MASTER DEGREE IN COMPUTER SCIENCE

THESIS IN SEMANTICS IN INTELLIGENT SYSTEMS

E-MEALIO: A CONVERSATIONAL AGENT BASED ON LLM FOR PROMOTING SUSTAINABLE EATING HABITS

Supervisor:

Prof. Cataldo Musto

Co-Supervisor:

Arianna Boldi, PhD

Candidate:

Antonio Raffaele Iacovazzi

ID: 756263

Academic Year 2023/2024

Abstract

Climate change is one of the most complex challenges humanity has ever faced. This phenomenon, driven by multiple factors, is closely linked to our dietary habits, with the overconsumption of processed foods and meat playing a significant role in the emission of greenhouse gases.

For this reason, a fundamental shift in eating habits is essential. However, changing something so deeply tied to our identity and daily lives is difficult without the right information and motivation.

This thesis presents an approach to encouraging more environmentally sustainable eating habits through an LLM-based agent called **E-Mealio**. Its objective is to help users better understand the impact of food on climate change while suggesting new recipes to gradually build a more sustainable lifestyle, meal after meal.

The agent is backed by **HeaSEv2**, a comprehensive dataset of recipes and ingredients, each with sustainability indexes computed from reliable sources. This ensures accurate data, minimizing the risk of generating misleading information that could confuse users.

Additionally, the conversation flow is structured through well-defined steps that trigger specific functionalities, reinforcing the agent's role as a specialized tool in its domain.

This work explores how recent advancements in generative artificial intelligence, combined with semantic search and tailored rule-based algorithms, can be leveraged to develop intelligent agents driven by precise information and actionable procedures. This approach integrates structured guidance with accurate sustainability metrics, making it a reliable tool for fostering meaningful behavioral change.

Contents

Part I

Introduction and Theoretical Foundations

Chapter 1

Introduction

According to new data from the World Meteorological Organization, 2024 has officially become the hottest year ever recorded, surpassing previous temperature records [1]. It also marks the first time the planet has exceeded the 1.5 degrees Celsius global warming threshold set by the Paris Climate Agreement.

This alarming situation is further exacerbated by recent political shifts, such as the United States' withdrawal from the Paris Agreement [2], which aimed to limit CO_2 emissions and prevent a rise in global temperatures beyond 1.5 degrees. The impacts of climate changes is becoming increasingly visible, with extreme weather events like flooding, droughts, and wildfires disrupting ecosystems and human communities alike.

While the primary response to this crisis requires political action, including increased investments in green technologies and renewable energy sources, it is equally critical to address the role of dietary choices. The food industry is a major contributor to greenhouse gas emissions, and our collective consumption habits significantly impact the rate of climate change.

This chapter examines the impact of food systems on the climate crisis, explores psychological techniques that foster engagement in tackling complex behavioral challenges, and establishes the scope and objectives of this thesis. The goal is to highlight sustainable dietary

alternatives and propose solutions that contribute to mitigating climate change.

1.1 Climate and Food: A Complex Problem

Climate change has multiple root causes, all of which share a common trait: they are driven by human activities and industrialization. Among these, the agrifood sector plays a particularly significant role. Numerous studies have attempted to quantify its contribution to global greenhouse gas emissions, yielding varying estimates. The Food and Agriculture Organization (FAO) reported in *Livestock's Long Shadow* [3] (2006) that livestock accounts for 18% of global emissions. However, a Worldwatch Institute report, *Livestock and Climate Change* [4] (2009), presented a much higher estimate of 51%, arguing that previous calculations had omitted key factors such as carbon sequestration loss from deforestation, underreported methane emissions, and the full impact of animal respiration. These differing estimates have sparked a global debate about the food industry's role in climate change, particularly regarding meat production. More recent studies from the FAO [5] estimate that the agrifood sector is responsible for approximately one-third of global greenhouse gas emissions, emphasizing the significant impact of this industry on the delicate balance of Earth's ecosystems. In addition to greenhouse gas emissions, several other environmental problems arise from the current state of the food industry. One of the most notable issues is deforestation. To meet the ever-growing demand for livestock production, increasing amounts of forested land are being cleared to make way for crop cultivation. This results in a reduced global capacity to absorb carbon dioxide CO_2 from the atmosphere, exacerbating climate change.

Closely related to deforestation is the ongoing loss of biodiversity. As forests and natural habitats are destroyed, countless species are pushed to the brink of extinction. The destruction of these ecosystems disrupts delicate environmental balances, leading to significant

consequences for wildlife and the planet's overall health.

Another critical environmental concern is water waste. From the water required to grow animal feed crops to the water consumed by the animals themselves, the agricultural industry is responsible for a large portion of global freshwater usage. This excessive demand for water resources contributes to water scarcity, particularly in regions already facing challenges in managing their freshwater supplies.

The current state of the food industry is also unsustainable from a social perspective. Food resources tend to flow toward wealthier countries, where there is an overabundance of products, while poorer countries often struggle to access basic food supplies. This economic imbalance exacerbates global inequality and creates a paradoxical situation: on one side, we face massive food waste, with large quantities of food discarded in affluent nations, and on the other, widespread malnutrition and food insecurity persist in many parts of the world. The inequality in food distribution highlights a severe contradiction in the global food system, where excess leads to waste, while scarcity drives hunger and poor nutrition.

All of these issues are happening while the food industry continues to actively contribute to the global warming phenomenon, which is already making parts of the world increasingly uninhabitable. Rising temperatures, extreme weather events, and environmental degradation are displacing entire populations, leading to mass migration. This, in turn, exacerbates social and political imbalances, creating further challenges for global stability.

For this reason, a renewed approach to our dietary habits is essential. A more livable and equitable planet can emerge from our power as consumers. While advocating for political and economic change, we must also foster new collective ways of living, understanding that our food choices can lead to a healthier planet and a better society[1].

[1]The book *We Are the Weather* (2019) by Jonathan Safran Foer [6] is recommended for readers seeking a deeper exploration of the topic, as it provides a comprehensive bibliography and a compelling call to action regarding

1.2 Nudging and Gamification as Way to Build New Habits

Developing a new habit, especially one that influences core values and daily routines such as eating habits, is a complex process. Studies suggest that it takes an average of 66 days for a new habit to become firmly established [7], even when the change is driven by strong evidence. For this reason, it is essential to develop strategies that help individuals adopt new habits by making the cognitive transition smoother and less demanding, while also leveraging these strategies in the development of systems and applications related to lifestyle and education. In particular, two main strategies emerge in this context: *nudging* and *gamification*.

1.2.1 Nudging

Nudging is a concept originally introduced in the field of cybernetics and later extensively explored by Richard Thaler and Cass Sunstein's book *Nudge: Improving Decisions About Health, Wealth, and Happiness*. It focuses on modifying the environment to encourage spontaneous behavioral changes that lead to desirable outcomes. Thaler and Sunstein defined their concept as the following [8]:

> A nudge, as we will use the term, is any aspect of the choice architecture that alters people's behavior in a predictable way without forbidding any options or significantly changing their economic incentives. To count as a mere nudge, the intervention must be easy and cheap to avoid. Nudges are not mandates. Putting fruit at eye level counts as a nudge. Banning junk food does not.

People's behavior does not always align with their values and needs, a phenomenon known as the *value-action gap*. This discrepancy arises due to differences in the brain's two modes

the role of food choices in climate change.

of functioning, as studied by Daniel Kahneman in *Thinking, Fast and Slow* [9]. In this seminal work, Kahneman describes the two main cognitive systems that govern decision-making. *System 1* is fast, automatic, and highly influenced by environmental cues, while *System 2* is slower, more deliberate, and considers explicit goals and intentions. In the presence of overwhelmingly complex decisions or time constraints, *System 1* takes over, providing fast judgments and decisions based on inner heuristics. However, this often leads to suboptimal choices that may not fully align with an individual's core values. Nudging "guides" *System 1* by modifying the decision-making environment, subtly increasing the likelihood of choosing a better option without restricting freedom of choice. Examples of nudging techniques include the *default option*, where the most desirable choice is pre-selected, increasing the likelihood of selection as alternative options require additional effort to access. Another example is the *social-proof heuristic*, which leverages peer influence by demonstrating that others within the same social context are making similar choices, encouraging the individual to follow suit.

Framing refers to how information is presented to enhance its appeal and perceived trustworthiness. For example, stating that a product is *"90% fat-free"* sounds healthier than saying it contains *"10% fat"*, making it a more desirable choice.

Finally, *feedback and reminders* help mitigate forgetfulness by providing timely notifications that encourage individuals to take specific actions. This technique is widely used in applications designed to help users stay committed to their goals. Nudging plays a crucial role in developing habits that help individuals adopt a more sustainable lifestyle. In this context, technological tools can assist in keeping users on track with their goals. A comprehensive review of nudging techniques that can encourage individuals to adopt more sustainable behaviors can be found in [10].

1.2.2 Gamification

Gamification refers to a set of techniques designed to make activities and behaviors more engaging, particularly those that may otherwise be perceived as tedious or uninteresting. Although the concept began to gain traction around 2008, it was coined in 2002 by video game developer Nick Pelling. A 2012 analysis by Nelson [11] reveals that gamification's roots can be traced to two key sources: the concept of "socialist competition" used by the USSR to motivate workers toward achieving goals, and the American "fun at work" management style, which aims to immerse employees in enjoyable activities to reinforce corporate culture.

Gamification incorporates recurring elements from the world of games, such as points, badges, leaderboards, and contextual narratives, to track progress and enhance motivation. Gamification is widely used in apps within the healthcare and education domains, such as *Duolingo* for language learning and *Fitbit* for fitness tracking. It is also applied in the workplace to keep employees engaged and aligned with company goals, fostering productivity and a sense of achievement, even if in this context it can degenerate in a masked version of micromanaging.

Nudging and gamification are often combined to leverage their respective strengths, creating a more effective framework for behavioral and habit formation. They become essential tools in any context where adopting new habits is crucial to achieving significant goals, such as combating climate change.

1.3 Scope of the Study

This study aims to develop a conversational agent designed to provide environmentally friendly suggestions in an informative and user-friendly manner. The agent will leverage a reliable dataset focused on food sustainability, specifically considering the concepts of *carbon foot-*

print and *water footprint*. The carbon footprint refers to the kilograms of CO_2 equivalent gases emitted in the production of one kilogram of a specific food item, while the water footprint represents the liters of water required during the production process. To ensure seamless interaction, the agent must effectively process natural language inputs and communicate its suggestions in an engaging and accessible way. For this reason, natural language processing will be handled by a large language model (LLM), providing the flexibility needed for dynamic and context-aware responses. The system will be designed to remain model-agnostic, allowing for easy adaptation to different LLMs if necessary. The agent will be accessible through a widely used and familiar platform by being developed as a bot within an existing application. Given the availability of public libraries and ease of integration, this project will implement the agent as a Telegram bot.

The agent will manage two main types of recommendations:

- **Simple recipe recommendation**: A recipe will be suggested based on the user's overall preferences, as specified in their profile, along with any real-time preferences expressed during the request (e.g., inclusion of specific ingredients). Additionally, if no explicit hints about ingredients are provided, the agent will consider the user's taste, inferred from their past recipe consumption. Recommendations will cover four main meal categories: breakfast, lunch, break, and dinner, with filtering based on sustainability information available in the dataset. The suggested recipe will be interactive, allowing the user to inquire about ingredient composition and sustainability. Furthermore, the user will have the option to accept, discard, or request an alternative suggestion.

- **Recipe improvement**: This recommendation type focuses on enhancing the sustainability of a given recipe provided by the user. The system will identify a similar recipe with a better sustainability score and highlight ingredient substitutions needed to trans-

form the original recipe into a more eco-friendly version. As with the previous recommendation type, the suggestion should be interactive, allowing the user to discuss the proposed changes, ask for clarifications, and either accept, discard, or request further modifications.

The recommendations and any follow-up information requested by the user should be conveyed in a polite and playful manner while remaining assertive. This approach aims to gently nudge the user toward accepting the suggested recipes.

Furthermore, the agent will be capable of collecting essential user information, such as allergies or dietary restrictions, and updating the user profile as needed. The user will also be able to inform the agent about recipes consumed throughout the day, even if they did not originate from the agent's recommendations. These registered information, along with accepted suggestions, will contribute to building the user's food diary. This diary will serve a dual purpose: it will help the agent refine its understanding of the user's tastes and provide a weekly analysis of food consumption from a sustainability perspective.

Finally, the agent must be equipped to answer any questions regarding the sustainability of one or more ingredients or recipes, including performing comparisons if necessary. It should also be able to address broader questions about food sustainability, enabling the agent to function not only as a recommender engine but also as a comprehensive expert on the topic.

The interaction should be structured using predefined steps to ensure that the conversation flows in a way that effectively guides the user toward the intended functionality. Any question or statement unrelated to the current topic of conversation or the agent's scope must be rejected, resetting the agent to a neutral state.

All information regarding recipe composition and the environmental impact of its ingredients must be computed using reliable data extracted from a validated database. This ensures accuracy and prevents hallucinations or incorrect assessments in the sustainability analysis of

a food item or recipe.

The agent should be able to contact users if no interaction has occurred within two days, serving as a reminder to further nudge them toward engaging with the system and its recommendations. Since this function may be perceived as intrusive, users should be asked during the profile setup whether they wish to enable such reminders.

No explicit gamification techniques will be implemented in this initial version, except for the weekly consumption analysis. This feature is designed to challenge users toward more sustainable consumption by highlighting instances where unsustainable recipes were consumed. However, the agent's overall architecture must remain flexible enough for future expansions, allowing the easy integration of features such as points, badges, and social comparisons with other users.

After the development phase, the conversational agent will be evaluated by a representative sample of users using a combination of techniques, such as think-aloud protocols and structured interviews. The evaluation process will be conducted with the support of an expert in usability psychology. The insights gathered from these tests will help assess the system's effectiveness and guide future improvements.

The remaining part of this work is structured as follows:

- **Chapter 2: Literature Review of Food Recommender Systems** – An exploration of the current landscape of recommender systems and AI-powered applications in the food domain.
- **Chapter 3: Theoretical Foundations of Transformers and Large Language Models** – A discussion on LLMs, the core technology underlying the proposed project.
- **Chapter 4: The HeaSEv2 Dataset** – An overview of the dataset construction process, which powers the E-Mealio agent.

- **Chapter 5: The E-Mealio Conversational Agent** – A detailed description of the architecture of the E-Mealio agent.
- **Chapter 6: Experiments and Results** – A review of the user-based qualitative and quantitative analyses, both followed by a discussion of the obtained results and possible system improvements.
- **Chapter 7: Conclusions and Future Developments** – A summary of the key findings and insights from this work, along with ideas for expanding its capabilities.

Chapter 2

Literature Review of Food Recommender Systems

This chapter analyzes the current state of recommender systems in the food domain. First, a brief introduction to recommender systems is provided to establish the proper context. Then, a review of the existing landscape of food recommender systems, both from commercial and academic perspectives, is presented. Finally, a brief analysis of other artificial intelligence systems that address food sustainability is included, given their relevance to the topic of this work.

2.1 Brief Introduction to Recommender System

The abundance of resources due to economic growth, both in media and products, can create an overwhelming experience, often leading to phenomenons like *analysis paralysis* [12] and *decision fatigue* [13], where individuals struggle to determine the right decision in the moment.

To help users navigate a stunning number of choices, a specialized field of research in computer science has emerged, focusing on filtering options to those most relevant while

anticipating user needs. This field, known as Recommender Systems, integrates artificial intelligence, information retrieval, information filtering, and psychology. Approaches vary by domain to ensure recommendations are not only relevant but also aligned with user preferences and contextual factors.

Recommender systems rely on three core entities: objects, users, and transactions. **Objects** represent the items being recommended within a specific domain, such as movies, books, or recipes. **Users** are individuals for whom recommendations are generated, each with unique preferences, behaviors, and past interactions. **Transactions** capture user-object interactions, providing valuable data that informs the recommendation process.

The functioning of a recommender system typically leverages multiple stages, each playing a crucial role in generating relevant recommendations.

1. **Training Phase**: The system collects and processes vast amounts of data, including past user interactions, item attributes, and contextual information. Machine learning and statistical analysis are then used to extract relevant patterns and build a model that captures user preferences and item characteristics.

2. **User Modeling Phase**: The system refines its understanding of individual users based on their interactions. More recent research like [14] explores enhancing this phase with multimodal data, such as text, images, and social interactions, to guide the user toward personalized recommendations.

3. **Filtering and Recommendation Phase**: Leveraging the model built during previous phases, the system applies filtering techniques like content-based filtering, collaborative filtering, or hybrid approaches to suggest the most relevant items. Factors such as item relevance, user similarity, and contextual cues help tailor recommendations to individual needs.

The primary approaches used to properly filters relevant items are the following:

- **Content-Based Approach**: This method identifies similarities between user data and potential recommendations. It utilizes past interactions, demographic details, and explicit feedback like reviews to determine relevant items. However, a key drawback is its tendency to suggest overly similar content, limiting serendipity, the ability to surprise users with unexpected yet valuable recommendations. A possible solution is to incorporate user preferences from different domains, broadening the system's understanding and diversifying recommendations.

- **Collaborative Filtering Approach**: This method predicts user preferences based on similarities with other users and is typically categorized into two perspectives: user-based and item-based collaborative filtering (CF).

  - **User-based CF**: This approach identifies users with similar feedback patterns and recommends items liked by those similar users. For example, if user A shares preferences with users B and C, and both B and C liked item X, previously unseen by A, the system is likely to suggest item X to A.

  - **Item-based CF**: This approach takes the reverse perspective. If item X was liked by users A, B, and C, and item Y was liked by users B and C, the system is likely to recommend item Y to A, assuming item X and Y appeal to the same user group.

  A practical way to represent this concept is through a user-item matrix, where each cell indicates a past interaction between a user and an item. User-based collaborative filtering (CF) identifies patterns among rows (users), while item-based CF compares columns (items). However, this approach faces the "cold-start" problem when new users join the system without prior interactions. To mitigate this, systems often suggest widely

liked items initially and refine recommendations dynamically as users engage with the content.

- **Hybrid Approach**: The previous two approaches can be combined into a more robust method that leverages both content-based modeling and user similarities to generate more accurate and diverse recommendations. This technique is widely used in platforms like Netflix and Spotify. Additionally, user connections within a service's social mechanisms can enhance recommendations, as shared interests among trusted connections help reinforce relevance. By integrating multiple strategies, hybrid approaches mitigate individual limitations and improve recommendation quality.

In the context of food recommendation, a hybrid approach can be highly effective by combining multiple factors, such as users' eating habits, their social connections, and the intrinsic properties of recipes, including taste and nutritional value. By integrating user behavior patterns with collaborative and content-based filtering, the system can offer personalized and nutritionally balanced suggestions. Additionally, leveraging social connections can enhance recommendations by incorporating shared preferences and trust among users within a community.

2.2 Landscape of Current Food Recommender Systems

The food and nutrition domain, with its unique challenges and characteristics, has garnered significant attention from both the commercial and academic worlds. Among notable food recommender systems, one of the most well-known examples was *Yummly* [15], which was discontinued at the end of 2024 [16]. This system was capable of profiling users based on their dietary restrictions, preferred or avoided ingredients, and taste preferences. Additionally, it allowed users to flag appreciated recipes, helping to identify popular dishes. Both aspects

contributed to its recommendation generation, which leveraged a hybrid approach.

A more recent commercial system is *Flavorish* [17], which incorporates image recognition and generative AI to function as both a recipe collector and a recommender system. It allows users to import recipes from various sources, including websites, handwritten notes, and even photos. Additionally, it provides personalized recipe recommendations based on a user's list of desired ingredients and dietary preferences, making it one of the most advanced publicly available systems.

On the academic side, research efforts have sought to extend food recommendation beyond individual user preferences to incorporate health-oriented insights. These approaches aim to suggest meals that align with broader well-being objectives, such as balanced nutrition, sustainability, or dietary constraints, providing a more holistic perspective on food recommendation.

A notable example is the work of Zhongqi Yang et al. on *ChatDiet* [18], which integrates a large language model (LLM) framework with causal discovery techniques applied to both individual user models and broader population data. This system provides food recommendations by interpreting user queries, retrieving relevant information, and generating structured natural language responses. However, its ingredient and recipe data are primarily derived from the LLM's latent knowledge, relying heavily on prompt engineering to incorporate necessary nutritional constraints for tailoring recommendations.

Another widely used approach involves leveraging ontologies to structure well-defined data, enabling reasoning over complex food-related relationships. For example, McKensy-Sambola et al. [19] developed an ontology-based nutritional recommender system to suggest meals for individuals suffering from obesity. Similarly, Adila and Baizal [20] employed an ontology-driven approach to recommend suitable recipes for patients with coronary heart disease. Ontologies offer a powerful framework, as they allow for complex queries and logical

reasoning over intricate domains. A prominent example of an ontology-based resource is *FoodKG*, developed by Haussmann et al. [21], which integrates multiple data sources into a comprehensive and structured dataset for food recommender systems.

In the domain of food sustainability, researchers have also explored ontologies that incorporate CO_2 emissions into their data models. Notably, the work of Yadav et al. [22] addresses this aspect, although it was not accessible for testing in the context of this study. Another noteworthy study, which also served as inspiration for this work, is that of Petruzzelli et al. [23]. Their approach utilizes food retrieval techniques to identify healthier and more sustainable alternatives based on an input recipe. Additionally, they leverage LLM capabilities to re-rank the most relevant options before presenting the final recommendation. A more detailed review of the current state of food recommender systems can be found in [24], while a broader analysis of the recommender systems designed to nudge users toward more sustainable behaviors across multiple domains, including food, is available in [25]. Despite the extensive research on this topic, few works have successfully combined the operational aspect of searching for sustainable recipes with the ability to deliver user-friendly messages that are both informative and designed to nudge behavior. This makes the focus of this work particularly unique, as it operates in a domain where direct comparisons remain limited.

2.3 Artificial Intelligence in the Food Sustainability Domain

Beyond recommender systems, artificial intelligence and data analysis play a crucial role in addressing sustainability challenges in the food sector from multiple perspectives. AI-driven innovations support various stages of the food lifecycle, from optimizing agricultural production and resource management to enhancing logistics and reducing waste.

For instance, Convolutional Neural Networks (CNNs) have been employed for monitoring soil and crop health, enabling early detection of diseases and potential issues, as demonstrated in the work of Mendoza-Bernal et al. [26]. Similarly, AI-powered sensors have been developed to monitor soil moisture levels, aiding reinforcement learning models in dynamically adjusting irrigation strategies, as seen in the work of Abioye et al. [27]. These advancements fall under the broader domain of Precision Agriculture, which enables the scientific use of resources to both maximize food production and minimize losses.

Another critical aspect of food sustainability is demand prediction, which helps mitigate food waste by aligning production with actual consumption needs. Time-series forecasting is one of the most widely used techniques for this purpose, as exemplified in the research of Yoo and Ho [28] and Taneja et al. [29]. A comprehensive review of artificial intelligence techniques addressing food sustainability can be found in [30].

Furthermore, data analysis is contributing to the development of more accurate methods for assessing the carbon footprint of recipes. The work of Long et al. [31] presents a comprehensive framework for modeling the carbon footprint of 388 recipes, considering every stage of food preparation, from ingredient production and cooking methods to the environmental impact of food waste and disposal. Although the number of recipes analyzed is limited, this framework is among the most comprehensive studies on the subject and serves as a valuable reference for future research.

Additionally, some studies are exploring how to adapt agricultural practices to rising global temperatures, aiming to enhance resilience and prevent food scarcity. A notable example is the work of Simsek-Senel et al. [32], which integrates data from multiple existing knowledge graphs to predict optimal adjustments in crop cultivation across the Netherlands. Their approach prioritizes sustainability without compromising the nutritional needs of the population, offering a data-driven strategy for climate-adaptive agriculture.

The challenges surrounding food and sustainability, from production to consumption, are complex and require advanced systems to address effectively. The scientific community is actively developing the necessary tools to tackle these issues, making them more manageable. It is then up to policymakers to recognize the potential of these resources, implement them, and raise public awareness to drive meaningful change.

Chapter 3

Theoretical Foundations of Transformers and Large Language Models

The E-Mealio conversational agent, the project at the center of this work, is powered by Large Language Models (LLMs), which serve both as tools for natural language processing tasks, such as text understanding and classification, and as generators of user-facing text.

LLMs are at the forefront of the ongoing revolution in artificial intelligence, offering highly flexible tools capable of performing a vast range of tasks. Their versatility and power have made them a key focus of current research, with significant efforts dedicated to understanding their emergent capabilities and optimizing their use for increasingly specialized applications and domains.

This chapter begins with an overview of the Transformer model, the neural network architecture that underpins modern LLMs, explaining its fundamental principles. It then explores key properties of LLMs that make them suitable as the backbone for intelligent applications.

Additionally, prompting techniques and the Retrieval-Augmented Generation (RAG) approach will be introduced as strategies to enhance their effectiveness. Finally, the chapter discusses critical ethical concerns associated with LLMs, highlighting the challenges and responsibilities involved in their development and deployment.

3.1 The Transformer Architecture

The Transformer architecture was first introduced in the seminal work *Attention Is All You Need* by Vaswani et al. [33]. In this paper, the attention mechanism was leveraged as the core component for building models capable of processing natural language. This marked a significant shift from previous approaches, where attention was primarily used as an auxiliary tool to mitigate memory fading in Recurrent Neural Networks (RNNs). The study demonstrated the fundamental capability of the attention mechanism to relate and compress information within sequential data, establishing it as a key principle in modern deep learning architectures. Their work also emphasizes the capability of the Transformer to be highly parallelizable, overcoming the main limitation of recurrent neural architectures, which strictly need to process input in a sequential manner.

The original proposed model, based on an encoder-decoder architecture, will be described. It is believed that understanding this cornerstone is sufficient to explore the later evolution of the model. Subsequently, a brief discussion of the encoder-only and decoder-only models will be provided.

The encoder-decoder Transformer model is structured, at its core, by the architecture depicted in the following diagram.

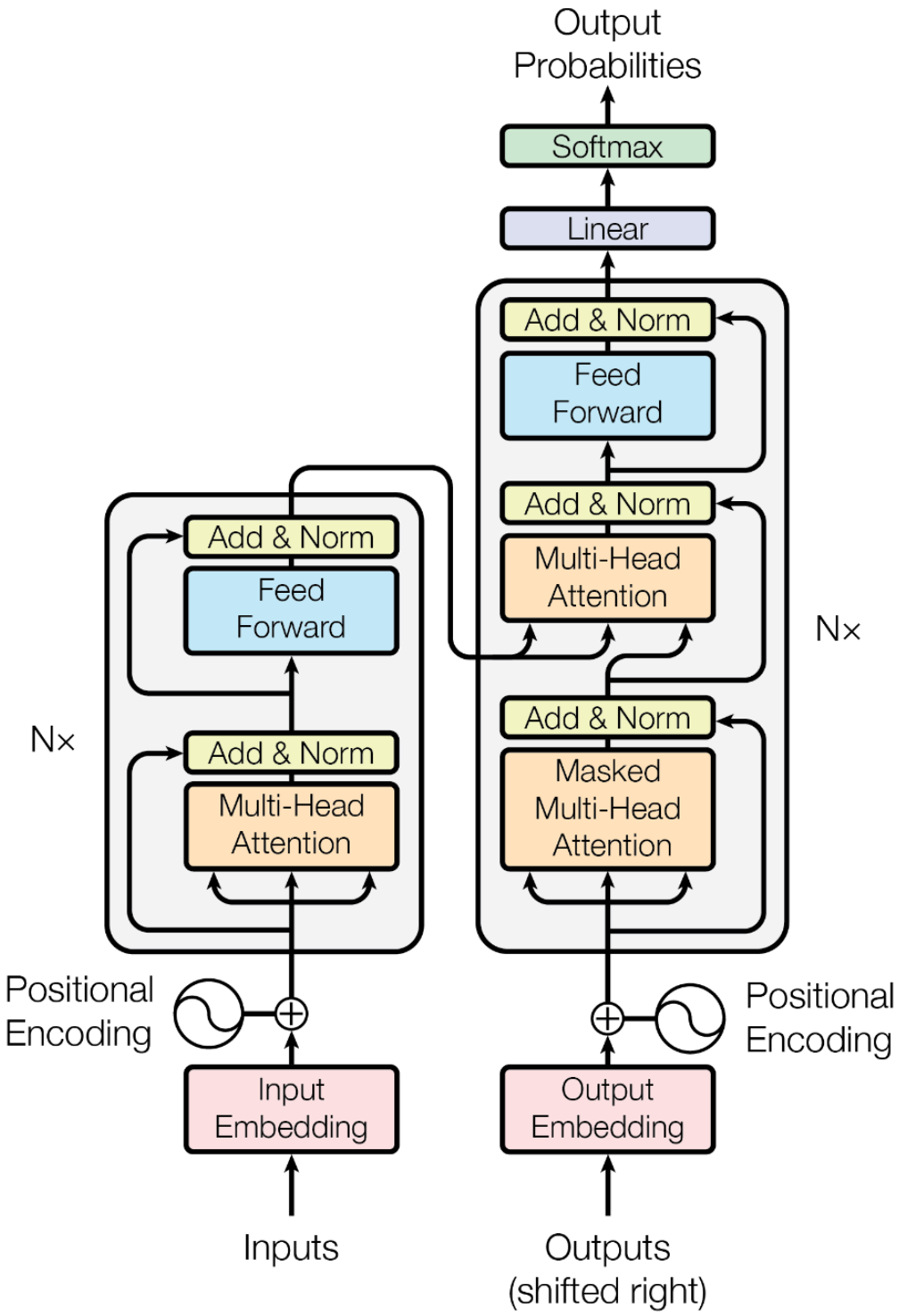


Figure 3.1: The original Transformer architecture.

The left part of the model is the *encoder*, which aims to build an intermediate representation of the input sentence by compressing latent knowledge. The right part is the *decoder*, which uses the intermediate representation produced by the encoder to generate the output sentence, relating each produced word to the complex representation created by the encoder.

The diagram in Figure 3.1 depicts just one encoder and one decoder for clarity. However, in reality, multiple encoders and decoders are stacked together to enhance the model's ability to handle more complex inputs. In the original manuscript, a stack of 6 encoders and 6 decoders was used.

3.1.1 The Encoder

Let's delve into the structure of the encoder of a Transformer by explaining all the transformation that are involved in an input sentence.

3.1.1.1 The Encoder Input: Tokenization and Embedding

The encoder is designed to process the input sequence in the form of embedded tokens. A token can be viewed as a single word; however, to handle a more restricted vocabulary, words are often split into two or more parts. For example, adverbs are frequently split into the base adjective and the "ly" suffix, resulting in two different tokens.

The tokenized words are then processed to be represented as vectors in an n-dimensional space. This process is called embedding, and it produces vectors that represent, in a mathematical form, the underlying semantics of each word. This results in interesting geometric properties, such as the famous example $\text{King} - \text{Man} = \text{Queen}$, where the vector representing the word "Queen" is obtained by subtracting the concept of being "Man" from the concept of being a "King."

This is typically achieved by training a Feed Forward Neural Network on a task like predict-

ing the next most probable word in a given corpus. Once the network is trained, the weights associated with each input (where each input is a word in the corpus) become the embedding for that word.

A well-known method in the literature to obtain embeddings is *word2vec* [34], which produces static embeddings for each word. Later, more sophisticated approaches were developed to generate dynamic embeddings capable of handling polysemy and contextual semantic differences, such as *ELMo* [35], based on bidirectional LSTM, and *BERT* [36], an encoder-only model that will be briefly explored later.

Transformer models can either use pre-trained embeddings or treat the embedding layer as a trainable set of weights, ultimately learning their own set of embeddings during training.

3.1.1.2 Representing the Position: The Positional Encoding

The embedded words provided as input do not contain any information about their position in the sentence, as each embedding represents a word independently. However, since the Transformer processes all the input words independently, a way to represent their position is needed in order to encode this information into their vector representations and exploit it in the subsequent phases of processing.

A simple and elegant solution to this problem is to add, to each component of the embedding vector of each word, a value that changes in a cyclical manner, similar to a trigonometric function. In practice, for each component of a d[1] dimensioned vector a value obtained by applying the following formulas is added.

$$PE(pos, 2i) = \sin\left(\frac{pos}{10000^{2i/d_{\text{model}}}}\right) \tag{3.1}$$

$$PE(pos, 2i+1) = \cos\left(\frac{pos}{10000^{2i/d_{\text{model}}}}\right) \tag{3.2}$$

[1] In the original manuscript the dimension of the embeddings d was fixed at 512.

Where i is a given component of the d-dimensional vector and *pos* is the position of the token for which we are computing the positional encoding vector.

The mechanism can be clearly understood by visualizing how the positional encodings are produced, as depicted by the following dummy example with an embedding vector of 4 components.

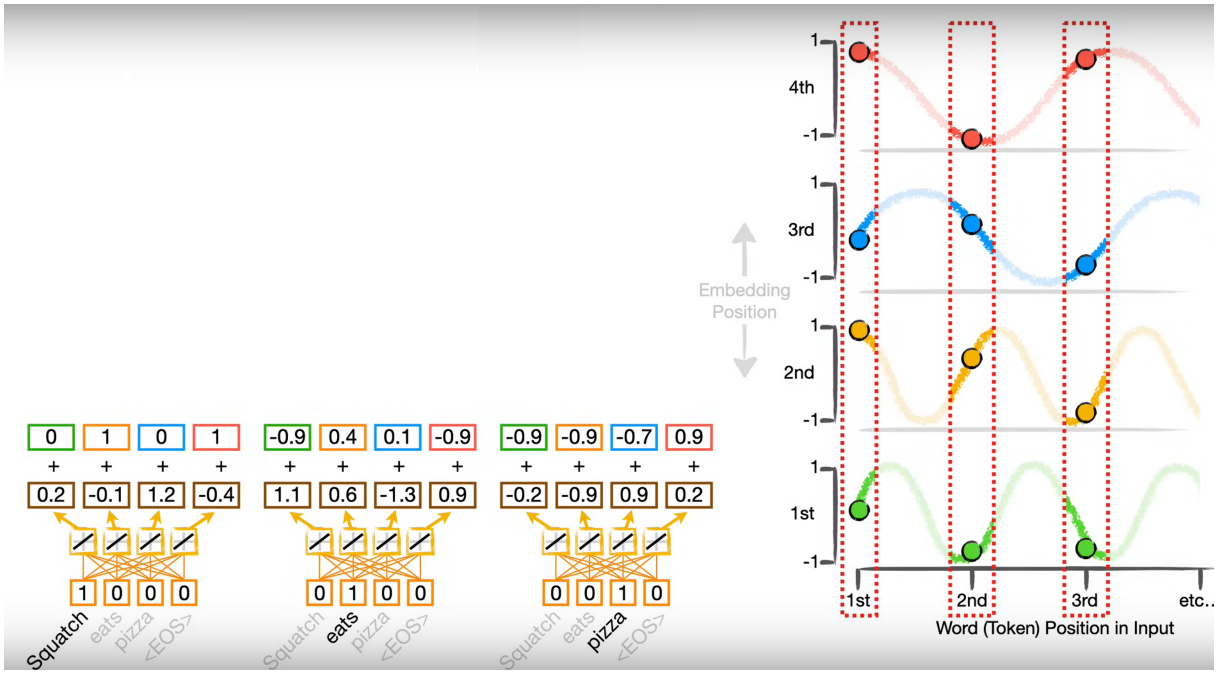


Figure 3.2: Positional encodings are derived by combining different sine and cosine functions at various frequencies, which are then added to the embedding vector. This visual example is based on the work of Josh Starmer from the YouTube channel StatQuest [37]. I would like to publicly thank him for his work, making it possible to grasp complex concepts in a simple yet highly effective way.

The combination of these d interleaved trigonometric values ensures that the positional encodings for different positions are sufficiently distinct. The periodic nature of the sine and cosine functions, with their varying frequencies, guarantees that tokens in different positions will receive unique positional encoding values, preventing any two positions from having

identical encodings.

3.1.1.3 Representing the Contextual Dependencies: The Self-Attention Mechanism

Self-attention is the core mechanism of Transformers. It allows the model to produce new representations of each input word, that capture the dependencies between the current word and all other words in the sequence. This is particularly useful when dealing with sentences like:

> *"The animal didn't cross the street because it was too tired."*

The word *it* is ambiguous: does it refer to *animal* or *street*? Given the context, it is easy for humans to understand that *it* refers to *animal*. However, this is not a trivial task for machine learning models.

Understanding these kinds of relationships and producing new representations that capture this information is the task of the self-attention mechanism, where the relationship between each word and all the other words is computed.

