a dedicated system prompt that guides its behavior. Here is the chain build in this node:

```
chain = prompt | self.doc_agent
```

As in the previous case, the output consists of the message generated by the agent, which is saved in the messages field of the state and then forwarded to the frontend for display.

7.5.2.5 Interation with Azure DevOps with ado_agent

The third branch of the graph handles interactions with the Azure DevOps platform. As in the previous node, a call is made to the specialized agent ado_agent, designed to operate with the tools available on the Azure DevOps MCP server.

At this stage, the prompt provided to the agent is quite general, since the exact functionalities of the chatbot related to DevOps have not yet been formalized. Consequently, the agent has access to all tools available on the MCP server, without a workflow-bound purpose.

The content of the node is therefore very similar to that of the doc_agent node, with the only difference being the invoked agent:

```
chain = prompt | self.ado_agent
```

Again, the output consists of the message generated by the agent, which is saved in the messages field of the state and forwarded to the frontend for display.

7.5.2.6 Management of the template files with template_agent_create, template_agent_update, template_agent_info

Since the three nodes dedicated to managing the template files all call the same agent, template_agent, which is equipped with specific tools for file management, they can be described together.

The first node, template_agent_create, invokes the specialized agent to create the template files. The prompt provides precise instructions: check that all necessary template files and their Markdown versions are present in the project folder indicated by the user; fill the files with the information available in the project wiki; report any missing information so that the user can add it.

The second node, template_agent_update, handles the updating of template files and the project wiki. The agent first updates the wiki with the new information received from the user and then updates the template files in both their original and Markdown versions.

The third node, template_agent_info, provides the user with the content of the template files, explains their meaning, summarizes them, or provides general information about the project folder.

In practice, all three nodes share a structure similar to the previous nodes: they invoke the agent with a specific prompt and return as output the message generated. The project name, already detected in the orchestrator node and saved in the state, must be provided as input, making the procedure straightforward:

```
prompt = ChatPromptTemplate.from_messages(
    [("human", prompts.TEMPLATE_AGENT_SETUP_PROMPT)]
)

chain = prompt | self.template_agent

response = await chain.ainvoke({"project": state.project_intent}, config=
    self.config)
```

The only difference between the three nodes lies in the prompt used; otherwise, the structure is identical, and the agent's message is returned as output, as in the other nodes of the graph.

7.5.3 The Edges

Once the nodes are implemented, it is necessary to define the relationships between them to build the complete structure of the graph. As explained in the introductory section on LangGraph, there are different types of edges. If the relationship between two nodes is deterministic, a standard edge is used, indicating that one node must be executed after the previous one. If instead the choice of the next node depends on parameters present in the state, conditional edges are used.

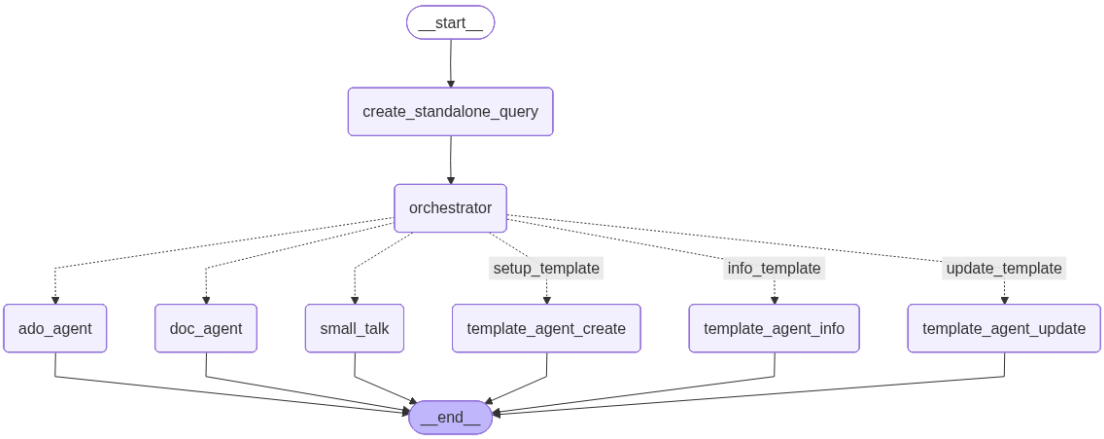

Figure 7.3: The LangGraph architecture of the POD Assistant.

As it can be seen from the image presented in the previous page and generated using the draw_mermaid_png method of LangGraph, two main types of edges can be observed: solid lines for standard edges and dashed lines for conditional ones. In this simple architecture, all edges are standard except for those originating from the orchestrator node, which are conditional. It is important to note that all nodes invoking an agent or the LLM have an edge ending at the end node, ensuring the completion of the graph execution and the return of the updated state to the frontend, responsible for displaying the interactions.

Below is the code used to build the graph with LangGraph, defining nodes and edges:

```
memory = MemorySaver()
graph_builder = StateGraph(GraphState)

graph_builder.add_node("create_standalone_query", self.
    create_standalone_query)
graph_builder.add_node("orchestrator", self.detect_intent)
graph_builder.add_node("small_talk", self.small_talk)
graph_builder.add_node("doc_agent", self.node_doc_agent)
graph_builder.add_node("ado_agent", self.node_ado_agent)
graph_builder.add_node("template_agent_create", self.
    node_template_agent_create)
graph_builder.add_node("template_agent_update", self.
    node_template_agent_update)
graph_builder.add_node("template_agent_info", self.
    node_template_agent_info)

graph_builder.add_edge(START, "create_standalone_query")
graph_builder.add_edge("create_standalone_query", "orchestrator")
graph_builder.add_conditional_edges(
            "orchestrator",
            lambda state: state.user_intent,
            {
                "small_talk": "small_talk",
                "doc_agent": "doc_agent",
                "ado_agent": "ado_agent",
                "setup_template": "template_agent_create",
                "update_template": "template_agent_update",
                "info_template": "template_agent_info"
            }
        )
graph_builder.add_edge("small_talk", END)
graph_builder.add_edge("doc_agent", END)
graph_builder.add_edge("ado_agent", END)
graph_builder.add_edge("template_agent_create", END)
graph_builder.add_edge("template_agent_update", END)
graph_builder.add_edge("template_agent_info", END)

compiled_graph = graph_builder.compile(checkpointer=memory)
```

As can be seen from the code provided, to create a graph with LangGraph, the following steps are taken:

- Define a StateGraph object, passing the custom state class.

- Add the nodes, specifying for each one the function to execute and the node name.
- For conditional edges, use the add_conditional_edges function with a lambda expression indicating all possible destinations based on the value of a state variable (in this case user_intent).
- Add the standard edges between the nodes.
- Compile the graph and manage short-term memory using the MemorySaver object provided by the LangGraph library.

7.6 Frontend with Gradio

The chatbot frontend was built using Gradio [29], a Python library that allows the creation of interactive web interfaces in a simple and fast way. The application appears as a traditional chat and includes a complex flow for state management, communication with the LangGraph, and streaming of the responses generated by the model.

The management of the message history is consistent with the architecture of the LangChain library, which represents each interaction as an object derived from BaseMessage. When a user sends a new message, the system converts the history maintained by the Gradio interface into a list of semantic messages: each user message becomes a HumanMessage, while each previous chatbot response is transformed into an AIMessage. In this way, the conversation state is not reduced to a simple sequence of strings but is reconstructed as a structured context, compatible with the LangGraph and its execution model.

After the conversion, the new user input is added to the list, which becomes the base of the state to be passed to the graph. This way, every time the user writes a message, the entire message history in that chat is sent to LangGraph.

If a new chat is created, the message history of the previous chat is no longer displayed, and only the new interactions are sent to the graph. This solution may seem suboptimal, and a more complex system with long-term memory management could be implemented, but considering a future production deployment, it was decided to follow the same methods currently used by other corporate solutions that manage chat history directly through the frontend, without backend memory management.

The interaction with the LangGraph is managed using the asynchronous method `astream`. For each new user message, a graph runner is created or retrieved, initialized with the conversation state represented by a GraphState object. This state contains the sequence of previous messages and allows the graph to modulate its execution according to the context. The graph execution does not produce a

single immediate response but generates a sequence of chunks that progressively represent parts of the message produced by the model or intermediate outputs generated by specific nodes in the graph, such as tool activations. Each chunk is then analyzed to determine whether it is model-generated content (for example, an AIMessageChunk) or a message related to tool invocation (ToolMessage). In the first case, the generated text is concatenated with the already collected content and streamed to the interface, allowing the user to see the response forming in real time. In the second case, the interface displays a temporary message indicating the ongoing activity, such as the invocation of an external tool.

The streaming mechanism is a central aspect of the user experience. Instead of waiting for the model to produce the entire response, the interface processes and shows the text chunks as they become available. This approach replicates the experience of modern chats and improves the perceived responsiveness of the system. From a technical point of view, streaming is managed through an asynchronous loop that monitors incoming chunks and, whenever new text content is received, forwards it to the Gradio interface with a slight artificial delay that helps make the typing rhythm smoother and more natural. If the graph produces special messages related to tools, the interface displays a placeholder with an icon and a brief description of the ongoing operation, as shown in the following image:

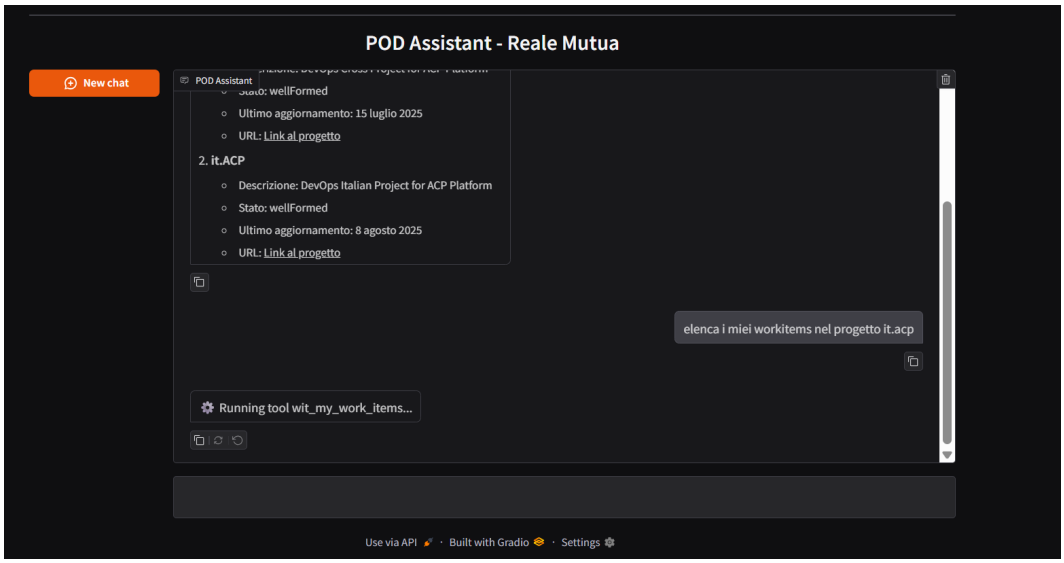

Figure 7.4: Screen capture of the frontend developed using Gradio.