3.1.1.4 Computing Attention for a Single Token

The attention of an embedded token $\mathbf{x}_1$ in relation to the other tokens $\mathbf{x}_2$, ..., $\mathbf{x}_n$, is computed as follows:

1. **Projection into Query, Key, and Value Representations**

   Each input token embedding $\mathbf{x}_i$ is transformed into three new vector representations: **Query** ($\mathbf{q}_i$), **Key** ($\mathbf{k}_i$), and **Value** ($\mathbf{v}_i$). This transformation is performed using three corresponding learnable weight **matrices**:

$$\mathbf{q}_i = \mathbf{x}_i \mathbf{W}^q, \quad \mathbf{k}_i = \mathbf{x}_i \mathbf{W}^k, \quad \mathbf{v}_i = \mathbf{x}_i \mathbf{W}^v$$

Where:

- $\mathbf{W}^q$ has dimensions (d, d_k).
- $\mathbf{W}^k$ has dimensions (d, d_k).
- $\mathbf{W}^v$ has dimensions (d, d_v).
- d is the dimension of the embedded token.

So, $\mathbf{W}^q$ and $\mathbf{W}^k$ share the same dimensions.

2. **Computing the Attention Score**

   The Query vector $\mathbf{q}_1$ of the attended token $\mathbf{x}_1$ is compared with all the Key vectors $\mathbf{k}_i$. This comparison is done using the **scaled dot-product**, where the dot product of $\mathbf{q}_1$ and $\mathbf{k}_i$ is divided by the square root of the dimension d_k to stabilize gradients:

$$\alpha_{1,i} = \frac{\mathbf{q}_1 \cdot \mathbf{k}_i}{\sqrt{d_k}}$$

   Finally, a **softmax** function is applied across all computed attention scores, producing a probability distribution over the importance of each token relative to $\mathbf{x}_1$:

$$s_1 = \text{softmax}(\alpha_{1,1}, \alpha_{1,2}, \ldots, \alpha_{1,n})$$

   Here, s_1 represents the attention weights that capture the relative importance of each token to the attended token $\mathbf{x}_1$.

3. **Computing the Final Contextualized Representation**

   The computed attention scores $s_{1,i}$ (which are the output of the softmax) are used to weight the corresponding Value vectors $\mathbf{v}_i$, generating a new context-aware representation of $\mathbf{x}_1$:

$$\mathbf{z}_1 = \sum_i^n s_i \mathbf{v}_i$$

Here, $\mathbf{z}_1$ is the new representation for $\mathbf{x}_1$, which is a weighted sum of the Value vectors $\mathbf{v}_i$, with weights s_i assigned by the softmax.

3.1.1.5 Efficient Self-Attention Computation Using Matrix Operations

To make the self-attention computation more efficient, rather than computing it for each token individually, all token embeddings are stacked together into a **tensor**. This allows the model to perform computations efficiently using matrix operations.

1. **Stacking Inputs into Matrices**

   The token embeddings are stacked to form the matrix $\mathbf{X}$ of dimension (n, d). Then, they are projected into Query, Key, and Value matrices using the learnable weight matrices $\mathbf{W}^q$, $\mathbf{W}^k$, and $\mathbf{W}^v$:

$$\mathbf{Q} = \mathbf{X}\mathbf{W}^q, \quad \mathbf{K} = \mathbf{X}\mathbf{W}^k, \quad \mathbf{V} = \mathbf{X}\mathbf{W}^v$$

2. **Computing Self-Attention for All Tokens**

   The attention scores are computed using matrix multiplication and scaling:

$$\mathbf{A} = \text{softmax}\left(\frac{\mathbf{Q}\mathbf{K}^T}{\sqrt{d_k}}\right)$$

   The final context-aware representations are obtained by multiplying the attention matrix $\mathbf{A}$ with the Value matrix:

$$\mathbf{Z} = \mathbf{A}\mathbf{V}$$

This formulation allows the Transformer to efficiently compute self-attention across all tokens in parallel, making it much faster than sequential models like RNNs.

3.1.1.6 Learning Multiple Dependencies: The Multi-Head Attention Layer

The previously described process relates to a single attention mechanism, which represents the entire input while accounting for a specific dependency between various tokens. However, in natural language, multiple dependencies exist between words, and a single attention mechanism may not be sufficient to capture all of them.

To address this, the attention is computed h times in parallel using different sets of learnable weight matrices $\mathbf{W}^q$, $\mathbf{W}^k$, and $\mathbf{W}^v$. Each set corresponds to a separate *"head"* and each head captures a different aspect of the relationships between tokens in the sequence.

Specifically, for each attention head i, the following operations are performed:

$$\mathbf{Q}_i = \mathbf{X}\mathbf{W}_i^q, \quad \mathbf{K}_i = \mathbf{X}\mathbf{W}_i^k, \quad \mathbf{V}_i = \mathbf{X}\mathbf{W}_i^v$$

where $\mathbf{X}$ is the matrix of input token embeddings, and the matrices $\mathbf{W}_i^q$, $\mathbf{W}_i^k$, and $\mathbf{W}_i^v$ are different for each attention head i. These weight matrices are learned during training, allowing each head to focus on different aspects of the input.

After computing the attention for each head, the resulting context-aware representations are concatenated together and projected back into the original dimension using a final learnable weight matrix $\mathbf{W}^O$:

$$\mathbf{Z} = \text{Concat}(\mathbf{Z}_1, \mathbf{Z}_2, \ldots, \mathbf{Z}_h)\mathbf{W}^O$$

Where:

- $\mathbf{Z}_i$ is the output of the attention computation for the i-th head.

- $\mathbf{W}^O$ is a weight matrix that projects the concatenated output back into the original dimensionality.

This process allows the model to capture a richer set of dependencies by using multiple attention heads that each focus on different types of relationships between tokens. The multi-head attention mechanism enables the model to process the sequence from multiple perspectives, improving its ability to understand complex patterns in the input data.

3.1.1.7 Preserving Information Flow: The Residual Connection

Residual connections are a key component of deep neural networks, helping to mitigate the vanishing gradient problem, which can make training deep models difficult. Essentially, they act as shortcuts that allow the original information to flow through deeper layers, ensuring that essential features are not lost during transformations.

In the Transformer architecture, residual connections play a crucial role in preserving information as it propagates through multiple layers. Specifically, they ensure that the representations refined by multi-head self-attention and feed-forward layers retain elements of the original input. This is particularly important because the self-attention mechanism can dynamically mix information across tokens, potentially altering individual token representations.

Mathematically, the residual connection is implemented as follows:

$$\mathbf{N} = \text{LayerNorm}(\text{Dropout}(\mathbf{Z}) + \mathbf{X})$$

where:

- $\mathbf{Z}$ represents the transformed representation produced by the multi-head attention or feed-forward layer.

- $\mathbf{X}$ is the input to the layer, which may be either the token embeddings (including positional encodings) or the output of the previous sublayer.

The normalization step, typically implemented using *Layer Normalization*, helps stabilize training, while *Dropout* acts as a regularization mechanism to reduce overfitting by randomly deactivating certain neurons during training. Together, these mechanisms allow the model to learn more robust and generalizable representations.

3.1.1.8 The Last Step: The Feed Forward Neural Network Layer

As previously described, each embedded input token has been enriched with positional information and relationships with other tokens in the sentence. To further refine this information, a feed-forward neural network (FFN) is applied independently to each processed input token. This final step can be viewed as asking, for each word, what it represents in the context of the processed language.

The operation of the Feed Forward Neural Network (FFN) can be represented as follows:

$$\text{FFN}(\mathbf{x}) = \text{Linear}_2(\text{ReLU}(\text{Linear}_1(\mathbf{x})))$$

Where:

- $\mathbf{x}$ is the input vector, with dimensionality d_{model},
- $\text{Linear}_1(\mathbf{x}) = \mathbf{x}\mathbf{W}_1 + \mathbf{b}_1$ is the first linear transformation, where the output dimensionality is typically larger than the input (usually $d_{\text{model}} \times n$ for some factor n),
- $\text{ReLU}(x) = \max(0, x)$ is the activation function applied element-wise after the first linear transformation,
- $\text{Linear}_2(\mathbf{x}) = \mathbf{x}\mathbf{W}_2 + \mathbf{b}_2$ is the second linear transformation, where the output dimensionality is d_{model}, matching the input size.

Finally, to the output of the Feed Forward Neural Network, another normalized residual connection is applied, as described earlier. This process ensures that the information flows smoothly through the layers while maintaining stability during training.

3.1.1.9 Stacking Multiple Encoder Blocks

The entire processing pipeline, consisting of embeddings, positional encoding, multi-head attention, residual connections, feed-forward neural networks, and a final residual connection, constitutes a single *Encoder Block*. This block is capable of capturing complex relationships between each input token. To enhance the model's capacity, multiple encoder blocks are often stacked sequentially. This stacking enables the model to capture both local (intra-sentence) and global (inter-sentence) dependencies within the input text.

Each encoder block receives the output of the previous one, progressively refining the token representations. By iterating this process on increasingly abstract representations, the model can learn more intricate patterns and dependencies throughout the sequence.

3.1.2 The Decoder

The decoder in a Transformer shares a similar internal structure with the encoder. It is also fed with token embeddings, which correspond to the training tokens (e.g., the known translated sentence) during training or the previously generated tokens during inference.

These embedded tokens are enriched with positional information before being processed by a *Masked Multi-Head Attention* layer. This differs from the standard multi-head attention used in the encoder, as it restricts each token's attention computation to itself and the preceding tokens in the sequence. While this is a natural constraint during inference, since the subsequent tokens have not yet been generated, it is crucial during training. The masking ensures that the decoder learns dependencies only from previously generated tokens, preventing

it from "seeing" information that would not be available during inference.

Following this, another *Multi-Head Attention* layer is applied. However, unlike the masked version, this attention mechanism enables each token in the decoder to attend to the contextual representations produced by the encoder. Specifically, it compares the *Query* representations from the decoder side with the *Key* and *Value* vectors from the encoder side. This interaction allows the decoder to incorporate relevant information from the fully processed source sequence. In the context of machine translation, this step ensures that the decoder effectively leverages the meaning of the input sentence to guide the generation of the output. This layer is commonly referred to as *Encoder-Decoder Attention*.

Next, a *Feed Forward Neural Network* is applied independently to each token's contextualized representation, similar to the encoder. Each of the three aforementioned steps is supported by *residual connections* and *layer normalization* to enhance information flow, stabilize training, and maintain the integrity of the learned representations.

This pipeline, consisting of *Masked Multi-Head Attention*, *Encoder-Decoder Multi-Head Attention*, and a *Feed-Forward Neural Network*, can be repeated multiple times. Each successive decoder layer produces a more abstract representation of the output context, enabling the model to capture long-range dependencies more effectively.

Finally, the output of this stack is linearly projected onto a space with dimensionality $|V|$, where V represents the token vocabulary of the output language. The linearly transformed output is then passed through a *softmax* layer, which converts the logits vector into a probability distribution. The token associated with the highest probability score is selected as the predicted token by the model.

The overall structure of the decoder enables it to iteratively generate output tokens while maintaining coherence and contextual consistency.

During inference, the decoder is initialized with a special *start token*, which signals the

beginning of the output generation process. Based on this token, the decoder generates the next token from the output vocabulary V. This newly generated token is then used, along with all previously generated tokens, to predict the next token. This *autoregressive* process continues until a special *end token* is produced, signaling the termination of the sequence generation.

The following image summarizes everything explained so far about Transformers, using the architecture diagram as a reference.

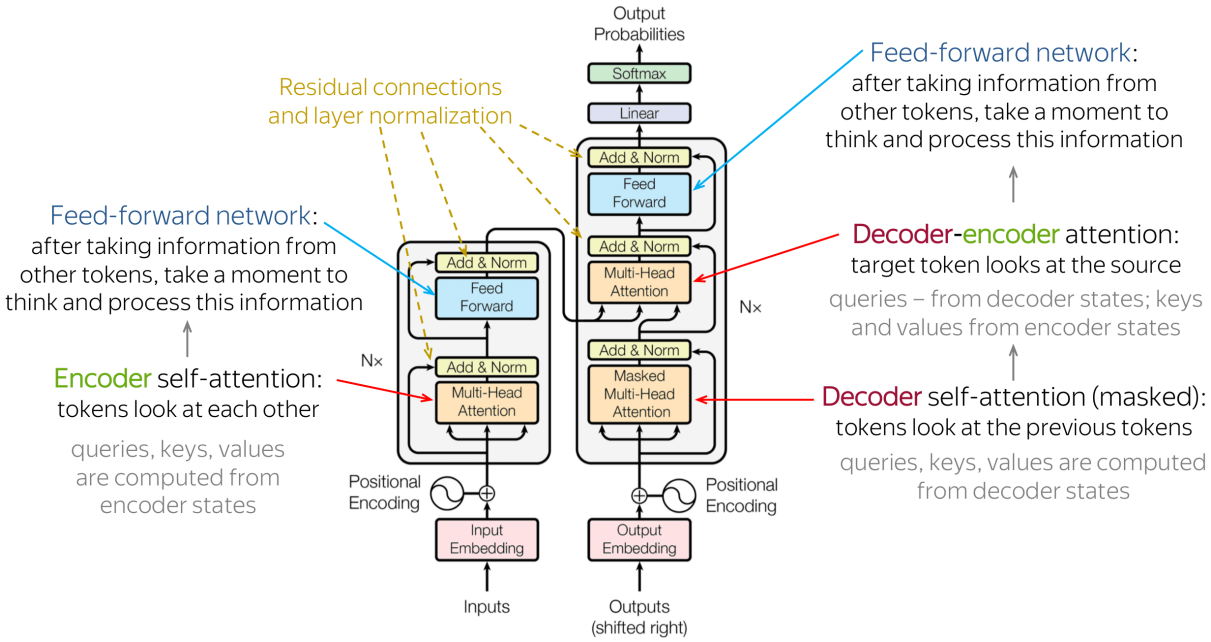


Figure 3.3: Overview of the Transformer architecture summarizing key components and concepts discussed.

3.1.3 Encoder-Only and Decoder-Only Transformers

The Transformer model, as previously described, utilizes both encoders and decoders to process input context and generate output. This architecture was originally inspired by sequence-to-sequence (seq-to-seq) models, which were traditionally implemented using recurrent neural networks (RNNs) such as LSTMs. However, further advancements have demonstrated that

these two components can be used independently to tackle different types of tasks while still producing meaningful results. Encoder-only architectures are particularly suited for tasks requiring deep contextual understanding, such as text classification and named entity recognition, whereas decoder-only architectures excel in generative tasks, such as text completion and language modeling.

On the encoder-only side, notable Transformer models like BERT are widely used for generating context-aware embeddings of sentences. These embeddings are particularly valuable as they capture semantic meaning at the sentence level, enabling tasks such as semantic search and document clustering. Such models are typically trained on tasks like reconstructing missing parts of text or next sentence prediction, where self-attention is leveraged effectively.

Encoder-only models can also serve as foundational models for various NLP tasks, such as text classification and sentiment analysis. By simply adding a fully connected layer on top and fine-tuning the model with a labeled dataset, they can be easily adapted to specific downstream applications with minimal effort.

On the decoder-only side, models such as GPT [38] and LLaMA [39] form the basis of the current generation of large language models (LLMs). These models are primarily trained on tasks like predicting the next token in a sequence, leveraging their autoregressive nature. By training on this task, these models learn a rich, latent representation of language that captures complex patterns and structures. This representation can then be fine-tuned and applied to a wide range of downstream applications, such as question answering, where models can be further refined using techniques like reinforcement learning. This allows LLMs to generalize their understanding of language and excel in tasks requiring deep contextual understanding.

3.2 Key Properties of Large Language Models

Decoder-only transformers are models that scale their capabilities as the model's dimensionality and the volume of training data increase. This fundamental property enables these models to achieve state-of-the-art performance in both text understanding and generation tasks. A significant milestone was reached with the 2020 paper *Language Models are Few-Shot Learners* [40], where GPT-3, with its 175 billion parameters, demonstrated the ability to adapt its behavior based on a provided input sentence, or *prompt*, without requiring any fine-tuning. This groundbreaking capability made GPT-3 and similar models versatile tools, functioning as general-purpose language manipulators capable of handling a wide range of tasks simply by varying the input prompt.

This advancement marked the beginning of a new era in artificial intelligence, where the focus on these models has increased exponentially, owing to their potential impact across multiple sectors of society.

Finally, these models can be easily adapted to handle a wide range of sequence-based data, learning to represent dependencies from various modalities such as audio and images, as well as complex sequential data like protein structures and DNA sequences. This adaptability allows the development of both general-purpose intelligent agents and domain-specific tools, which can be leveraged by researchers to significantly accelerate scientific progress [41] [42].

3.3 Using Large Language Models as Tools: Prompt Engineering and RAG

Large Language Models (LLMs) are currently used as general-purpose tools across a wide range of tasks, thanks to their ability to adapt their behavior based on a set of instructions

in a given context. This flexibility has led to the development of new areas of study aimed at understanding the emergent capabilities of these models. As a result, techniques like prompt engineering and Retrieval-Augmented Generation (RAG) have been invented to more effectively leverage the power of LLMs.

Prompt engineering focuses on how to effectively instruct these models to perform complex tasks by carefully crafting the input prompts. This allows users to get the most out of the model's capabilities. On the other hand, Retrieval-Augmented Generation (RAG) is a technique that enhances the model's responses by steering them toward the content of a provided document, making it especially useful for personalizing the model's output and ensuring that the responses align more closely with specific content or knowledge.

3.3.1 Main Prompt Engineering Techniques

Prompt engineering can be defined as [43]:

> Prompt engineering is a relatively new discipline for developing and optimizing prompts to efficiently use language models (LMs) for a wide variety of applications and research topics. Prompt engineering skills help to better understand the capabilities and limitations of large language models (LLMs).

Several techniques have been developed, with the most notable being *Zero-shot Prompting*, *Few-shot Prompting*, and *Chain of Thought Prompting*. These techniques are briefly discussed here. For a more in-depth analysis, refer to [44].

3.3.1.1 Zero-shot Prompting

Zero-shot Prompting is the most straightforward technique used when interacting with LLMs. It simply consists of providing a prompt that instructs the model to perform a task on a given

input (e.g., translation or text classification) while relying solely on the model's latent knowledge representation to generate a response, without providing any examples of compliant outputs.

For instance, the following prompt:

> Classify the text as neutral, negative, or positive.
>
> Text: I think the vacation is okay.
>
> Sentiment:

is an example of Zero-shot Prompting. The expected output could be:

Neutral

This technique is effective for simple tasks where the model's internal contextualization of the prompt is sufficient to generate an accurate response.

However, Zero-shot Prompting is often unreliable for domain-specific tasks where the model may lack the necessary specialized knowledge. To overcome this limitation, fine-tuning the model with a dataset containing correct examples can significantly improve its performance on non-trivial tasks [45]. On the other hand, fine-tuning a model is not always feasible, especially for non-technical users or when working with a closed-source model. To overcome these limitations, fine-tuning can be avoided by leveraging more precise and specialized prompting techniques.

3.3.1.2 Few-shot Prompting

An effective way to guide the model toward the desired behavior is by providing a set of examples within the prompt to serve as references when generating the response. This technique, known as Few-shot Prompting, leverages the input examples to influence how the model's

latent knowledge representation is applied to produce the output. An extended version of this prompting technique was applied in the prompt that manages the main hub of the E-Mealio agent.

An example of Few-shot Prompting is following prompt:

> A ”whatpu” is a small, furry animal native to Tanzania. An example of a sentence that uses the word whatpu is: We were traveling in Africa and we saw these very cute whatpus.
>
> To do a ”farduddle” means to jump up and down really fast. An example of a sentence that uses the word farduddle is:

Where only one example is provided. The expected output could be:

> *When we won the game, we all started to farduddle in celebration.*

In this example, it is evident how the provided example and the definition were used to generate a compliant response. For more difficult tasks, it is possible to experiment with increasing the number of demonstrations (e.g., 3-shot, 5-shot, 10-shot, etc.).

When providing examples, it is ideal to present them in a coherent and well-formatted manner. However, studies like [46] have shown that even non-coherent or poorly formatted examples can still positively influence the model when generating its response, demonstrating the high abstraction capabilities of these models.

Although more powerful than the Zero-shot technique, Few-shot Prompting can still fail when a multi-step process is required to provide a correct answer. This limitation can be overcome by the next prompting technique presented.

3.3.1.3 Chain of Thought Prompting

Chain of Thought Prompting was a game-changer in the study of the emergent properties of Large Language Models. Introduced in [47], it enables LLMs to perform tasks that were previously thought to be impossible for language-based models. This technique involves providing an excerpt of a reasoning process within the input prompt, which is then leveraged as an example for generating the desired result.

Unlike Few-shot Prompting, which simply lists plausible results, Chain of Thought Prompting explicitly outlines the entire reasoning process. It mimics the internal dialogue that humans engage in when reasoning about complex problems, guiding the model to follow a logical progression.

For instance, the following prompt:

> Q: If I have 50 apples, I eat 10 of them and I throw away other 2. How many apples do I have?
>
> A: You have 50 apples. If you eat 10, 50-10=40. If you throw away other 2, 40-2=38
>
> Q: If I have 35 apples, I eat 1 of them and I throw away other 4. How many apples do I have?
>
> A: You have 35 apples. If you eat 1, 35-1=34. If you throw away other 4, 34-4=30
>
> Q: If I have 23 apples, I eat 6 of them and I throw away other 4. How many apples do I have?
>
> A:

is an example of Chain-of-Thought Prompting, where multiple examples of reasoning is provided.

The expected output could be:

You have 23 apples. If you eat 6 of them, you have 23 - 6 = 17 apples. If you throw away other 4, you have 17 - 4 = 13 apples. The answer: 13.

This technique has evolved into different forms, such as Zero-Shot Chain of Thought, which, as discovered in [48], enables the production of realistic reasoning by simply applying the phrase "Let's think step by step" to the original prompt.

Subsequently, given the dramatic improvement in response quality, a new generation of LLMs have been trained to generate Chain of Thought (CoT) reasoning by themselves before providing the final response. Models of this type include OpenAI's o1 [49], trained by combining reinforcement learning and examples of reasonable CoT, and DeepSeek-R1-Zero [50], which is trained to discover CoT on its own through reinforcement learning on well-defined problems.

The nature of this technique, which relates reasoning to language manipulation, has changed the perception of these models, from "simple" next token predictors to tools capable of reasoning about complex problems. This shift has important philosophical consequences regarding the nature of reasoning and how it manifests in intelligent agents, both biological and artificial. It deserves to be studied deeply from both engineering and humanistic perspectives to fully understand its implications.

3.3.2 Adapting Model Answers to a Specified Domain: Retrieval Augmented Generation

Retrieval-Augmented Generation (RAG) is a technique that integrates prompt engineering with a dedicated text-retrieval pipeline to create dynamic prompts. Relevant information is retrieved and injected into the model's input, providing reliable context for generating responses. This approach enhances factual consistency, improves the reliability of generated

content, and mitigates the issue of ”hallucination,” where responses are generated based on linguistic plausibility rather than real-world knowledge. RAG serves as the foundation of the E-Mealio agent, ensuring greater accuracy and trustworthiness in the provided answers.

First described by Meta AI researchers [51], the RAG technique involves searching for relevant documents based on a given textual input. This is typically done by computing the similarity between the input and document embeddings, where Encoder-Only Transformers can be leveraged. The retrieved relevant sources are then provided as part of the LLM's prompt, allowing the model to generate an answer that directly references the supplied context.

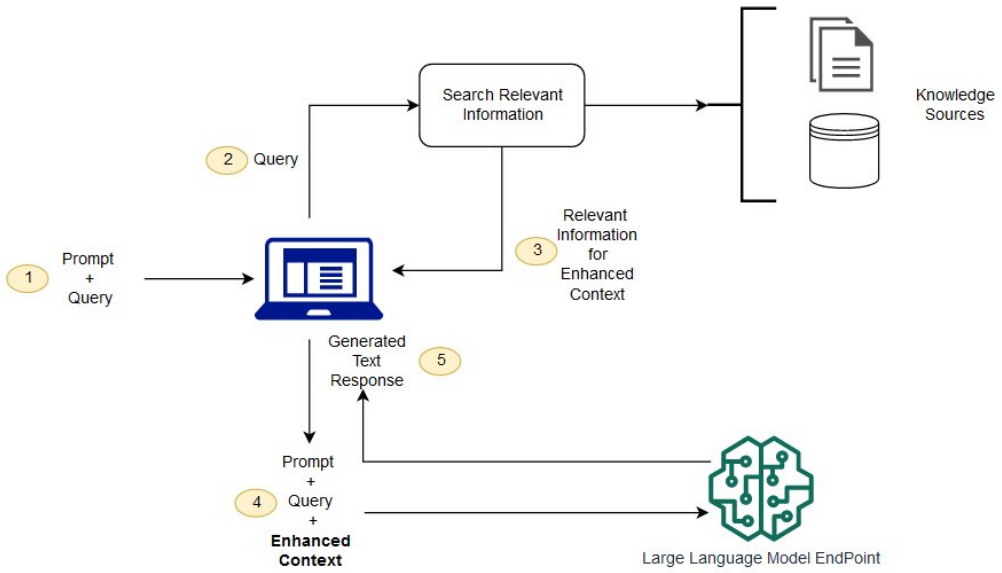


Figure 3.4: Overview of the architecture of a RAG-based system.

This makes RAG particularly useful in situations where factual information evolves over time. Given that LLMs' parametric knowledge is static, RAG allows language models to bypass retraining, enabling access to the latest information for generating more reliable and up-to-date outputs. More recently, these retrieval-based approaches have gained popularity and are being integrated into widely used LLM interfaces, such as ChatGPT [52], to improve capabilities and reliability.

3.4 Ethical Considerations of Large Language Models and Related Products

The evolution of LLMs marks a new ”summer” for AI, with growing investments from both public and private sources. This abundance of resources and focus is allowing the field to expand exponentially. However, given the significance of this technological revolution, an analysis of the ethical consequences of such powerful tools should not be dismissed. It is crucial to harness this power for the common good, rather than allowing it to be used as a means of control and social manipulation.

Several considerations can be explored, such as the impact of these tools on the educational system. They can easily be used uncritically to bypass assignments and proper study, which could hinder learning. Furthermore, the ability of LLMs to spread misinformation is a major concern. This can occur both as an unintended side effect due to the hallucinatory nature of these models and as a tool for social engineering, potentially exploited by governments or politicians. Another crucial aspect is the development of virtual companions that can express emotions and behave as if they were fully human. This scenario, wonderfully depicted in the 2013 movie *Her* by Spike Jonze, has become a reality. While fascinating, this raises several ethical concerns, thoroughly examined in the 2023 articles *Chatbots Are Not People* by Rick Claypool [53] and *The Problem With Counterfeit People* by philosopher Daniel C. Dennett [54]. These works explore the psychological and social implications of normalizing the presence of highly realistic, non-human agents that can exhibit human-like behavior and even empathy.

Some key findings from are:

- There is a risk that conversational systems with commercial or political agendas will be used to influence people. These systems can potentially access vast amounts of data on

their targets, detect emotional cues like facial expressions and pupil dilation, and adjust their responses in real time to enhance persuasion.

- There is a risk of users becoming emotionally entangled with conversational conversational systems marketed as virtual friends and romantic partners in ways that can foster dependence, encourage harmful behaviors, and undermine social bonds between real people.
- There is a risk of charismatic conversational systems being designed to assume the roles of authority figures such as therapists, doctors, teachers, lawyers, and life coaches, whose advice and instructions can have a disproportionate effect on decisions individuals make.
- Prioritizing interactions with machines – even if they are genuinely validating – can risk further isolating people, potentially eroding social bonds and leaving individuals depending on synthetic sociality that is, at its core, transactional rather than relational.

A potential solution to these problems is to design systems that, despite their ability to produce meaningful text, are still perceived as tools rather than sentient beings. This can be achieved by placing constraints on the topics a chatbot can discuss, reducing their ability to mimic realistic conversations. Additionally, providing these systems in broader contexts, such as apps or software, where their role is understood as that of a ”helper” rather than a sentient agent, could mitigate these concerns.

While these aspects may seem beyond the scope of this work, they have significantly influenced the presented agent's design. The conversation structure reinforces that the chatbot functions strictly as a tool rather than a sentient entity, supporting the idea of developing beneficial technologies while ensuring that human interaction remains between people, not chatbots.

Part II

System Architecture, Design and Experiments

Chapter 4

HeaSEv2 Dataset

The conversational agent object of this work is powered by a refined and expanded version of the HeaSE dataset created as university project work by Giovanni Tempesta and Michele Di Carlo [55], here called *HeaSEv2*.

The initial version of the HeaSE dataset was developed by integrating information from the HUMMUS dataset [56] with sustainability data on carbon footprint (CFP) and water footprint (WFP) from the SU-EATABLE LIFE project [57], following a methodology similar to the one described in a previous work by Iacovazzi and Gigantelli [58]. Funded by the European Union, the SU-EATABLE LIFE project compiles reliable Life Cycle Assessment (LCA) data on CFP and WFP for a wide range of common ingredients. Additionally, it provides categorized values for ingredient typologies, ensuring researchers have consistent and trustworthy data for their studies.

Starting from the HUMMUS dataset, a data cleaning process was applied to refine and structure the information. After this process, the final dataset included 93,748 recipes and 79,195 ingredients, provided as two separate CSV files. These ingredient data were then enriched with sustainability information by performing substring searches within the SU-EATABLE

LIFE dataset food items. For unmatched ingredients, semantic similarity techniques were applied to find the most relatable match. While this represented a promising first step in linking a large recipe dataset with grounded sustainability data, further refinements were necessary to enhance its accuracy and reliability.

The following sections outline the steps taken to evolve the HeaSE dataset into HeaSEv2.

4.1 Reducing the Number of Ingredients

An initial enhancement to the HeaSE dataset involved simplifying the ingredient list. The original list, which contained 79195 entries, included a substantial amount of duplication. Many ingredients were distinguished solely by minor variations in flavor, branding, or the use of descriptive adjectives.

The simplification procedure utilized an LLM-based approach to automate the creation of a more streamlined ingredient list, effectively minimizing duplication. For each recipe in the HeaSE dataset, the corresponding instance of the *ingredients* field in the HUMMUS *recipe* dataset was extracted. Then, a call to the GPT API[1] was made to generate a simplified and standardized version of each provided ingredients list.

The model was instructed by means of the following prompt.

[1]The model employed for this task was GPT-4o-mini [59], chosen for its balance between precision and cost efficiency.

Ingredients Streamliner System Prompt

Given a noisy text provided by the user: Extract the ingredients' name using the simplest noun possible in singular form, like a vocabulary entry; then derive the related quantity using the available information. Quantities can be expressed in different ways, like in ounces, grams, cups, tablespoons, or units of the ingredient. If you can't derive a quantity, just provide the ingredient's name. Do not made up quantities.

Use the following output structure: ['ingredient_name1 _ quantity1 __ unit of measure1', 'ingredient_name2 _ quantity2 __ unit of measure2', ... , 'ingredient_nameN _ quantityN __ unit of measureN']

Do not output anything else outside what requested.

The prompt was designed to simplify the naming of each ingredient and standardize the presentation of quantity and unit measurements.

To illustrate how the model processes ingredient extraction, the following examples present original ingredient list alongside their corresponding structured output. The model simplifies ingredient names, converts them to singular form where applicable, and standardizes quantity representation. If a quantity is unavailable, only the ingredient name is retained.

Original ingredients:

{[('pork roast', '1 time(s) (3 -4 lb) '),
('jar apricot preserves', '1 time(s) (12 ounce) '),
('jar horseradish sauce', '1 time(s) (12 ounce) '),
('medium onion, sliced', '1 time(s) ')],

[('bunch romaine lettuce', '1 time(s) '),
('cauliflower, cut into bite size pieces', '2 time(s) cups '),

('mushrooms, sliced', '8 time(s) ounces'),

('pomegranate seeds', ' time(s) ')]}

Processed simplified ingredients:

{['pork _ 3–4 __ lb',

'apricot preserve _ 12 __ ounce',

'horseradish sauce _ 12 __ ounce',

'onion _ 1 __ unit'],

['lettuce _ 1 __ bunch',

'cauliflower _ 2 __ cups',

'mushroom _ 8 __ ounces',

'pomegranate _ __ ']}

In this example, the model extracts the core ingredient names while removing unnecessary descriptors. The quantities are formatted uniformly with appropriate units of measure.

The resulting streamlined ingredient list was saved as the updated version of the *ingredient* field of the HeaSEv2 *recipe* dataset, while the original ingredient list used as input was preserved in a new field called *original_hummus_db_ingredients* to ensure no loss of original information.

Finally, each unique ingredient from the new simplified lists was extracted and used to rebuild the ingredient dataset. This approach ensured consistency and prevented mismatches between the items in the *ingredient* dataset and the ingredients reported in the *recipe* dataset.

The whole procedure successfully reduced the ingredient list from 79195 entries to 9086, achieving an 88.5% reduction in the number of distinct items reported.

However, this method did not entirely eliminate duplicates, as several similar ingredients remain with slight variations in their names. Additionally, some ingredients were overly sim-

plified, resulting in overly generic terms and a corresponding loss of useful information.

Further refinements of this procedure may be necessary to enhance reliability and minimize information loss. These improvements could involve optimizing the prompt design or exploring alternative approaches to better handle the rewriting of ingredient lists.

Despite these limitations, the new ingredient list is surely more compact and can be leverage to simplify the association with the SU-EATABLE LIFE food item list.

4.2 Association of Ingredients Footprints

Another problem of the original version of the dataset concerns the strategy adopted to match the HeaSE ingredient name with the corresponding SU-EATABLE LIFE ingredients in order to inherit its sustainability data. As already mentioned, the primary technique adopted involved a substring search between the single words of an HeaSE ingredient and the words in the ingredients of the SU-EATABLE LIFE dataset. This approach led to several spurious and inconsistent associations, where ingredients that should be mapped to the same SU-EATABLE LIFE ingredient were instead assigned to different ingredients, introducing noise into the dataset.

Furthermore, this approach does not account for the possibility of linking the sustainability of certain HeaSE ingredients, which are absent from the SU-EATABLE LIFE list, to broader but still valuable information about the sustainability of the ingredient type. Leveraging such generalized data is crucial for providing sustainability estimates while avoiding potentially inaccurate or unrelated associations.

These problems was addressed by first producing a new, simplified, version of the SU-EATABLE LIFE dataset, and then using a new matching algorithm based on semantic similarities. These steps are detailed in the following subsections.

4.2.1 Construction of the ”Revisited CSEL” Dataset

A new simplified version of the SU-EATABLE LIFE dataset was built. This new version, called *Revisited CSEL*[2], provides an unique list of ingredients and a unique list of typologies, in order to presents in each list both the information about carbon footprint and water footprint, that was previously separated in distinct lists in the original SU-EATABLE LIFE dataset. The approach adopted to merge such information is based on [58] and primarily consists of combining the data from the sheet containing CFP values with the one containing WFP values, aligning all common ingredients. However, the version used in this work introduces several improvements, resulting in the so-called ”revisited” version. Changes involved the simplifying of the concept of ”*typology*”, which is derived by merging both *typology* and *sub-typology* of the SU-EATABLE LIFE dataset, reducing the complexity of the data hierarchy.

Furthermore, the derived typology ”*VEGETABLES*” was introduced by computing an average of both the original typologies ”*VEGETABLES NOT HEATED GREENHOUSE*” and ”*VEGETABLES OPENFIELD*” in order to provide a unique CFP and WFP value related to vegetable ingredients that have no match with a specific ingredient in the Revisited CSEL dataset.

This revisited and comprehensive version of the SU-EATABLE LIFE was then leveraged in the subsequent data-matching phase.

4.2.2 Improved Ingredient Semantic Matching

In order to overcome the limitation of the substring-based approach adopted in the HeaSE dataset, a new algorithm, based on semantic similarity ad agentic automated decision was developed. The algorithm is based on the light weight model *sentence-transformers/all-mpnet-*

[2]CSEL stands for ”*Comprehensive SU-EATABLE LIFE*,” referring to the fact that it integrates both CFP and WFP information into a unified dataset.

base-v2 [60] for embeddings computation, and on GPT-4o [61] for the prompt-based agentic part. The LLM call within the algorithm is instructed using the following prompts.

Ingredient Matcher System Prompts

SYSTEM_PROMPT_ING: Given an ingredient name, choose the most relatable ingredient from the list analyzing it element by element. A list could have no relatable ingredients. Write ONLY the name of the matched ingredient as reported in quotes. If there are no relatable ingredients, return "NO_MATCH".

SYSTEM_PROMPT_TYP: Given an ingredient name, choose the most relatable typology from the list analyzing it element by element. A list could have no relatable typologies. Write ONLY the name of the matched typology as reported in quotes. If there are no relatable typologies, return "NO_MATCH".

The algorithm works as follows:

Algorithm 1 Ingredient Matching Algorithm

1: **function** MapIngredientsToDatabase($ingredient\_list$)

Require: $ingredient\_list$ is a CSV file containing ingredients to map

Require: Reference databases for ingredients and typologies along with their CFP and WFP data are available (Revisited CSEL dataset)

Require: An embedder model is available for text encoding

Require: An LLM model is configured and accessible

Ensure: returns a CSV file with mapped ingredients and their environmental footprints

2: $results \leftarrow$ empty DataFrame with columns [ingredient, cfp, wfp, mapping_type, mapped_item]

3: Initialize embedder model

4: $K \leftarrow 3$ ▷ Number of similar items to consider

```
    database_embeddings_I ← encode all ingredients from reference database
    database_embeddings_T ← encode all typologies from reference database
    for each ingredient in ingredient_list do
        embedding ← encode(ingredient)
        similar_ingredients ← GetTopKSimilar(embedding, database_embeddings_I, K)
        answer ← QueryLLM(ingredient, similar_ingredients)
        if answer ≠ "NO_MATCH" then
            Map ingredient and store environmental footprints
        else
            similar_typologies ← GetTopKSimilar(embedding, database_embeddings_T,
K)
            answer ← QueryLLM(ingredient, similar_typologies)
            if answer ≠ "NO_MATCH" then
                Map typology and store environmental footprints
            else
                Store as unmatched ingredient
            end if
        end if
        Append new row to results
    end for
    Save results to CSV file
end function

function GetTopKSimilar(embedding, embeddings_list, K)
```

Require: $embedding$ is the encoded ingredient to match

Require: $embeddings\_list$ contains reference embeddings

Require: K is the number of similar items to return

Ensure: returns K most similar items sorted by similarity

$similarity \leftarrow$ CosineSimilarity($embedding, embeddings\_list$)

$top\_k \leftarrow$ ArgsortDescending($similarity$)[1 to K]

return top_k

end function

function QUERYLLM($ingredient, similar\_items$)

Require: $ingredient$ is the item to match

Require: $similar\_items$ is the list of potential matches

Ensure: returns the best match or "NO_MATCH"

Construct query string with ingredient and similar items

$answer \leftarrow$ Call LLM with appropriate system prompt (SYSTEM_PROMPT_ING or SYSTEM_PROMPT_TYP)

return $answer$

end function

To better understand the mechanism, the following examples of ingredient matching are provided. In these examples, the *Ingredient* refers to the original ingredient without sustainability information, and the *Similar ingredients* are those that are most closely related to the original ingredient in the Revisited CSEL dataset, selected based on semantic similarity. Finally, the *Matched item* is the ingredient chosen as a reference by the LLM prompt described earlier.

Ingredient	Mapping Details
Olive Oil	Ingredient : olive oil Similar ingredients : [”OLIVE OIL”; ”COCONUT OIL”; ”OLIVE OIL REFINED”;] Matched item: OLIVE OIL ingredient cfp wfp mapping_type mapped_item olive oil 3.27 14415 ingredient OLIVE OIL
Cinnamon	Ingredient : cinnamon Similar ingredients : [”CINNAMON”; ”PEPPERMINT”; ”GARLIC”;] Matched item: CINNAMON ingredient cfp wfp mapping_type mapped_item cinnamon 0.84 15526 ingredient CINNAMON
Garlic	Ingredient : garlic Similar ingredients : [”GARLIC”; ”GARLIC POWDER”; ”ONION”;] Matched item: GARLIC ingredient cfp wfp mapping_type mapped_item garlic 0.25 589 ingredient GARLIC
Oil	Ingredient : oil Similar ingredients : [”PEANUT OIL”; ”OLIVE OIL”; ”PALM OIL”;] Matched item: PEANUT OIL ingredient cfp wfp mapping_type mapped_item oil 2.11 7529 ingredient PEANUT OIL

Similarly, an example regarding typology matching when no suitable ingredients was found is reported.

Ingredient	Mapping Details
Zest	Ingredient : zest Similar ingredients : [”LIME”; ”RAISIN”; ”MALT”] No suitable match with similar ingredients was found. Similar typologies : [”SPICES”; ”FRUIT JUICE”; ”FRUIT”] ingredient cfp wfp mapping_type mapped_item zest 2.15 748.0 typology FRUIT
Syrup	Ingredient : syrup Similar ingredients : [”CANE SUGAR”; ”BUTTERED MILK”; ”CANE MOLASSES”] No suitable match with similar ingredients was found. Similar typologies : [”SUGAR”; ”SWEETS”; ”FRUIT JUICE”] ingredient cfp wfp mapping_type mapped_item syrup 0.78 1294.5 typology SUGAR
Molasses	Ingredient : molasses Similar ingredients : [”CANE MOLASSES”; ”RAISINS”; ”MALT”] No suitable match with similar ingredients was found. Similar typologies : [”SUGAR”; ”BISCUITS”; ”DRIED FRUIT”] ingredient cfp wfp mapping_type mapped_item molasses 0.78 1294.5 typology SUGAR

Ingredient	Mapping Details
Leaf	Ingredient : leaf Similar ingredients : [”CARROT”; ”TOMATO”; ”RADISH”] No suitable match with similar ingredients was found. Similar typologies : [”VEGETABLES”; ”SEEDS”; ”FRUIT”] ingredient cfp wfp mapping_type mapped_item leaf 1.865 336.0 typology VEGETABLES

The resulting matches were then manually reviewed and annotated to filter out any mismatches or incorrect associations. The algorithm successfully mapped 6689 ingredients to their corresponding data in the *Revisited CSEL* dataset, while the remaining 2397 ingredients were labeled as NO_MATCH due to the absence of a plausible corresponding ingredient or typology.

Of the 6689 matches, approximately 722[3] were classified as incorrect, resulting in an overall accuracy of approximately 89%. The mismatched ingredients were then corrected by manually associating them with the most plausible ingredient or typology. For several of them, external data not provided by the *Revisited CSEL* dataset were used. The origin of these external data sources is discussed in the following subsection, as they were primarily utilized to assign a proper CFP value to ingredients initially labeled as NO_MATCH.

4.2.3 Integration of External Data for NO_MATCH Ingredients

Although the SU-EATABLE LIFE dataset is comprehensive and accurate, it does not include several ingredients found in the HeaSE dataset. Furthermore, its typologies fail to adequately

[3]This number is based on an initial tracked phase of verification. Additional minor inconsistencies were identified during subsequent reviews, but these corrections were made on the fly without tracking the exact number. However, the actual number of misclassified items is not significantly higher than the reported figure.

classify certain mismatched ingredients, such as alcoholic products, industrial sweets, and complex ingredients composed of multiple components.

To assign accurate CFP values to these ingredients[4], additional datasets were identified and leveraged. The datasets used include:

- **Open Food Facts** [62]: A large, user-contributed database of commercial food products. It reports sustainability data obtained from Agribalyse [63], a French project focused on food LCA.
- **CarbonCloud** [64]: A database of commercial food products managed by the company of the same name, which specializes in food LCA.
- **LiveLCA** [65]: A small-scale project curated by Ditte Juhl and Matthew The, compiling sustainability information for a broad range of food items.

These datasets were used to manually map some of the previously mentioned 722 mismatched ingredients. The values obtained from these resources were fully compatible with those provided by the SU-EATABLE LIFE dataset, as CFP data in all sources is expressed in kg CO_2 eq per kg or liter of food commodity.

At the end of this process, all ingredients whose CFP data were linked using a dataset other than SU-EATABLE LIFE were grouped and used as references to map CFP data to the 2397 NO_MATCH ingredients, leveraging a matching algorithm similar to Algorithm 1.

The results consisted of 454 ingredients still labeled as NO_MATCH, which were subsequently ignored, and 1943 mapped ingredients that were then manually reviewed. Approximately half of the generated matches were discarded and manually corrected by selecting

[4]The data integration described here focuses solely on CFP data, as, to the best of my knowledge, no comprehensive WFP datasets were available at the time of this research. However, given the objectives of the conversational agent in this work, CFP data alone are sufficient to assess the environmental impact of a recipe or food item.

a more appropriate reference from those already used to map the correct ingredients or by searching for a new suitable match in the previously listed resources.

For some ingredients, new special labels were manually assigned instead of assigning a CFP value. These labels are as follows:

- **TOO_GENERIC**: Used to mark ingredients for which the renaming procedure was too ambiguous or destructive and should be redone.
- **NOT_FOOD**: Used to mark items that are not food, such as tools used in the recipe preparation or errors in processing text, often caused by issues in the LLM streamlining process.
- **NO_DATA**: Used to mark ingredients for which no relatable data could be found anywhere.

At the end of the data-mapping procedure, the results are as follows:

- 7270 ingredients mapped using SU-EATABLE LIFE derived information.
- 1237 ingredients mapped using data from the other discussed data sources.
- 454 ingredients remain labeled as NO_MATCH.
- 68 ingredients mapped as TOO_GENERIC.
- 25 ingredients mapped as NO_DATA.
- 24 items mapped as NOT_FOOD.

4.3 Refined Sustainability Formula

By combining the ingredient composition of a recipe with the obtained CFP and WFP data, it was then possible to compute a sustainability index. This index allows the application to rank each recipe based on its environmental sustainability.

The sustainability index formula was derived from those used in [55] and [58], where each recipe's score is calculated as the sum of the scaled sustainability indices of its ingredients. The scaling process ensures that ingredients with higher environmental impact contribute more significantly to the overall recipe score. However, the original formula had an important limitation: each ingredient's sustainability index was determined by a linear combination of CFP and WFP data, making it mandatory for each ingredient to have both values. Since the sustainability impact of a recipe is influenced more by carbon emissions than by water resource use, the new scoring system was designed to handle both types of data separately. This allows for the use of ingredients that have a CFP value but no WFP value, thus ensuring that such ingredients can still contribute to the sustainability score.

The first step involved normalizing the CFP and WFP values of each ingredient to the range $[0; 1]$ in order to work with more manageable numbers later. This was achieved using the following formula:

$$\text{Normalized Value} = \frac{\text{Value} - \text{Min Value}}{\text{Max Value} - \text{Min Value}}$$

Where:

- Value represents the original CFP or WFP value of the ingredient.
- Min Value is the minimum value of the corresponding CFP or WFP dataset.
- Max Value is the maximum value of the corresponding CFP or WFP dataset.

Using these normalized values, two different recipe sustainability scores were computed, one for CFP and the other for WFP. The applied formula for calculating the CFP sustainability score is the following:

$$\text{CFP Score} = \sum_{i=1}^{N} \text{CFP}_i \cdot e^{-i}$$

Where:

- CFP_i represents the normalized CFP value of the i-th ingredient in the sorted list of ingredients.
- N is the total number of ingredients in the recipe.
- e^{-i} is the exponentially decaying factor, where i is the position of the ingredient in the sorted list (higher-ranked ingredients have higher weight).

For WFP, the same formula is applied, simply using the normalized WFP scores for each ingredient. The formula for the WFP score is:

$$\text{WFP Score} = \sum_{i=1}^{N} \text{WFP}_i \cdot e^{-i}$$

Where:

- WFP_i represents the normalized WFP value of the i-th ingredient in the sorted list of ingredients.

Finally, by using both of these sustainability indices, an overall index is computed as a linear combination of the two, scaled by two factors, α and β, which are used to tune the weight of the two components of the score. A value of $\alpha = 0.8$ and $\beta = 0.2$ was chosen in order to give more weight to the CFP, given its higher relevance in the sustainability domain. The overall sustainability score is computed as:

$$\text{Overall Sustainability Score} = \alpha \cdot \text{CFP Score} + \beta \cdot \text{WFP Score}$$

Finally, another normalization was applied in order to scale the obtained values to the range $[0; 1]$.

To conclude, the CFP and WFP coverage of each recipe was computed as the ratio of the number of ingredients for which such information is available, to the total number of ingredients in the recipe. The coverage is useful for determining the reliability of the sustainability index when using it to inform decisions about a recipe.

4.4 Recipe Grouping

To facilitate a more intuitive use of the dataset, three different groups of recipes were defined: the unsustainable recipes group labeled as 2, the sustainable group labeled as 1, and the highly sustainable group labeled as 0. These groups were identified using the following heuristics:

- If a recipe contains unsustainable ingredients, such as beef, or its sustainability score is comparable to these recipes, it is assigned to group 2.
- If a recipe contains animal-derived products but is more sustainable than those in group 2, it is assigned to group 1. Other recipes with a similar sustainability score are also placed in this group.
- If a recipe does not contain animal-derived ingredients, or its sustainability score is comparable to such recipes, it is assigned to group 0.

The thresholds for each group were determined by sorting the recipes and analyzing large subsets to identify significant changes in ingredient composition that align with the aforementioned rules. The identified thresholds are as follows:

- Recipes with a normalized sustainability score above 0.15 belong to group 2. A total of 19132 recipes belong to this group.
- Recipes with a normalized sustainability score in the range $]0.04; 0.15]$ belong to group 1. A total of 64817 recipes belong to this group.
- Recipes with a normalized sustainability score below 0.04 belong to group 0. A total of 9799 recipes belong to this group.

The group of a recipe in the dataset is stored in the field *sustainability_label* of the recipe dataset.

4.5 Allergy Data Integration

To provide recommendations that account for user allergies, the need to map allergens for each ingredient became apparent. This task was undertaken by Scavo Beatrice and Valentino Federico as exam related task connected to this work. To enrich the dataset with this information, they referred to the official list of food allergens provided by the European Union [66] and used this list as a reference to instruct the GPT-4o model, interacting with it directly through the ChatGPT console using the following prompts[5].

[5]The prompts are reported in Italian, as it was the language actually used by the authors.

ChatGPT Allergens Mapping Prompt pt.1

Modifica il documento xlsx, dove è contenuta una lista di ingredienti. Per ogni ingrediente (nome nella colonna B) fai una ricerca su internet definendo:

- Se l'ingrediente è composto da più sotto-ingredienti;

- Se l'ingrediente in sé (o i suoi sotto-ingredienti) contengono gli allergeni, definiti dalle macro-categorie di allergeni nel pdf in allegato.

Prima di fare queste operazioni, traduci il contenuto del pdf in inglese. I risultati della ricerca, mettili nelle colonne K ed L.

Subsequently, further refinements were made to the identified macro-categories using the following prompt structure:

ChatGPT Allergens Mapping Prompt pt.2

Continua a raffinare il file, prendendo il nome nella colonna B ed E. Cerca cosa sono e da cosa sono composti. Se sono o sono composti da allergeni, segnalalo.

Per esempio, PASTA è fatta da farina/grano quindi il suo allergene è glutine.

These prompts leveraged the capabilities of the ChatGPT console to process .xlsx files directly and perform web searches, enriching the ingredient list with corresponding allergens based on the reference .pdf file where applicable. The following allergen categories were considered:

- Cereals containing gluten (wheat, rye, barley, oats, spelt, kamut, and their hybridized strains)
- Crustaceans and crustacean-based products
- Eggs and egg-based products
- Fish and fish-based products
- Peanuts and peanut-based products

- Soy and soy-based products
- Milk and milk-based products (including lactose)
- Tree nuts (almonds, hazelnuts, walnuts, cashews, pecans, Brazil nuts, pistachios, macadamia nuts)
- Celery and celery-based products
- Mustard and mustard-based products
- Sesame seeds and sesame-based products
- Sulfur dioxide and sulfites (in concentrations higher than 10 mg/kg or 10 mg/L)
- Lupin and lupin-based products
- Mollusks and mollusk-based products

The mapping was then manually reviewed, similar to the CFP and WFP verification phase, to ensure accuracy, with some minor manual adjustments.

Upon completion of the mapping phase, the resulting data was used to populate two new fields in the ingredients dataset: *allergen*, a boolean field indicating whether an ingredient is an allergen, and *allergen_type*, which specifies the associated allergen using the aforementioned allergens list.

Subsequently, for each recipe, the presence of allergens was checked by iterating over the list of simplified ingredients. The *allergens* field was populated with the list of allergens associated with the recipe's ingredients. Finally, 14 new tags were introduced to indicate the absence of specific allergens. If a recipe did not contain any allergens, all 14 allergen-free tags were assigned. The new tags are as follows:

- gluten-free
- crustacean-free
- egg-free
- fish-free
- peanut-free

- soy-free
- lactose-free
- nut-free
- celery-free
- mustard-free
- sesame-free
- sulfite-free
- lupin-free
- mollusk-free

4.6 Disabling Oversimplified Recipes

The process of ingredient streamlining applied to the original HUMMUS Dataset ingredients list was unable to effectively handle some minor discrepancies in the original dataset. These discrepancies, which are common in large datasets, led to some recipes having a significantly reduced number of ingredients after the simplification process. As a result, these recipes were scored as not impactful, even though the original ingredient list referred to non-sustainable ingredients.

To avoid using and recommending recipes with incomplete data, a simple data cleaning process was performed. This process involved identifying recipes where the difference in the number of ingredients between the original Hummus ingredients list and the simplified list was equal or greater than 4. This analysis led to the identification of 440 recipes that had too few ingredients compared to the original list. These recipes were labeled as disabled using a new boolean field created for this purpose.

4.7 Properties of the HeaSEv2 Dataset

The HeaSEv2 dataset, updated using the previously described technique, became a useful and concrete way to address sustainability-related tasks, using reliable data and a relatively compact set of ingredients. The dataset results in a number of 75129 recipes whose both CFP and WFP coverage were equal to or higher than 70%, an empirically chosen threshold useful for selecting which recipes are backed with sufficient reliable information when proceeding with the work. This means that about the 80% of the dataset is backed up with enough information about the sustainability of its recipes and their involved ingredients.

In order to grasp the quality of the proposed approach, the list of the top 10 least sustainable recipes and the top 10 most sustainable recipes, both having the coverage values equal to or above the threshold, are reported:

Recipe	**Normalized Sustainability Score**
Amy Truong Zucchini Low-Carb Lasagna	1.000
Baked ”ziti” W/ Tofu Shirataki	0.983
Buffalo Burgers	0.969
The Gobsmacked Leaning Tower of Buffalo Burger	0.965
Hiller’s Buffalo/Venison Chili	0.949
Buffalo Skewers	0.942
The Italian Buffalo	0.929
Spanish Bean Soup - Fiery!	0.782
Al Kabsa - Traditional Saudi Rice (& Meat) Dish	0.755
”Secret Ingredient” Beef Stew	0.634

Table 4.3: Worst 10 recipes from the sustainability score perspective. Despite their name, the first two recipes use buffalo meat, the most unsustainable ingredient in the dataset, making them the overall most unsustainable recipes.

Recipe	Sustainability Score
Grape Granita - Sorbet - No Cooking Required - One Ingredient	0.00055
Easy Granita	0.00055
Fried Mashed Potatoes	0.00069
Baked Potato-Onion Wrap-Ups	0.00069
Apple and Carrots Juice	0.00091
Morning Bright Eye	0.00091
Apple Carrot Juice	0.00091
Sliced Mango	0.00094
Easy Applesauce	0.00100
Carrot Cucumber Juice	0.00106

Table 4.4: Top 10 recipes from the sustainability score perspective.

For completeness, the best and worst recipes from both the CFP and WFP perspectives are also reported.

Recipe	CFP Score
Amy Truong Zucchini Low-Carb Lasagna	1.06372
Baked ”ziti” W/ Tofu Shirataki	1.06351
Buffalo Burgers	1.04906
Buffalo Burgers, Won’t You Come out Tonight?	1.04887
The Gobsmacked Leaning Tower of Buffalo Burger	1.04525
Hiller’s Buffalo/Venison Chili	1.02895
Bison / Buffalo Pot Roast	1.02487
Grilled Jalapeno Buffalo Burgers	1.01450
Buffalo Skewers	1.01012
The Italian Buffalo	1.00560

Table 4.5: Worst 10 recipes from the carbon footprint score perspective.

Recipe	CFP Score
Sliced Mango	0.00393
Grape Granita - Sorbet - No Cooking Required - One Ingredient	0.00393
Easy Granita	0.00393
Baked Potato-Onion Wrap-Ups	0.00407
Fried Mashed Potatoes	0.00407
Apple and Carrots Juice	0.00429
Morning Bright Eye	0.00429
Apple Carrot Juice	0.00429
Easy Applesauce	0.00433
Sweet Potato/Carrot Fat Substitute for Recipe	0.00454

Table 4.6: Best 10 recipes from the carbon footprint score perspective.

Recipe	WFP Score
Cameline Sauce	1.50680
Couscous Pilaf With Saffron Cream	1.37205
Saffron-Scented Halibut with Spinach, Zucchini and Tomato	1.37117
Oriental Rice Pudding With Rhubarb	1.07139
Indian Moghul Chicken Biryani	1.04054
Al Kabsa - Traditional Saudi Rice (& Chicken) Dish	1.04002
Taste of Heaven - Mushroom and Carrots Biryani	1.04000
Chai Tea from Scratch	1.03998
Curried Chicken Moghlai	1.03981
Moroccan B'stella	1.03878

Table 4.7: Worst 10 recipes from the water footprint score perspective.

Recipe	WFP Score
Tomato Beer	0.00023
Black and Berry (Beer)	0.00029
Beer Punch	0.00031
Short Cut Spinach and Cheese Souffle Stuffed Tomatoes	0.00042
Pine Melon Crush	0.00044
Pineapple Carrot Juice	0.00044
Stewed Tomatoes and Zucchini	0.00048
Jazz up the Cranberry Sauce	0.00051
Fantabulous Roasted Bell Peppers	0.00051
Basic Salad Mix (Salad Spinner)	0.00054

Table 4.8: Best 10 recipes from the water footprint score perspective.

Furthermore, other interesting properties are inherited from the original HeaSE dataset. The most interesting ones are the presence of nutritional values for each recipe, the presence of descriptive text and instructions about the preparation of the recipe, and the presence of property tags that help filter the recipes when searching for something that respects certain constraints.

The most useful tags are the following:

- *main-dish*: This tag identifies recipes that are typically served as the main course of a meal.
- *dinner*: This tag identifies recipes that are usually consumed as dinner, often used in conjunction with the *main-dish* tag.
- *lunch*: This tag identifies recipes typically consumed as lunch, often used in conjunction with the *main-dish* tag.
- *breakfast*: This tag identifies recipes typically consumed as breakfast.
- *snack*: This tag identifies recipes usually consumed as a snack.

- *15-minutes-or-less*: This tag identifies recipes that can be prepared in about 15 minutes.
- *30-minutes-or-less*: This tag identifies recipes that can be prepared in about 30 minutes.
- *kosher*: This tag identifies recipes that adhere to Hebrew dietary restrictions.
- *vegan*: This tag identifies recipes that are vegan.
- *vegetarian*: This tag identifies recipes that are vegetarian.
- All the 14 new *allergen-free* tags: The 14 newly introduced tags help identify recipes that are free from specific allergens, making it easier for users with dietary restrictions to find suitable options.

All these tags can be used to easily filter the dataset in a simple and direct manner, enabling users to select recipes that align with their preferences and requirements across various aspects. Additionally, new tags can be easily added, further enhancing the dataset's expressiveness and allowing for more refined searches.

For a comprehensive review of the tags already present in the initial version of the dataset, please refer to [56].

4.7.1 Dataset Recap

This subsection provides a concise summary of the key statistics previously presented about the HeaSEv2 dataset. It includes aggregated data on ingredients and recipes, emphasizing sustainability metric coverage, allergen information, and relevant categorical tags.

Ingredients:

- Total number of ingredients: 9086
- Number of ingredients with CFP data: 8509
- Number of ingredients with WFP data: 6956
- Number of ingredients with both CFP and WFP data: 6956

- Number of ingredients without both CFP and WFP data (NO_MATCH or NO_DATA ingredients): 479
- Number of ingredients reporting an allergen: 2747
- Number of items in the dataset that are not actually ingredients (TOO_GENERIC or NOT_FOOD removable entries): 96

Recipes:

- Total number of recipes: 93748
- Number of recipes with a CFP coverage greater than 70%: 93292
- Number of recipes with a WFP coverage greater than 70%: 75129
- Number of recipes with both a CFP and WFP coverage greater than 70%: 75129
- Number of recipes tagged as *main-dish*: 17305
- Number of recipes tagged as both *main-dish* and *lunch*: 1687
- Number of recipes tagged as both *main-dish* and *dinner*: 2717
- Number of recipes tagged as *breakfast*: 4941
- Number of recipes tagged as *snack*: 1983
- Number of recipes tagged as *vegetarian*: 11173
- Number of recipes tagged as *vegan*: 3043
- Number of recipes tagged as *kosher*: 1371
- Number of recipes tagged as *15-minutes-or-less*: 14049
- Number of recipes tagged as *30-minutes-or-less*: 16223
- Number of healthy recipes (with healthiness_label equal to 0): 28148
- Number of recipes reporting at least one allergen: 61539
- Number of disabled recipes: 440

4.8 HeaSEv2 Dataset Structure

The final version of the HeaSEv2 dataset has the following structure[6].

Recipe:

- **title** (String): Recipe title.
- **sustainability_score** (Float): Sustainability Score in range $[0; 1]$.
- **sustainability_label** (Integer): Allows identifying sustainable recipes.
- **recipe_id** (Integer): A unique identifier for each recipe.
- **description** (String): Recipe description.
- **author_id** (Integer): Identifier of the recipe's author on Food.com.
- **duration** (Integer): Time required to prepare the recipe (in minutes).
- **directions** (String): Step-by-step preparation instructions.
- **ingredients** (String): List of ingredients used in the recipe (after recipe simplifying process).
- **serves** (Integer): Number of servings the recipe yields.
- **last_changed_date** (Date): Last modification date of the recipe.
- **food_kg_locator** (String): Knowledge graph locator for the recipe.
- **recipe_url** (String): URL linking to the original recipe source.
- **tags** (String): Categorization tags associated with each recipe.
- **new_recipe_id** (Integer): Updated identifier for the recipe.
- **new_author_id** (Integer): Updated identifier for the recipe's author.

[6]The data-type in brackets specifies the appropriate data type to use when loading the CSV files into a database or an in-memory data structure. The underlined field names are the one that was modified or added in the v2 version.

- **average_rating** (Float): Average user rating of the recipe.
- **number_of_ratings** (Integer): Number of ratings received.
- **servingsPerRecipe** (Integer): Number of servings per recipe.
- **servingSize [g]** (Float): Weight of a single serving in grams.
- **calories [cal]** (Float): The total calorie content of the recipe.
- **caloriesFromFat [cal]** (Float): The amount of calories derived from fat.
- **totalFat [g]** (Float): Total fat content in grams.
- **saturatedFat [g]** (Float): Amount of saturated fat in grams.
- **cholesterol [mg]** (Float): Cholesterol content in milligrams.
- **sodium [mg]** (Float): Sodium content in milligrams.
- **totalCarbohydrate [g]** (Float): Total carbohydrates in grams.
- **dietaryFiber [g]** (Float): Dietary fiber content in grams.
- **sugars [g]** (Float): Total sugars in grams.
- **protein [g]** (Float): Protein content in grams.
- **direction_size** (Integer): Number of steps in the recipe instructions.
- **ingredients_sizes** (Integer): Number of ingredients used.
- **who_score** (Float): A healthiness score based on WHO methodology, ranging from 0 to 14, with 14 being the best.
- **fsa_score** (Float): A healthiness score based on the UK Food Standards Agency (FSA) nutrient profiling system, ranging from 0 to 8, with 8 being the best.
- **nutri_score** (Float): A nutritional score for each recipe, graded from A (best) to E (worst).
- **ingredient_food_kg_urls** (String): Knowledge graph URLs for each ingredient.
- **ingredient_food_kg_names** (String): List of ingredient names as per knowledge graph.

- **healthiness_label** (Integer): A categorical label indicating recipe healthiness ranging from 0 to 2. The lower the value the better the healthiness is.
- **percentage_covered_cfp** (Float): Percentage of CFP-based sustainability coverage.
- **percentage_covered_wfp** (Float): Percentage of WFP-based sustainability coverage.
- **cfp_sustainability** (Float): Sustainability score based only on CFP values.
- **wfp_sustainability** (Float): Sustainability score based only on WFP values.
- **overall_sustainability** (Float): Overall sustainability score for the recipe combining CFP and WFP data.
- **original_hummus_db_ingredients** (String): Original ingredient representation from the Hummus database.
- **simplified_ingredients** (Strings): Simplified ingredient names (without unit of measure and respective values) used for processing.
- **allergens** (String): List of allergens present in the recipe.
- **disabled** (Boolean): Indicates whether the recipe is disabled or active.

Ingredients:

- **ingredient** (String): The name of the ingredient.
- **cfp** (Float): The CFP (carbon footprint) value for the ingredient.
- **wfp** (Float): The WFP (water footprint) value for the ingredient.
- **mapping_type** (String): The type of mapping applied to the ingredient (ingredient or typology).
- **mapped_item** (String): The mapped item associated with the ingredient.
- **data_origin** (String): The source from where the ingredient data was obtained. No value imply the Revisited CSEL dataset as source.

- **cfp_normalized** (Float): The normalized CFP value for the ingredient.
- **wfp_normalized** (Float): The normalized WFP value for the ingredient.
- **allergen** (Boolean): Indicates whether the ingredient is an allergen (true/false).
- **allergen_type** (String): The type of allergen the ingredient represents, if applicable.

Chapter 5

E-Mealio Conversational Agent

Figure 5.1: The E-Mealio logo.

The E-Mealio conversational agent is the core focus of this work. As previously discussed in Chapter 1, it is designed as a Telegram bot that provides trustworthy recommendations for sustainable meals. This chapter outlines the agent's key functionalities, its overall architecture, and the technical choices that shape its implementation.

5.1 E-Mealio Functionalities

The foundational step in the development of the E-Mealio conversational agent was to identify a coherent set of functionalities that would embody the overall concept of a sustainable meals expert, as discussed in Section 1.3.

The analysis identified nine core functionalities, categorized into three groups: Registering User Functionalities, Main Functionalities, and Asynchronous Functionalities.

5.1.1 Registering User Functionalities

The following functionality is meant to be used during the user's first interaction to register their data and ensure a personalized experience moving forward.

1: User Data Acquisition at First Use Upon the first interaction, the agent collects essential user information, including dietary restrictions, allergies, and personal details such as name, surname, date of birth, and nationality. While dietary restrictions and allergies are actively used to personalize recommendations, personal details are mandatory solely for registration purposes and are not currently utilized for recommendation. However, they are included to potentially analyze user behavioral patterns and may be leveraged in the future to refine recommendation techniques. Lastly, the agent requests permission to send reminders if the user remains inactive for two days.

The identified dietary restrictions currently available are the following: Vegan, Vegetarian, and Kosher.

These restrictions were chosen due to the current availability of recipes in the HeaSEV2 dataset that include such tags. Further restrictions can be easily added, provided there is an update to the tags of each recipe, including the one that represents the compatibility of that

restriction with the recipe.

The managed allergies are the following: Gluten, Crustaceans, Egg, Fish, Peanut, Soy, Lactose, Nuts, Celery, Mustard, Sesame, Sulfites, Lupin, and Mollusks. These allergies are the 14 recognized by the EU in the alimentary sector and are addressed by the HeaSEV2 dataset.

5.1.2 Main Functionalities

The idle agent waits for a user request in a state called ”the hub,” which is interpreted and redirected to the appropriate functionality. If this state is invoked for the first time, or when the user sends a message unrelated to any functionality, the agent will send a presentation message. This message informs the user about the agent's capabilities and provides a set of sample phrases that can be used to trigger the various functionalities. The following tasks can be activated through this hub:

2: Recipe Recommendation for Daily Meals The agent provides recommendations for meals, including breakfast, lunch, dinner, and snacks, following these constraints:

- **Mandatory implicit constraints:**
  - Dietary restrictions specified during profile creation.
  - Allergies, if provided during profile setup.
- **Optional implicit constraints:**
  - Recipes already consumed or rejected in the past week are excluded.
- **Optional explicit constraints:**
  - Preference for recipes labeled as highly healthy.
  - Preference for recipes with medium or short preparation times.

If no results are found due to excessive constraints, the agent will automatically remove some optional constraints and explain the reason for their removal.

The user can also specify a list of preferred and undesired ingredients. Rather than acting as a strict filter, this list reorders the initially retrieved result set based on the user's preferences. If no specific ingredient preferences are provided, the agent suggests the closest recipe to the vector representing the user's taste for a given meal type (e.g. breakfast, lunch, dinner, or break)[1]. If the taste vector is not yet available, the agent recommends the most sustainable recipes among those retrieved using the specified filters.

Once a recommendation is provided, the user can discuss, modify, accept, or reject it. If the conversation shifts to an unrelated topic, the agent resets itself and returns to the hub. Eventually accepted or rejected recipes are saved in the user history.

3: Recipe Improvement The user can request an improved version of a given recipe by providing its ingredients and, optionally, its name. If the name is not provided, it is automatically derived from the ingredient list. If the user's request lacks essential details, such as a proper ingredient list, the agent prompts the user to provide the necessary information before proceeding.

Once the information is provided, the system follows these steps:

- Using the ingredient list provided by the user, a sustainability score is computed following the formula described in Section 4.3. This recipe is considered the "base" recipe.
- The recommendation system retrieves recipes with a better sustainability score than the base recipe. A semantic similarity search is then performed to identify the closest match among the more sustainable recipes, leveraging both the ingredient list and the recipe

[1]From now on, the term *meal type* will always refer to the following four categories: breakfast, lunch, dinner, or break

name.

- The original and target recipes are compared in a prompt, instructing the LLM to suggest ingredient substitutions for improved sustainability.

As for the recipe recommendation, the user can discuss, modify, accept, or reject the suggestion. If the conversation deviates significantly, the agent resets and returns to the hub. Accepted or rejected recipes are saved in the user history for future reference.

4: User Profile Recap and Update The agent summarizes the user's profile and asks if any changes are needed. If so, an update process similar to the initial data collection (task 1) is initiated.

5: Weekly Food Diary Recap The agent analyzes the user's food consumption over the past week based on the recipes they have accepted or confirmed as eaten. Along with this analysis, it provides general suggestions on how to make more sustainable choices in the future, encouraging the user toward a more eco-friendly lifestyle. Additionally, the user can engage in a discussion about the provided insights.

6: Sustainability Expert The agent can answer questions regarding:

- Specific ingredients, recipes (by name or ingredient list), or broader topics (e.g., carbon footprint, water consumption).
- Comparative analysis when multiple ingredients or recipes are involved.

The user can further discuss the response, but if the topic shifts drastically or they indicate understanding, the conversation ends and returns to the hub.

7: Food Diary Users can log their consumed meals by providing the meal type, the ingredients involved, and the recipe name. If the recipe name is not provided, it can be automatically derived from the ingredients list. If neither the meal type nor the ingredient list is provided, the agent explicitly requests these details. Custom recipes are saved in the database but do not have an ID, meaning they do not contribute to the exclusion filter for previously consumed recipes in the recommendation system. These logged recipes are later utilized in functionality 5 to analyze and provide insights into the user's dietary sustainability.

5.1.3 Asynchronous Functionalities

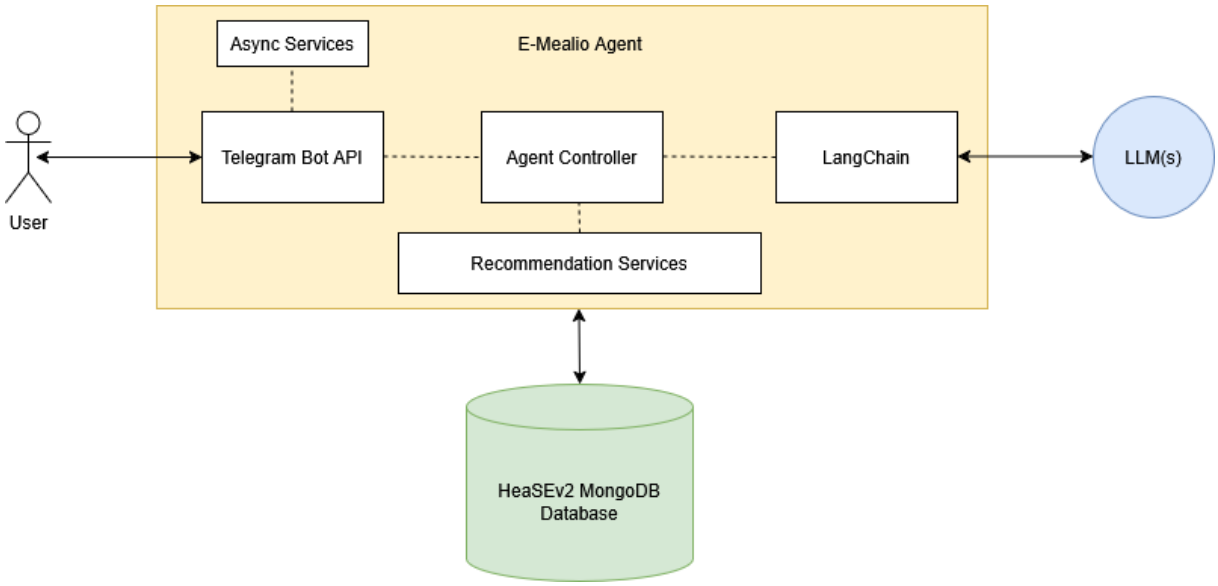
These functionalities do not involve actions that the user can directly invoke by interacting with the agent, but rather actions that the agent performs on a schedule, based on certain conditions.