Overall, the frontend built with Gradio does not simply act as an input and output layer, but plays an active role in maintaining the conversational context, orchestrating the message exchange with LangGraph and managing the streaming. For additional images of the Gradio interface, refer to the appendix in Chapter 10.

7.7 Experiments and results

Before arriving at the architecture presented in this chapter, several experiments were conducted to test the agents' ability to manage the tools at their disposal.

One of them focused on the degree of freedom granted to the AI agents. Multiple versions of graphs were created with LangGraph, each designed according to the level of control intended for the agents. A graph with fewer nodes represents a solution with less control and greater freedom of decision-making for the agents, while a graph with many nodes resembles a nearly deterministic workflow, where the desired sequence of actions is explicitly defined and agents are forced to use specific tools or perform well-defined tasks at each step.

The initial approach involved creating a small graph with just a few nodes, granting agents a high level of autonomy, and testing the results on the various tasks described in the previous sections. The outcome, however, was disappointing: agents often failed to understand what they were supposed to do and, if they misinterpreted a user query, would misuse tools and return incorrect outputs. This behavior could even be risky when working with files or directly accessing company platforms such as DevOps. In many cases, files were overwritten, or the agents fell into loops of tool calls until hitting the maximum iteration limit. Such issues typically arose when user requests were unclear, and, without a well-defined workflow guiding them, agents resorted to calling whatever tools they deemed appropriate.

An alternative solution involved building a graph with LangGraph, where agent behavior was explicitly constrained within a well-defined workflow. This resulted in a graph with many nodes, making the chatbot's behavior almost deterministic. In this setup, agents had very limited autonomy and were triggered with precise prompts that specified the actions to perform. The main drawback of this approach was the need to anticipate every possible scenario; overlooking even a single case could lead the system to provide incorrect answers or exhibit unintended behaviors. The results were satisfactory only when the user's request was correctly identified and the corresponding portion of the graph was executed. Overall, this solution proved fragile and error-prone, since it forced agents to operate in ways they were not designed for.

The final solution was therefore a compromise between the two extremes, resulting in the graph presented in this chapter. This approach achieved reliable outputs by giving agents a balanced degree of freedom: they were guided by the instructions defined in each node but not forced into overly specific tasks that would have made the workflow too rigid and reduced its generalizability.

For this project, no dedicated test dataset was created, as the work was conceived as a proof of concept to explore the technical feasibility of using the MCP protocol for a virtual assistant. Since the solution was intended for internal purposes only, no business stakeholders were involved to evaluate its practical value. At the time of development, both the evaluation criteria and the chatbot's final features were still undefined. The main objective was to better understand the MCP protocol and assess its potential for integration into company projects. For this reason, no formal testing sessions were carried out to validate the results.

Chapter 8

CGA Copilot

After presenting the first project carried out during the internship, this chapter focuses on the second project: the CGA Copilot. This Proof of Concept addresses a different yet equally important challenge: making it easier for insurance agents to consult and interpret complex technical documents, such as the General Conditions of Insurance (CGA).The chapter describes the project objectives, the solutions implemented, and the strategies adopted to optimize access to the information contained in insurance documents.

8.1 Introduction

The goal of this PoC is to develop a chatbot capable of supporting insurance agents in interpreting technical insurance documents. In particular, the focus is on the General Conditions of Insurance (CGA), documents that are typically long and complex, detailing available coverages, exclusions, deductibles, policy limits, and other contractual clauses.

The main purpose of the chatbot is to simplify the consultation and search for information contained in the CGA, making it easier to understand technical aspects and reducing the time needed to find specific answers.

The approach followed in this work is based on a traditional RAG model. Documents are first split into highly precise chunks and indexed in a vector database to enable accurate retrieval. Before querying the vector database, query rewriting is applied to refine and optimize the queries, improving the relevance of the retrieved content. The rewritten queries are then executed within a well-structured workflow built with LangGraph, enabling the execution of multiple queries and providing structured control over the data flow to the language model. This combination ensures that the retrieved information is both comprehensive and precisely aligned with the user's intent before being fed into GPT-based models.

8.2 The problem

The project was born from the need to facilitate access to and understanding of a complex insurance document: the "*Condizioni Generali di Assicurazione*" (CGA) of the product "*Azienda Reale*" aimed at companies and provided by Reale Mutua. This is a legal and technical document, written in PDF format, containing detailed information on insurance conditions, offered coverages, policyholder obligations, exclusions and coverage limitations, as well as specific sections such as a glossary of the terms used.

The complexity of the document represents a significant challenge for users: its length, structured layout, and specialized terminology make it difficult to find specific information and fully understand the content. Users often need to navigate through many sections and interpret technical terms, which can be frustrating and inefficient.

The main problem, therefore, concerns the ability of policyholders to quickly access relevant information and understand its meaning without reading the entire document. The goal is to reduce search time, improve the accuracy of responses to user questions, and make technical terminology clearer.

General Conditions of Insurance, such as those addressed in this PoC, can be very long and detailed. The document used, for example, is 170 pages long and divided into ten sections, each corresponding to a purchasable coverage. Each section contains detailed lists of covered and non-covered items, as well as summary tables of deductibles, excesses, and indemnity limits.

Given that these are complex and highly structured documents, it is essential to create an accurate RAG system capable of retrieving information only from the relevant coverages. Many sections address similar topics but refer to different coverages that can be purchased separately. Therefore, it is crucial to select only the chunks that are actually relevant to correctly answer user questions, detecting the intent of the request and accurately identifying the section of the document it refers to.

A fundamental requirement of the chatbot is also to always return the page or range of pages from which the information was extracted, ensuring traceability and transparency of the answers. This is particularly important in regulated or corporate contexts, where the user must be able to quickly verify the source of the information obtained.

In documents like CGA, which contain similar sentences spread across different sections, correctly answering user questions is a challenge for traditional RAG implementations that rely solely on similarity between query and chunks. To provide reliable answers, the system must adopt a more targeted and contextualized approach, capable of identifying the correct section, extracting the relevant information, and indicating its origin in the document.

8.3 The AI solution

To address the problem of retrieval and response generation on complex documents such as CGA, a solution based on an advanced RAG system integrated with conversational orchestration architectures was developed. The main goal was to ensure that the chatbot's answers were complete, accurate, and contextualized, despite the length and complexity of the document.

The solution starts with data preprocessing: the original PDF is transformed into a structured index in JSON format, representing titles, numbering, and hierarchy of sections, chapters, paragraphs, and subparagraphs, along with their start and end pages. This approach allows mapping each portion of text to a specific node, maintaining traceability and context. Chunks that are too long or unclear, such as glossary entries or extended sections, are further segmented using LLMs to create coherent and manageable informational units for retrieval.

Once segmented, the chunks are transformed into embeddings and indexed in a vector database on Azure AI Search. Here, the combination of textual, vector, and semantic search allows retrieving relevant information even from complex documents, using cosine similarity and nearest neighbor algorithms like HNSW. The hierarchical path of the chunks is included in semantic queries, ensuring that the system can provide responses with precise references to the original content's position in the document.

For query orchestration and conversational context management, a graph was built with LangGraph. Thanks to this structure, the chatbot can distribute queries across multiple sections of the document, combining complementary information and reducing the risk of providing incomplete answers. For example, when a question is asked about whether something is covered by a specific coverage, the system can simultaneously search both the chapter on insured items and the chapter on excluded items, providing a more comprehensive answer.

Moreover, each user query is enriched by the LLM itself, which reformulates it in technical-insurance language and integrates the context of previous interactions. This process improves the relevance of the answers and ensures greater consistency in the output generated by the chatbot.

Overall, this system effectively handles structured and complex documents, delivering precise and contextualized answers with source traceability, and ultimately offering a far better consultation experience than reading the PDF directly.

8.4 Data preprocessing

The first step in developing the RAG system involved data preparation, a crucial element to ensure accurate and consistent retrieval. In particular, a JSON file was generated to represent the hierarchical structure of the document, including titles, section, chapter, paragraph, and subparagraph numbering, as well as their respective start and end pages.

To obtain this structure, the pages corresponding to the document's index were first extracted from the PDF and converted into Markdown format using Azure Document Intelligence, which employs OCR techniques for precise text extraction. Subsequently, an LLM (GPT-4.1) was used to automatically transform the raw text into a JSON file matching the desired structure, eliminating the need for manual operations and laying the foundation for a fully automated and scalable preprocessing process for multiple documents.

A sample of the generated JSON file is shown below:


```
{
 "sections": [
  {
   "title": "GLOSSARIO",
   "start": 15,
   "end": 28,
   "chapters": []
  },
  {
   "title": "NORME COMUNI A TUTTE LE GARANZIE",
   "start": 29,
   "end": 35,
   "chapters": [
    {
     "number": "1",
     "title": "OBBLIGHI DEL CONTRAENTE/ASSICURATO",
     "start": 29,
     "end": 30,
     "sub_chapters": [
      {
       "number": "1.1",
       "title": "Dichiarazioni relative alle circostanze influenti sulla valutazione ...",
       "start": 29,
       "end": 29,
       "sub_sub_chapters": []
      }
     ]
    }
   ]
  }
 ]
}
```


After creating the index, the remaining pages of the document were also converted into Markdown format and subsequently segmented into chunks corresponding

to sections, chapters, paragraphs, and subparagraphs, each associated with the corresponding node in the JSON index. The chosen approach was to use as a basic unit any portion of text between two consecutive titles, regardless of the hierarchical level (section, paragraph, or subparagraph).

To perform this operation, the index was "flattened" into an ordered list of titles, removing the nested hierarchical structure. Using the page information already included in the index, it was then possible to extract the Markdown content of the relevant pages and, with the support of regular expressions, isolate the text between one title and the next. It was necessary to include the page following the reference page, as the next title could appear beyond the page indicated as the end of the section.

Three fields were then added to the JSON index for each entry:

- **text**, containing the portion of the document associated with the title;
- **path**, describing the full hierarchical path of the entry, ensuring traceability and precise identification of the source of the information;
- **context**, providing additional text from the same sub-chapter as the chunk, which offers more context to the LLM during retrieval without introducing unrelated or misleading information.

This last field is particularly important because, during the RAG process, the vector search is performed using the embedding of the *text* field, but the *context* field is returned instead. The *context* field is longer than the *text* field and therefore provides the LLM with more contextual information to generate a more accurate response. This approach has also been adopted in other company projects and has proven useful for producing precise answers by supplying additional context. Only the chunks immediately before and/or after the current chunk, belonging to the same sub-chapter and covering the same topic, are included as context. This ensures that the LLM receives relevant information without being confused or misled by unrelated content.

However, during preliminary tests, it became apparent that some chunks were still too large, particularly those spanning multiple pages. The glossary, for example, consists of 14 pages and contains numerous terms, making it difficult to query efficiently. To address these challenges, each glossary entry was treated as an independent chunk, and chunks larger than 4000 characters were further subdivided with the support of an LLM. The LLM identified logical subdivisions not present in the original index and added them as an additional hierarchical level in the JSON file.

At the end of the process, the document was represented by 593 structured chunks, each enriched with a textual embedding of the *text* field, making it ready for the indexing phase.

8.5 Creation of the Index on Azure AI Search

Once the chunks were prepared, they were uploaded to a vector database based on Azure AI Search, which became the core of the retrieval system. The index was designed to support textual, vector, and semantic searches in a hybrid mode, thus taking full advantage of the platform's capabilities.

The index was designed to have the following set of fields:

- **id**: primary key of the document;
- **text**: textual content indexable and searchable with the IT_LUCENE lexical analyzer (optimized for Italian);
- **context**: textual content to retrieve;
- **title** and **path**: additional metadata to improve retrieval context;
- **page_start** and **page_end**: references to the origin pages of the chunk;
- **embeddings**: vector representation of the *text* field (3072 dimensions), obtained through the embedding model integrated via LangChain.

Indexing was enhanced with two key configurations:

- **Vector Search**: based on the HNSW (Hierarchical Navigable Small World) algorithm with cosine similarity metric.

  The chosen parameters (m=4, ef_construction=400, ef_search=500) optimized the efficiency and accuracy of vector search.

- **Semantic Search**: enriched with a configuration prioritizing the *title* field and the content of *text* and *path*. This improved the re-ranking of results through the semantic relevance score (reranker_score).

Regarding retrieval, different search methods were tested:

- TEXT: traditional full-text search.
- VECTOR: purely vector-based search.
- SEMANTIC: semantic search based on the integrated language model.
- HYBRID: combination of textual and vector search.
- HYBRID_SEMANTIC: the most advanced mode, combining semantic search with vector retrieval, with final ordering based on the re-ranking score.

Among these, the semantic hybrid mode (HYBRID_SEMANTIC) proved the most effective, as it ensures coherent results even for complex or ambiguous queries.

In semantic hybrid mode, the query is first converted into an embedding and used to perform a vector search, which is then combined with semantic ranking and traditional textual search. This allows the system to handle ambiguous or complex queries, providing the top n most relevant results, enriched with contextual information such as title, path, and reference pages.

A key element in optimizing retrieval was the use of semantic re-ranking.

After the search is executed, the results are further evaluated by the Azure AI Search semantic model, which assigns each chunk a relevance score.

This re-ranking phase is particularly useful when queries contain ambiguity or polysemous terms, as it allows the system to prioritize results that not only match at the textual or vector level but are also closer to the actual meaning of the query.

In practice, the search occurs in two phases:

- **initial retrieval** (textual/vector/semantic), identifying a set of candidate documents;
- **semantic re-ranking**, where candidates are reordered based on the score calculated by the semantic model.

The integration of re-ranking had a significant impact on system quality, ensuring more accurate, coherent, and contextualized responses, especially in scenarios with complex queries or open-ended questions.

8.6 The Graph

The next step involved integrating the data and the vector index with LangGraph, used as a conversational architecture to orchestrate model queries and responses. In this PoC, no agents were implemented: the graph does not manage interactions between multiple agents, but defines a deterministic workflow of calls to individual LLMs, each with a prompt specifying the operations to perform.

Unlike previous projects, where it was necessary to provide diversified tools via agents with specific functionalities, here the task is more limited and can be managed without agents. It is still possible to integrate agents in a similar application, for example one dedicated to retrieval and another responsible for validating the extracted chunks and, if necessary, reformulating queries to improve results. However, this solution may lead to longer response times and the risk of loops if the system does not immediately find a satisfactory answer, reducing the smoothness of interaction.

To ensure reliability and short response times, an almost deterministic graph was chosen, where each node has behavior defined by the prompt and no decision-making autonomy is left.

The main goal of the graph is to improve chunk retrieval in the RAG phase, refining the search and selecting the correct portions of the document.

To achieve this, user queries are reformulated not only to create a standalone query, integrating the context of previous conversations, but also to highlight the key words essential for accurate retrieval. This step is particularly important when the user expresses questions in everyday language or refers to specific real cases. Since retrieval is based on semantic similarity between query and chunks, the presence of terms that are too specific or concrete may lead to selecting irrelevant chunks. Therefore, the query is reformulated in technical-insurance language, and key concepts are abstracted to reflect the style of the CGA documents.

However, during some initial tests, a recurring problem emerged: for questions regarding the coverage of a guarantee, the system mainly retrieved chunks from the chapter *"What can I insure"*, neglecting those from *"What is not insured"*. This caused incomplete or incorrect answers, as the exclusions of the guarantees are described exclusively in the second section. To solve this issue, the system performs two queries to the vector database, one for each chapter, ensuring access to all the information needed to provide accurate answers.

The adoption of the graph thus significantly improved the overall quality of responses compared to a simple RAG system, effectively balancing information from different sections of the document. Moreover, the query reformulation mechanism and the integration of conversational context further increase the relevance of results and ensure consistency in the responses, enhancing the overall user experience.

The resulting LangGraph architecture for the CGA Copilot is illustrated in Image 8.1.

Building on this visualization, we can now proceed, similarly to the approach taken in the POD Assistant project, to describe in detail the graph's architecture, its state, the individual nodes, and their relationships through the edges.

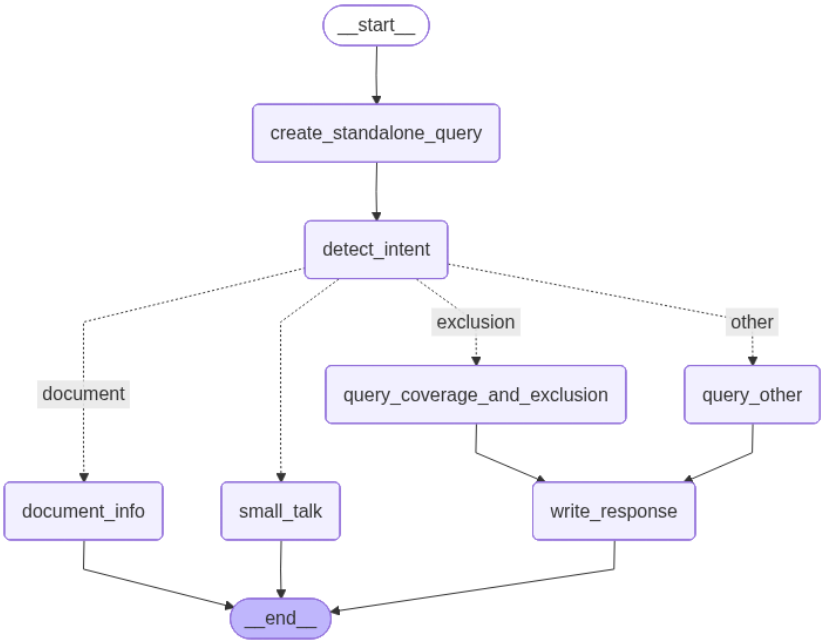

Figure 8.1: The LangGraph architecture of the CGA Copilot.

8.6.1 The State

Before going into the details of the individual nodes of the graph, it is essential to define the state and the various fields that make up its class in this case as well.

Below is the definition of the GraphState class used in this project:

```
class GraphState(BaseModel):
    product: ProductClass
    messages: List[BaseMessage] = []
    standalone_query: str = ""
    retrieved_documents: List[str] = []
    intent: str = ""
```

The fields are:

- **product**: represents the structured information related to the product under analysis. In this PoC, the focus was on a single CGA; however, the architecture has been designed with scalability in mind, allowing the integration of multiple documents in the future. In this specific case, the product considered corresponds exclusively to the CGA *Azienda Reale RN*.
- **messages**: keeps the history of exchanged messages, useful for tracking the conversational context.

- **standalone_query**: represents the user query in an independent form, obtained after a rewriting phase.
- **retrieved_documents**: collects the most relevant documents retrieved by the system and available for the subsequent nodes.
- **intent**: represents the main intent of the user.

8.6.2 The Nodes

8.6.2.1 Standalone query

Immediately after the start node, there is the create_standalone_query node, whose function is identical to that described for the POD Assistant: it generates a standalone query, that is, a self-contained query that includes all the necessary context to be correctly interpreted by the LLM, without relying on the entire conversation history.

As in the project described in the previous chapter, the CGA Copilot graphical interface also allows obtaining the complete list of messages exchanged between the user and the AI assistant, which is used as context for generating the standalone query. In the case of a new chat, the history is reset, and the backend receives only the messages from the current conversation, allowing multiple independent chats, each with its own context.

The node solves the problem of implicit references to previous interactions, typical of human dialogue, by creating a clear and complete query that allows the LLM to respond without ambiguity. For example, if the user asks for information on specific CGA documents, the standalone query automatically includes the necessary references, avoiding misunderstandings due to overly concise questions.

From a technical point of view, the implementation is identical to that of the POD Assistant: the node sends a targeted prompt to the LLM together with the current conversation history. The generated query is then used to obtain coherent and precise responses.

8.6.2.2 Detect intent

Immediately after generating the standalone query, the detect_intent node comes into play, responsible for identifying the user's intent, that can be classified into the following categories:

- **small talk**: general conversations between the user and the LLM;
- **document_info**: requests for general information about the document, its structure, or content not indexed in the Azure AI Search database, which require a separate handling from the RAG;

- **coverage/exclusions**: questions related to guarantees, which require generating a double query to separately interrogate the sections *"What can I insure"* and *"What is not insured"*;
- **other document information**: requests concerning other sections or content of the document.

This classification allows the flow to be correctly routed to the next nodes of the graph, ensuring coherent and relevant responses.