Reminder Sending The agent tracks all user interactions and records the time of the last interaction. Every day at 12:00, if two days have passed since the last interaction with a user and the reminders are enabled, the bot will automatically send a reminder to nudge the user to engage in a new conversation.

User Taste Vector Computation To personalize the agent's recommendations, each month, the system retrieves the user's food consumption history from the previous month. A random sample of ten recipes is selected for each meal type, and a taste vector is computed using the ingredient embeddings. This vector captures the user's preferences for specific meal types. The resulting taste vectors are stored for future use, transforming the agent into a content-based recommender system that delivers more customized meal suggestions based on the user's tastes.

5.2 E-Mealio Agent Software Architecture

The previously presented functionalities needed to be orchestrated and integrated seamlessly to enable users to interact naturally with the agent while also ensuring efficient extraction of information from the recipe and ingredient dataset. This requirement shaped the project into the following architecture:



The E-Mealio Agent is composed of five main components:

- **Agent Controller** – Manages the conversational flow using a finite state machine and a collection of dedicated LLM prompts. It's the core mechanism of the E-Mealio Agent.
- **LangChain Service** – Responsible for invoking LLM APIs by providing the user request, prompt, and, when necessary, contextual memory of the conversation. Additionally, the generated output is processed to extract relevant information, including the agent's next state and any supplementary details required for executing subsequent tasks.
- **Recommendation Service** – Implements algorithms to extract the most suitable recipe based on the user's request.

- **Telegram Bot API** – Allows the agent to interact with users through the Telegram application.
- **Async Services** – Managed by means of a subcomponent of the Telegram Bot API, they are responsible for handling asynchronous functionalities (see Subsection 5.1.3).

The **HeaSEv2 MongoDB Database** is a MongoDB instance used to store and manage data from the HeaSEv2 dataset. Additionally, it contains collections for internal agent purposes, such as user representation and log tracking.

The **LLMs** in the diagram represent APIs from various LLM providers, such as OpenAI or Anthropic. Since LangChain handles these integrations, any compatible LLM can be plugged in. However, for this project, GPT-4o was primarily used.

The agent is implemented in **Python 3**, leveraging the rich ecosystem of libraries available for Natural Language Processing (NLP). Key dependencies include the aforementioned LangChain, as well as numpy and scikit-learn.

5.2.1 The LangChain Library and Motivation for Its Usage

As described in Section 3.3, various strategies have emerged to leverage LLMs as tools for developing structured, LLM-powered software. In this rapidly growing field, LangChain [67] has become one of the most widely recognized choices for building such applications.

Developed in 2022 by Harrison Chase, LangChain enables seamless integration with multiple LLM models, providing frameworks that simplify the implementation of state-of-the-art techniques for prompt manipulation and orchestration. These include various prompting strategies discussed earlier, memory mechanisms, and agents, intended as specialized prompts used to determine the appropriate sequence of instructions for completing a task.

In the context of this work, LangChain was primarily used to make E-Mealio an LLM-agnostic agent, allowing for easy adaptation to different underlying models through its stan-

dardized API. Another extensively utilized feature of LangChain is its *Memory* module, which enables the system to track user interactions and provide context-aware responses.

Other key functionalities of LangChain, such as its built-in RAG module and LangGraph (designed for implementing structured workflows that determine when and how prompts should be used), were intentionally not adopted in this project. Instead, custom implementations were developed to better suit the specific requirements of E-Mealio, avoiding excessive reliance on external decisions that could have significantly impacted its design.

5.2.2 Source Code Availability and Package Structure

The source code of the project is openly available on GitHub at the following repository: `https://github.com/aiacovazzi/Sustainable-Food-Chat-Agent`

The source code will not be reported directly. When a general description of an algorithm is necessary, a pseudocode representation will be provided. However, for each main component of the agent, a reference to the source code file that implements it will be included.

To facilitate the understanding of source code references appearing later in this document, a brief explanation of the project's package structure is provided.

```
projectRoot
├── dto
│   ├── Ingredient.py
│   ├── Log.py
│   ├── Recipe.py
│   ├── Response.py
│   ├── User.py
│   └── UserHistory.py
└── persistence
```

```
│   ├── IngredientPersistence.py
│   ├── LogPersistence.py
│   ├── MongoConnectionManager.py
│   ├── UserHistoryPersistence.py
│   └── UserPersistence.py
├── service
│   ├── async
│   │   └── ComputeMonthlyUserTasteService.py
│   ├── bot
│   │   ├── EmbedderService.py
│   │   ├── LangChainService.py
│   │   └── LogService.py
│   ├── domain
│   │   ├── FoodHistoryService.py
│   │   ├── IngredientService.py
│   │   ├── RecipeService.py
│   │   └── UserDataService.py
│   ├── ExpertRecipeService.py
│   ├── ImproveRecipeService.py
│   └── SuggestRecipeService.py
├── ChatbotController.py
├── Contants.py
├── TelegramBot.py
├── test.py
└── Utils.py
```

The *dto* package contains all Data Transfer Object (DTO) classes used to map database entities onto objects that can be used in the code. DTO classes include methods for JSON serialization and deserialization, enabling seamless communication with MongoDB through the *jsonpickle* library.

The *persistence* package contains all classes responsible for direct interaction with the database, performing CRUD operations for each entity.

The *service* package contains the agent's services, which perform complex data processing to implement the agent's functionalities. This package is further divided into three subpackages:

- *bot*: Contains general utility functions.
- *async*: Manages asynchronous functionalities.
- *domain*: Includes domain-specific utility functions that require complex orchestration of the functions in the *persistence* package.

The *projectRoot* also contains files that implement the Telegram bot, the controller, the constants file (which includes all prompts and token definitions), a utilities file (used for general-purpose functions like escaping curly braces and conversions), and the test file containing unit tests to ensure the correct behavior of the agent after edits or refactoring.

The following subsections will explore each component of the E-Mealio Agent in detail, mentioning also the Python files that implement these components.

5.2.3 HeaSEv2 MongoDB Database and the Data Layer

The key property of the E-Mealio agent is its integration with the extensive and well-structured HeaSEv2 dataset. To ensure efficient access and manipulation, the dataset was imported into a MongoDB database. This choice was driven by four main factors:

1. **Flexible Data Representation**: MongoDB allows for a schema-less representation of entities, avoiding the rigid constraints of relational databases. This is particularly beneficial for recipes, which may have varying numbers of metadata.

2. **JSON-Like Structure for LLM Compatibility**: Data in MongoDB is stored in BSON (a binary JSON format), making it naturally compatible with modern LLM models and RAG techniques.

3. **Potential Integration with MongoDB Atlas**: The availability of the Atlas platform, a SaaS solution for hosting MongoDB databases that also offers advanced semantic search algorithms. While Atlas was not utilized in the current version of the project in favor of a more customized recipe semantic search implementation, future versions could leverage its capabilities to enhance performance and scalability.

4. **Scalability for Future Production Environments**: MongoDB is designed for horizontal scalability, which allows efficient handling of increasing data volumes and user interactions. Although this scalability is not yet leveraged in the current implementation, it remains a key advantage for potential deployment in a production environment.

5.2.3.1 HeaSEv2 MongoDB Database Structure

The implemented database, named **emealio_food_db**, contains the two parts of the HeaSEv2 dataset described in Section 4.8, structured as follows.

The **ingredients** collection provides all the fields previously described, with the addition of a new field:

- **ingredient_embedding** (Array of Floats): Stores the embedding of the ingredient name (the **ingredient** field).

The **recipe** collection provides all the fields previously described, with the addition of two new fields:

- **title_embedding** (Array of Floats): Stores the embedding of the recipe title.
- **ingredients_embedding** (Array of Floats): Stores the sum of the embeddings for each ingredient involved in the recipe composition, using the **simplified_ingredients** field as a reference.

The embeddings were computed using the **Alibaba-NLP/gte-large-en-v1.5** model. This choice was driven by its moderate memory usage of 1.62 GB, its model dimension with 434 million parameters, and its average score of 81.43 on Semantic Textual Similarity (STS)[2], as reported by the legacy MTEB benchmark page.

Some empirical tests were performed before adopting this embedder. However, a more thorough comparison of which embedding model maximizes the agent's performance remains an open task for future improvements.

To enhance query execution performance, both in the **ingredient** collection and in the **recipe** collection, a set of indexes has been defined.

Indexes for the **ingredient** collection:

- An ascending index on the *ingredient* field (the ingredient name).

Indexes for the **recipe** collection:

- An ascending index on the **recipe_id** field.
- An ascending index on the **title** field.
- A compound index on the following fields:

[2]For this task, the metric is computed as the Spearman correlation of the cosine similarity with the ground truth similarity label.

- **sustainability_label** (ascending index).
- **disabled** (ascending index).
- **percentage_covered_cfp** (descending index).
- **percentage_covered_wfp** (descending index).

This compound index supports the main query for recipe extraction used in the recommendation service.

The **emealio_food_db** also provides ad hoc entities unrelated to the HeaSEv2 dataset, allowing the storage of internal data such as users, their food consumption history, and system logs, those collection are resumed by the following tables:

users:

- **_id** (ObjectId): The MongDB identifier for the record (indexed in ascending order).
- **username** (String): The Telegram username of the user.
- **id** (String): the telegram identifier for the user (indexed in ascending order).
- **name** (String): The first name of the user.
- **surname** (String): The surname of the user.
- **dateOfBirth** (String): The date of birth of the user in string format.
- **nation** (String): The nationality of the user.
- **allergies** (Array of Strings): A list of allergies that the user has.
- **restrictions** (Array of Strings): A list of dietary restrictions for the user.
- **reminder** (Boolean): Indicates whether the user has a reminder set.
- **tastes** (Array of Float Arrays): A list of embeddings representing taste preferences of the user for each meal type (breakfast, lunch, break, dinner).

users_food_history:

- **_id** (ObjectId): The MongDB identifier for the record (indexed in ascending order).
- **userId** (String): The Telegram user identifier for the user associated with the recipe (indexed in ascending order).
- **recipeId** (Integer): The recipe identifier (recipe_id) in the database.
- **recipe** (Object): Contains the details of the stored recipe.
  - **name** (String): The name of the recipe.
  - **id** (Integer): The recipe identifier (recipe_id) in the database.
  - **ingredients** (Array of Objects): A list of ingredients used in the recipe. Each ingredient has the following fields:
    - **name** (String): The name of the ingredient.
    - **cfp** (Float): Carbon Footprint (CFP) value associated with the ingredient.
    - **wfp** (Float): Water Footprint (WFP) value associated with the ingredient.
  - **sustainabilityScore** (Float): A score representing the overall sustainability of the recipe.
  - **instructions** (String): URL that links to the recipe's preparation instructions.
  - **description** (String): A brief description or context about the recipe.
  - **removedConstraints** (Array of Strings): A list of constraints that were removed from the query in order to extract the recipe.
  - **mealType** (String): The type of meal (Breakfast, Lunch, Break or Dinner).
- **date** (String): The date and time of when the recipe was saved.
- **status** (String): The current status of the recipe, indicating whether it was derived by an agent suggestion (accepted or declined) or asserted by the user.

logs:

- **_id** (ObjectId): The MongDB identifier for the record (indexed in ascending order).
- **logContent** (String): The content of the log entry, typically describing an action or event.
- **date** (Timestamp): The date and time when the log entry was created.
- **userId** (String): The identifier of the user associated with the log entry (indexed).
- **agent** (String): The agent that triggered or created the log entry (System are software utilities log, User are the sentence written by the user, Agent are the LLM's produced response).

5.2.3.2 The E-Mealio's DTOs

On the agent implementation side, under the package *projectRoot/dto*, are located the classes that manage the internal representation of the previously described collections.

It is important to note that the classes **Ingredient** and **Recipe**, located in the files *projectRoot/dto/Ingredient.py* and *projectRoot/dto/Recipe.py*, do not include all the fields present in the MongoDB collection. Instead, these classes only contain the fields that are relevant to the internal representation of the corresponding concepts within the agent.

The structure of these classes is as follows:

Ingredient Class Fields:

- **name**: The name of the ingredient.
- **cfp**: The CFP (carbon footprint) value for the ingredient.
- **wfp**: The WFP (water footprint) value for the ingredient.

Recipe Class Fields:

- **name**: The name of the recipe.
- **id**: A unique identifier for the recipe.
- **ingredients**: A list of **Ingredient** objects associated with the recipe.
- **sustainabilityScore**: The sustainability score of the recipe.
- **instructions**: URL that links to the recipe's preparation instructions.
- **description**: A textual description of the recipe.
- **removedConstraints**: A list of constraints that were removed from the query in order to extract the recipe
- **mealType**: The type of meal (e.g., "Dinner", "Lunch").

These classes are the ones actually used to represent recipes in the **users_food_history** collection.

The other provided DTOs are **Log**, located in *projectRoot/dto/Log.py*, **User**, located in *projectRoot/dto/User.py*, and **UserHistory**, located in *projectRoot/dto/UserHistory.py*, which essentially reflect the structure already described in the MongoDB collection analysis.

There is one last DTO class named **Response**, located in *projectRoot/dto/Response.py*, that contains the following fields:

Response Class Fields:

- **answer**: The string that must be displayed to the user.
- **action**: A token controlling the agent's behavior. It is derived by parsing the LLM response and extracting the parts of the response that match a regex that captures the string *TOKEN* followed by a decimal number.
- **info**: Stores additional information between agent phases. This field is derived by parsing the LLM response and extracting the content enclosed in curly brackets.

- **memory**: An object of type *ChatMessageHistory* that stores the conversation history.
- **modifiedPrompt**: Stores the modified prompt for generating the answer. This is useful when we want to generate a response using a message that differs from the last message sent by the user.

This class is used to represent the response provided by the LangChain Service that will be described later on.

Every DTO class is provided with methods that allow converting an object into JSON and vice-versa by means of the *jsonpickle* library. This library is used to serialize Python objects into JSON format and deserialize JSON back into Python objects, handling more complex data types that the standard JSON module might not be able to serialize directly.

5.2.3.3 The E-Mealio's Persistence

To enable the agent to retrieve and modify the MongoDB collection, the *projectRoot/persistence* package contains classes that manage the previously described entities using CRUD[3] operations. These classes support standard Create, Read, Update, and Delete actions on the MongoDB collection, allowing the agent to interact with the stored data efficiently.

For the **Recipe** and **Ingredient** entities, additional functionality is provided to retrieve entities based on semantic similarity. This is achieved by leveraging the embedding fields of the name of each entity. By utilizing cosine similarity, it is possible to search for the most similar recipe or ingredient given a search string. This allows for more intuitive and flexible retrieval of entities based on content similarity, as opposed to relying solely on exact matches.

The persistence classes are frequently accessed through the domain service classes located in *projectRoot/service/domain*. These classes not only encapsulate the persistence layer but

[3]Not all entities have a full set of CRUD operations. For ingredients and recipes, it is not possible to add or modify the original data.

also provide more complex workflows, such as data assembly and processing, that support the operations carried out in the agent's two main services (i.e., recipe suggestion service, recipe improvement service) and/or in the agent controller.

5.2.4 The Telegram Bot API Layer

The Telegram Bot API layer is responsible for managing end-user communication through the widely used messaging application Telegram. This choice was driven by the simplicity of implementation and the potential for widespread adoption. The absence of the need for a dedicated app makes the bot more accessible, especially for users who may be hesitant to install additional applications on their smartphones.

The bot was registered through the service "The BotFather," which allows for the definition of information such as the bot's name, description, profile image, and URL. It also provides the bot token, which is used in the code to bind the implemented behavior to a specific bot.

The actual bot implementation is located in the file *projectRoot/TelegramBot.py*. This file uses the *python-telegram-bot* library to handle receiving and responding to messages. Additionally, it is responsible for executing asynchronous services via the *apscheduler* library.

The entry point of the bot is the *main()* method, which sets up the *ConversationHandler* object to receive and react to user messages. It also assigns asynchronous processes to the scheduler along with a cron parameter that defines when those processes should be executed.

The *ConversationHandler* is designed to:

- Extract the user from the agent database using the Telegram user ID as the key and determine whether the user is new. If so, the registration phase is invoked.
- Launch the hub at the first interaction.
- Launch the specific task based on which task has been defined by the agent controller as the next one.

The *ConversationHandler* is typically used to define and manage different states in a bot's interaction flow, which is based on the Finite State Machine (FSM) model. However, in this project, state management is handled by the agent controller to offer a more customized implementation that better fits the unique needs of the E-Mealio agent. This approach also allows for greater flexibility and avoids tightly coupling the logic to the Telegram bot framework.

Instead of defining multiple states for different phases of the conversation, this implementation defines only a single state, *INTERACTION*. This state is solely responsible for managing the continuous conversation between the user and the agent. The conversational flow, including tracking intermediate user inputs and updating the state, is accomplished by passing all necessary data through a simple, custom contextual object named *user_data*.

The *user_data* object is defined within the bot context to track the conversational flow and user data. The *user_data* object contains the following information:

- **userData**: Contains the user data as defined in the *projectRoot/dto/User.py* DTO class. The object is retrieved from the database at the start of a conversation, or it is created during the first interaction and then stored. This data is passed back and forth to the controller and used by it when necessary.
- **action**: Contains the next token representing the action that must be invoked. This value is returned by the agent controller.
- **memory**: For certain functionalities, the ability to keep track of a portion of the conversation is enabled. This allows the LLM model to respond based on the entire interaction during that phase. This field stores that memory to be passed again to the controller when needed.
- **info**: Similar to memory, this field stores information used by the controller to provide the LLM layer with extracted details. If these pieces of information are built through several conversational steps, this field tracks them and passes them again to the agent

controller.

- **callbackMessage**: If populated by the agent controller, this field contains a fictitious user message that is immediately passed back to the controller. This enables generating two responses in certain cases, such as when an interaction is closing and the hub needs to be presented again.

All these fields are populated through interactions with the agent controller, the actual core of the E-Mealio agent.

This structure allows for a more flexible and modular approach to conversation handling, which could easily be adapted for different messaging platforms or even for a non-Telegram-based interface if needed.

The implementation automatically handles multiple users by providing a new instance for each one, eliminating the need for specific effort in implementing multi-user support.

5.2.5 The Agent Controller and the LangChain Service Layer

The Agent Controller serves as the core of the E-Mealio Agent, managing the conversational flow by interpreting user requests and directing them to the appropriate prompt for processing. It is implemented in the file *projectRoot/ChatbotController.py*.

The controller is structured as a finite state machine (FSM), as described below[4].

[4]The diagram presented is a simplified version of the actual implemented FSM, as each state may pass through intermediate states used for validating the user's request before proceeding.

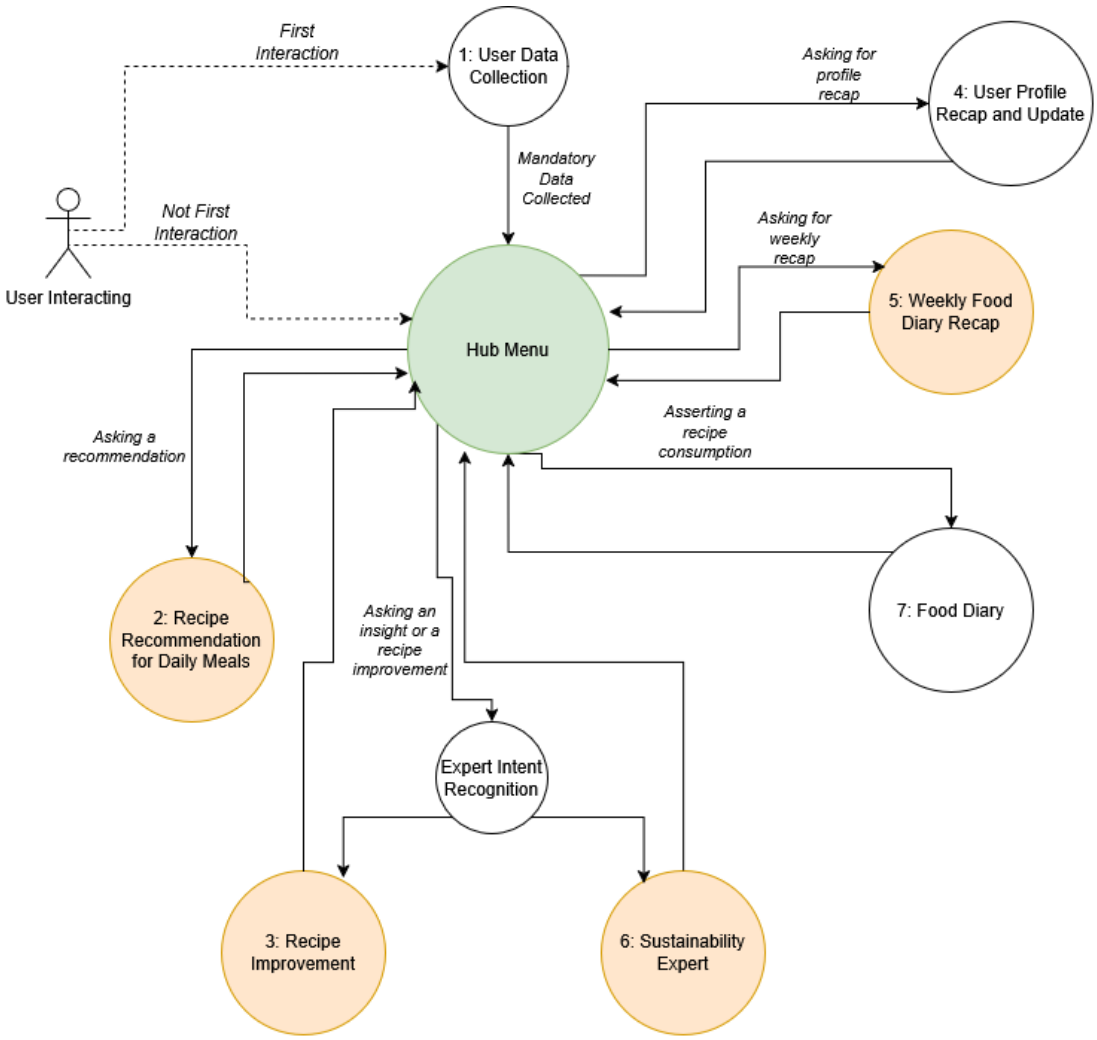


Figure 5.2: Abstract finite state machine overview of the controller.

Each numbered state in the finite state machine (FSM) corresponds to a specific functionality outlined in Section 5.1. The green central state represents the hub where the agent remains idle, awaiting a user request. Dashed arrows indicate states where the user can initiate an interaction. Orange states denote conversation-continuation states, leveraging the LangChain memory. This allows the user to request further details and clarifications before returning to the central hub.

The controller selects the appropriate action using a token linked to a specific LLM prompt. Each prompt is designed to consistently generate a token alongside the response, as well as

any additional information needed for subsequent processing.

The prompts are executed by the LangChain Service, implemented in the file *projectRoot/service/bot/ChatbotController.py*, it is a lightweight software module responsible for invoking the configured LLM using the LangChain library and parsing its output into the Response DTO class.

After receiving the Response DTO from the LangChain Service, the controller checks whether it contains a valid response to be presented to the user, which leads to two possible outcomes.

- If an answer is present, it is forwarded to the Telegram layer for displaying it. The Telegram layer internally stores the token from the *Response* DTO into its *user_data* object to maintain the conversational flow and resumes the conversation using the appropriate FSM state if the user send a new messages as response.
- If the Response DTO does not contain an answer, the controller immediately executes the next task. This process continues in a loop until a final answer is generated. This iterative mechanism enables multi-step prompt processing when necessary, allowing complex requests to be handled seamlessly.

The controller's workflow can be summarized by the following algorithm:

Algorithm 2 Answer Router Algorithm

1: **function** AnswerRouter($userData, userPrompt, token, memory, info$)

Require: $userData$ contains user identification and settings

Require: $userPrompt$ is the input query from the user

Require: $token$ is the current FSM token

Require: $memory$ contains conversation history

Require: $info$ contains additional context information

Ensure: returns a Response object with answer, action, memory, modifiedPrompt, and info

2: $response \leftarrow$ CreateEmptyResponse()

3: **while** $response.answer$ is empty **do**

4: $response \leftarrow$ AnswerQuestion($userData, userPrompt, token, memory, info$)

5: **if** An error occurred **then**

6: Log the error

7: **return** error message

8: **end if**

9: $token \leftarrow response.action$

10: $info \leftarrow response.info$

11: $memory \leftarrow response.memory$

12: **if** $response.modifiedPrompt$ is not empty **then**

13: $userPrompt \leftarrow response.modifiedPrompt$

14: **end if**

15: **end while**

16: **return** $response$

17: **end function**

18: **function** AnswerQuestion($userData, userPrompt, token, memory, info$)

Require: $userData$ contains user identification and settings

Require: $userPrompt$ is the input query from the user

Require: $token$ identifies the current conversation state

Require: $memory$ contains conversation history

Require: $info$ contains additional context information

Ensure: returns a Response object with answer, action, memory, modifiedPrompt, and info

19: **if** token == p.TASK_0_HOOK **then** ▷ Task 0 is related to user data acquisition.

20: $response \leftarrow$ ExecuteLLMchain(p.GET_DATA_PROMPT_BASE_0, userPrompt, temperature, userData)

21: **return** $response$

22: ▷ ...other branches related to user data acquisition.

23: **else if** token == p.TASK_1_HOOK **then** ▷ Task 1 is related to presenting the agent hub.

24: $memory \leftarrow$ None

25: $response \leftarrow$ ExecuteLLMchain(p.STARTING_PROMPT, userPrompt, temperature, userData)

26: **return** $response$

27: **else if** token == p.TASK_2_HOOK **then** ▷ Task 2 is related to recipe recommendation.

28: $response \leftarrow$ ExecuteLLMchain(p.TASK_2_PROMPT, userPrompt, temperature, userData)

29: **return** $response$

30: ▷ ...other branches related to recipe recommendation.

31: **else**

32: ▷ Similar logic for tokens:

33: ▷ TASK_MINUS_1_HOOK: Memory cleaning

34: ▷ TASK_3_HOOK to TASK_3_60_HOOK: Recipe improvement

35: ▷ TASK_4_HOOK to TASK_4_50_HOOK: Profile management

36: ▷ TASK_5_HOOK to TASK_5_10_HOOK: History retrieval

37: ▷ TASK_6_HOOK to TASK_6_40_HOOK: Sustainability expert

38: ▷ TASK_7_HOOK to TASK_7_20_HOOK: Recipe consumption diary

39: **end if**

40: **end function**

In the *AnswerQuestion* function, the *ExecuteLLMchain* function corresponds to a call to the LangChain Service layer. The actual *temperature* value used in each call depends on the specific execution branch and was determined empirically during development.

Each branch of the *AnswerQuestion* function, in addition to invoking the LangChain Service, may also perform auxiliary operations such as data preparation (e.g., converting between Python objects and JSON or vice versa), calling some core services like the recipe recommender or the recipe improver and manage interactions with the persistence layer, like in the case of user data construction.

The *AnswerQuestion* function serves as the main controller, orchestrating both LLM calls and core services. It is responsible for making key decisions regarding the execution flow and determining the necessary operations at each step.

Most tokens, represented by constants from the TASK_*N*_HOOK set, are associated with an LLM prompt designed to instruct the model to perform specific actions or generate text based on a given context or request. It is also possible to bind a token to a simple algorithmic execution or to a function call if necessary. This behavior depends on the implementation of the controller branch that is triggered by the corresponding token.

The behavior of the *ExecuteLLMchain* is synthesized in the following algorithm:

Algorithm 3 LLM Chain Execution Algorithm

1: $llm \leftarrow$ LLM LangChain wrapper (ChatOpenAI or ChatAnthropic)

2: **function** ExecuteLLMchain($inPrompt, inQuery, temp, userData, memory, memEnabled$)

Require: $inPrompt$ is the system prompt for the LLM

Require: $inQuery$ is the user's prompt to process

Require: $temp$ controls randomness of LLM output

Require: $userData$ contains user identification information

Require: $memory$ contains conversation history (optional)

Require: $memEnabled$ indicates if memory should be used

Ensure: returns a Response object with answer, action, info, memory, and empty modified prompt

Set llm temperature to $temp$
if $memory$ is None **and** $memEnabled$ is True **then**
    $memory \leftarrow$ Initialize new ChatMessageHistory
    Add $inPrompt$ to $memory$ as user message
end if
$prompt \leftarrow$ GetPrompt($inPrompt, memory$)
$outputParser \leftarrow$ Initialize new StrOutputParser
$chain \leftarrow$ Combine $prompt$, llm, and $outputParser$
if $memory$ is not None **then**
    $answer \leftarrow chain.invoke(query : inQuery, memory : memory.messages)$
else
    $answer \leftarrow chain.invoke(query : inQuery)$
end if
$action \leftarrow$ GetToken($answer$)
$info \leftarrow$ GetInfo($answer$)
$answer \leftarrow$ CleanAnswerFromTokenAndInfo($answer, info$)
if $memory$ is not None **then**
    Add $inQuery$ to $memory$ as user message
    Add $answer$ to $memory$ as AI message
end if
$response \leftarrow$ new Response($answer, action, info, memory,''$)
return $response$
end function

26: **function** GETPROMPT($inputPrompt, memory$)

Require: $inputPrompt$ is the system prompt for the LLM

Require: $memory$ contains conversation history (can be None)

Ensure: returns a properly formatted prompt template

27: **if** $memory$ is not None **then**

28: **return** ChatPromptTemplate with system message, memory placeholder, and user query

29: **else**

30: **return** ChatPromptTemplate with system message and user query only

31: **end if**

32: **end function**

Where *GetToken*, *GetInfo*, and *CleanAnswerFromTokenAndInfo* simply extract the relevant portions of the response using regex.

The algorithm primarily serves as an orchestrator for the following LangChain objects:

- The LLM wrapper (*ChatOpenAI* or *ChatAnthropic*), selected by defining an appropriate constant. This wrapper abstracts the configurations required to call the LLM's API.
- The prompt template (*ChatPromptTemplate*), which defines the structure of the prompt sent to the model, specifying both the system prompt and the user query.
- The *ChatMessageHistory*, which stores previous messages from both the user and the agent. This allows the agent to incorporate past conversation context, enabling more coherent and context-aware responses.
- The *StrOutputParser*, which parses the LLM's response into a string format.

The strength of this approach lies in its flexibility; it allows tasks to be easily added, removed, or modified, thereby adapting the agent's behavior dynamically. Moreover, this framework can be applied to completely different domains due to its generalizable structure.

5.2.5.1 Prompts Invoked by the LangChain Service Layer

All prompts, along with their token definitions, are located in *projectRoot/Constants.py*, grouped by functionality.

Each prompt is identified by a constant name, which the controller uses to invoke it through the LangChain service. Every prompt constant name is paired with a corresponding token constant that shares the same numeric identifier in its name, making it easy to associate the correct token with its related prompt. For example, the constant *TASK_2_HOOK* is assigned to the value *"TOKEN_2"*, which is then used to invoke *TASK_2_PROMPT* in the controller.

Each prompt will be presented and discussed in detail in Section 5.3 when explaining the implementation of each functionality.

In essence, each defined prompt:

- Defines the agent's role and the task at hand.
- Specifies the expected input and output format (often involving JSON) or the general content of the message that the prompt should manage.
- Uses tokens to indicate transitions between conversation states, often handling multiple outcomes or self-loops in cases where re-asking the same question is necessary.

5.2.6 The Recipe Recommendation Service

One of the key capabilities of the agent is to recommend meals for different times of the day, such as breakfast, lunch, a break, or dinner.

To enable this functionality while leveraging the rich semantic properties of the **HeaSEV2** dataset, a dynamic query-based service has been implemented. This service extracts recipes that match the user's explicit requirements, personal profile, and general taste preferences.

The Recipe Recommendation Service is implemented in the file: *projectRoot/service/Sug-*

gestRecipeService.py.

Its core function is:

get_recipe_suggestion (mealDataJson, userData)

This function returns a **Recipe DTO object** containing the recommended recipe.

5.2.6.1 Input Parameters

The function takes two input objects, both provided by the **controller layer**:

-3pt**userData**: An instance of the **User DTO**, containing relevant user data such as:

- – Allergies.
  – Dietary restrictions.
  – Taste preferences embeddings for each meal type.

- **mealDataJson**: A JSON object constructed by the agent controller using the *TASK_-2_PROMPT*. This prompt extracts structured information from a natural language user request and returns a JSON object with the following fields:

```
{
  "mealType": "Breakfast | Lunch | Dinner | Break",
  "recipeName": "string",
  "sustainabilityScore": float,
  "ingredients_desired": ["string", "string"],
  "ingredients_not_desired": ["string", "string"],
  "cookingTime": "short | medium | not_relevant",
  "healthiness": "yes | not_relevant"
}
```

The *recipeName* and *sustainabilityScore* fields are not used in the recommendation phase when responding to a standard user request. However, they are essential for the **improvement service**, which builds upon the recommendation service by adjusting the request parameters (as discussed later).

All other collected data are used to refine the selection and extract the most suitable recipe.

5.2.6.2 Recipe Recommender Service Execution

The **recipe recommendation process** begins by defining the following **MongoDB query**:

```
{
  "$and": [
    { "  sustainability_label  ": { "$in": [0, 1] } },
    { " percentage_covered_cfp ": { "$gte": 70 } },
    { "percentage_covered_wfp": { "$gte": 70 } },
    { " disabled ": false },
    { "TAGS_SUSTAINABILITY" },
    { "TAGS_RESTRICTIONS" },
    { "ALLERGENES" },
    { "TAGS_MEAL_TYPE" },
    { "TAGS_USER_HISTORY" },
    { "TAGS_HEALTHINESS" },
    { "TAGS_MEAL_DURATION" }
  ]
}
```

This query consists of two types of constraints:

1. **Fixed Conditions**: The extracted recipes must all satisfy the same base conditions:

- The sustainability level must be either 0 or 1, ensuring that highly impactful recipes are not recommended.
- The suggested recipe must have at least **70% coverage** for both **CFP (Carbon Footprint)** and **WFP (Water Footprint)** across all its ingredients. This threshold ensures reliability when analyzing a recipe's environmental impact.
- All disabled recipes must be excluded from the selection process.

2. **Dynamic Conditions**: Some parts of the query are represented by TAGS placeholders. These placeholders are dynamically populated based on the user's request and applied only if the corresponding conditions are met.

This approach allows for a flexible and personalized recipe selection process while maintaining a structured filtering mechanism.

The service then analyzes the provided input data to determine which dynamic conditions hold and properly assigns their values. This process is carried out as follows:

First, a set of variables representing the tags that need to be substituted is defined:

```
tagsSustainability = ""
tagsRestrictions = ""
allergenes = ""
tagsMealType = ""
tagsUserHistory = ""
tagsHealthiness = ""
tagsMealDuration = ""
```

Each of these variables will replace the corresponding placeholders in the query. If a variable is assigned an actual value, it introduces an additional filter in the query. Conversely, if a variable remains empty, it will not affect the filtering process.

The embeddings for the desired and undesired ingredients are calculated by summing the individual embeddings of each ingredient listed in the *ingredients_desired* and *ingredients_not_desired* fields from the *mealDataJson* input parameter. These embeddings are generated using the **Alibaba-NLP/gte-large-en-v1.5** model, which is also used to define the embedding fields in the database.

The same approach is applied to the recipe name provided in the *recipeName* in *mealDataJson*. If the recipe name is specified, its embedding is computed using the same model. This embedding is particularly leveraged when the recommendation service is invoked by the recipe improvement service.

If the *ingredients_desired* list is not provided, the *desiredIngredientsEmbedding* is instead initialized using the user's taste vector for the specific meal type requested. This approach ensures consistency when applying the semantic ranking function later in the process. However, if *ingredients_desired* is explicitly provided, it overrides the user's taste preferences.

Then the actual tags variable valorization is performed as follows:

- If the *sustainabilityScore* value is provided in the *mealDataJson* input parameter, the following piece of json query is produced and replaced to the TAGS_SUSTAINABILITY in the recipe extraction query:

  ```
  {
    " sustainability_score ": { "$lt ": SUSTAINABILITY_VALUE }
  }
  ```

  However this valorization actually happen only when the recommendation service is called passing through the recipe improvement service.
- If the field *restrictions* in the *userData* object is populated, the following JSON query snippet is generated and replaces the *TAGS_RESTRICTIONS* placeholder in the recipe

extraction query:

```
{
    "$and": [
     {"tags": { "$regex": "RESTRICTION_1" }},
     {"tags": { "$regex": "RESTRICTION_2" }},
      ...
     {"tags": { "$regex": "RESTRICTION_N" }}
   ]
}
```

where each *RESTRICTION_I* corresponds to an actual restriction value present in the list.