The architecture of the node is based on a model similar to the POD Assistant, but it has been adapted to the document context of the CGA Copilot. A dedicated prompt is constructed to include the standalone query, and the LLM generates a structured output using Pydantic, limiting the response to the intent, which corresponds to the classification of the user's query.

8.6.2.3 Small talk

Once the user intent is determined, the graph can branch into different paths. Among these, as we have seen, one is dedicated to small talk interactions, that is, conversations that do not require the intervention of specialized LLMs or the use of external tools.

The implementation of the "small talk" node in the CGA Copilot is very similar to the one already described for the POD Assistant: a direct call is made to the LLM, using as input the standalone query generated by the previous node. The only significant difference concerns the prompt, which in this case introduces a specific constraint: the assistant must only handle generic or courtesy interactions, without answering questions that go beyond the insurance context and, in particular, the CGA documentation.

In this way, the LLM keeps a behavior consistent with the project goals, answering greetings, thanks or generic requests, but avoiding providing non-relevant information. The output is then returned as an AI message and shown on the frontend, following the same logic of history management described in the previous chapter.

8.6.2.4 Information on the document

When the user intent is about asking for general information on the CGA, the flow of the graph goes to the document_info node. This node has the role of managing specific questions on the insurance documents, using the structure of the document itself, the index and some additional information that can be useful, such as the short description of the CGA at the beginning of the document, which is not included in the index for the RAG.

From an implementation point of view, the node is almost the same as the previous one, the only difference is in the way the dedicated prompt is built to query the LLM. The input is not only the standalone query, but also contextual information such as the product name, its description and the structure of the document. These additional elements allow the model to correctly place the question inside the insurance context.

The answer of the LLM is then wrapped in an AI message and returned as the only output of the node, which replaces the list of messages in the state. In this way the frontend can directly show the answer to the user.

Note: to provide the structure of the document (that is, its index) to the LLM, the choice was made not to insert it manually, but to automate the process to make it also easy to scale. For this purpose, the index was saved on Azure AI Search, a solution that also turned out to be suitable for reasons related to the technological infrastructure used in the PoC.

In particular, to avoid relying on external storage services for the support files, the information was stored in a single chunk and retrieved directly by selecting the content that represents the document's index using the document name as the key. This approach ensures simple and centralized access to the structure of each document.

If in the future it will be necessary to integrate more support documents, it will be enough to add new chunks to the index on Azure and apply a filter on the fields, so that only the relevant information can be quickly retrieved.

8.6.2.5 Query on coverage or exclusion

The query_coverage_and_exclusion node plays a crucial role in the flow, because it is the point where the user standalone query is rewritten to better fit the language of the document, improving the effectiveness of the retrieval.

As highlighted before, it was observed that when the user asked questions about coverage, the RAG system retrieved almost only content from the chapters *"What can I insure"*, while ignoring those of *"What is not insured"*. This aspect is particularly critical: the same case proposed by the user can belong both to coverage and to exclusions, and without a balanced search it would not be possible to give complete and correct answers.

From an implementation point of view, the node works in a way similar to the others already described, with the difference that it is divided into two distinct phases, both based on queries to Azure AI Search.

The first phase, focused on coverage, begins with the construction of a query optimized to identify the sections that address the topic in terms of coverage, using a dedicated prompt. The LLM rewrites the user's question into a form suitable

for vector search, and the resulting query is then used to query Azure AI Search through the search_query method with the parameters described in Chapter 8.5:

```
snippets = azure_search.search_query(query_covered, top=4)
```

The second phase, dedicated to exclusions, relies on a different prompt through which the LLM builds a complementary query aimed at identifying the parts of the document that describe exclusions or non-covered conditions.

This query is also used to query Azure AI Search, and the obtained snippets are added to those of the previous phase.

At the end of the process, the node therefore returns a set of documents that includes both the passages related to coverage and those on exclusions, which are then stored in the state. In particular, the 4 most relevant chunks are retrieved for each of the two queries (coverage and exclusion), for a total of 8 chunks. This ensures a good balance between quality of the answers, generation time and operating costs of the solution.

Note that the retrieved text is not limited to the chunk itself, but encompasses the entire context of the sub-chapter in which it appears, meaning that more text is retrieved than is used for the vector search.

8.6.2.6 Query on other topics

The query_other node has the role of managing all those user requests that do not belong to the particular cases such as coverage and exclusions, and that therefore need a more general formulation of the query.

From an implementation point of view, the node uses a specific prompt that guides the LLM in rewriting the question. The standalone query is in fact rewritten into a format more suitable for vector search, as done for the previous node, keeping the semantic content but adapting it to the terminology and the style present in the document.

Once generated, the query is used to query the search engine through the search_query method in the same way as the previous node. In this case, the first 5 most relevant chunks are extracted after reranking. In this way we can ensure to have enough chunks to answer in an accurate way. This value, as well as the one of the previous node (4 chunks per phase, for a total of 8), was obtained empirically by carrying out a series of tests and checking that it turned out to be the optimal number to provide enough context to the LLM and at the same time avoid making it process too much text, thus increasing the costs. These snippets are finally stored in the state and made available to the next nodes of the graph.

8.6.2.7 Write response

The write_response node represents one of the final steps of the generation flow, where the final answer to be shown to the user is produced. After the intent has been classified and the relevant documents have been retrieved from the search system, this node has the task of integrating the retrieved contents with the user query and with the conversational context, returning a structured and coherent output.

The functioning is relatively straightforward: the already selected documents are passed to the LLM together with eventually additional contextual elements, such as the document index, the product name, the description and the standalone query generated before. These pieces of information are combined in a specific prompt that instructs the model to draft an answer according to a predefined template, aimed at ensuring consistency and clarity.

The template is divided into several sections:

- Initial feedback: a short and immediate answer, that allows the user to get quick confirmation to their request;
- Detail with sources: an in-depth explanation supported by textual citations extracted from the document, with precise references to the source (page number, section, chapter and paragraph);
- Conclusion/summary: a synthesis, useful especially in case of long answers;
- List of sources: a complete and exhaustive list of only the sources used for the answer.

If no relevant documents are found, the node intercepts the condition and returns an informative message that invites the user to reformulate the question, avoiding in this way that the assistant produces arbitrary or unsupported content.

The generated output is an AIMessage, built from the text produced by the LLM. In case of errors in the generation, the exception is handled and a fallback message is produced, so as to ensure continuity of the conversation.

The write_response node therefore plays a key role in the architecture: it represents the synthesis moment where the information processed in the previous steps comes together and is transformed into a final answer, clear and immediately usable by the user.

8.6.3 The Edges

Once the nodes were implemented, it was necessary to define the relations between them in order to build the complete structure of the graph. Below is the image of the graph, generated through the `draw_mermaid_png` method. Also in this case, as for the POD Assistant, most of the connections are of the standard type, while the most relevant branching is the one starting from the detect_intent node, which routes the execution towards different nodes depending on the intent detected in the user query.

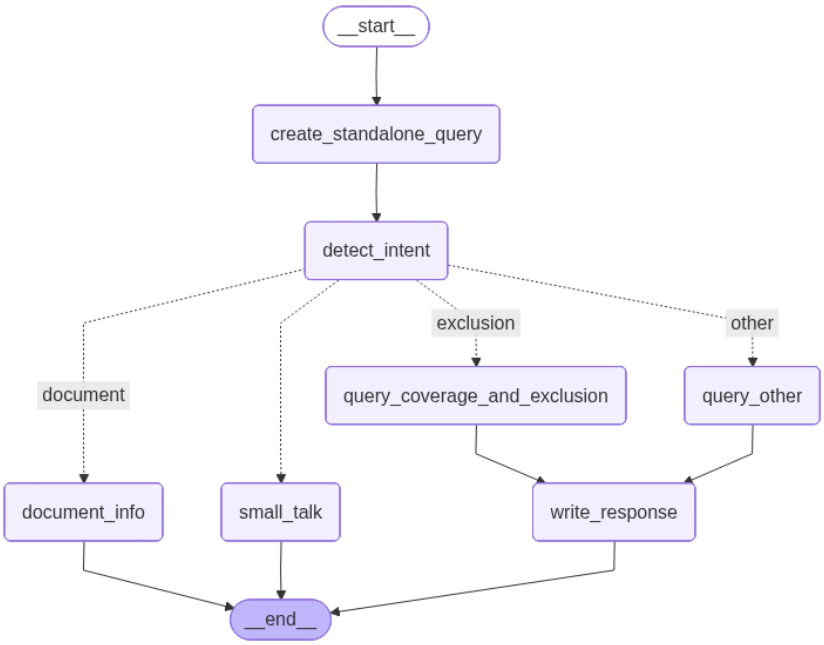

Figure 8.2: The LangGraph architecture of the CGA Copilot.

The basic path of the graph starts from the initial node, connected to create_standalone_query, which has the task of rewriting the user's question in a form more suitable for the search. After that, the execution passes to the detect_intent node, which, thanks to a conditional edge, decides the continuation of the flow based on the value of the intent variable saved in the state. In particular, queries related to coverage and exclusions are directed to the query_coverage_and_exclusion node, while generic requests are routed to query_other. Informational intents lead instead to the document_info node, while lighter conversational interactions are managed by the small_talk node.

Once the relevant contents have been retrieved, the graph converges again towards the response generation phase. Both query_coverage_and_exclusion and query_other flow into the write_response node, which processes the extracted

documents and builds the final answer to be returned to the user. This node, together with small_talk and document_info, represents one of the last steps before the conclusion of the graph: all outgoing edges from them end at the final `END` node, which marks the conclusion of the processing and the return of the updated state.

The following code shows the construction of the graph with LangGraph, including the definition of the nodes, the standard edges and the conditional ones:

```
memory = MemorySaver()
graph_builder = StateGraph(GraphState)

graph_builder.add_node("create_standalone_query",  partial(nodes.
    create_standalone_query, llm=self.llm))
graph_builder.add_node("detect_intent", partial(nodes.detect_intent, llm=
    self.llm))
graph_builder.add_node("query_coverage_and_exclusion", partial(nodes.
    query_coverage_and_exclusion, llm=self.llm, azure_search=self.
    azure_search))
graph_builder.add_node("query_other", partial(nodes.query_other, llm=self
    .llm, azure_search=self.azure_search))
graph_builder.add_node("write_response", partial(nodes.write_response,
    llm=self.llm))
graph_builder.add_node("small_talk", partial(nodes.small_talk, llm=self.
    llm))
graph_builder.add_node("document_info", partial(nodes.document_info, llm=
    self.llm))

graph_builder.add_edge(START, "create_standalone_query")
graph_builder.add_edge("create_standalone_query", "detect_intent")
graph_builder.add_conditional_edges(
    "detect_intent",
    lambda state: state.intent,
    {
        "coverage": "query_coverage_and_exclusion",
        "exclusion": "query_coverage_and_exclusion",
        "document": "document_info",
        "other": "query_other",
        "small_talk": "small_talk",
    }
)
graph_builder.add_edge("query_coverage_and_exclusion", "write_response")
graph_builder.add_edge("query_other", "write_response")
graph_builder.add_edge("write_response", END)
graph_builder.add_edge("small_talk", END)
graph_builder.add_edge("document_info", END)

compiled_graph = graph_builder.compile(checkpointer=memory)
```

As can be seen, the process of building the graph does not differ much from the previous project: the nodes are defined and associated with their respective functions, the deterministic edges are added, and then the conditional edges are specified to handle the possible logical branches. Finally, the graph is compiled and made executable, with the management of temporary memory entrusted to the MemorySaver object provided by LangGraph.

8.7 Frontend with Gradio

The frontend of the CGA Copilot chatbot was developed using Gradio, as was done for the POD Assistant project. In this case too, the application takes the form of a traditional chat and includes the logic for managing the conversational state, interacting with the LangGraph graph, and streaming the responses generated by the model.

The management of the message history follows the same approach already described for the POD Assistant: the history maintained by Gradio is converted into a list of semantic messages compatible with LangChain, where user messages become HumanMessage and chatbot messages become AIMessage. The new input is added to this list and then passed to the graph as the context of the conversation. In this project as well, long-term memory management on the backend was not implemented, maintaining instead the simpler solution compatible with other company applications, where the history is handled directly by the frontend.

An important difference compared to the POD Assistant concerns the construction of the conversation state. In this case, a GraphState object is created that contains not only the sequence of messages, but also the insurance product in reference. These pieces of information allow the CGA Copilot graph to modulate responses based on the product and the relevant document resources.

Interaction with the graph occurs via the asynchronous method `astream`, as done in the POD Assistant. In this case too, execution does not produce an immediate output, but a sequence of chunks. In the CGA Copilot, however, only some nodes of the graph (small_talk, write_response, document_info) are allowed to generate text to be returned to the user. When a valid AIMessageChunk is received, the content is progressively concatenated and sent to the Gradio interface, with a small artificial delay that makes the writing appear more natural. In the absence of relevant output, a fallback message is displayed, while any errors are intercepted and reported with a dedicated message.

The streaming mechanism is identical to that implemented in the POD Assistant: the user does not have to wait for the generation to finish but can follow the progressive formation of the response, achieving a perception of greater responsiveness.

8.8 Experiments and results

Before evaluating the proposed solution, preliminary experiments were conducted using a traditional RAG system. The results, however, were very disappointing, as the system simply failed to provide meaningful outputs. For documents as complex as insurance contracts, characterized by highly technical language and subtle semantic nuances, a simple query on a vector database is insufficient to

retrieve the correct chunks needed for the LLM to respond accurately. Directly using the user's question to search the database rarely retrieves the most relevant information. This is mainly due to mismatches between the user's language and the document's wording, as well as the frequent presence of real-world references or examples not explicitly described in the text, which must remain abstract enough to cover multiple scenarios.

These limitations clearly highlight the need for an enhanced approach. The solution presented in this work addresses the problem through query rewriting and the generation of multiple queries, which proved essential to ensure the retrieval of the correct chunks that is an operation that represents a fundamental step in determining the overall success of the chatbot.

To evaluate the performance of this improved system, a batch evaluation approach was adopted. This method involved the automatic processing of a predefined set of questions and the recording of the responses generated for subsequent qualitative and quantitative analysis.

The evaluation dataset was not created arbitrarily. Instead, it was prepared directly by the business team, composed of professionals with specific expertise in the insurance domain. This ensured that the questions reflected realistic scenarios and that the evaluation provided not only technical feedback, but also an assessment of the substantive correctness of the answers in relation to the contractual documents and operational practices. The file provided was in Excel format and contained 51 questions, each accompanied by the references with the relevant page numbers.

Once the dataset was defined, the evaluation script was developed to read the Excel file and process each question, sending it to the chatbot through the graph implemented with LangGraph. The responses were then saved in the corresponding column of the Excel file, next to the original question. This setup enabled a direct comparison between the system's answers and the expected ones, while also allowing the annotation of qualitative feedback in the dedicated column.

8.8.1 Test with different models

After verifying the effectiveness of the enhanced retrieval strategy, additional experiments were conducted to assess how different LLMs would perform when integrated into the system. In particular, the goal was to observe whether the choice of model could influence the overall accuracy and reliability of the chatbot's answers when working with the same dataset and retrieval pipeline.

The first tests were carried out using OpenAI's GPT-4o and GPT-4.1 models. As shown in Table 8.1, in both cases the chatbot produced 2 wrong answers and 2 partially correct answers.

The partially correct answers refer to situations where the chatbot provided information that was technically accurate but retrieved from sections of the document

different from those indicated as ground truth by the business team. While the content was correct, relying on alternative passages increases the risk of misleading deductions, potentially resulting in inaccurate outputs.

The wrong answers, on the other hand, stemmed from two distinct issues. In one case, the error was due to an incorrect chunk being retrieved during the retrieval phase, preventing the system from answering properly. In the other case, the correct chunk was retrieved, but the LLM failed to formulate the answer accurately.

The question in this case is:

> *Se le strutture portanti verticali del fabbricato, i solai, le pareti esterne sono in materiali incombustibili, in quale classe ricade il fabbricato?*

And a part of the correctly retrieved chunk is the following:

> *I locali nei quali viene svolta l'attività dichiarata sul modulo di polizza hanno caratteristiche costruttive ascrivibili ad una delle seguenti classi:*
>
> - *Classe 1 strutture portanti verticali e del tetto e solai in cemento armato o laterizi; pareti esterne e copertura in materiali incombustibili;*
> - *Classe 2 strutture portanti verticali e del tetto, solai, pareti esterne e copertura in materiali incombustibili;*
> - *Classe 3 strutture portanti verticali, solai, pareti esterne e copertura in materiali incombustibili; strutture portanti del tetto in materiali combustibili;*
> - *Classe 4 differenti dalle precedenti.*

The chatbot's answer was:

> Se le strutture portanti verticali del fabbricato, i solai e le pareti esterne sono tutti in materiali incombustibili, il fabbricato rientra nella Classe 2. In questa classe, anche la copertura deve essere in materiali incombustibili.
>
> Fonti consultate
>
> Sezione Incendio — 6.4 Classificazione dei fabbricati (pag. 48)

The correct answer in this case would be Classe 3 if we assume that the roof's load-bearing structures are combustible materials, since this was not specified by the user. An even better answer would manage this uncertainty explicitly: differentiating the response based on the type of material of the roof's load-bearing

structures, and confirming the class only if the user provides more details, thus avoiding incorrect deductions.

To address this issue, a filter was introduced to remove irrelevant chunks based on the reranker score. The filter works as follows: if the reranker score (on a 0–4 scale) between two retrieved results differs by more than 0.5, the list of chunks is truncated at that point, and only the top chunks are returned. For this question, in fact, only 2 out of 5 chunks were actually relevant, and it was expected that reducing the context given to the LLM could improve its reasoning on these types of questions.

Despite the filter working correctly and providing only the most relevant chunks to the LLM (2 in this case), no improvement was observed in the response formulation. It was therefore concluded that the issue was not due to excessive or confusing context caused by irrelevant chunks, but more likely due to limitations of the model itself.

For this reason, newer OpenAI models such as GPT5-mini and GPT5 with reasoning low were tested.

As a first experiment, GPT5 with reasoning low was used only for the answer generation step, since this is the most critical part where the LLM must reason carefully to assemble all the pieces and avoid incorrect responses. The results were satisfactory: the model answered correctly even to the critical question presented earlier. However, response times were significantly longer, with nearly one minute required to generate a single response, clearly impractical for a real user interaction.

For this reason, GPT5 with reasoning low was quickly abandoned, and GPT5-mini was tested instead. In this case too, the LLM managed the critical question correctly, answering:

> Se le strutture portanti verticali, i solai e le pareti esterne sono in materiali incombustibili, il fabbricato rientra nella classe 2 a condizione che anche la copertura (tetto) sia in materiali incombustibili. Se invece la copertura o le strutture portanti del tetto sono in materiale combustibile, il fabbricato ricade nella classe 3. (La classe 1 richiede invece strutture portanti in cemento armato o laterizi, quindi non corrisponde alla sola caratteristica di materiali incombustibili.)
>
> Fonti consultate
>
> Sezione INCENDIO — 6 CHE COSA POSSO ASSICURARE - 6.4 Classificazione dei fabbricati (pag. 48)

Thus, even without the reasoning capability, the model was still able to provide the correct answer. The only drawback was response times: while not as long as GPT5 with reasoning low, they were still noticeably slower than GPT4o and

GPT4.1.

Table 8.1 shows the results obtained with GPT4o, GPT4.1, and GPT5-mini, along with their respective response times.

Model	Partially Correct	Incorrect	Avg. Response Time (s)
GPT-4o	2	2	15
GPT-4.1	2	2	18
GPT-5 mini	2	0	24

Table 8.1: Comparison of model performance and response times on a test set of 51 questions.

After carrying out this analysis, it was decided to upgrade the model and use GPT-5-mini only for the formulation of the final answer, while GPT-4.1 was employed for all the other phases, as it proved to be slightly better than GPT-4o in reformulating queries for the vector database. The choice of GPT-5-mini for the final answer was motivated by its performance: it did not produce any wrong answers and, out of 51 questions, only 2 responses (3.92%) were classified as partially correct, since the information was accurate but cited from different sections of the text.

This improvement in accuracy, however, came with a trade-off in response time. The answers now take slightly longer to generate, which is not ideal for a virtual assistant. Nevertheless, considering the complexity of the data and the context in which the system is used, these times remain reasonably acceptable. In this scenario, it is preferable to wait a few extra seconds for a correct answer rather than receive a faster but potentially incorrect one. After all, an inaccurate response could lead an insurance agent to provide misleading information to a client, with potentially significant consequences.

Note: in the count of correct answers, those whose response can be found in several sections were also included. This is because the question is generic, and therefore the system, not knowing from which section to take the answer, gives some examples taken from different sections but then asks the user to specify the reference section if they want a more precise answer.