- Similarly to the previous case, if the field *allergies* in the *userData* object is populated, the following JSON query snippet is generated and replaces the *TAGS_ALLERGENES* placeholder in the recipe extraction query:

```
{
    "$and": [
     {"tags": { "$regex": "ALLERGEN_1-free" }},
     {"tags": { "$regex": "ALLERGEN_2-free" }},
      ...
     {"tags": { "$regex": "ALLERGEN_N-free" }}
   ]
}
```

where each *ALLERGEN_I* corresponds to an actual allergen value present in the list. The suffix *-free* is appended because this notation is used in the HeaSEV2 dataset to identify recipes that do not contain the specified allergen.

- The meal type is always specified in the *mealDataJson* input parameter if the recommen-

dation service is directly called by the user. For each possible value (Breakfast, Lunch, Break, or Dinner), a different JSON query snippet is generated to replace the *TAGS_MEAL_TYPE* placeholder in the recipe extraction query.

For **Breakfast**:

```
{
    "tags": { "$regex": "breakfast" }
}
```

For **Lunch**:

```
{
    "$and": [
        { "tags": { "$regex": "main-dish" } },
        { "tags": { "$regex": "lunch" } }
    ]
}
```

For **Break**:

```
{
    "tags": { "$regex": "snack" }
}
```

For **Dinner**:

```
{
    "$and": [
        { "tags": { "$regex": "main-dish" } },
        { "tags": { "$regex": "dinner" } }
    ]
```

}

In the cases of **Lunch** and **Dinner**, the *main-dish* tag is also enforced to prevent recommending appetizers or minor courses. This constraint ensures that the E-Mealio agent suggests a single main course for each meal type.

If the recommendation service is not invoked by the user but instead used as an auxiliary service by the improvement service, no specific meal type is provided. However, to avoid biasing the recommendations toward recipes such as drinks or minor courses, and to reduce the number of recipes held in memory when searching for an alternative recipe, the following JSON query snippet replaces the *TAGS_MEAL_TYPE* placeholder:

```
{
    "$or": [
        { "$and": [ { "tags": { "$regex": "main-dish" } }, { "tags": { "$regex": "
            dinner" } } ] },
        { "$and": [ { "tags": { "$regex": "main-dish" } }, { "tags": { "$regex": "
            lunch" } } ] },
        { "tags": { "$regex": "breakfast" } },
        { "tags": { "$regex": "snack" } }
    ]
}
```

Further details on how the recommendation system operates within the context of recipe improvement will be provided in a dedicated subsection of this chapter.

- Another key property of the recommendation service is to prevent suggesting the same recipe more than once within the same week. This ensures proper diversification in the proposed recipes.

To achieve this, the service retrieves the user's recipe consumption history from the past seven days using the method *get_user_history_of_week* from the auxiliary service *FoodHistoryService*. The extracted recipe IDs are then used to populate the following JSON query snippet, which replaces the *TAGS_USER_HISTORY* placeholder:

```
{
    " recipe_id ": { "$nin": [ RECIPE_ID_1, RECIPE_ID_2, ..., RECIPE_ID_N ] }
}
```

where each *RECIPE_ID_I* corresponds to an actual recipe ID retrieved using *get_user_history_of_week*.

It is important to note that only recipes present in the database are considered for exclusion. Recipes manually asserted by the user, which do not have a corresponding database ID, are not included in the exclusion process.

- To leverage the presence of the *healthiness_label* field in the recipe database, the recommendation service provides the option to filter only recipes labeled as healthy.

  This is achieved by adding the following JSON query snippet, which replaces the *TAGS_HEALTHINESS* placeholder, whenever the *healthiness* field in the *mealData* input parameter is set to "yes":

  ```
  {
      " healthiness_label ": 0
  }
  ```

- Finally, to utilize the presence of time-related tags in the recipe database, such as *"15-minutes-or-less"* or *"30-minutes-or-less"*, the recommendation service allows users to specify a desired cooking time.

  This is done via the *cookingTime* field in the *mealDataJson* input parameter. If this field

is set, the following JSON query snippet is generated and replaces the *TAGS_MEAL_DURATION* placeholder. The exact query depends on the value of *cookingTime*:

If *cookingTime* is set to "short":

```
{
    "tags": { "$regex": "15-minutes-or-less" }
}
```

If *cookingTime* is set to "medium":

```
{
    "tags": { "$regex": "30-minutes-or-less" }
}
```

Given the presence of multiple filters, it is possible that their combination may lead to an empty result set during recipe extraction. To prevent the system from returning no recipe at all, a **flexible extraction procedure** is implemented. This procedure distinguishes between **mandatory filters**, which must always be applied, and **optional filters**, which can be relaxed if necessary to broaden the search results.

The **mandatory filters** are:

- **Sustainability filter** (if present in the *tagsSustainability* variable). This filter ensures that only recipes with a sustainability score lower than the one specified in the *mealDataJson* input parameter are extracted. However, this filter is only applied when the recommendation service is invoked through the **recipe improvement service**.
- **Dietary restriction filter** (if present in the *tagsRestrictions* variable). This filter is mandatory to ensure that the system does not recommend recipes that violate the user's dietary restrictions.

- **Allergen filter** (if present in the *allergenes* variable). This filter prevents the system from suggesting recipes containing ingredients that the user is allergic to.
- **Meal type filter** (present in the *tagsMealType* variable). This filter ensures that the recommended recipe is appropriate for the specified meal type (e.g., Breakfast, Lunch, Break, or Dinner). Additionally, this filter helps limit the number of recipes held in memory, optimizing the recommendation process.

The **optional filters** are listed below, ordered from the least important to the most important:

- **Recipe preparation duration filter** (if present in the *tagsMealDuration* variable). This filter ensures that the recommended recipes have a preparation time compatible with the user's preference. However, if no recipes satisfy both the mandatory and optional filters, this filter is the first to be removed.
- **Recipe healthiness filter** (if present in the *tagsHealthiness* variable). This filter ensures that the recommended recipes align with the user's preference for healthy meals. However, if no recipes satisfy both the mandatory and optional filters, this filter is the second to be removed.
- **User weekly consumption filter** (if present in the *tagsUserHistory* variable). This filter prevents the system from recommending recipes that the user has already consumed or rejected within the past week. However, if no recipes satisfy both the mandatory and optional filters, this filter is the last to be removed.

Whenever an optional filter is removed, it is recorded in the *removedConstraints* list. This list is then included in the recipe object passed to the LLM model, allowing it to inform the user that some constraints were relaxed to successfully complete the search process.

The removal of optional filters is performed iteratively whenever the search query returns an empty result set. This process continues until either:

- A valid recipe is found, or
- All optional filters are removed without yielding a non-empty result set.

If the recommendation service is unable to retrieve a compatible recipe after removing all optional filters, it returns *None* to the caller, which will handle the situation accordingly.

After executing the query, the recipe result set is ordered based on a value called *taste_score*. This score is calculated as follows:

- The *taste_score* is assigned to the extracted recipe dataframe as a new column and initialized at value 0.
- If the *desiredIngredientsEmbeddings* value is provided, the cosine similarity between the *desiredIngredientsEmbeddings* and the recipe embedding of each extracted recipe is computed. The result is stored in a new column of the extracted recipe dataframe called *cosine_similarity_desired*. This column is then summed to the *taste_score* column.
- If the *notDesiredIngredientsEmbeddings* value is provided, the cosine similarity between the *notDesiredIngredientsEmbeddings* and the recipe embedding of each extracted recipe is computed. The result is stored in a new column called *cosine_similarity_not_desired*. This column is then subtracted from the *taste_score* column.
- If the *recipeNameEmbedding* value is provided, the cosine similarity between the *recipeNameEmbedding* and the recipe embedding of each extracted recipe is computed. The result is stored in a new column called *cosine_similarity_recipe_name*. This column is then summed to the *taste_score* column.

Summarizing, the *taste_score* is computed as:

$$taste\_score = cosine\_similarity\_desired - cosine\_similarity\_not\_desired + cosine\_similarity\_recipe\_name$$

The *cosine_similarity_not_desired* value is subtracted because a higher similarity with the *notDesiredIngredientsEmbeddings* suggests that the recipe might contain ingredients the user does not want.

The *cosine_similarity_recipe_name* is used specifically when the recommendation service is called by the recipe improvement service. In this case, the alternative recipe should be similar to the base recipe. A similarity in the recipe names indicates that they are likely variants of the same recipe.

After sorting, the most relevant recipe, i.e., the one with the highest *taste_score*, is selected for recommendation. It is then converted into a *RecipeDTO* object and returned to the caller.

If the *taste_score* computation is not possible because both the *desiredIngredientsEmbeddings* and the *notDesiredIngredientsEmbeddings* are not provided, simply the most sustainable recipe using the *sustainabilityScore* column is recommended.

When this happens, it means that the user didn't provide any indication about desired ingredients, and their taste vector regarding preferences for that meal type is still not available.

This service is flexible enough to recommend tailored recipes accounting for preferences and requirements in a way that strongly leverages the semantics of the HeaSEv2 dataset.

All the previously described logic are implemented by means of the following algorithm:

Algorithm 4 Recipe Suggestion Algorithm

1: **function** GETRECIPESUGGESTION($mealDataJson, userData$)

Require: $mealDataJson$ contains meal preferences and constraints

Require: $userData$ contains user information including restrictions and allergies

Ensure: Recipe object that matches user preferences or None if no match found

2: $queryTemplate \leftarrow$ Initialize MongoDB query template with core constraints

3: $mandatoryReplacement \leftarrow$ empty list for required query constraints

4: $notMandatoryReplacement \leftarrow$ empty list for optional query constraints

```
    desiredIngredientsEmbedding ← empty numpy array
    notDesiredIngredientsEmbedding ← empty numpy array
    recipeNameEmbedding ← empty numpy array
    mealData ← Decode JSON meal data
    if mealData.recipeName is not empty then
        recipeNameEmbedding ← Embed recipe name
    end if
    if mealData.ingredients_desired is not empty then
        desiredIngredientsEmbedding ← Embed desired ingredients
    else
        tastes ← GetUserTastes(userData.id, mealData.mealType)
        if tastes is not empty then
            desiredIngredientsEmbedding ← Convert tastes to numpy array
        end if
    end if
    if mealData.ingredients_not_desired is not empty then
        notDesiredIngredientsEmbedding ← Embed non-desired ingredients
    end if
    recipes ← Access recipes collection from database
    if mealData.sustainabilityScore is not empty then
        Add sustainability filter to tagsSustainability
    end if
    if userData.restrictions is not empty then
        Build restrictions filter in tagsRestrictions
    end if
```

30: **if** $userData.allergies$ is not empty **then**

31: Build allergies filter in $allergenes$

32: **end if**

33: Build meal type filter in $tagsMealType$ based on $mealData.mealType$

34: $userHistory \leftarrow$ GetUserHistoryOfWeek($userData.id$)

35: **if** $userHistory$ is not empty **then**

36: Build user history filter in $tagsUserHistory$

37: **end if**

38: **if** $mealData.healthiness$ is "yes" **then**

39: Set healthiness filter in $tagsHealthiness$

40: **end if**

41: **if** $mealData.cookingTime$ is specified **then**

42: Set meal duration filter in $tagsMealDuration$

43: **end if**

44: Add primary filters to $mandatoryReplacement$ list

45: Add optional filters to $notMandatoryReplacement$ list

46: $numberOfFoundRecipes \leftarrow 0$

47: $numReplacement \leftarrow$ length of $notMandatoryReplacement$

48: $removedConstraints \leftarrow$ empty list

49: **while** $numberOfFoundRecipes = 0$ AND $numReplacement > 0$ **do**

50: $queryText \leftarrow$ QueryTemplateReplacement($mandatoryReplacement$, $notMandatoryReplacement$, $numReplacement$, $queryTemplate$)

51: $query \leftarrow$ Decode JSON query

52: $suggestedRecipes \leftarrow$ Find recipes matching query

53: $numberOfFoundRecipes \leftarrow$ Count matching documents

$numReplacement \leftarrow numReplacement - 1$

if $numberOfFoundRecipes = 0$ AND current constraint is not empty **then**

Append current constraint name to $removedConstraints$

end if

end while

if $numberOfFoundRecipes = 0$ **then**

return None ▷ No recipe found

end if

$suggestedRecipe \leftarrow$ GetPrefRecipeByTaste($suggestedRecipes$, $desiredIngredientsEmbedding$, $notDesiredIngredientsEmbedding$, $recipeNameEmbedding$, $userData$)

if $suggestedRecipe$ is None **then**

Sort $suggestedRecipes$ by sustainability score (ascending)

$suggestedRecipe \leftarrow suggestedRecipes[0]$

end if

$suggestedRecipe \leftarrow$ GetRecipeById(suggestedRecipe.recipe_id)

$suggestedRecipe \leftarrow$ ConvertToEmealioRecipe($suggestedRecipe$, $removedConstraints$, $mealType$)

return $suggestedRecipe$

end function

function GetPrefRecipeByTaste($recipeSet, desIngr, notDesIngr, recipeName, userData$)

Require: $recipeSet$ is a collection of recipes to evaluate

Require: $desIngr$ contains embeddings of preferred ingredients

Require: $notDesIngr$ contains embeddings of ingredients to avoid

Require: $recipeName$ contains embedding of desired recipe name

Require: $userData$ contains user identification information

Ensure: returns the recipe with highest taste score or None if no preferences

if $desIngr$ is empty **and** $notDesIngr$ is empty **then**

    return None

end if

$recipes\_df \leftarrow$ Convert $recipeSet$ to DataFrame

Initialize $recipes\_df.taste\_score \leftarrow 0.0$

if $desIngr$ is not empty **then**

    $recipes\_df.sim\_desired \leftarrow$ CosSimilarity($recipes\_df.ingr\_embedding$, $desIngr$)

    $recipes\_df.taste\_score \leftarrow recipes\_df.taste\_score + recipes\_df.sim\_desired$

end if

if $notDesIngr$ is not empty **then**

    $recipes\_df.sim\_not\_desired \leftarrow$ CosSimilarity($recipes\_df.ingr\_embedding$, $notDesIngr$)

    $recipes\_df.taste\_score \leftarrow recipes\_df.taste\_score - recipes\_df.sim\_not\_desired$

end if

if $recipeName$ is not empty **then**

    $recipes\_df.sim\_name \leftarrow$ CosSimilarity($recipes\_df.title\_embedding$, $recipeName$)

    $recipes\_df.taste\_score \leftarrow recipes\_df.taste\_score + recipes\_df.sim\_name$

end if

$preferred\_recipes \leftarrow$ Sort $recipes\_df$ by $taste\_score$ descending

$highestTasteScoreRecipe \leftarrow preferred\_recipes[0]$

return $highestTasteScoreRecipe$

end function

The *ConvertToEmealioRecipe* and *QueryTemplateReplacement* functions are omitted for brevity. However, the former essentially converts a recipe object retrieved from the database into a

Recipe DTO object containing only the relevant information, while the latter progressively replaces constraints in the query template, ensuring that optional constraints are omitted as the input parameter *numReplacement* decreases.

5.2.7 The Improvement Service

The second core functionality of the E-Mealio agent is its ability to suggest recipe improvements that enhance sustainability. To manage the complexity of this task, the system leverages the presence of similar recipes that differ in a subset of ingredients, where one version is more sustainable than the other. Given these two recipes, the LLM model is tasked with highlighting the differences and providing the user with a coherent ingredient swap.

Given this requirement, the primary task involves a search process similar to the one performed by the recommender service. For this reason, the recommender service was designed in advance with the capability of identifying recipes with a sustainability score below a predefined threshold.

The implemented improvement service, located in *projectRoot/service/ImproveRecipeService.py*, is responsible for extracting recipe information provided by the user, such as ingredient CFP and WFP values, and computing the sustainability score of the given recipe. The sustainability label is then determined by applying the same thresholds described in Section 4.4.

If the computed label is 2, indicating that the recipe is not sustainable, the recipe suggestion service is invoked, using only the recipe name and ingredient list. Otherwise, the current sustainability score is also included in the request to ensure that the retrieved recipe has a lower sustainability score than the one provided by the user.

The recipe name and ingredient list are used to perform a semantic search, identifying recipes that are as close as possible to the one provided by the user.

5.2.8 The Async Services

As described in Subsection 5.1.3, the agent includes two asynchronous functionalities designed to encourage user engagement and build trust in the system.

The first asynchronous functionality allows the agent to send reminders to users who have opted in during profile creation.

The second functionality involves the monthly computation of a taste vector for each user, representing their preferences for different meal types, that are then used by the recipe recommender service to provide customized option to users.

5.2.8.1 Sending Reminders

This feature is implemented directly within the Telegram Bot API layer, leveraging its built-in capability for scheduling jobs. The system uses a cron job set to trigger daily at **12:00 PM**, a time chosen because it is close to the usual lunch hour.

At each trigger, the system checks the *lastInteractionDate* of all users who have reminders enabled (i.e., their *reminder* field is set to *True*). If the difference between the current date (truncated to remove time) and the user's *lastInteractionDate* is two days or more, a reminder message is sent, prompting them to engage with the bot.

5.2.8.2 Monthly Taste Computation

This functionality is triggered by a cron job in the Telegram Bot API Layer, which runs on the first day of each month. It calls a service implemented in *projectRoot/service/async/ImproveRecipeService.py* through the method *compute_monthly_user_taste()*.

This method iterates over all registered users and, for each user, invokes the taste computation method *compute_user_taste(user)*. This method extracts the user's food history from the

past 30 days and computes a taste vector for each meal type using *compute_taste(userHistory, mealType)*.

The *compute_taste()* method samples up to 10 consumed recipes for the given meal type. It then calculates the embedding of each sampled recipe by summing the embeddings of its ingredients. Finally, the recipe embeddings are summed, and the resulting vector is stored as the user's taste profile for that specific meal type.

The *compute_user_taste(user)* method behavior can be summarized by the following algorithm:

1: **function** ComputeUserTaste($user$)

Require: $user$ contains user identification and profile information

Ensure: Updates user's taste profiles for each meal type

2: $userHistory \leftarrow$ GetUserHistoryOfMonth($user.id$)

3: $breakfastTaste \leftarrow$ ComputeTaste($userHistory$, 'Breakfast')

4: $lunchTaste \leftarrow$ ComputeTaste($userHistory$, 'Lunch')

5: $dinnerTaste \leftarrow$ ComputeTaste($userHistory$, 'Dinner')

6: $breakTaste \leftarrow$ ComputeTaste($userHistory$, 'Break')

7: $tastes \leftarrow$ CreateObject()

8: $tastes.breakfast \leftarrow$ ConvertToList($breakfastTaste$)

9: $tastes.lunch \leftarrow$ ConvertToList($lunchTaste$)

10: $tastes.dinner \leftarrow$ ConvertToList($dinnerTaste$)

11: $tastes.break \leftarrow$ ConvertToList($breakTaste$)

12: UpdateUserTastes($user.id$, $tastes$)

13: **end function**

14: **function** ComputeTaste($userHistory, mealType$)

Require: $userHistory$ contains user's meal consumption history

Require: $mealType$ specifies the type of meal to analyze

Ensure: Returns embedding vector representing user's taste for specified meal type

if $userHistory$ is empty or null **then**

return null

end if

$meals \leftarrow$ EmptyList()

$tasteEmbedding \leftarrow$ ZeroVector(1024)

for each $singleMeal$ in $userHistory$ **do**

if $singleMeal.recipe.mealType = mealType$ **then**

$meals$.Append($singleMeal.recipe.ingredients$)

end if

end for

if $|meals| > 10$ **then**

$meals \leftarrow$ RandomSample($meals$, 10)

end if

if $|meals| = 0$ **then**

return null

end if

for each $meal$ in $meals$ **do**

$recipeEmbedding \leftarrow$ GetRecipeEmbedding($meal$)

$tasteEmbedding \leftarrow tasteEmbedding + recipeEmbedding$

end for

return $tasteEmbedding$

end function

function GetRecipeEmbedding($recipe$)

Require: $recipe$ is a list of ingredients

Ensure: Returns embedding vector representing the recipe

38: $recipeEmbedding \leftarrow$ ZeroVector(1024)

39: **for** each $ingredient$ in $recipe$ **do**

40: $ingFromDb \leftarrow$ GetIngredientByName($ingredient.name$)

41: **if** $ingFromDb$ is null **then**

42: $ingFromDb \leftarrow$ GetMostSimilarIngredient($ingredient.name$) ▷ Perform similarity search

43: **end if**

44: $embedding \leftarrow ingFromDb.ingredient\_embedding$

45: $recipeEmbedding \leftarrow recipeEmbedding + embedding$

46: **end for**

47: **return** $recipeEmbedding$

48: **end function**

5.3 Functionality Implementation Details and Examples

In Section 5.1, a general overview of the implemented functionalities was provided. This section offers a deeper dive into the implementation details of each functionality, examining how each is orchestrated using the technical components discussed in Section 5.2, with the help of sequence diagrams[5], and providing details about the system prompts used by the LLM layer. Each functionality is numbered according to the same order as presented in Section 5.1 for coherence.

5.3.1 The Main Hub

The *Main Hub* of the agent is the functionality loaded at the start of an interaction with a registered user, or when the command */start* is sent while the current managed state is **state 1**.

The main hub is powered by the following prompt.

[5] A representative example for each functionality will be provided. These examples will illustrate the general orchestration of the components.

TASK 1 System Prompt: Main Hub

You are a food recommender system named E-Mealio with the role of helping users choose more environmentally sustainable foods. The following numbered tasks are your main functionalities:

2) Start a recommendation session if the user doesn't know what to eat. Be careful: if the user mentions a break, they are referring to a snack. This task is usually triggered by sentences like "I don't know what to eat", "I'm hungry", "I want to eat something", "I would like to eat", "Suggest me something to eat", "Recommend me something to eat" etc. This task is also triggered when asking for new food suggestions starting from a previous one using a sentence like "Suggest me a recipe with the following constraints:"

3) Act as a sustainability expert if the user asks for properties of recipes or specific foods, or if the user asks for the sustainability improvement of a recipe. This task is usually triggered by sentences like "What is the carbon footprint of INGREDIENT/RECIPE?", "How much water is used to produce a kg of INGREDIENT/RECIPE?", "How can I improve the sustainability of INGREDIENT/RECIPE?", "Tell me about INGREDIENT/RECIPE" etc. where RECIPE is the actual recipe and INGREDIENT is the actual ingredient. The user can also mention more than one item (recipe/ingredient) in their request. Sustainability improvement requests often have terms like "more sustainable," "improve," "better," and so on... Recipes can be referred to by their name or just by their ingredients, however, the user must always provide the list of ingredients. This task is also triggered if the user asks for broad information about sustainability and climate change, like "What is the carbon footprint?", "What is the water footprint?", "What is food waste?", "What is global warming?", "What is climate change?", "How is food related to climate change?", "What is CO2?", "What is food sustainability?" etc.

Those are general examples; the user can ask about any environmental concept, but the main topic is environmental sustainability.

4) Summarize the user profile and eventually accept instructions to update it. This task is usually triggered by sentences like ”Tell me about my data,” ”What do you know about me?”, ”What is my profile?” etc.

5) Talk about the history of consumed food in the last 7 days. This task can be triggered by sentences like ”What did I eat in the last 7 days?”, ”Tell me about my food history”, ”What did I eat last week?”, ”Summarize my recent food habits” etc.

7) Keep track of recipes that the user asserts to have eaten, in order to subsequently evaluate the sustainability of the user’s food habits. This task is usually triggered by sentences like ”I ate a pizza”, ”I had a salad for lunch”, ”I cooked a carbonara” etc. Recipe tracking requires the list of ingredients for the recipe.

Each number is the identifier of a specific task.

Put maximum effort into properly understanding the user request in the previous categories. Be careful not to classify a question of type 2 as a question of type 3 and vice versa. Questions of type 3 are usually more specific and contain a recipe or a food.

Follow these steps to produce the output:

- If the user asks a question that triggers a functionality of type 2, 3, 4, 5, or 7, just print the string ”TOKEN X” where X is the number of the task. Do not write anything else.

- If the user ask a question about you, or asks how to use or invoke one of your previously mentioned numbered tasks (included recipe sustainability improvement and sustainability expertise), execute the following steps:

Print the string ”TOKEN 1”, then continue by providing a detailed explanation of how to invoke such functionality by referring to to previuosly mentioned example sentences and instructions. Do NOT mention the number of the task, just the functionality.

- Otherwise, execute all the following steps:

Print the string ”TOKEN 1”, then always continue by doing the following steps:

If the user wrote a greeting, answer with a greeting too. Otherwise, if it was an unrelated message or you simply don’t know how to respond, decline politely.

Subsequently, regardless to previous steps, introduce yourself by mentioning your name, and describe your capabilities. For each task, provide an example of a phrase that can trigger it. In the bullet point of task, start each of them with a representative emoji instead of a number or a symbol. Put an empty row between each task to improve readability. Do not forget to include your ability to answer general questions about sustainability as additional point. Add a reminder about using the /start command to begin a new conversation and return to the starting point. Conclude your message with a funny food joke.

- Finally, if you weren’t able to understand the user’s request: Print the string ”TOKEN 1”, then write a message where you tell the user that you didn’t understand the request because it wasn’t relatable to any of the functionalities you can perform. Then, present your capabilities as described above and conclude with a funny food joke.

Always maintain a respectful and polite tone.

The prompt primarily defines the core capabilities of the agent, assigning each function a unique identifier. These identifiers are later used by the controller to generate the appropriate tokens required by the controller for loading new prompts. Additionally, the prompt provides instructions on how the agent should introduce itself to the user, explain its functionalities, and guide the user in invoking each feature effectively.

The Main Hub behavior can be visualized by means of the following finite state machine graph.

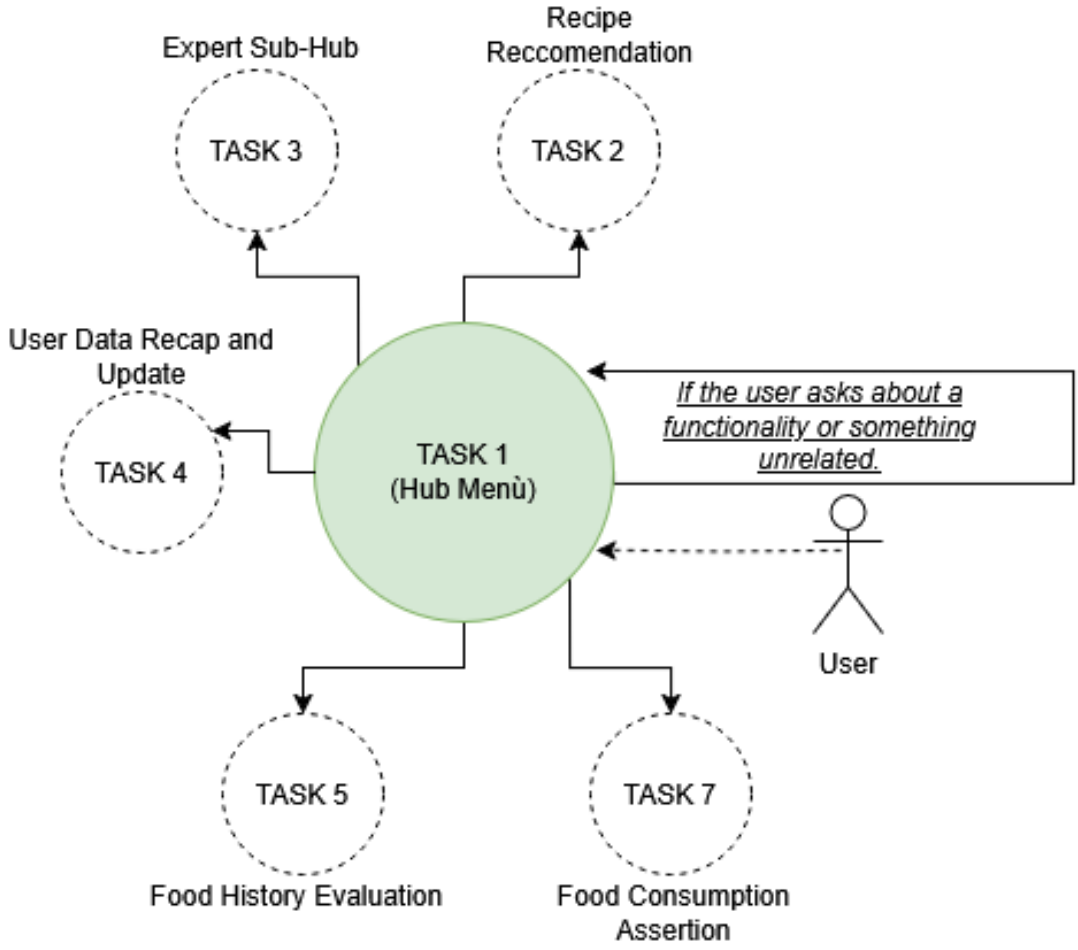


Figure 5.3: Finite State Machine representation of the Main Hub (**TASK 1**) behavior. **TASK 6** is managed by the recipe expert sub-hub (**TASK 3**), so it is not directly connected to the main hub. The dashed arrow represents the connection between the user and the state that enables the interaction. The dashed circles represent states that, when activated, would cause the agent to transition to a different task. In contrast, the solid circles represent states that are the focus of the current discussion within the task. This convention will also be maintained in subsequent FSM figures. All the connected states are complex FSMs themselves, each capable of returning to **TASK 1**. The structure of each task will be detailed later.

Basically it detect the intention of the user producing the relative controller state token. The Main Hub control flow when the user sends a greeting, triggering the agent's greeting, can be described by means of the following sequence diagram.

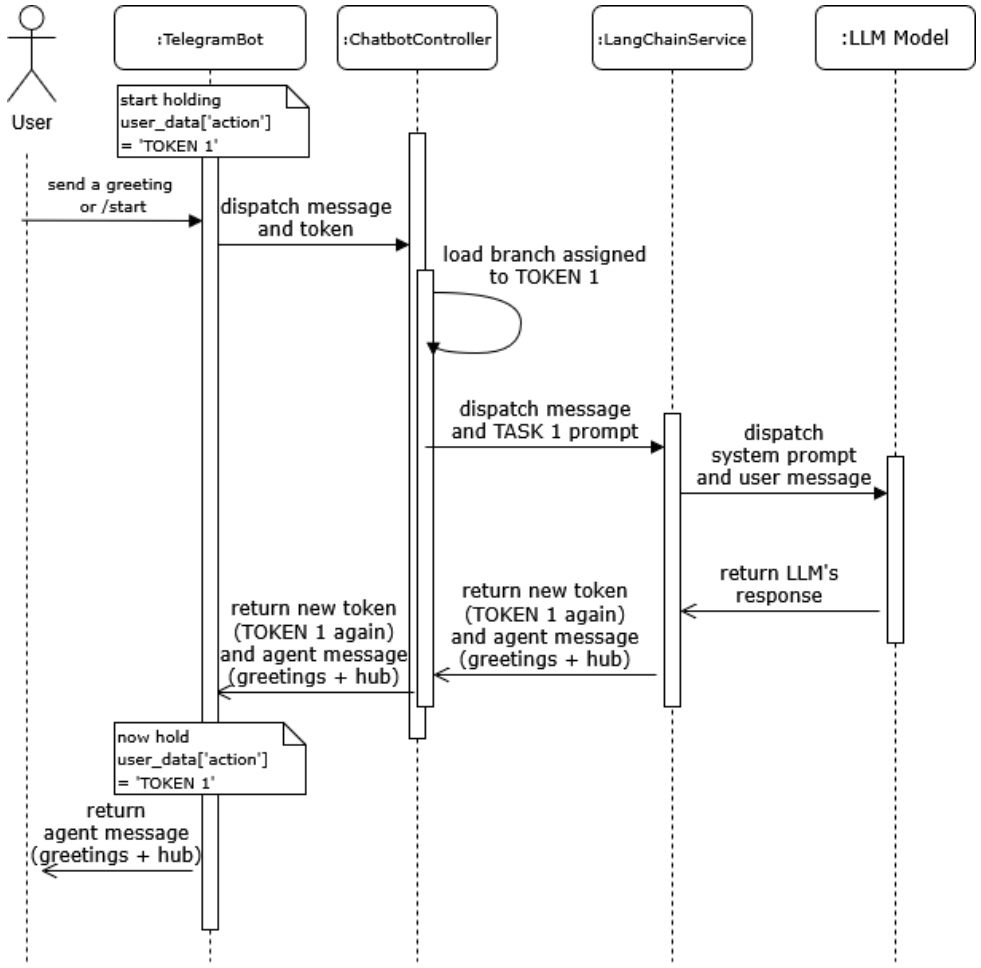


Figure 5.4: Sequence diagram describing the Main Hub control flow when the user send a greeting to the agent.

The same flow is followed when the user asks how to invoke a certain functionality. Also in that case, the underlying system prompt instruct to produce the controller token "TASK 1".

When instead a specific functionality is requested, the interaction between the agent's components can be described by means of the following sequence diagram.

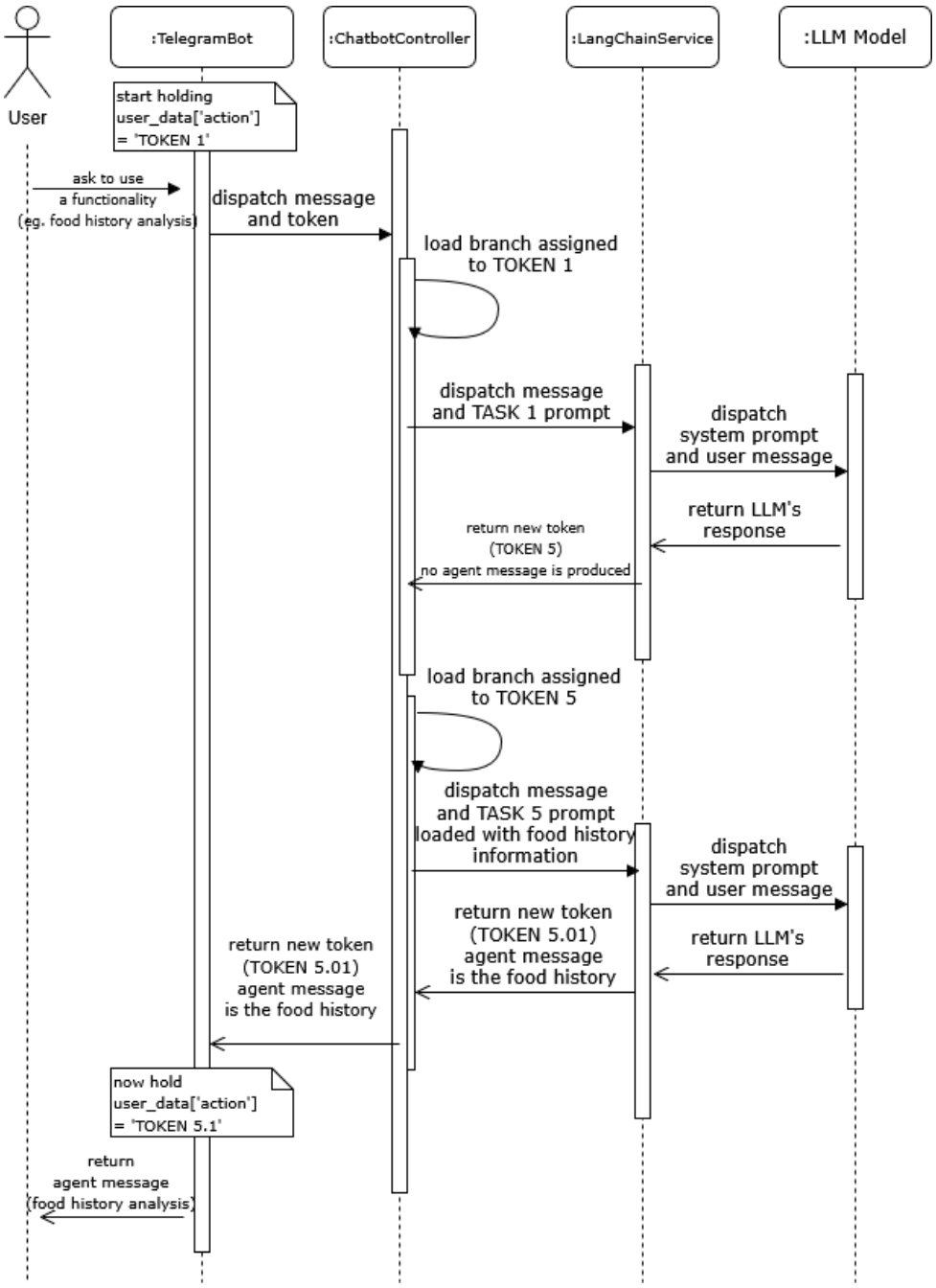


Figure 5.5: Sequence diagram describing the Main Hub control flow when the user ask to invoke a functionality to the agent.