Although the dataset is not very large and does not fully reflect the typical questions of real chatbot use (being prepared "offline"), the results show that the system is capable of providing good quality answers. The experiment thus confirms the effectiveness of the LangGraph-based approach, which achieved excellent performance even on complex documents such as CGA.

It is also worth noting that similar tests conducted previously on a traditional RAG system produced unsatisfactory results, making the conventional approach

of limited usefulness for such complex and technical insurance documentation. The Proof of Concept developed here demonstrates that integrating a workflow orchestration graph with traditional retrieval can significantly improve the quality of answers, making the system suitable for real use cases.

8.8.2 Evaluation questions

The following are examples of questions from the test set provided by the business team, along with the corresponding answers generated by GPT5-mini and the correct references from which the answers should be derived in the document.

ID 6

Question: L'indicizzazione se presente in polizza si applica sempre?

Chatbot answer:

No. L'indicizzazione si applica solo se sulla polizza è indicato "SI" e non è automatica in ogni caso. Principali regole previste:

- È attiva solo se espressamente riportata sul modulo di polizza.
- Adegua mensilmente i valori e i limiti delle prestazioni (escluse le franchigie); il premio viene invece aggiornato solo alla scadenza annuale. In caso di sinistro si fa riferimento agli importi rivalutati calcolando il rapporto tra l'indice ISTAT del terzo mese precedente il sinistro e quello del terzo mese precedente la stipula.
- Le parti possono rinunciare all'indicizzazione inviando comunicazione almeno 60 giorni prima della scadenza annuale, a condizione che siano state pagate almeno tre annualità; in tal caso la polizza prosegue con premio, valori e limiti raggiunti alla scadenza dell'annualità in corso.
- Se l'incremento annuo dell'indice risulta inferiore al 2%, viene comunque applicato un aumento minimo del 2% a premio, valori e limiti (escluse le franchigie).

Fonti consultate:

- *Norme comuni a tutte le garanzie* — 4 Informazioni sul premio - 4.4 Clausola di indicizzazione (pagg. 33-34)

Correct page: 33

In this case, as can be seen, the chatbot was able to answer correctly, also citing the exact page provided by the business team.

ID 30

Question: Le merci poste ad un'altezza inferiore ai 12 cm sono risarcibili a termini di polizza per la garanzia inondazione ed alluvione?

Chatbot answer:
La polizza esclude esplicitamente i danni alle merci la cui base è posta ad un'altezza inferiore a 10 cm dal pavimento: tali beni non sono indennizzabili, salvo le merci che per caratteristiche non possono essere messe su pallet o supporti di almeno 10 cm (in quel caso l'esclusione non si applica). Inoltre sono esclusi i beni posti in locali interrati/seminterrati e i beni all'aperto.
Per la soglia di 12 cm non c'è una previsione specifica: Non risulta esclusa esplicitamente nel documento.
Fonti consultate

- *Sezione Catastrofali* — 18.2 Rischi esclusi per la garanzia inondazioni, alluvioni (pag. 81-82)
- *Sezione Incendio* — 6 Che cosa posso assicurare (esclusioni per allagamento: merci con base <10 cm, locali interrati/seminterrati, beni all'aperto) (pag. 38-48)

Correct page: 81

ID 51

Question: Dove è l'esclusione, nella sezione RC, per i danni da evento atmosferico che dovessero colpire i veicoli dei clienti di un'autofficina/carrozzeria/gommista?

Chatbot answer:
Nel testo della Sezione Responsabilità Civile non è indicata un'esclusione specifica per i danni da evento atmosferico ai veicoli dei clienti. Anzi la garanzia per i veicoli di terzi (in sosta nell'ubicazione, in consegna/custodia o durante carico/scarico) è prevista con franchigia di €250 e limite di €150.000 (31.3.5). È presente invece, all'art. 31.1 (punto 12), l'esclusione dei rischi di circolazione per i veicoli soggetti all'obbligo di assicurazione di legge, che non riguarda i danni subiti mentre i veicoli sono custoditi o in sosta. Quindi non risulta esclusa esplicitamente nel documento.
Fonti consultate:

- *Sezione Responsabilità Civile* — 31.3.5 Veicoli di terzi (pag. 120-121)
- *Sezione Responsabilità Civile* — 31.1 Rischio assicurato - Ambito di Applicazione (pag. 118-119)

Correct pages: 119; 122

In this case, not finding the exclusion mentioned by the user, the chatbot

correctly states that it cannot find it, without making wrong assumptions. This is a type of question that, if not handled properly, could lead to incorrect answers, as the LLM might otherwise hallucinate or invent information.

ID 17

Question: In caso di sinistro entro quando devo avvisare la mia agenzia dell'evento occorso?

Chatbot answer:

Il termine dipende dalla garanzia attivata nella polizza aziendale. Esempi tratti dal documento:

- Tutela Legale: l'assicurato deve segnalare il sinistro immediatamente non appena ne viene a conoscenza; la copertura copre anche sinistri avvenuti durante la vigenza e denunciati entro 24 mesi dalla cessazione della polizza.
- Tutela Ambientale: l'avviso scritto all'Agenzia o a Reale Mutua deve essere fatto entro 3 giorni da quando se ne è avuta conoscenza.
- Responsabilità Civile: la denuncia scritta all'Agenzia o a Reale Mutua va effettuata entro 10 giorni da quando se è venuti a conoscenza dell'evento.
- Elettronica: l'avviso all'Agenzia o a Reale Mutua deve essere inviato entro 10 giorni dalla conoscenza del sinistro.

Se desideri il termine preciso relativo a una sezione specifica della tua polizza, indicami quale.
Fonti consultate:

- *Sezione Tutela Legale* — 49.2 Termini di denuncia del sinistro e fornitura dei mezzi di prova (pag. 159)
- *Sezione Tutela Ambientale* — 44.1 Obblighi in caso di sinistro (pag. 146)
- *Sezione Responsabilità Civile* — 35.1 Obblighi in caso di sinistro (pag. 131)
- *Sezione Elettronica* — 15.2 Obblighi in caso di sinistro (pag. 73-74)

Correct pages: 58

This question is a typical example of a generic query, where the answer depends on the section of the document the user wants to refer to. Since no specific coverage was mentioned, the chatbot provided an answer citing different ones and asked the user to clarify their interest to give the correct details. In this case, the correct page number referred to the fire section, while the chatbot retrieved from other sections, but the answer was still considered correct.

ID 50

Question: Quante sono le sezioni di polizza?

Chatbot answer:
Le sezioni di polizza sono dieci. Ecco l'elenco:

1. Sezione Incendio (pagg. 36–64)
2. Sezione Elettronica (pagg. 65–76)
3. Sezione Catastrofali (pagg. 77–89)
4. Sezione Danni Indiretti (pagg. 90–103)
5. Sezione Furto (pagg. 104–117)
6. Sezione Responsabilità Civile (pagg. 118–131)
7. Sezione RC Prodotti (pagg. 132–141)
8. Sezione Tutela Ambientale (pagg. 142–147)
9. Sezione Tutela Legale (pagg. 148–161)
10. Sezione Assistenza (pagg. 161–170)

Regarding question 50, instead, the user asks to list all the sections of the policy. This type of question is handled by a specific branch of the system that manages questions about the document's structure. This branch does not perform any query to the vector database and thus does not rely on the RAG mechanism. Instead, it can answer correctly because it has direct access to the document's table of contents, which includes the start and end pages of all chapters and subchapters.

Chapter 9

Conclusions

This thesis has investigated the potential of Generative AI agents, RAG systems, and the MCP protocol applied to knowledge management and operational efficiency in the insurance sector. The work carried out at the Data Science Center of Excellence of Reale Mutua Assicurazioni led to the creation of two PoC: the POD Assistant and the CGA Copilot.

The POD Assistant showed how multi-agent architectures can support internal teams by automating repetitive and low-value tasks, such as writing mandatory project documentation, consulting Scrum and company guidelines, and interacting with Azure DevOps through MCP. Even if limited to a PoC, the project demonstrated the potential benefits of an agent-based approach and, in particular, of the MCP protocol. The latter is emerging as a new standard to allow LLMs to interact directly with external services and platforms. Its adoption opens significant opportunities, giving immediate access to a wide ecosystem of tools through natural language. However, since it is a recent technology, there are still open issues related to security and the handling of sensitive data: it is necessary to guarantee the authenticity of MCP servers and the reliability of the protection measures implemented.

The CGA Copilot addressed another important challenge: the ability of an LLM to answer questions based on long documentation, technical in nature and characterized by specialized language. The first tests with a traditional RAG approach showed limits in terms of accuracy and relevance of the answers. To overcome these issues, a hybrid pipeline was developed, combining vector search and semantic reranking, orchestrated through the LangGraph framework. This solution significantly improved the reliability and completeness of the answers, showing the possibility of applying RAG systems in complex scenarios with strict regulatory constraints such as the insurance sector.

Overall, the work has highlighted that:

- AI agents can increase internal efficiency by modularizing tasks and integrating with existing company tools;
- MCP reduces integration times and costs, supporting interoperability between AI systems and external platforms;
- hybrid retrieval pipelines, enhanced by LangGraph, represent a promising approach to make LLMs more reliable in complex and regulated domains.

However, both solutions remain at the Proof of Concept stage. They show technical feasibility and potential value, but further developments are needed to reach a production level, especially in terms of scalability, robustness under heavy workloads, and secure handling of sensitive company data.

9.1 Future works

Looking to the future, the first goal will be to transform the prototypes into solutions ready for use in real contexts. The POD Assistant can be enriched with new features and fully integrated into company workflows, while the CGA Copilot will initially be deployed in controlled environments, allowing its usability to be evaluated directly with real insurance agents.

For these solutions to grow sustainably, the issue of scalability will need to be addressed. Multi-agent architectures will have to be optimized to support higher data volumes and a larger number of users. This will be accompanied by monitoring and logging systems, essential for tracking performance, identifying errors, and guiding improvement actions.

At the same time, it will be possible to explore advanced functionalities that make the solutions more flexible and reliable. The introduction of human-in-the-loop mechanisms will allow automation to be balanced with human supervision, especially in more delicate decisions, while the use of more sophisticated multi-agent collaborations will enable coordinated management of complex workflows and structured decision-making processes.

In this direction, it is interesting to consider local intervention strategies (e.g., an agent alerts the supervisor only in case of high uncertainty) or hierarchical mechanisms (automatic agent first, escalation to a human operator if needed). Adaptive control can also be explored: the agent decides whether to delegate or intervene based on confidence indicators, error costs, or domain constraints. Multi-agent collaborations can be made dynamic: on-demand activation, modular agents that combine for new tasks, or orchestrator agents that coordinate sub-tasks.