This sequence represents the archetypal interaction between the agent's components, involving multiple exchanges between the controller and the LangChain layer before generating

a response. The internal state of the bot is then updated accordingly to ensure the conversation progresses in alignment with the new state.

An example of real interaction with this functionality appears as follows.

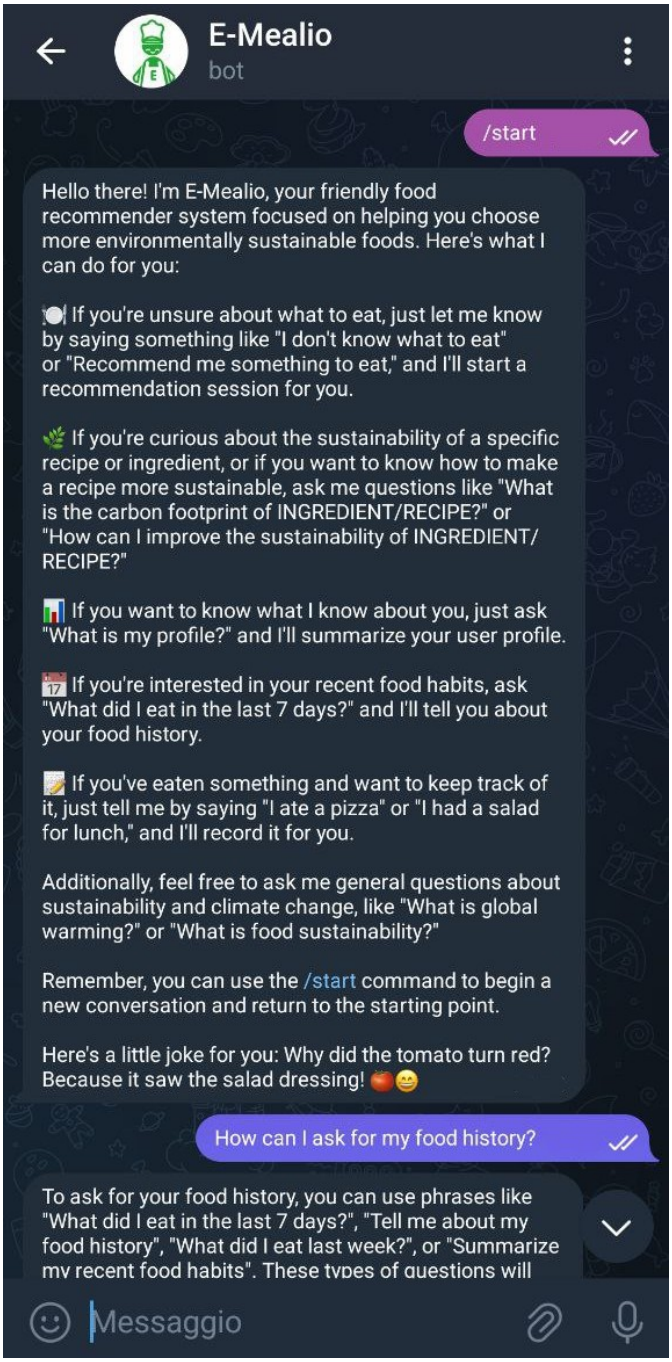


Figure 5.6: Starting a new conversation directs the user to the hub.

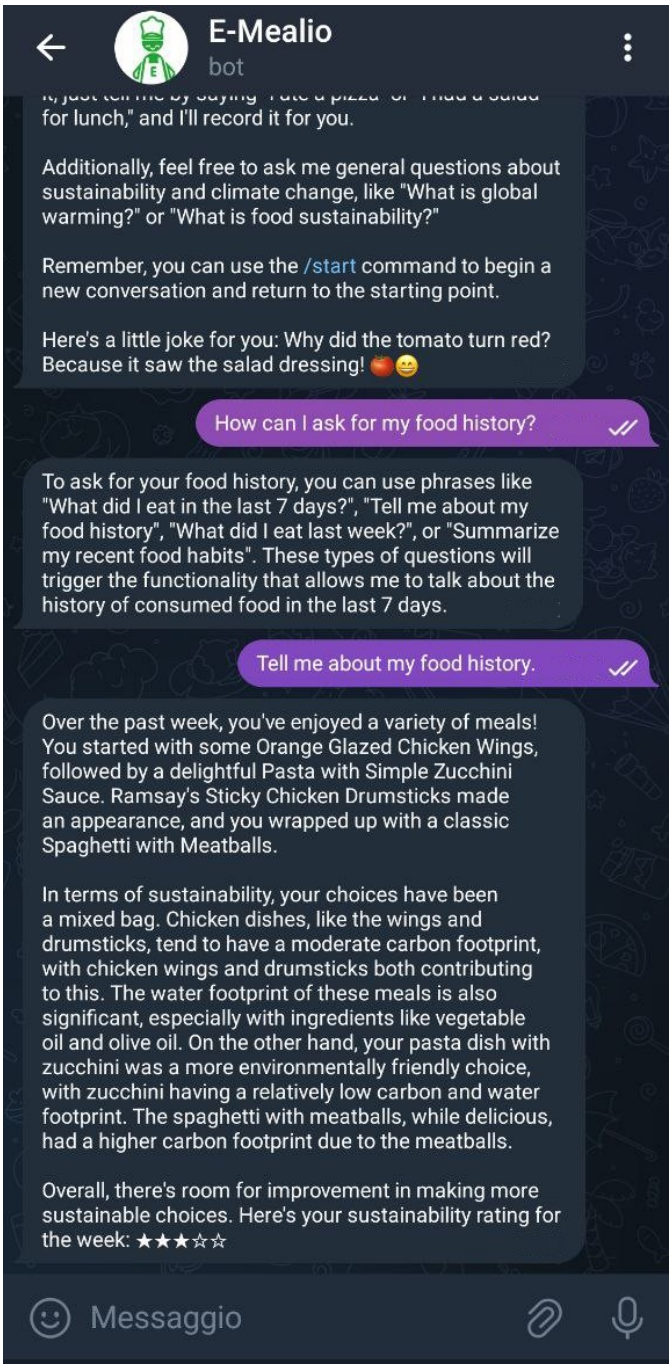


Figure 5.7: The agent responds to questions about its functionalities. While in the hub state, the agent also processes instructions that trigger specific functionalities. In this example, the food history function is requested, and the agent retrieves and provides the relevant information.

5.3.2 Functionality 1: User Data Acquisition

The *User Data Acquisition* function is invoked directly by the Telegram Bot Layer when an unregistered user interacts with the agent.

The functionality is implemented by means of the following prompts[6].

First, the necessity to acquire the user's data is presented by means of the prompt related to the controller state **0**, which is directly passed by the Telegram Bot Layer.

TASK 0 System Prompt: Presenting User Data Acquisition
You are a food recommender system named E-Mealio and have the role of collecting data about the user. User data has the following structure: *USER_DATA_STRUCTURE_TEMPLATE_WITH_MANDATORINESS* Follow these steps to produce the output: - Print the string "TOKEN 0.1", then ask the user to provide you with all the information above. Tell the user that the information can be provided in a easy conversational form.

This prompt simply instructs the LLM model to ask the user for the required information and moves the controller to the **state 0.1**, where the agent is tasked with receiving the user's data.

USER_DATA_STRUCTURE_TEMPLATE_WITH_MANDATORINESS is a placeholder substituted with the following portion of prompt.

[6]In the code, the identifier of these prompts is 0, even though in this document the functionality is related to id 1. This is mentioned to avoid confusion.

System Prompt Portion: User Data Structure Template with Mandatoriness

name: the name of the user. Mandatory.

surname: the surname of the user. Mandatory.

dateOfBirth: the date of birth of the user in the format DD/MM/YYYY. Mandatory.

nation: the nation of the user. If the user provides their nationality instead of a country name, infer the corresponding nation and set it as the nation field. Mandatory.

allergies: a list of foods that the user cannot eat. The possible constraints are [”gluten”, ”crustacean”, ”egg”, ”fish”, ”peanut”, ”soy”, ”lactose”, ”nut”, ”celery”, ”mustard”, ”sesame”, ”sulfite”, ”lupin”, ”mollusk”]. If the user mentions a term related to an allergy item, match it to the closest predefined constraint and use that item as a constraint. Optional.

restrictions: a list of alimentary restrictions derived from ethical choices or religious beliefs. The possible constraints are [”vegan”, ”vegetarian”, ”kosher”]. Optional.

This portion is the same used in other prompts of the *User Data Acquisition* task and subsequently in the *User Profile Recap and Update* task.

The state 0.1 is related to the following prompt.

TASK 0.1 System Prompt: Collecting User Data

You are a food recommender system named E-Mealio and have the role of collecting data about the user. User data has the following structure:

USER_DATA_STRUCTURE_TEMPLATE_WITH_MANDATORINESS

The user could provide you with this information in a conversational form or via a structured JSON.

Follow these steps to produce the output:

- If the user answers something unrelated to this task: Print the string ”TOKEN 0.1”, then write a message that gently reminds the task you want to pursue. If you received both unrelated conversational information and JSON data, specify what mandatory information is missing.

- Otherwise: Print the string ”TOKEN 0.2”, then print a JSON with the information collected until now. Set the absent information as an empty string (for atomic fields) or an empty list (for list fields).

Do not include in the JSON any markup text like ”“‘json“‘”.

Do not make up any other question or statement that is not included in the previous ones.

This prompt instructs the agent to collect the user's data in a JSON format, generating the controller token for the next state, **0.2**. It also instructs the agent to detect unrelated messages, allowing the system to remain in the same state and ask the user for their personal information again if necessary.

Additionally, this prompt is used to re-collect any missing information. In such cases, the already collected data is provided as a JSON string alongside the user's message.

The controller branch related to **state 0.2** is responsible for detecting if all the mandatory

information has been provided, using the following prompt.

TASK 0.2 System Prompt: Evaluating User Data

You are a food recommender system named E-Mealio, and your role is to collect data about the user. User data has the following structure:

USER_DATA_STRUCTURE_TEMPLATE_WITH_MANDATORINESS

The user will provide you with a JSON containing some information about their profile. Follow these steps to produce the output:

- If all the mandatory information is collected, print the string "TOKEN 0.3". Do not write anything else.

- Otherwise, if the user hasn't provided all the mandatory information: Print the string "TOKEN 0.1", then ask them for the remaining information.

This prompt instructs the agent to verify whether all mandatory information is present. When invoked by the controller, the JSON containing the collected data is passed to the prompt replacing the user message. If any information is missing, the agent returns to the information acquisition state **0.1** and explicitly asks the user for the missing details.

If all the necessary information is provided, the persistence state token **0.3** is generated.

When the controller is in **state 0.3**, it presents the collected information using the related prompt and stores the data gathered so far.

The prompt related to **state 0.3** is the following.

TASK 0.3 System Prompt: Presenting Collected User Data

You are a food recommender system named E-Mealio, and your role is to collect data about the user.

The user will provide their profile in a JSON format.

Follow these steps to produce the output:

- Print the string "TOKEN 0.4", then summarize what you have collected in a conversational form. Do not refer to the user's tastes, last interaction, or user id and user nickname. Finally ask for permission to send reminders about the bot's usage if the user forgets to use the system.

Tell also that the reminders will be sent every two days if the user doesn't use the system.

This brief prompt instructs the agent to respond with the collected information and explicitly request the user's consent to receive usage reminders. Here too, the data is passed as a JSON, replacing the user message. This new sub-task is assigned to **state 0.4**, which is triggered by generating the corresponding controller token.

The **state 0.4** invoke the following prompt.

TASK 0.4 System Prompt: Asking Consent For Reminders

You are a simple intent detection system.

You previously asked the user if they want to receive reminders about the bot's usage.

The user will answer with an affirmative (ok, yes, sure, etc.), a negative (no, I don't want, etc.), or ask what kind of reminder you will send.

Follow these steps to produce the output:

- If the user's answer is affirmative, print the string "TOKEN 0.5". Do not write anything else.

- If the user's answer is negative, print the string "TOKEN 0.6". Do not write anything else.

- If the user asks what kind of reminder you will send: Print the string "TOKEN 0.4", then answer that you will send a reminder about the bot's usage every two days if the user doesn't use the system. Then ask again if they want to receive the reminder.

- If the user answers something unrelated to a yes/no question: Print the string "TOKEN 0.4" then explain that you need a yes/no answer and ask again if they want to receive the reminder.

This prompt makes the agent act as an intent recognition system, allowing it to detect all the possible ways to accept or decline the reminder sending offer.

It also instructs the agent on what to say when a non-compliant answer is provided, or when more information about the reminder is requested, allowing the controller to remain in the same state.

Depending on the response, a different state token is produced. Both states manage the persistence of the consent information accordingly. These states do not invoke the LangChain Service again with a new token; they simply provide a static message and return to the hub

by transitioning to **state 1**.

The overall behavior of the User Data Acquisition task can be described by the following finite state machine graph.

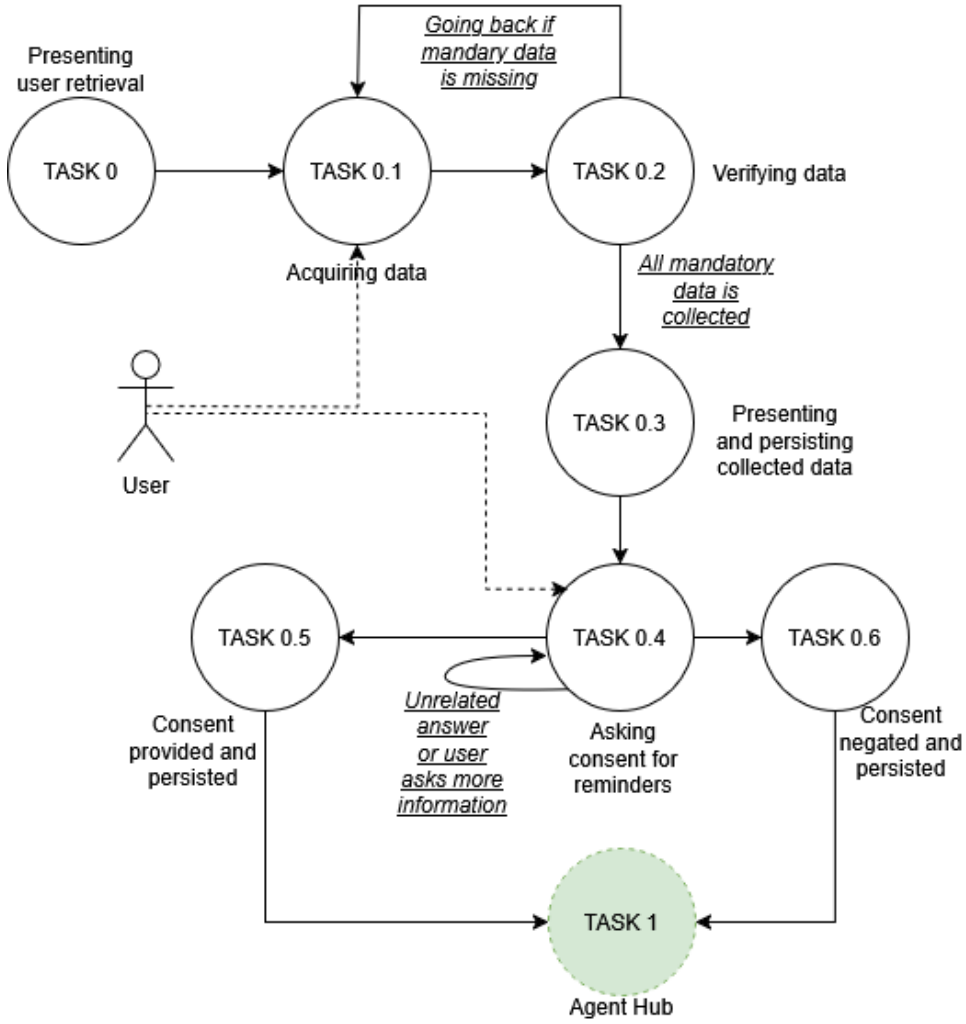


Figure 5.8: Finite State Machine representation of the Agent Data Collection functionality.

An example of interaction between the agent's components when handling this task is illustrated in the following sequence diagram. The example depicts the scenario where the user provides all the necessary information and consents to receiving reminders.

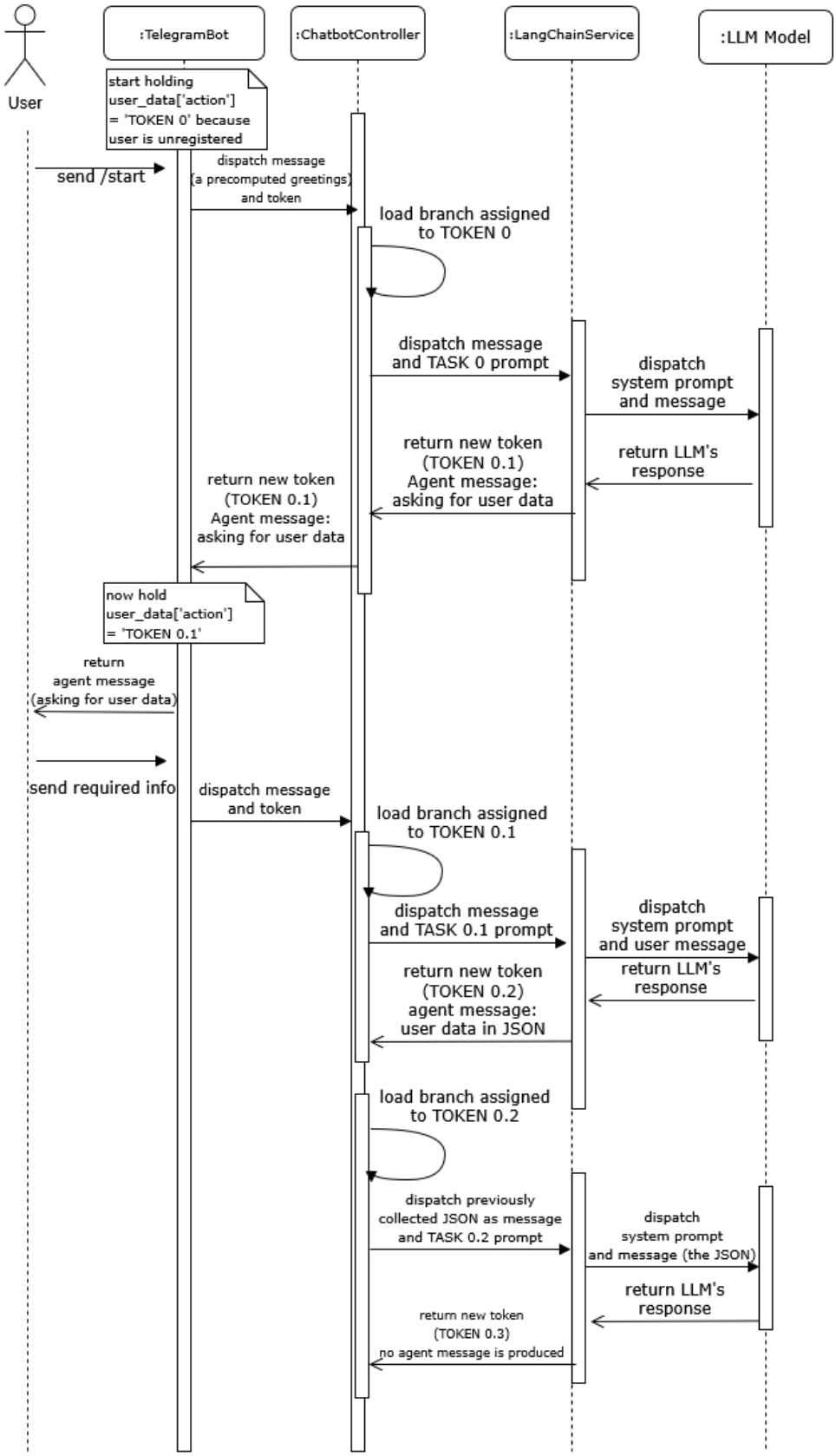
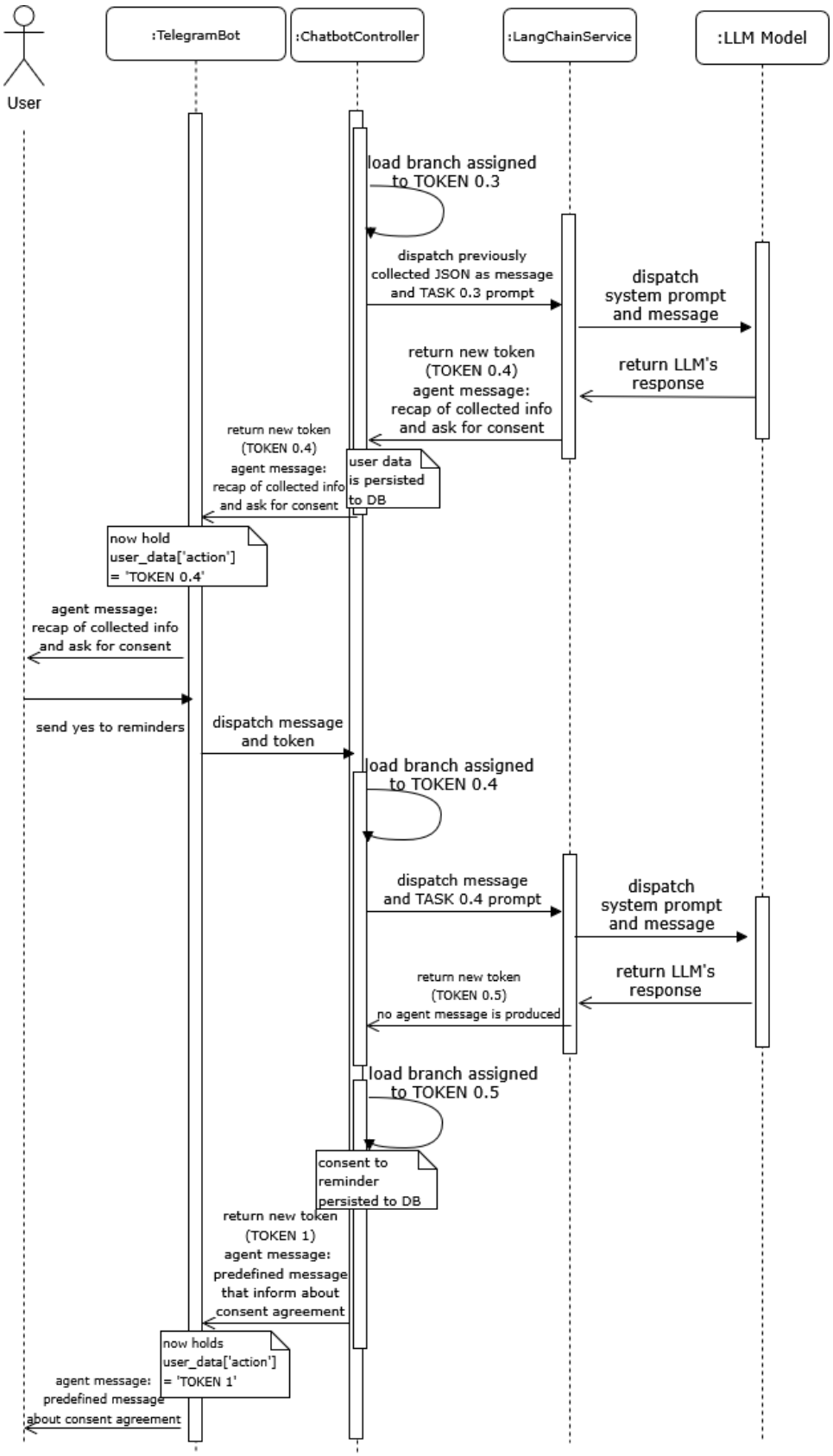


Figure 5.9: Continue in the next page.


User
:TelegramBot
:ChatbotController
:LangChainService
:LLM Model
load branch assigned to TOKEN 0.3
dispatch previously collected JSON as message and TASK 0.3 prompt
dispatch system prompt and message
return LLM's response
return new token (TOKEN 0.4) agent message: recap of collected info and ask for consent
user data is persisted to DB
return new token (TOKEN 0.4) agent message: recap of collected info and ask for consent
now hold user_data['action'] = 'TOKEN 0.4'
agent message: recap of collected info and ask for consent
send yes to reminders
dispatch message and token
load branch assigned to TOKEN 0.4
dispatch message and TASK 0.4 prompt
dispatch system prompt and message
return LLM's response
return new token (TOKEN 0.5) no agent message is produced
load branch assigned to TOKEN 0.5
consent to reminder persisted to DB
return new token (TOKEN 1) agent message: predefined message that inform about consent agreement
now holds user_data['action'] = 'TOKEN 1'
agent message: predefined message about consent agreement

In the sequence diagram, it is clear how the interaction between the controller and the LangChain service continues back and forth until a proper response is generated for the user and sent back through the stack to the Telegram Bot layer.

An example of an actual interaction with the agent performing this task is as follows.

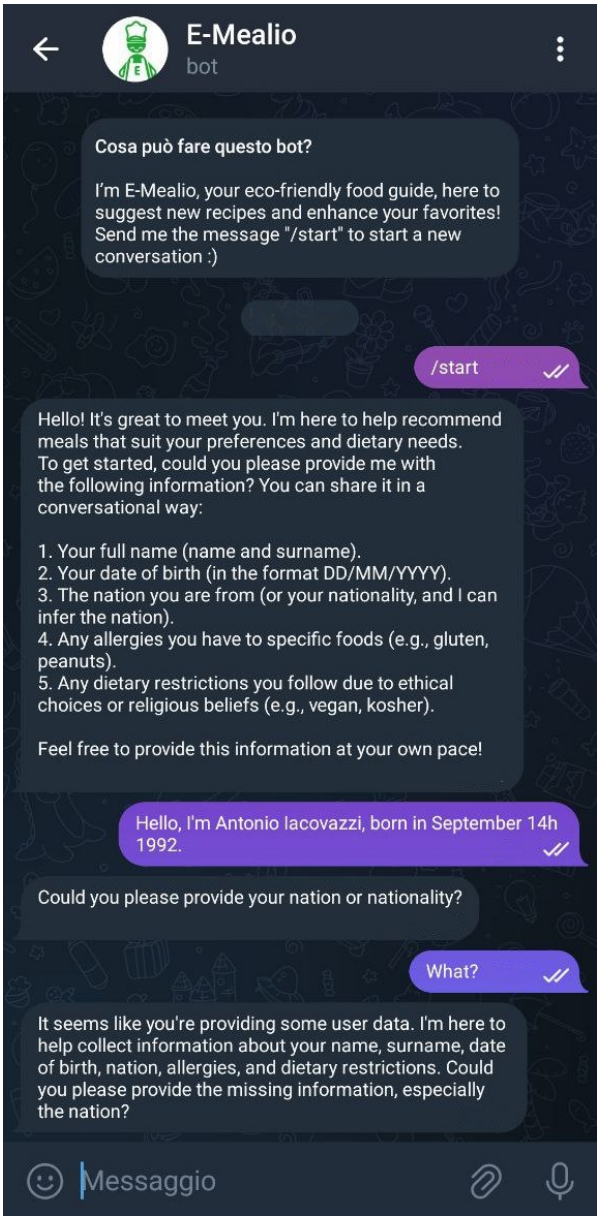


Figure 5.10: When no user profile is found in the database, initiating a new conversation activates the user registration functionality. After receiving the user's response, the agent detects any missing information and prompts the user to complete the required details.

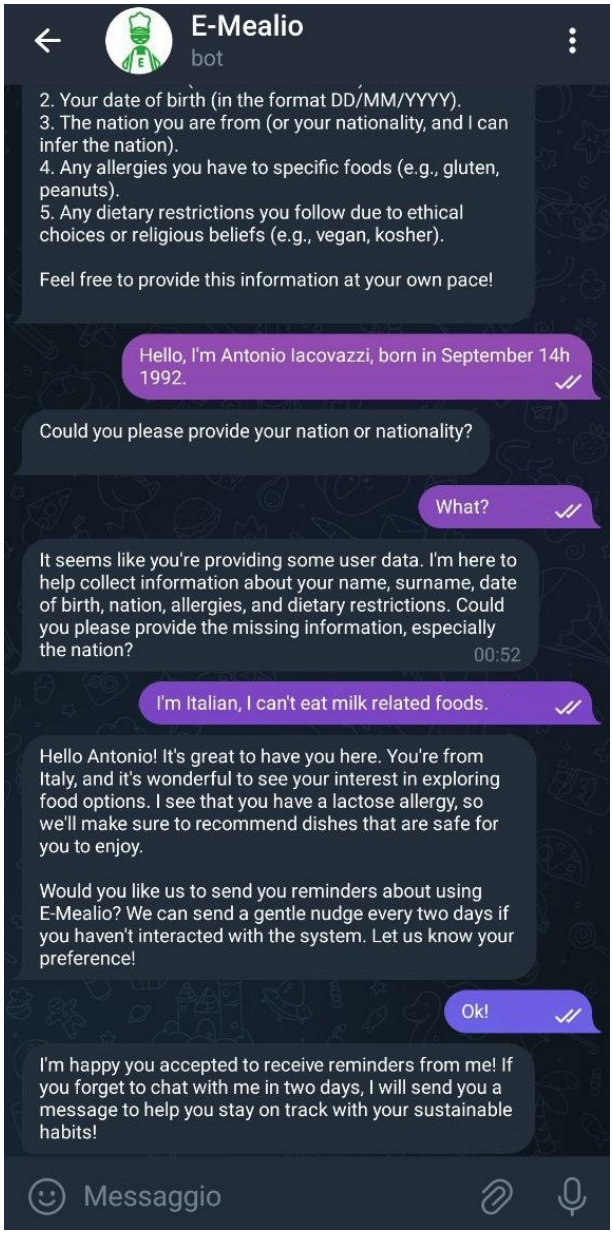


Figure 5.11: Continuing from the previous example, the agent detects additional relevant information beyond the mandatory fields. Once all data is collected, it is stored and presented to the user. The agent then requests consent for reminders. After receiving a response, a predefined message is displayed to inform the user, and the agent transitions to the main hub state.

5.3.3 Functionality 2: Recipe Recommendation

The *Recipe Recommendation* functionality is the core capability of the E-Mealio agent. While the actual computation of recommended recipes is handled by the service described in Subsection 5.2.6, the process of gathering user preferences and presenting the extracted data is managed through a series of LLM prompts, orchestrated by the agent controller.

First, when a user triggers the recipe recommendation functionality while interacting with the main hub, the system transitions to **state 2**. This state corresponds to the first prompt in the recommendation process, which is responsible for collecting all necessary information required to invoke the recommendation service. The prompt responsible for this data acquisition is the following.

TASK 2 System Prompt: Asking Recipe Recommendation Data

You are a food recommender system named E-Mealio, and you have to collect the information needed in order to suggest a meal.

The meal suggestion data is structured as follows:

mealType: the type of meal. The possible values are [”Breakfast”, ”Lunch”, ”Dinner”, ”Break”]. Mandatory.

recipeName: the name of the recipe. Keep it empty.

sustainabilityScore: the sustainability score of the recipe. Keep it empty.

ingredients_desired: a list of ingredients that the user would like to have in the recipe. Optional.

ingredients_not_desired: a list of ingredients that the user would not like to have in the recipe. Optional.

cookingTime: the time that the user has to cook the meal. The possible values are [”short”, ”medium”, ”not_relevant”]. Optional.

healthiness: the level of healthiness that the user wants to achieve. The possible values are [”yes”, ”not_relevant”]. Optional.

You can infer the kind of meal by using information about the time of day with the following rules:

This morning → Breakfast

Today → Lunch

This noon → Lunch

This evening → Dinner

Tonight → Dinner

Snack → Break

Something quick (or similar) → Break

The user may provide you with the information about the meal in a conversational form and also via a structured JSON. Conversational information and JSON can be provided together.

Follow these steps to produce the output:

- Print the string ”TOKEN 2.05”, then print a JSON with the information collected up to that point. Include every field, but set the absent information as an empty string (for atomic fields) or an empty list (for list fields). Do not ask any additional questions or make any other statements that are not part of the previous instructions.

Do not include in the JSON any markup text like ”“‘json“‘”.

This prompt follows a similar strategy to the one seen for **TASK 0.1**. It defines all the necessary information along with their respective constraints and a small set of rules to derive

the required details directly from the user's message.

It is important to note that when this prompt is invoked, the corresponding user message it processes is actually the recipe recommendation request sent by the user to the hub state. The hub, reacting to the user's request, produces the state token **2**, which then triggers this prompt while passing along the recipe recommendation request. For this reason, it is always ensured that a proper request is sent, and no additional validation is performed before generating the output.

The prompt generates a JSON containing all the data required to invoke the recommendation service, including fields that are not strictly necessary for the recommendation itself but are required when the service is called by the recipe improvement functionality. This approach ensures that all expected fields are present in the input, even if left blank.

Finally, the produced next state token is **2.05**, an intermediate state that ensures the presence of the meal type in the generated JSON. This is necessary because the original user message may not explicitly mention the required meal type needed to extract the recipe from the appropriate subset.

This check is performed by the corresponding controller branch, which verifies the existence of the meal type in the JSON. If it is missing, a predefined message is sent back to the user, saving the previously collected information and returning to **state 2**. At this point, the **state 2** prompt processes the user message again, this time incorporating the JSON with the previously collected information, allowing it to finally assemble the complete JSON.

If the meal type is properly assigned, the controller branch associated with **state 2.05** transitions to **state 2.10**, which invokes the recipe recommendation service, passing along all the gathered information.

Once the service generates a response, the resulting JSON containing the recommended recipe is passed to the prompt associated with **state 2.10**. This prompt is then forwarded to the

LangChain service, enabling its memory and allowing the system to track the conversation. As a result, the agent can adapt its responses based on the entire conversation context. This tracked portion of the conversation is utilized by **state 2.20**, which is the token produced by the recommendation presentation prompt.

The prompt is as follows.

TASK 2.10 System Prompt: Presenting Recommended Recipe

You are a food recommender system with the role of helping users choose more environmentally sustainable foods.

Your role is to suggest the following recipe {suggestedRecipe} given the following constraints {mealInfo} and the user profile {userData}.

Follow these steps to produce the output:

- Print the string ”TOKEN 2.20”, then explain why the suggested recipe is a good choice for the user, focusing on the environmental benefits it provides.

If there are constraints in the ”removedConstraints” field of the suggested recipe, explain that those constraints were removed in order to provide a plausible suggestion that otherwise would not be possible.

Do not mention missing constraints if the ”removedConstraints” field is empty.

Use information about the carbon footprint and water footprint of the ingredients to support your explanation, but keep it simple and understandable.

Refer to numbers of CFP and WFP, but also provide an idea of whether those values are good or bad for the environment.

The sustainability score is such that the lower the value, the better the recipe is for the environment. It ranges from 0 to 1.

Do not provide it explicitly but use a Likert scale to describe it printing from 0 to 5 stars (use ascii stars, using black stars as point and white stars as filler).

Provide the URL that redirects to the recipe instructions.

Then, highlightning the following part using an emoji: Persuade the user to accept the suggestion by explicitly asking if they want to eat the suggested food. Explain also that the response will be saved in the user's profile for track the consumption of the recipe and allow the evaluation of the user's sustainability habits.

Write an empty row for better readability before the final part.

Finally close the message also by suggesting the user to ask more details about the recipe or the ingredients if they want.

Be succinct, using up to 200 words. Maintain a respectful and polite tone.

In this prompt, it is possible to observe the presence of placeholders enclosed in curly brackets, which are dynamically replaced when the prompt is loaded by the controller. The *suggestedRecipe* placeholder contains the data regarding the recipe to be suggested, *mealInfo* holds the constraints derived from the user's request during the previous phase, and *userData* includes the user's general information along with their allergies and dietary restrictions.

When executed, the prompt generates the next state token **2.20** and utilizes the gathered information to construct a well-formed suggestion for the user, following the provided instructions. This process exemplifies Retrieval-Augmented Generation (RAG), a key technique that enables the agent to remain flexible in its responses while ensuring the provided information originates from verified sources extracted through a structured procedure.

It also possible that the recommendation service is completely unable to extract a proper recipe to suggest, in that case the agent moves toward a different state, the 2.10.1. This state is related to the following prompt.

TASK 2.10.1 System Prompt: Presenting No Recommended Recipe

You are a food recommender system with the role of helping users choose more environmentally sustainable foods.

Your role is to suggest a recipe that respects the constraints {mealInfo} and the user profile {userData}, but unfortunately, no recipe that meets the constraints was found.

Follow these steps to produce the output:

- Print the string ”TOKEN 1”, then explain why no recipe was found and suggest that the user remove some constraints in order to obtain a recipe. Conclude by inviting the user to ask for a new suggestion or start a new conversation.

Be succinct, using up to 150 words, and don’t provide further hints about possible options. Maintain a respectful and polite tone.

In this case, the placeholders are populated to construct a well-reasoned explanation for the agent’s failure, suggesting that the user relax or modify some constraints.

The subsequently generated state token is **1**, indicating that the agent returns to the hub state, ready to handle new requests.

Returning to the case where a suggestion is successfully generated, the next state token **2.20** transitions the agent into a state where a discussion about the recommended recipe can take place, leveraging the previously enabled LangChain memory system.

The prompt related to this state is as follows.

TASK 2.20 System Prompt: Recipe Suggestion Conversation and Confirmation

You are a food recommender system with the role of helping users choose more environmentally sustainable foods.

You will receive the message history about a food suggestion previously made by you.

Follow these steps to produce the output:

- If the user asks a question about the food suggestion previously provided: Print the string "TOKEN 2.20", then answer the question and persuade the user to accept the suggestion by explicitly asking if they want to eat the suggested food.

- If the user likes the recipe and/or accepts the suggestion, print the string "TOKEN 2.30". Do not write anything else.

- If the user doesn't like the recipe and/or declines the suggestion, print the string "TOKEN 2.40". Do not write anything else.

- If the user asks for a new food suggestion, print the string "TOKEN 2.50". Do not write anything else.

- If the user asks or tells something completely unrelated to the current suggestion, sustainability, or asks about another recipe:

Print the string "TOKEN -1", then write a message where you tell the user that is a question about another topic. Finally softly invite the user to start a new conversation.

Always maintain a respectful and polite tone.

The prompt is loaded with previous messages exchanged between the user and the agent using LangChain memory, eliminating the need to explicitly include the conversation history in the prompt.

The prompt is instructed to recognize if the user's message is intended to discuss the proposed recipe. In that case, the produced state remains **2.20**, allowing the agent to stay in the

same state while expanding the conversational context memory. Alternatively, the prompt can recognize the user's intention to accept, reject, or request a new suggestion, producing specific states for each case (**2.30**, **2.40**, and **2.50**, respectively).

In states **2.30** and **2.40**, the controller transitions into a branch where it can invoke the persistence service, enabling the system to register the user's decision and then clearing the memory before moving back to the hub. Conversely, if the user requests a new suggestion, the branch related to **2.50** resets the memory, saves the currently suggested recipe as temporarily rejected, reloads the initial user request that triggered the recommendation process, and returns to the hub to initiate a new recommendation. The entire process is then executed again, starting from **state 2**, this time suggesting a different recipe.

Finally, the **2.20** prompt is designed to recognize completely unrelated messages. In such cases, it produces the state token **-1**, which triggers a controller branch that resets the memory and transitions back to the hub via token **1**, making the agent ready to handle new requests.

The overall behavior of the Recipe Recommendation task can be summarized by the following finite state machine graph.

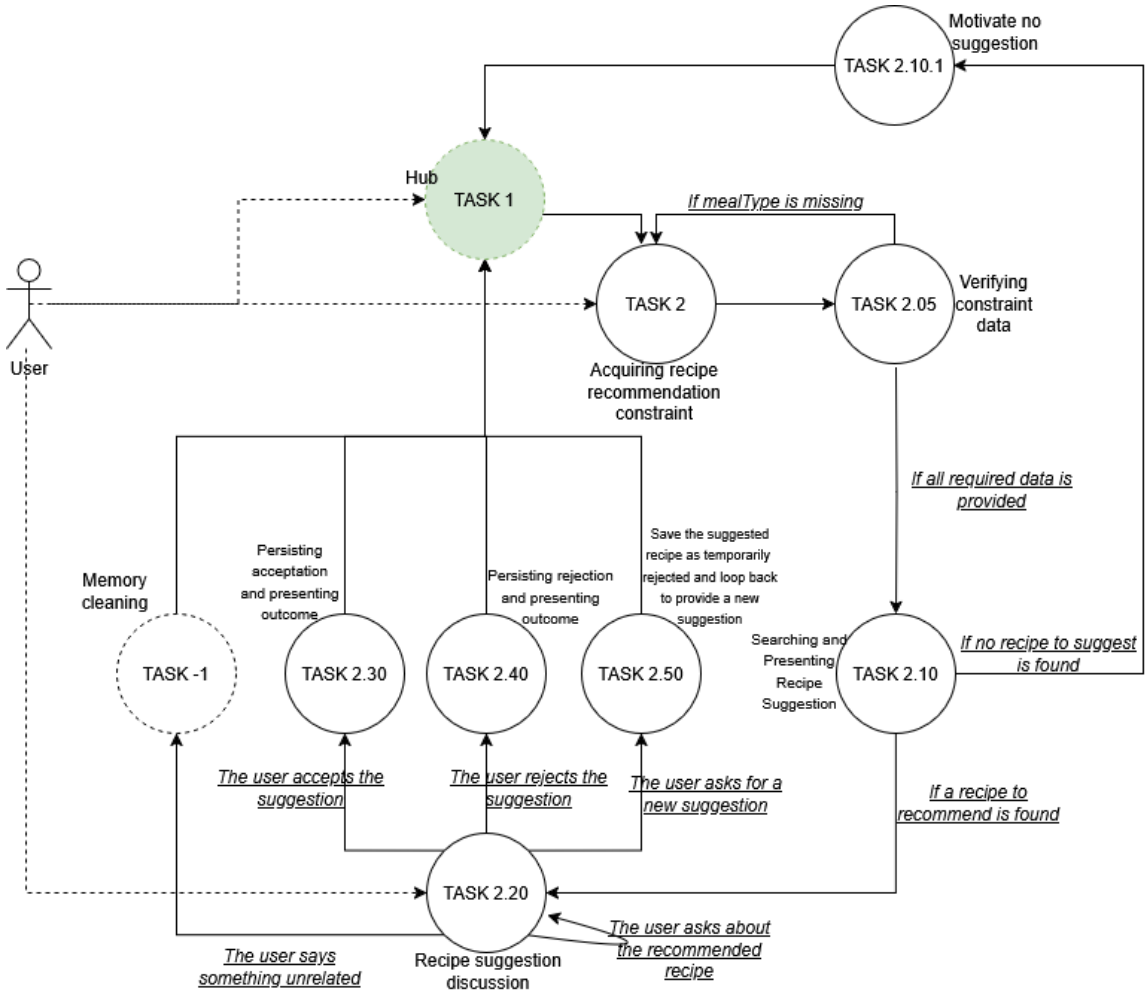


Figure 5.12: Finite State Machine representation of the Recipe Suggestion functionality.

An example of interaction between the agent's components when handling this task is illustrated in the following sequence diagram. The example depicts a scenario where the user requests a recipe suggestion without specifying the meal type. The system then prompts the user for this missing detail, after which a recipe is suggested. The user proceeds to ask a question about the recipe and ultimately accepts the suggestion.

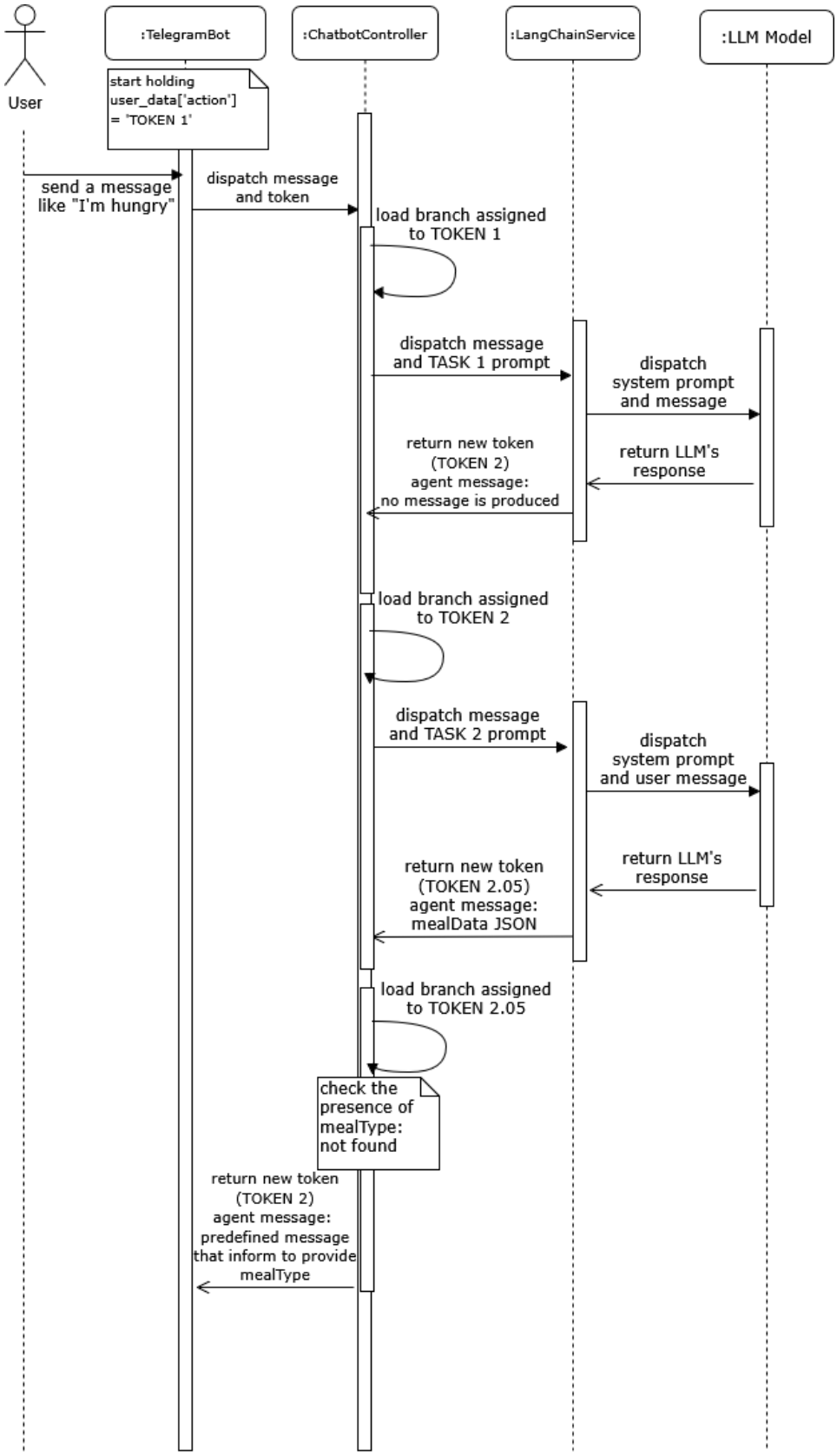


Figure 5.13: Continue in the next page.

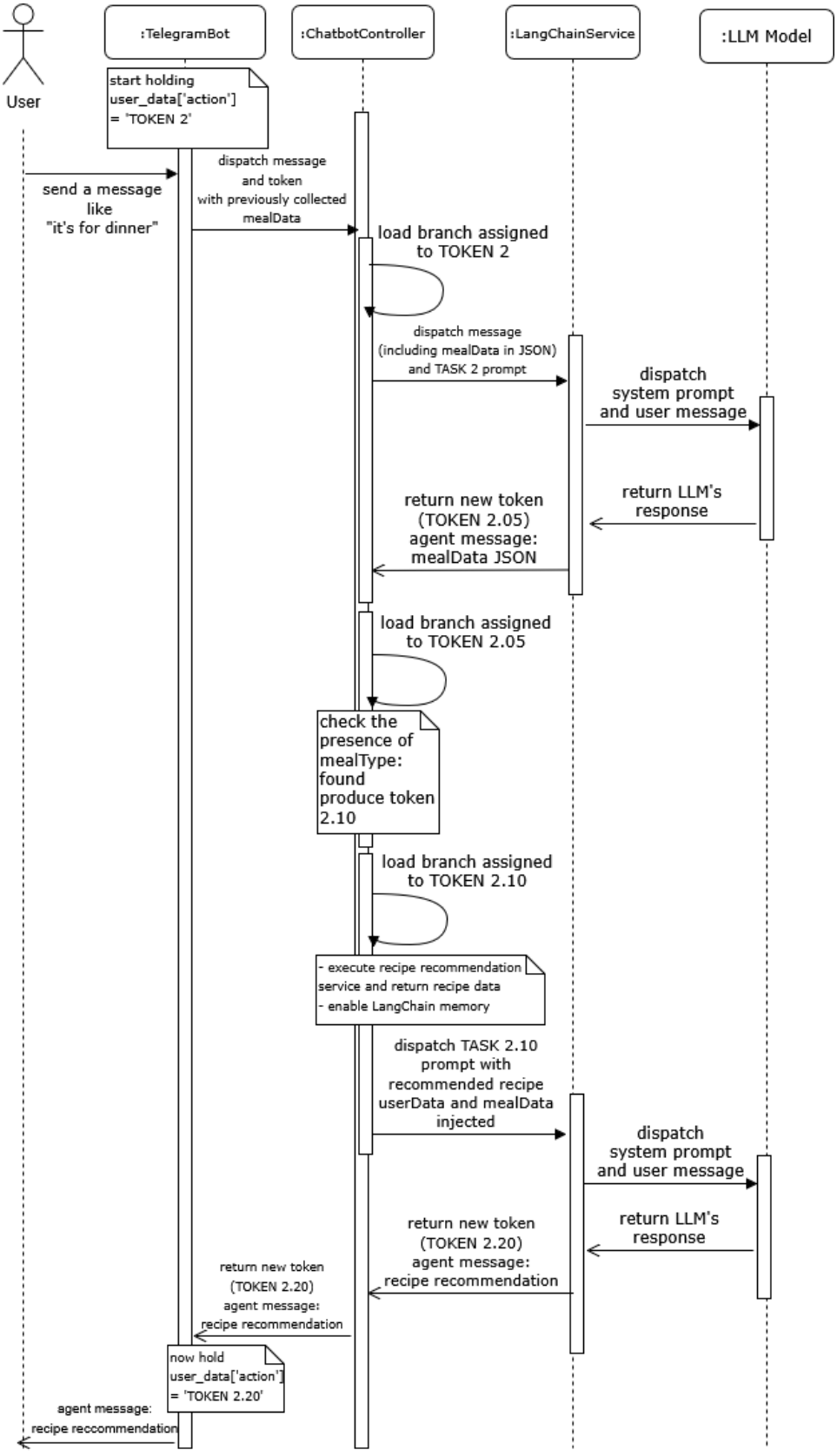
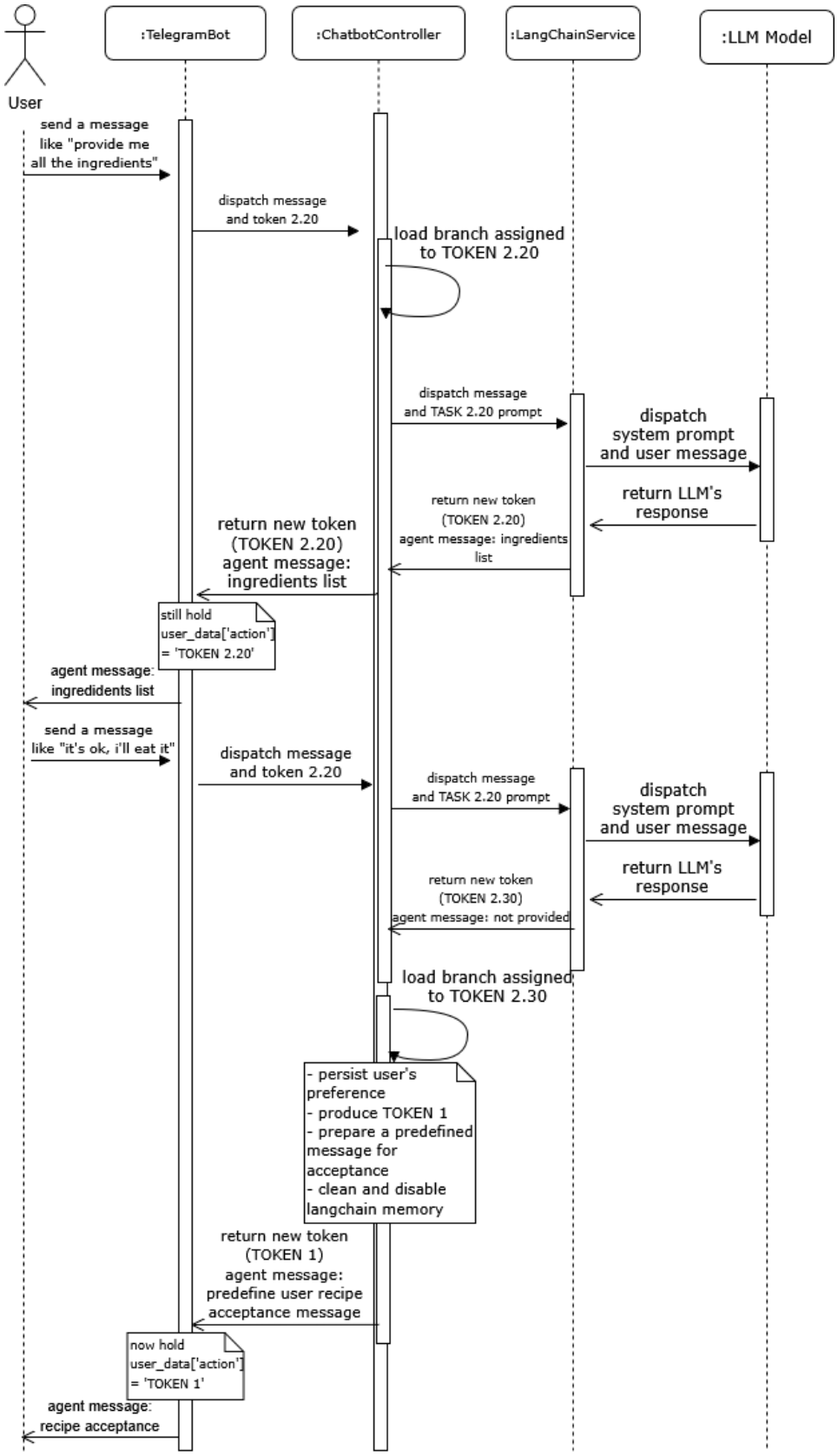


Figure 5.14: Continue in the next page.


User
:TelegramBot
:ChatbotController
:LangChainService
:LLM Model
send a message like "provide me all the ingredients"
dispatch message and token 2.20
load branch assigned to TOKEN 2.20
dispatch message and TASK 2.20 prompt
dispatch system prompt and user message
return LLM's response
return new token (TOKEN 2.20) agent message: ingredients list
return new token (TOKEN 2.20) agent message: ingredients list
still hold user_data['action'] = 'TOKEN 2.20'
agent message: ingredidents list
send a message like "it's ok, i'll eat it"
dispatch message and token 2.20
dispatch message and TASK 2.20 prompt
dispatch system prompt and user message
return LLM's response
return new token (TOKEN 2.30) agent message: not provided
load branch assigned to TOKEN 2.30
- persist user's preference
- produce TOKEN 1
- prepare a predefined message for acceptance
- clean and disable langchain memory
return new token (TOKEN 1) agent message: predefine user recipe acceptance message
now hold user_data['action'] = 'TOKEN 1'
agent message: recipe acceptance

An example of an actual interaction with the agent performing this task is as follows.

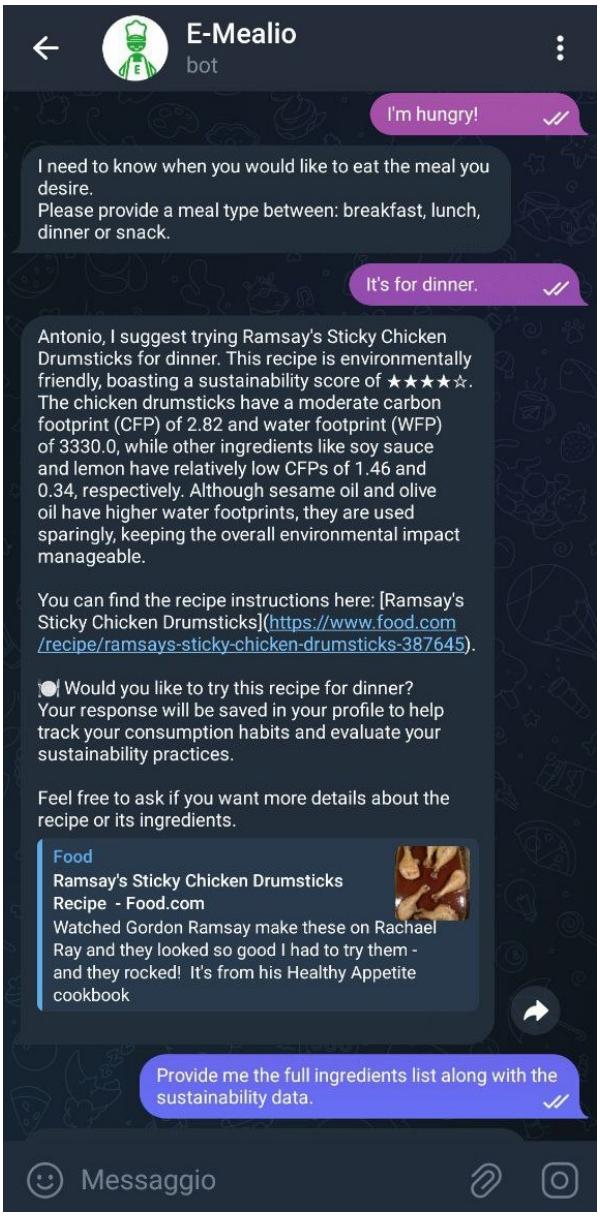


Figure 5.15: The user requests a meal recommendation by specifying a meal type. The agent responds with a suitable recipe suggestion.

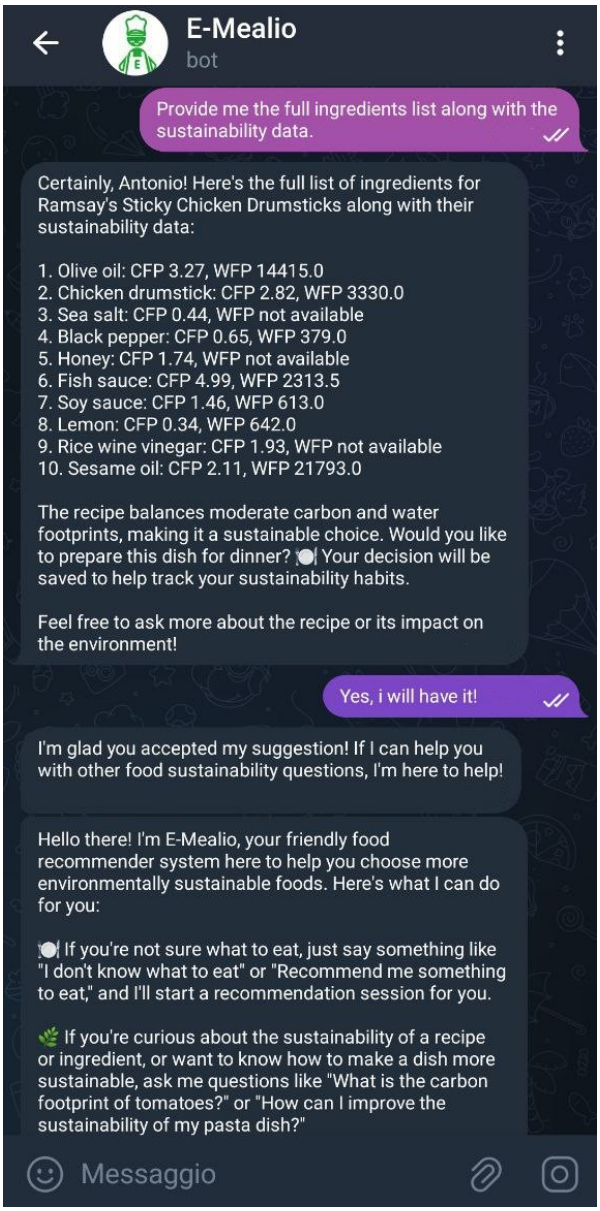


Figure 5.16: Continuing from the previous example, the user briefly discusses the recommended recipe before accepting it. Once the discussion ends, the agent returns to the hub.

5.3.4 The Sustainability Expert Sub-Hub

The agent's hub, when required to handle both the recipe sustainability improvement task and the sustainability expert task, produces the token **3**, which transitions to the so-called *Sustainability Expert Sub-Hub*.

The introduction of a sub-hub was necessitated by difficulties encountered when attempting to separate the two tasks within the main hub using only prompt-based distinctions. In an earlier version of the agent, these two functionalities were distinct within the main hub and associated with separate tokens. However, likely due to the complexity of the context, the LLM frequently confused one functionality with the other.

This issue led to the current solution: an intermediate hub dedicated to distinguishing whether a user request pertains to improving a recipe's sustainability or seeking general sustainability information.

By this way, the main hub only needs to determine whether the request pertains to sustainability. The dedicated prompt is then responsible for classifying the request into one of the two functionalities, producing the token **3.10** if it relates to recipe sustainability improvement or **6** if it concerns a general sustainability inquiry.

The prompt used to implement the sustainability expert sub-hub is as follows.

TASK 3 System Prompt: Sustainability Expert Sub-Hub

The user will provide you with a sentence containing a recipe, a food item, or a sustainability/environmental concept.

Follow these steps to produce the output:

- You have to distinguish between two types of questions:

1) If the question is about the sustainability of a recipe, ingredients, or an environmental concept, print the string "TOKEN 6". Do not write anything else.

2) If the question is about the sustainability improvement of a recipe, print the string "TOKEN 3.10". Do not write anything else.

How to distinguish between the two types of questions:

- A question of type 1 is usually a general question about the overall sustainability of recipes or foods, asked as an informative question.

- A question of type 2 is usually about the sustainability improvement of a recipe or food, or a statement in which the user expresses interest in eating a recipe.

The prompt first instructs the model on the task it needs to execute and then provides additional details to help distinguish between the two functionalities.

The interaction of this functionality with the rest of the agent will be analyzed when the two related functionalities are discussed.

5.3.5 Functionality 3: Recipe Sustainability Improvement

Another core functionality of the E-Mealio agent is its ability to guide the user in improving the sustainability of a recipe by suggesting ingredient substitutions here called *Recipe Sustainability Improvement.* As described in Subsection 5.2.6, the recipe improvement process is carried out by invoking the recipe recommender service. This service is provided with information about the sustainability score to surpass, the list of ingredients, and the recipe name, which is used to find the closest alternative recipe that can serve as a reference for ingredient substitutions. Besides the strict extraction of the proposed improved recipes, this functionality, similar to the recipe recommendation feature, must also handle the acquisition of the base recipe, evaluate whether all the required information is available, and manage the acceptance or the rejection of the improved version.

The starting point of the recipe improvement functionality is the prompt loaded when the agent is in **state 3.10**, which is triggered by the sustainability expert sub-hub upon a user's request for recipe improvement.

The prompt is as follows.

TASK 3.10 System Prompt:Base Recipe Data Collection

You are a food recommender system with the role of helping users improve the sustainability of a given recipe.

You will receive an improvement request containing a recipe expressed as a list of ingredients and, optionally, the recipe name. The recipe data can be provided in a conversational form or via a structured JSON. They can also be provided together.

By extracting the information in the user message and in the JSON (if available), you will provide a JSON with the following structure:

name: the recipe name provided by the user, or derive it from the ingredients if not provided.

ingredients: the list of ingredients for the recipe exactly as provided by the user. Do not make up any ingredients. The ingredients list is usually provided by the user as a list of ingredients separated by commas. Populate this field as a list of strings.

This JSON will be used in the next task for the improvement of the recipe.

Follow these steps to produce the output:

- If the ingredients are provided, print the string ”TOKEN 3.20” followed by the JSON.

- Otherwise: Print the string ”TOKEN 3.15” followed by the JSON, then write a message telling the user that the recipe with the given name is not processable without a proper ingredient list and ask them to provide it.

Do not include in the JSON any markup text like ”“‘json“‘“.

This prompt is used to collect the mandatory information about the base recipe. Specifically, to accurately determine the sustainability score of the base recipe, the ingredient list must be provided beforehand. This prevents the system from approximating the base recipe

by searching for a recipe solely based on its name.

The prompt instructs the LLM to verify whether a proper list of ingredients has been provided and to construct a JSON representation of the base recipe. If all the mandatory information is present, the token **3.20** is generated along with the constructed JSON. Otherwise, the token **3.15** is generated, accompanied by a JSON containing the data collected so far and a message informing the user that the ingredient list is mandatory.

In **state 3.15**, the agent waits to receive the ingredient list. If the list is correctly provided, the agent transitions to **state 3.10**; otherwise, if an unrelated message is received, it remains in **state 3.15**.

The related prompt is as follows.

TASK 3.15 System Prompt: Acquiring Ingredients

You are a food recommender system with the role of helping users improve the sustainability of a given recipe. You previously asked the user to provide the ingredients of the recipe.

Follow these steps to produce the output:

- If the user provides the ingredients list, print the string ”TOKEN 3.10”.

- If the user provides something unrelated to this task:

Print the string ”TOKEN 3.15”, then write a message where you simply remind them of your current purpose.

When the required ingredient list is provided, the prompt, by simply generating the next state token, forces the controller to forward the received message, along with the previously generated JSON, managed by the controller, to the prompt responsible for base recipe data acquisition in **state 3.10**.

Once the mandatory information is acquired, the agent transitions to **state 3.20**, where

it invokes the recipe improver service described in Subsection 5.2.7. If an improved recipe is found, the agent loads the prompt related to **state 3.20**, which presents the improved recipe while highlighting the necessary ingredient substitutions. This prompt is invoked with LangChain memory enabled, allowing the user to engage in a discussion about the improved version, similar to the recommendation functionality. Conversely, if no improved recipe is found (e.g., when the user's restrictions are too strict to allow for viable alternatives), the prompt related to **state 3.20.1** is loaded. This prompt is responsible for explaining the reason behind the agent's failure to provide an improved recipe.

The prompt related to **state 3.20** is as follows.

TASK 3.20 System Prompt: Presenting Recipe Improvements

You will receive two recipes as JSON structures: the base recipe {baseRecipe} and the sustainably improved recipe {improvedRecipe}.

Your task is to suggest to the user what to substitute in the base recipe in order to obtain the improved recipe.

Follow these steps to produce the output:

- Print the string "TOKEN 3.30", then write a message explaining, using the provided carbon footprint data and the differences in the ingredients, why the improved recipe is a better choice from an environmental point of view. Provide instructions on how to substitute the ingredients in the base recipe to obtain the improved recipe. Be clear on what ingredients to remove and what to add. Use information about the carbon footprint and water footprint of the ingredients to support your explanation, but keep it simple and understandable. Refer to numbers of CFP and WFP, but also provide an idea of whether those values are good or bad for the environment.

The sustainability score is such that the lower the value, the better the recipe is for the environment. It ranges from 0 to 1. Do not provide it explicitly but use a Likert scale to describe it printing from 0 to 5 stars (use ascii stars, using black stars as point and white stars as filler).

Then, highlight this request using an emoji, ask if the user wants to accept the improvement. Explain also that the response will be saved in the user's profile for track the consumption of the recipe and allow the evaluation of the user's sustainability habits.

Write an empty row for better readability before the final part.

Close the message also by suggesting the user to ask more details about the recipe or the ingredients if they want.

Be succinct, using up to 200 words. Maintain a respectful and polite tone.

The prompt is loaded with both the previously acquired base recipe and the retrieved improved version. It then instructs the LLM to generate the necessary ingredient substitutions required to enhance the base recipe, obtaining a more sustainable version. Finally, it ask the user to decide whether to accept the suggested improvements, following the same approach as the basic recommendation functionality. Additionally, the prompt produces the **3.30** state token, transitioning the agent into a state where the user can discuss the improved recipe, as well as accept, reject, or request an alternative suggestion.

The prompt related to **state 3.20.1** is the following.

TASK 3.20.1 System Prompt: Presenting Failed Recipe Improvement

You are a food recommender system with the role of helping users choose more environmentally sustainable foods. Your role is to suggest an ingredient substitution to improve the base recipe {baseRecipe} given the user profile {userData}, but unfortunately, no recipe that meets the user constraints was found.

Follow these steps to produce the output:

- Print the string "TOKEN 1", then explain that no recipe was found given the current restrictions provided in the user profile, suggesting modifying the profile by removing some of them. Conclude by inviting the user to ask for a new suggestion or start a new conversation.

Be succinct, using up to 150 words, and don't provide further hints about possible options. Maintain a respectful and polite tone.

This prompt, similar to the one associated with **state 2.10.1**, simply informs the user of the reasons why an improvement suggestion could not be provided. After presenting this information, the agent transitions back to the hub state by producing the token **1**.

The prompt related to **state 3.30**, related to improvement discussion and acceptance is the following.

TASK 3.30 System Prompt: Improved Recipe Discussion

You are a food recommender system with the role of helping users choose more environmentally sustainable foods. You will receive the message history about a sustainability improvement of a recipe previously made by you.

Follow these steps to produce the output:

- If the user asks questions about the recipe improvement previously provided: Print the string ”TOKEN 3.30”, then answer to the question, and persuade them to accept the consumption of the improved recipe.
- If the user accepts the improvement suggestion, print the string ”TOKEN 3.40”. Do not write anything else.
- If the user declines the improvement suggestion, print the string ”TOKEN 3.50”. Do not write anything else.
- If the user asks for a new improvement suggestion, print the string ”TOKEN 3.60”. Do not write anything else.
- If the user asks or tells something unrelated to the improvement and/or sustainability: Print the string ”TOKEN -1”, then write a message where you tell the user that is a question about another topic. Finally softly invite the user to start a new conversation. Maintain a respectful and polite tone.

This prompt, similarly to the one related to **state 2.20**, is loaded with the LangChain memory enabled, allowing the user to ask questions about the previously provided recipe improvement. In such cases, it will produce the state token **3.30**, remaining in the same state while providing a response to the asked question. The prompt also recognizes if the user has accepted or rejected the proposed improvement, or if they want a new suggestion, essentially replicating the behavior observed in the recommendation functionality by producing state tokens **3.40**, **3.50**, and **3.60**. In each case, an interaction with the persistence layer occurs, saving the acceptance, rejection, or temporary rejection of the proposed improved recipe.

When another improvement is requested, the original user request for an improved recipe is sent back to the **3.10** state, restarting the improvement process.

When the recipe improvement is accepted or rejected, a proper predefined message is sent

back to the user, producing also the state token **1**, moving the agent back to the main hub.

It can also recognize if the user asks for something completely unrelated. In such cases, the state token **-1** is produced, forcing a memory reset and returning the agent to the hub.

The overall behavior of the Recipe Sustainability Improvement task can be summarized by the following finite state machine graph.

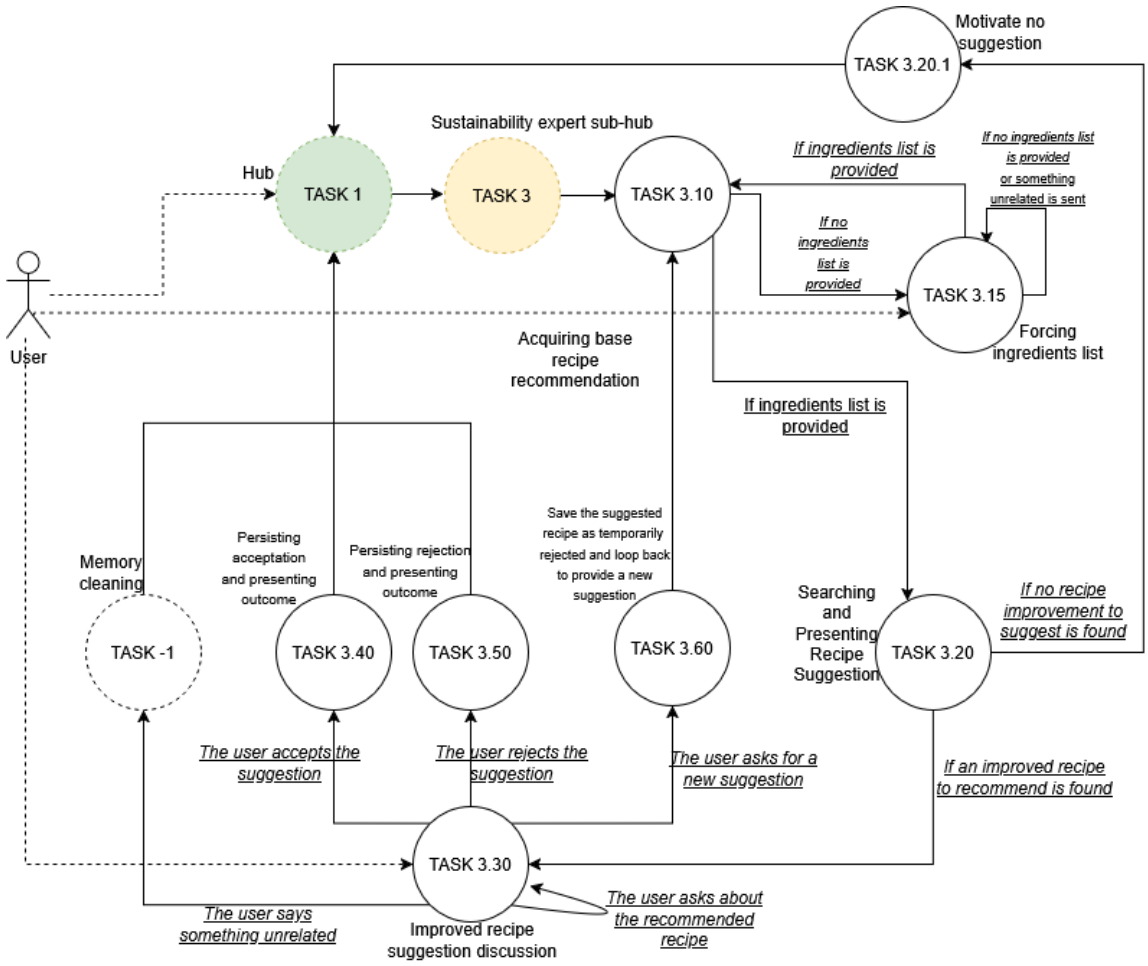


Figure 5.17: Finite State Machine representation of the Recipe Sustainability Improvement functionality.