In this perspective, another promising direction concerns the evaluation of chatbot performance through the *LLM-as-a-judge* paradigm, where large language models are employed to assess the quality of generated responses. Frameworks such as *Ragas* [30] already provide structured methodologies for measuring reliability, factual consistency, and overall usefulness of answers, offering a complementary evaluation to traditional human-based or metric-based approaches.

The *LLM-as-a-judge* approach is particularly relevant for RAG systems: the idea is to have a *critical* LLM evaluate the outputs of another model according to predefined rubrics (e.g., accuracy, adherence to documentation, usefulness, completeness). Ragas is a library specifically designed for this purpose, providing metrics such as faithfulness, answer_correctness and helpfulness, while supporting an iterative alignment workflow between automatic judgments and human evaluations.

Nevertheless, the limitations should be critically considered: trustworthiness of LLM scores, transparency of criteria, explainability of judgment decisions, and the risk of systematic biases. A robust approach likely combines LLM-as-a-judge with human validation, cross-checking scores, and complementary metrics.

Finally, future developments may also explore the release of these multi-agent systems using standards such as the *Agent-to-Agent Protocol* (A2A) [31]. This would enable agents to directly communicate and interoperate across different platforms and organizational boundaries, opening up opportunities for building larger ecosystems of cooperative AI systems.

The A2A protocol is a new open standard designed to allow autonomous agents to communicate, negotiate, and collaborate independently of their underlying implementation. A2A defines key concepts such as the *Agent Card* (a JSON description of capabilities, endpoints, and credentials), capability discovery, task negotiation, sub-task assignment, long-term task management, and asynchronous push-based communication.

A2A is designed with interoperability, security, and modularity in mind. It integrates with existing standards, facilitating enterprise adoption. It can also work with Model Context Protocol to connect agents to external tools: agents use MCP for tool access and A2A for inter-agent communication.

For enterprise systems, adopting A2A would make agents modular and interoperable: e.g., a legal analysis agent could cooperate with pricing, customer service, or compliance agents, exchanging data and tasks securely.

In conclusion, this thesis has shown that agent-based AI and RAG systems are practical tools that can transform the way companies access and manage knowledge. Furthermore, this work contributes to the company's ongoing efforts to leverage Artificial Intelligence, exploring its application in complex domains such as the insurance sector. The projects developed, even as Proofs of Concept, enabled the

testing of solutions that improve knowledge management and assess the adoption of more advanced conversational architectures, laying the foundation for future developments toward increasingly reliable, scalable, and effective AI systems.

Looking ahead, combining robust evaluation (e.g., LLM-as-a-judge), agent interoperability protocols (such as A2A), and scalable production pipelines represents the most promising path to transform these prototypes into resilient, cooperative, and adaptive enterprise systems.

Chapter 10

Appendix

10.1 POD Assistant frontend

In Figure 10.1 the initial view of the application is shown when running it with the Gradio interface. Some example questions are provided to suggest possible actions that the user can perform to start interacting with the assistant. On the top left side, there is a *New chat* button, which opens a new page with an empty chat history, similar to the behavior of the most popular LLM-based web interfaces. This feature is particularly useful when the user wants to switch topics and prevent previous interactions from being considered as context for future ones.

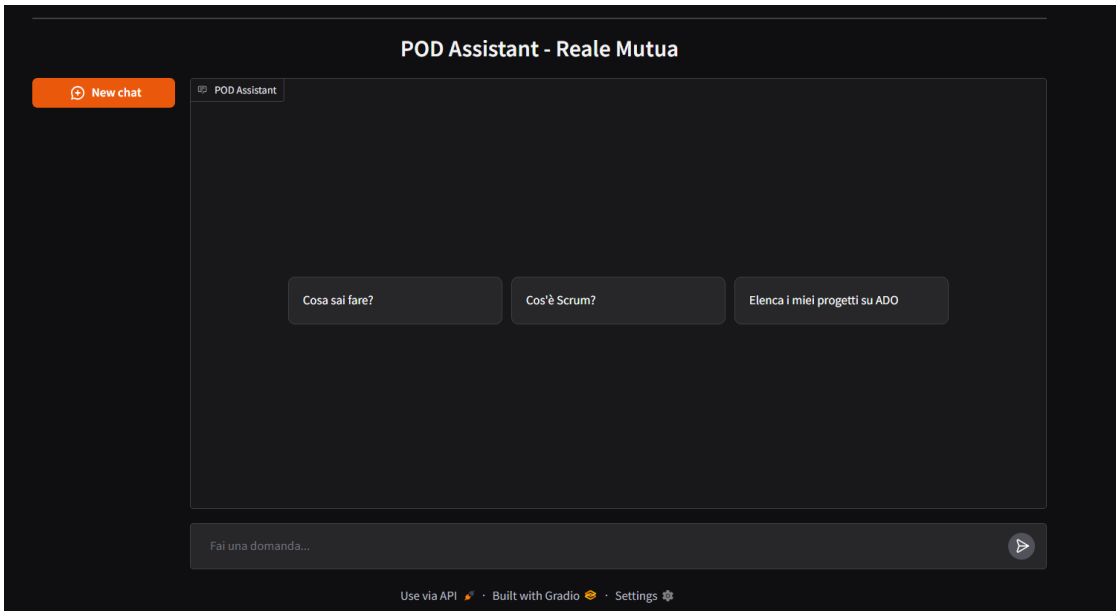

Figure 10.1: The initial view of the application with the Gradio interface.

In practice, this functionality enables resetting the message history and managing the conversational context directly through the Gradio interface, eliminating the need to implement a dedicated memory manager for handling multiple chats. Gradio automatically sends to the backend only the list of messages belonging to the current chat, meaning that the context is inherently limited to the ongoing conversation.

In Figure 10.2, an example of an interaction is shown in which the use of agent tools is required. The interface transparently displays which tools are executed in order to answer the user's questions. While this behavior can be disabled, it can be helpful in cases where processing takes longer, as it reassures the user that the system is functioning and generating a response.

Moreover, as shown on the left side, the user can navigate across different chats covering various topics. It is possible to switch from one chat to another, with the corresponding message history being passed to the backend and used as context for the assistant.

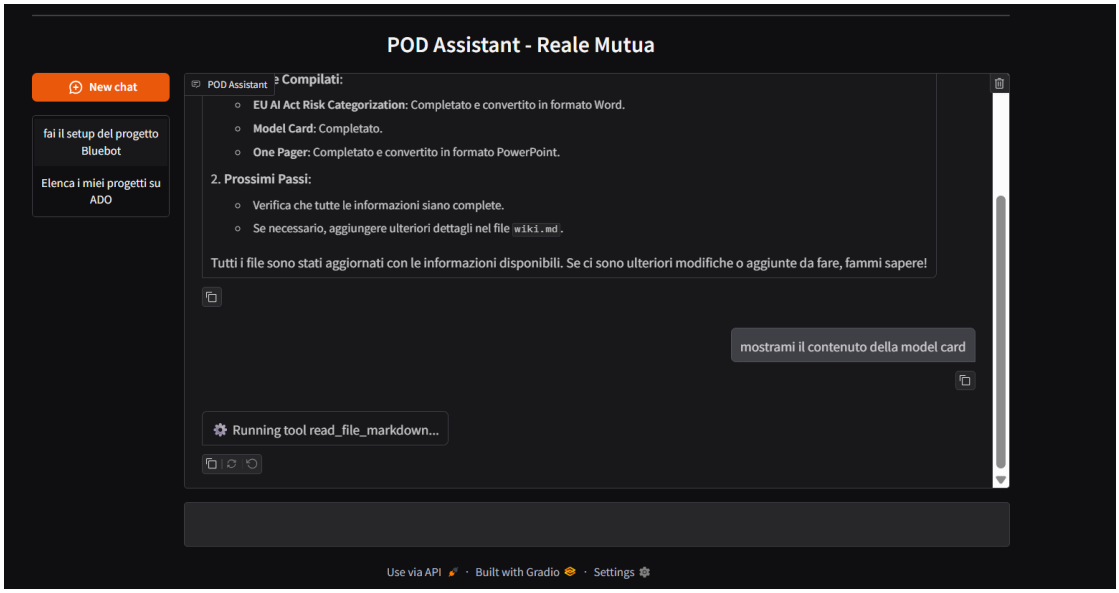

Figure 10.2: Example of the Gradio interface with multiple parallel chats. On the left, the list of conversations allows the user to switch between different topics, while on the right the selected chat is displayed with the full history of the interaction with the assistant.

10.2 CGA Copilot frontend

The interface of the CGA Copilot is essentially the same as the one described for the POD Assistant, since it is based on the same Gradio frontend. As in the previous case, some example questions are provided at the beginning, which can guide the user in formulating queries and understanding the type of information that the assistant can retrieve.

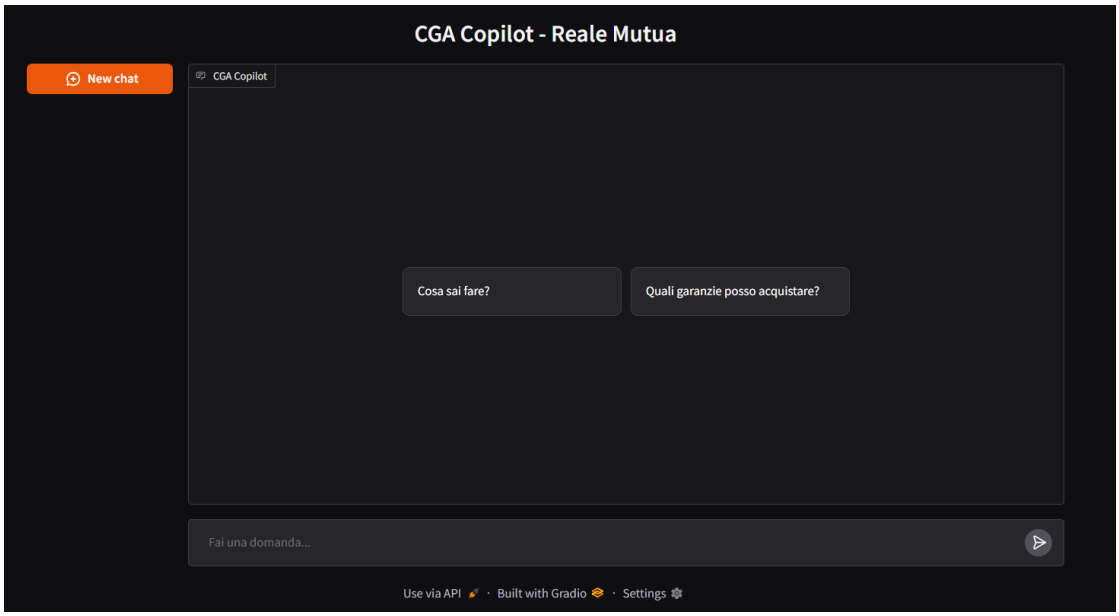

Figure 10.3: The initial view of the CGA Copilot application with the Gradio interface, showing example questions to guide the user.

In practice, the workflow and the main features remain unchanged: the user can start a new chat through the dedicated button, switch across different conversations on the left panel, and view the interaction history in the main window. The system automatically manages the conversational context by considering only the messages of the currently selected chat.

Figures 10.3 and 10.4 illustrate the initial view and an example interaction with the CGA Copilot.

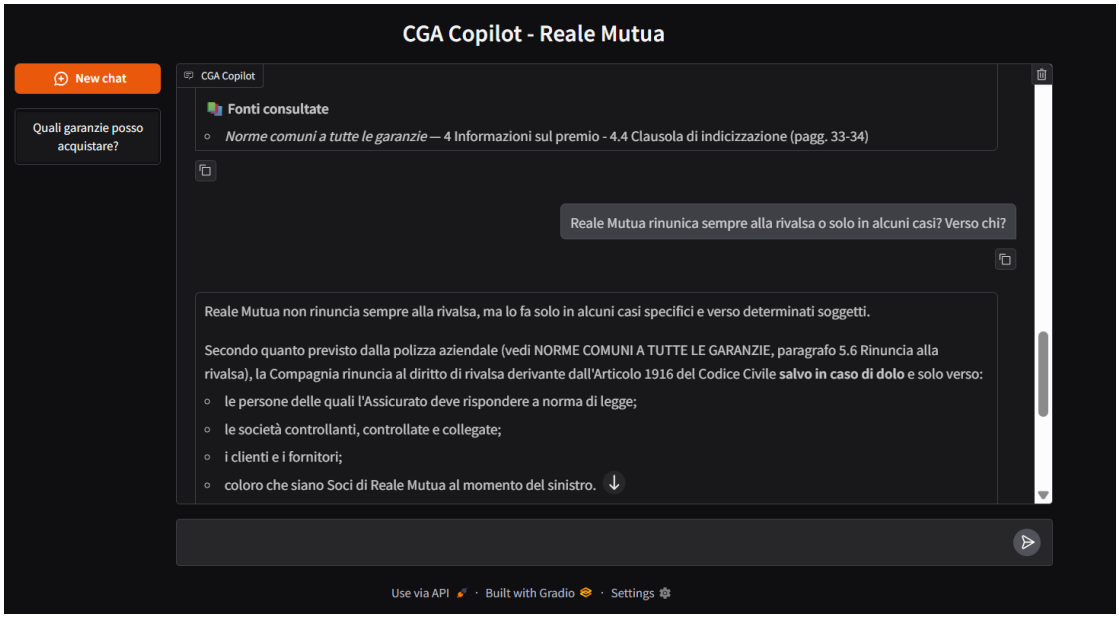

Figure 10.4: Example of interaction with the CGA Copilot.

Bibliography

[1] Alex Singla, Alexander Sukharevsky, Lareina Yee, Michael Chui, and Bryce Hall. *The State of AI in early 2024: Gen AI adoption spikes and starts to generate value*. Accessed: 2025-09-05. May 2024. URL: https://www.mckinsey.com/capabilities/quantumblack/our-insights/the-state-of-ai-2024 (cit. on p. 1).

[2] Alexander Sukharevsky, Dave Kerr, Klemens Hjartar, Lari Hämäläinen, Stéphane Bout, Vito Di Leo, and Guillaume Dagorret. *Seizing the agentic AI advantage*. Accessed: 2025-09-05. June 2025. URL: https://www.mckinsey.com/capabilities/quantumblack/our-insights/seizing-the-agentic-ai-advantage (cit. on p. 1).

[3] Tom Coshow. *Intelligent Agents in AI Really Can Work Alone. Here's How.* Accessed: 2025-09-05. Oct. 2024. URL: https://www.gartner.com/en/articles/intelligent-agent-in-ai (cit. on p. 1).

[4] Reale Group. *Reale Group*. Accessed: 2025-09-05. 2025. URL: https://www.realegroup.eu/IT/ (cit. on p. 4).

[5] Google Cloud. *What are AI agents? Definition, examples, and types*. Accessed: 2025-09-05. 2025. URL: https://cloud.google.com/discover/what-are-ai-agents?hl=en (cit. on p. 5).

[6] Anna Gutowska. *What Are AI Agents?* Accessed: 2025-09-05. 2024. URL: https://www.ibm.com/think/topics/ai-agents (cit. on p. 5).

[7] Google Research. *ReAct: Synergizing Reasoning and Acting in Language Models*. https://research.google/blog/react-synergizing-reasoning-and-acting-in-language-models/. 2022 (cit. on pp. 5, 7).

[8] Anna Gutowska. *What is a Multi-Agent System?* Accessed: 2025-09-05. 2025. URL: https://www.ibm.com/think/topics/multiagent-system (cit. on p. 5).

[9] Google Cloud. *What is Human-in-the-Loop (HITL) in AI & ML?* Accessed: 2025-09-05. 2025. URL: https://cloud.google.com/discover/human-in-the-loop?hl=en (cit. on p. 5).

[10] Kate Whiting. *What are the risks and benefits of 'AI agents'?* Accessed: 2025-09-05. Dec. 2024. URL: https://www.weforum.org/stories/2024/12/ai-agents-risks-artificial-intelligence/ (cit. on p. 5).

[11] Rina Diane Caballar and Cole Stryker. *AI agent frameworks: Choosing the right foundation for your business*. Accessed: 2025-09-05. 2025. URL: https://www.ibm.com/think/insights/top-ai-agent-frameworks (cit. on p. 5).

[12] Amy Brennen. *What is an agentic framework?* Accessed: 2025-09-05. Feb. 2025. URL: https://www.moveworks.com/us/en/resources/blog/what-is-agentic-framework (cit. on p. 5).

[13] CrewAI. *CrewAI – The Leading Multi-Agent Platform*. Accessed: 2025-09-05. 2025. URL: https://www.crewai.com/ (cit. on p. 5).

[14] LangChain. *LangChain Documentation*. Accessed: 2025-09-05. 2025. URL: https://python.langchain.com/docs/introduction/ (cit. on p. 5).

[15] LangChain. *Why LangGraph?* Accessed: 2025-09-05. 2025. URL: https://langchain-ai.github.io/langgraph/concepts/why-langgraph/ (cit. on pp. 5, 19).

[16] Fast-Agent Team. *Fast-Agent – MCP Native Agents and Workflows*. Accessed: 2025-09-05. 2025. URL: https://fast-agent.ai/ (cit. on p. 5).

[17] LlamaIndex. *LlamaIndex – Build Knowledge Assistants over your Enterprise Data*. Accessed: 2025-09-05. 2025. URL: https://www.llamaindex.ai/ (cit. on p. 5).

[18] LlamaIndex. *LlamaIndex Documentation*. Accessed: 2025-09-05. 2025. URL: https://docs.llamaindex.ai/en/stable/ (cit. on p. 5).

[19] World Economic Forum. *The Future of Jobs Report 2025*. Accessed: 2025-09-05. Jan. 2025. URL: https://www.weforum.org/publications/the-future-of-jobs-report-2025/ (cit. on p. 12).

[20] The LangChain Team. *Multi-agent architectures*. https://langchain-ai.github.io/langgraph/concepts/multi_agent/#multi-agent-architectures. Accessed: 2025-10-03. 2025 (cit. on p. 14).

[21] Wikipedia contributors. *Model Context Protocol*. Accessed: 2025-09-05. 2025. URL: https://en.wikipedia.org/wiki/Model_Context_Protocol (cit. on p. 24).

[22] Model Context Protocol. *Model Context Protocol Documentation*. Accessed: 2025-09-05. 2025. URL: https://modelcontextprotocol.io/docs/getting-started/intro (cit. on pp. 24, 25).

[23] Norah Sakal. *MCP vs API: Model Context Protocol Explained*. Accessed: 2025-09-05. Mar. 2025. URL: https://norahsakal.com/blog/mcp-vs-api-model-context-protocol-explained/ (cit. on p. 24).

[24] Shivam Bhardwaj. *MCP Servers: Model Context Protocol Servers Explained*. https://dev.to/shivamp0987/mcp-servers-model-context-protocol-servers-explained-2d5p. Accessed: 2025-09-14. 2025. URL: https://dev.to/shivamp0987/mcp-servers-model-context-protocol-servers-explained-2d5p (cit. on p. 25).

[25] Elastic. *What is Retrieval Augmented Generation (RAG)? A Comprehensive Guide*. https://www.elastic.co/what-is/retrieval-augmented-generation/. Accessed: 2025-09-15. 2025 (cit. on pp. 27, 29, 30).

[26] Adam Fourney and (e altri) Bansalg. *MarkItDown: Python tool for converting files to Markdown*. https://github.com/microsoft/markitdown. Versione rilasciata: 0.1.3, licenza MIT. 2025 (cit. on p. 39).

[27] Jessica Tegner. *PyPandoc: Thin wrapper for Pandoc*. https://github.com/JessicaTegner/pypandoc. Version v1.13. Accessed: 2025-09-10. 2025. URL: https://github.com/JessicaTegner/pypandoc (cit. on p. 42).

[28] Microsoft. *Azure DevOps MCP Server*. https://github.com/microsoft/azure-devops-mcp. Accessed: 2025-09-10. 2025. URL: https://github.com/microsoft/azure-devops-mcp (cit. on p. 43).

[29] Abubakar Abid, Ali Abdalla, Ali Abid, Dawood Khan, Abdulrahman Alfozan, and James Zou. *Gradio: An open-source Python library for building machine learning model demos*. https://www.gradio.app/. Accessed: 2025-09-12; Based on *Gradio: Hassle-Free Sharing and Testing of ML Models in the Wild*, arXiv preprint. 2019 (cit. on p. 55).

[30] Ragas Contributors. *Ragas Documentation*. https://docs.ragas.io/en/stable/concepts/. Accessed: 2025-10-03. 2024 (cit. on p. 86).

[31] A2A Protocol Contributors. *Agent-to-Agent Protocol (A2A)*. https://a2a-protocol.org/dev/. Accessed: 2025-10-03. 2024 (cit. on p. 86).