The sequence diagram below illustrates an interaction where the user requests a recipe improvement but does not provide the ingredient list. The system prompts the user for the missing list, suggests an improvement, and the user ultimately reject it after a brief discussion.

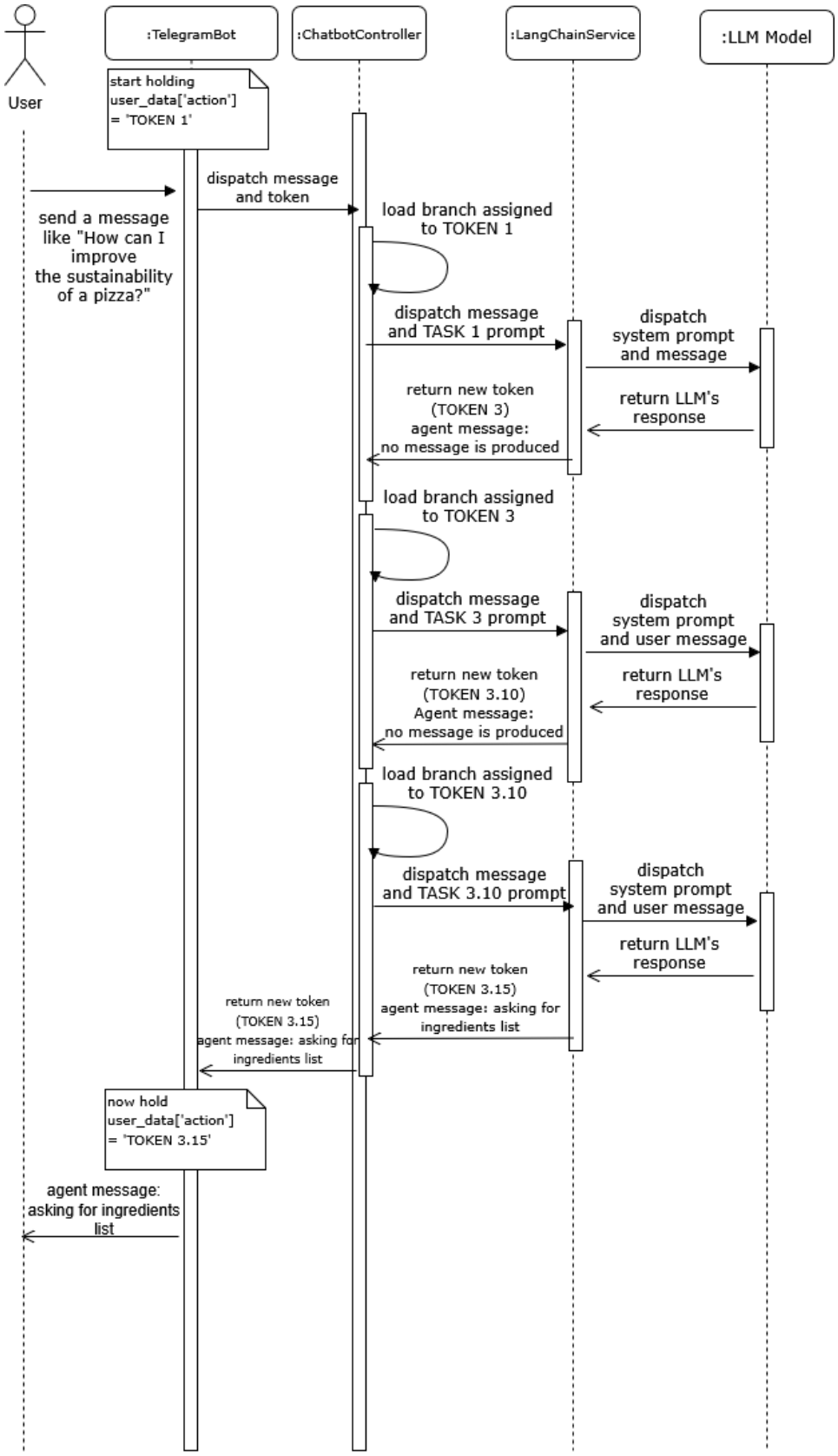


Figure 5.18: Continue in the next page.

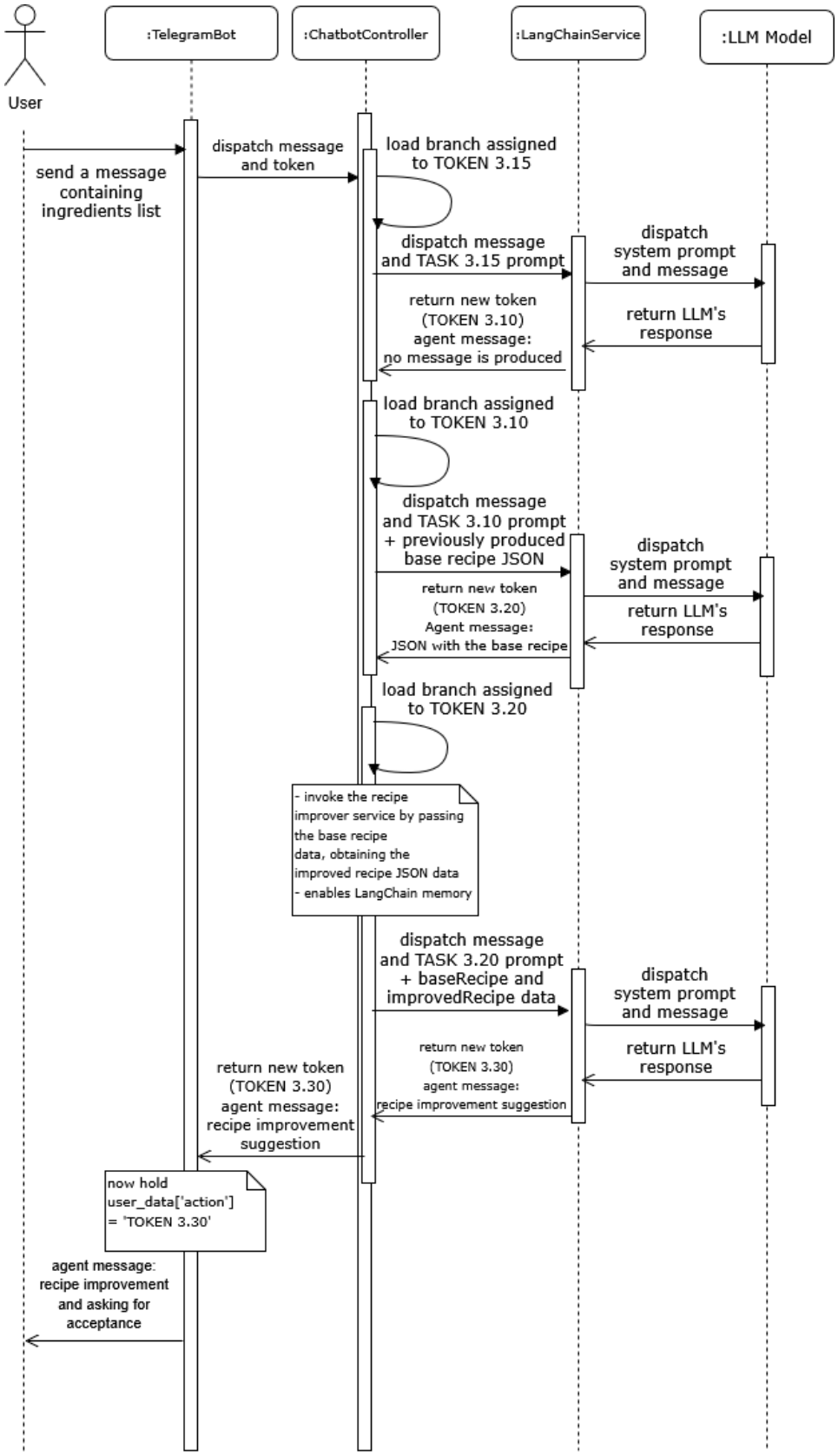
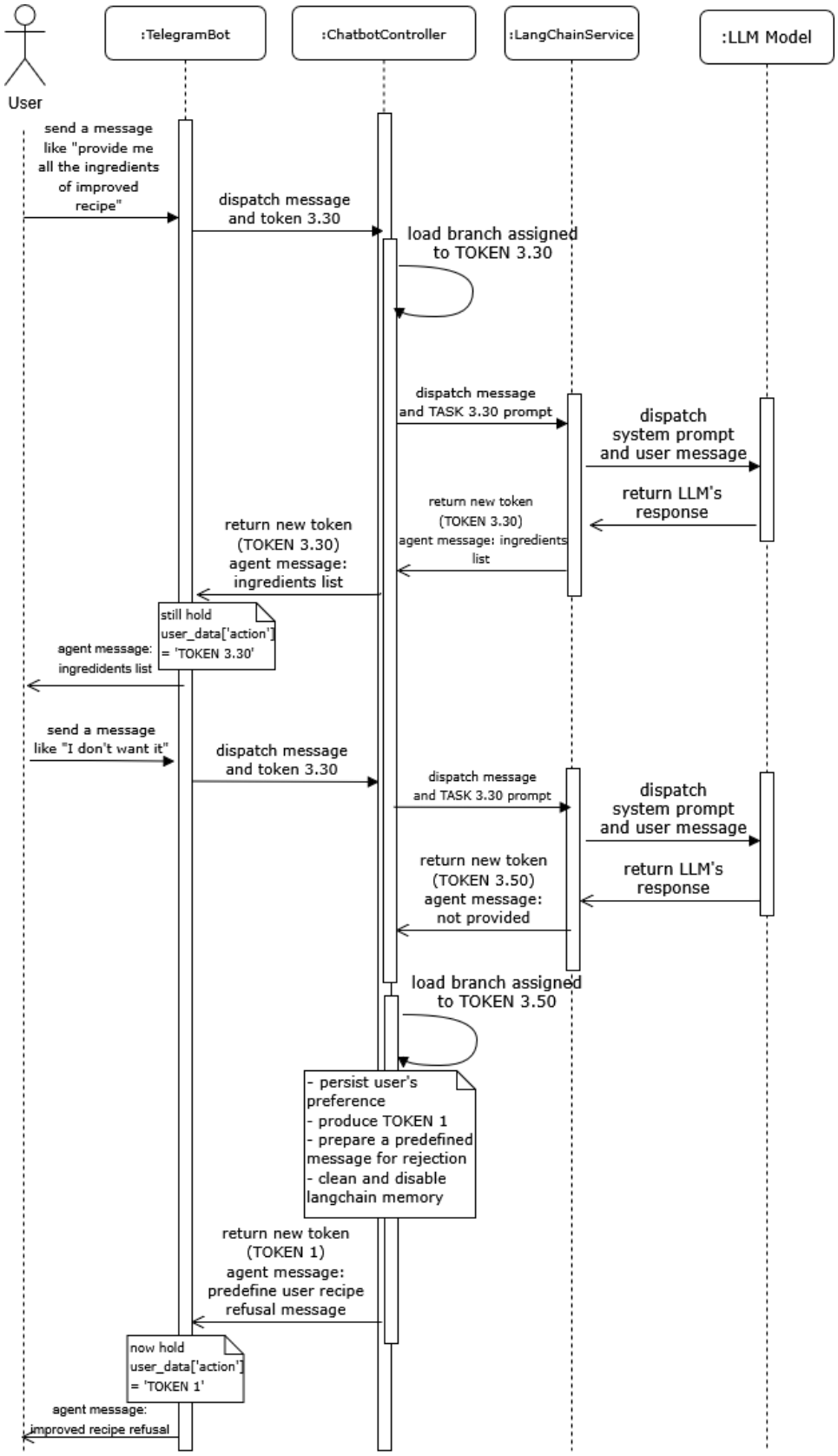


Figure 5.19: Continue in the next page.


User
:TelegramBot
:ChatbotController
:LangChainService
:LLM Model
send a message like "provide me all the ingredients of improved recipe"
dispatch message and token 3.30
load branch assigned to TOKEN 3.30
dispatch message and TASK 3.30 prompt
dispatch system prompt and user message
return LLM's response
return new token (TOKEN 3.30) agent message: ingredients list
return new token (TOKEN 3.30) agent message: ingredients list
still hold user_data['action'] = 'TOKEN 3.30'
agent message: ingredients list
send a message like "I don't want it"
dispatch message and token 3.30
dispatch message and TASK 3.30 prompt
dispatch system prompt and user message
return LLM's response
return new token (TOKEN 3.50) agent message: not provided
load branch assigned to TOKEN 3.50
- persist user's preference
- produce TOKEN 1
- prepare a predefined message for rejection
- clean and disable langchain memory
return new token (TOKEN 1) agent message: predefine user recipe refusal message
now hold user_data['action'] = 'TOKEN 1'
agent message: improved recipe refusal

An example of an actual interaction with the agent performing this task is as follows.

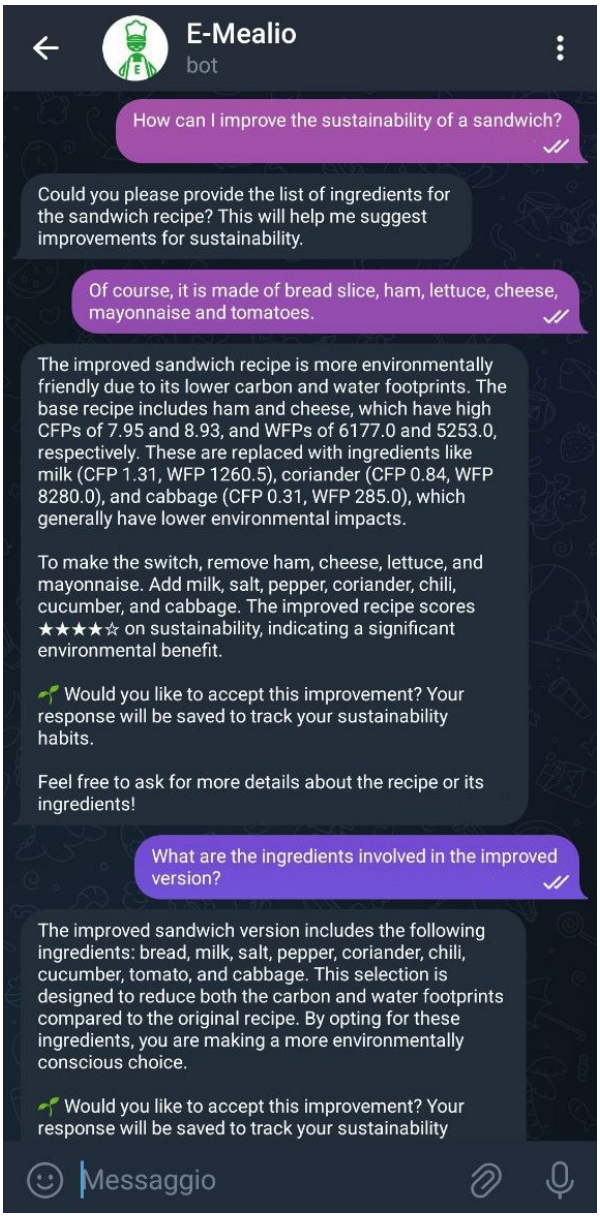


Figure 5.20: The user requests a sustainability improvement for a recipe. The agent asks for the ingredient list, which the user provides. Based on this, the agent suggests improvements, and the user begins discussing them.

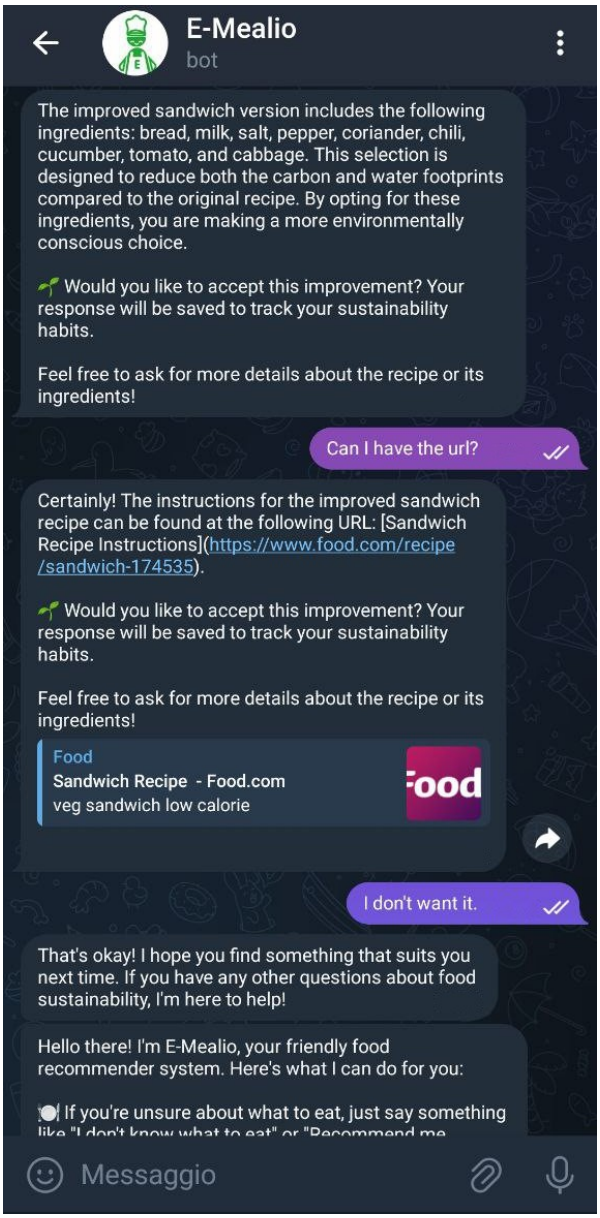


Figure 5.21: Continuing from the previous interaction, the discussion continues until the agent provides a URL for the improved recipe. The user ultimately decides to reject the recipe, and the agent responds before returning to the hub.

5.3.6 Functionality 4: User Profile Recap and Update

The E-Mealio agent has the ability to retrieve and update the user profile, in a similar way to the previously described user data collection functionality.

This feature, referred to as *User Profile Recap and Update*, is powered by a series of prompts designed to display the user profile, request updates, verify if any new valid information is provided, and persist those updates.

The profile recap functionality is implemented through the following prompt.

TASK 4 System Prompt: User Data Recap

You are a food recommender system named E-Mealio with the role of helping users choose more environmentally sustainable foods.
The user will provide you with some information about their profile, structured as a JSON. Follow these steps to produce the output:
- Print the string ”TOKEN 4.10”, then answer the user by generating a summary of the provided data, ignoring the information about tastes, last interaction and user id. Then ask if the user wants to update any information.
Maintain a respectful and polite tone.

This prompt instructs the LLM model to generate a response summarizing the current user's profile while also asking whether they wish to update it. The profile data is passed by the controller's branch associated with **state 4**, replacing the user message. This approach serves as an alternative to the placeholder-based method when the prompt expects only a single piece of data.

Through this prompt, the next state token **4.10** is produced, prompting the user to confirm whether they want to update their profile.

The prompt related to **state 4.10** is as follows.

TASK 4.10 System Prompt: Asking For User Data Updating

You are a simple intent detection system. You will receive an answer from the user about whether they want to update their profile. Follow these steps to produce the output:
- If the user's answer is affirmative, print the string "TOKEN 4.20". Do not write anything else.
- If the user's answer is negative or unrelated to a yes/no answer, print the string "TOKEN 1". Do not write anything else.

This prompt simply instructs the LLM model to recognize an affirmative response to the previously asked question, producing the state token **4.20**. If the response is negative or unrelated, it generates the state token **1**, returning the agent to the main hub.

The prompt related to state **4.20** is the following.

TASK 4.20 System Prompt: Presenting User Data Updating

You are a food recommender system named E-Mealio and have the role of collecting data about the user. User data has the following structure:
USER_DATA_STRUCTURE_TEMPLATE
This information is intended to be the new information that the user wants to update in their profile.
Follow these steps to produce the output:
- Print the string "TOKEN 4.30", then remind the user of the information that can be updated.

This prompt summarizes all the information the user can update, including their constraints, and generates the next state token **4.30**, which handles receiving the updates.

The *USER DATA STRUCTURE TEMPLATE* placeholder refers to essentially the same prompt portion as the one previously reported in Subsection 5.3.2, but without mentioning the mandatoriness of each field, as it is not relevant in this context. Additionally, it includes information about the *reminder* field.

It is provided here for completeness.

System Prompt Portion: User Data Structure Template

name: the name of the user.

surname: the surname of the user.

dateOfBirth: the date of birth of the user in the format DD/MM/YYYY.

nation: the nation of the user. If the user provides their nationality instead of a country name, infer the corresponding nation and set it as the nation field.

allergies: a list of foods that the user cannot eat. The possible constraints are [”gluten”, ”crustacean”, ”egg”, ”fish”, ”peanut”, ”soy”, ”lactose”, ”nut”, ”celery”, ”mustard”, ”sesame”, ”sulfite”, ”lupin”, ”mollusk”]. If the user mentions a term related to an allergy item, match it to the closest predefined constraint and use that item as a constraint.

restrictions: a list of alimentary restrictions derived from ethical choices or religious beliefs. The possible constraints are [”vegan”, ”vegetarian”, ”kosher”].

reminder: a boolean value that indicates whether the user allows receiving reminders.

The prompt related to **state 4.30** is as follows.

TASK 4.30 System Prompt: Receiving User Data Updates

You are a food recommender system named E-Mealio and have the role of collecting data about the user. User data has the following structure:

USER_DATA_STRUCTURE_TEMPLATE

The user could provide you with part of this information in a conversational form or via a structured JSON. This information is intended to be the new information that the user wants to update in their profile.

Follow these steps to produce the output:

- If the user answers something unrelated to this task: Print the string "TOKEN 4.30", then write a gently reminder of the task you want to pursue.

- Otherwise: Print the string "TOKEN 4.40", then print a JSON with only the information collected until now. Do not include the information that has not been provided by the user.

Do not include in the JSON any markup text like "“‘json“‘". Do not make up any other questions or statements that are not the previous ones.

This prompt allows the system to receive updated user information, instructing the LLM to generate a JSON summarizing the provided data with the new values. Additionally, it produces the next state token **4.40**, temporarily updating the stored *userData* object with the new information and transitioning the agent to the user data validation phase.

If the user sends an unrelated message, the state token **4.30** is produced, keeping the agent in the same state and asking again to provide updated information.

The prompt related to state **4.40** is the following.

TASK 4.40 System Prompt: Validating User Data Updates

You are a food recommender system named E-Mealio and have the role of collecting data about the user.

User data has the following structure:

USER_DATA_STRUCTURE_TEMPLATE_WITH_MANDATORINESS

reminder: a boolean value that tells if the user allows receiving reminders. Optional.

The user will provide you with a JSON containing only part of this information about their profile in order to update it.

Follow these steps to produce the output:

- If the JSON refers to some information that is marked as mandatory, and all are filled in, print the string "TOKEN 4.50". Do not write anything else.

- Otherwise, if the JSON refers to some information that is marked as mandatory but is null or empty:

Print the string "TOKEN 4.30", then ask for the remaining information.

This prompt instructs the LLM model to verify if the provided JSON refers to mandatory fields and ensures that they are actually valorized. If there is some missing information, the user is explicitly asked to provide it again, moving back to the state **4.30**.

Of course, only the provided information are verified. If some mandatory field isn't explicitly stated in the provided JSON, it means that the user simply doesn't want to update that field.

If the provided JSON is compliant, the state token **4.50** is produced, invoking the user data persistence layer, allowing the system to finally store the updated information.

The state **4.50** is related to the following prompt.

TASK 4.50 System Prompt: Presenting User Updated Information

You are a food recommender system named E-Mealio and have the role of collecting data about the user.

The user will provide their profile in a JSON format.

Follow these steps to produce the output:

- Print the string ”TOKEN 1”, then summarize what you have collected in a conversational form, ignoring the information about tastes, last interaction and user id.

This prompt is called by the agent controller, which provides it with the user data in place of the canonical user message. It instructs the LLM model to present the newly updated information and then transitions to state **1**, bringing the agent back to the hub.

The overall behavior of the User Profile Recap and Update functionality can be summarized by the following finite state machine graph.

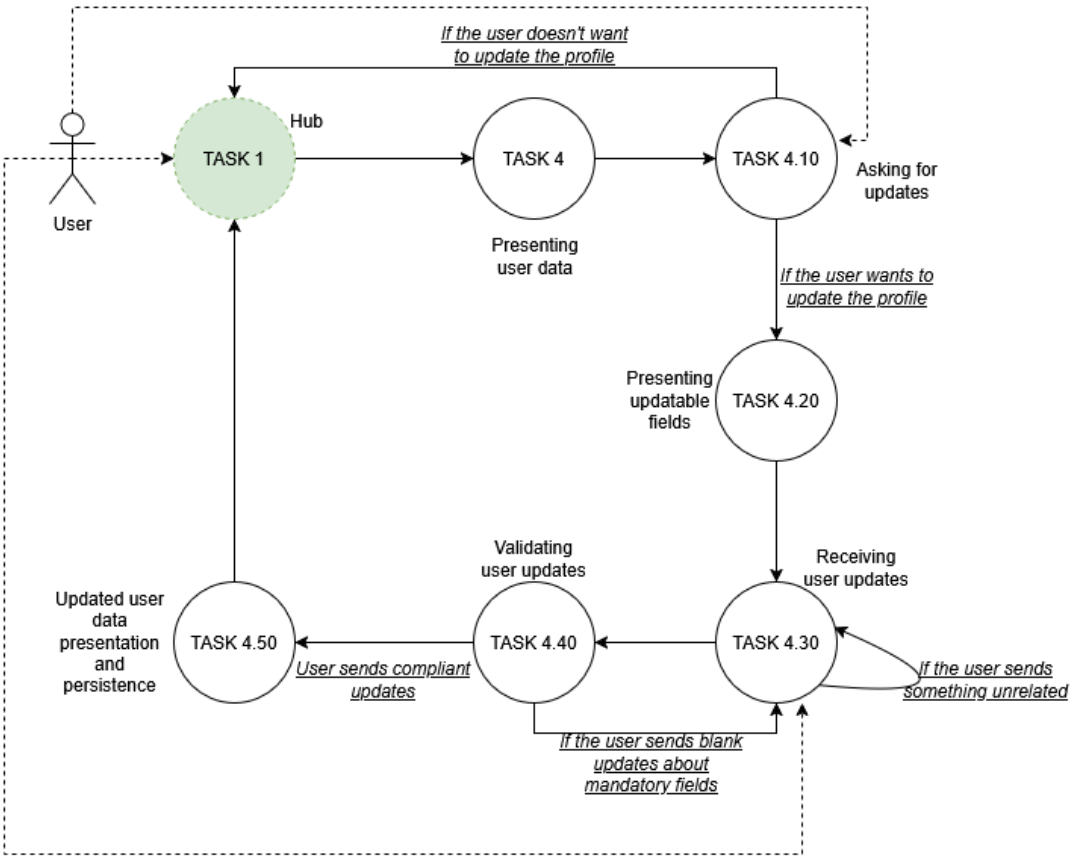


Figure 5.22: Finite State Machine representation of the User Profile Recap and Update functionality.

An example of interaction between the agent's components when handling this task is illustrated in the following sequence diagram. The example depicts a scenario where the user first asks to recap their own data, then consents to update it, updates the allergy list, and finally, the newly updated user data are persisted.

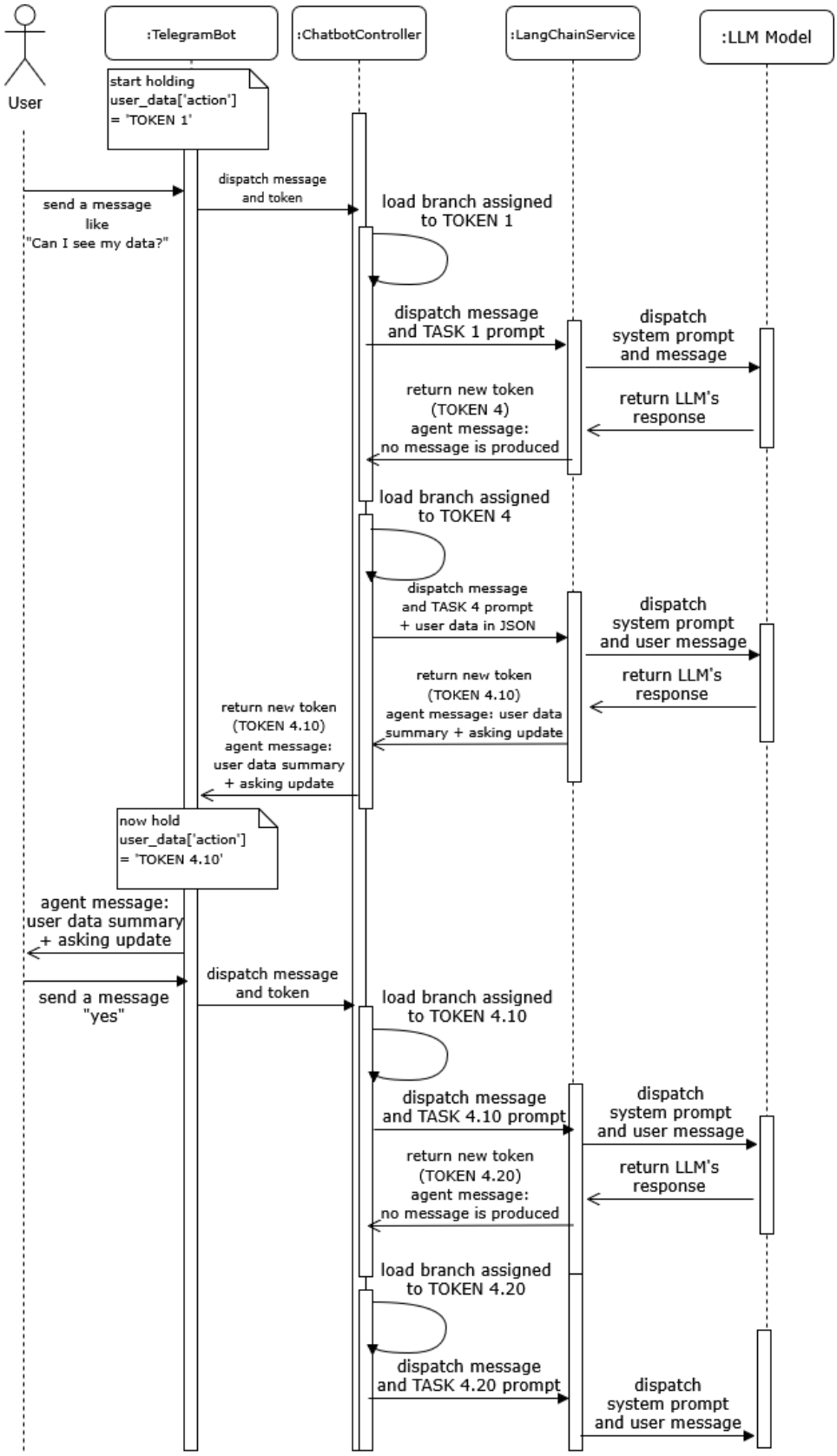
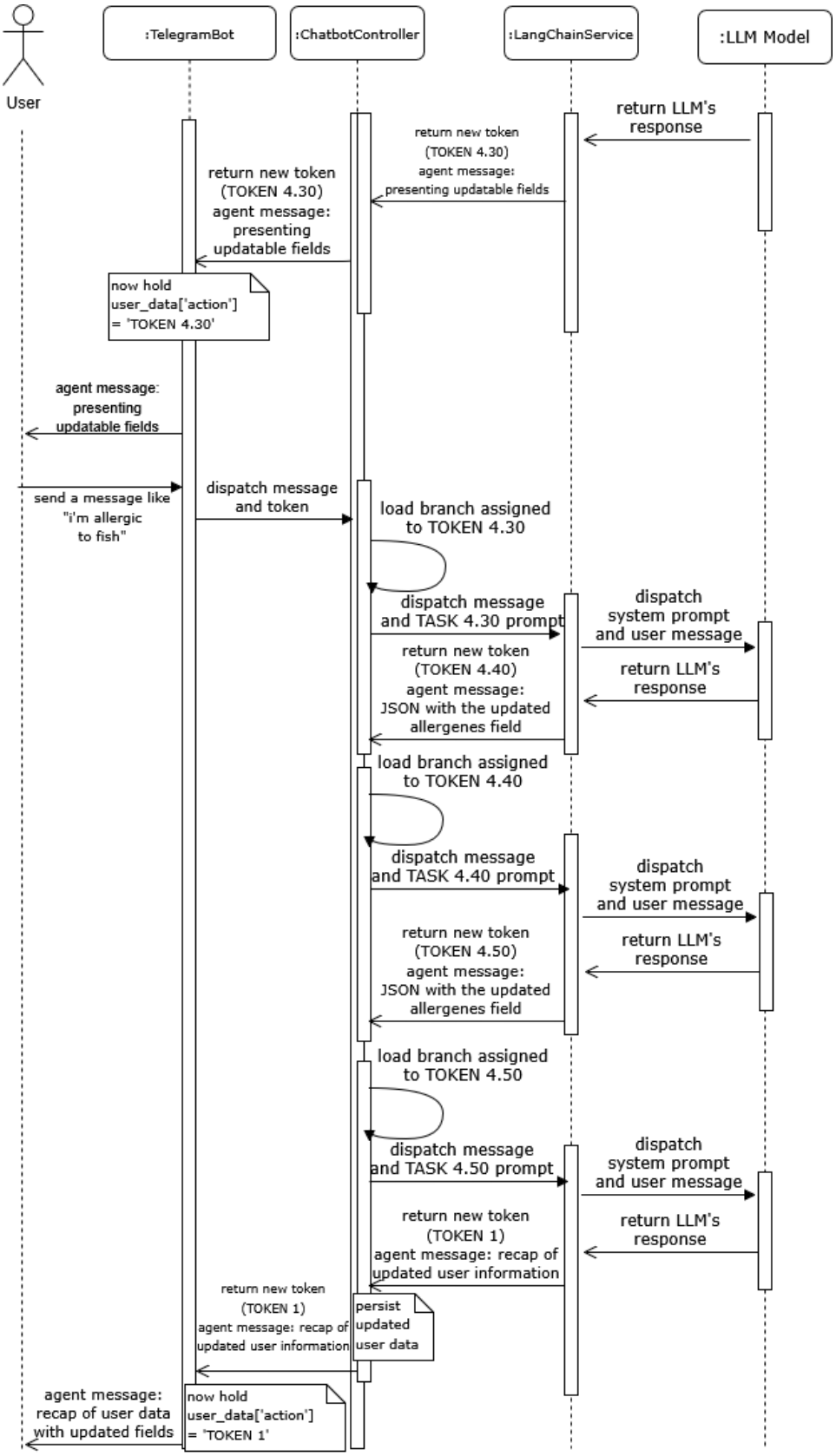


Figure 5.23: Continue in the next page.


User
:TelegramBot
:ChatbotController
:LangChainService
:LLM Model
return LLM's response
return new token (TOKEN 4.30) agent message: presenting updatable fields
return new token (TOKEN 4.30) agent message: presenting updatable fields
now hold user_data['action'] = 'TOKEN 4.30'
agent message: presenting updatable fields
send a message like "i'm allergic to fish"
dispatch message and token
load branch assigned to TOKEN 4.30
dispatch message and TASK 4.30 prompt
dispatch system prompt and user message
return LLM's response
return new token (TOKEN 4.40) agent message: JSON with the updated allergenes field
load branch assigned to TOKEN 4.40
dispatch message and TASK 4.40 prompt
dispatch system prompt and user message
return LLM's response
return new token (TOKEN 4.50) agent message: JSON with the updated allergenes field
load branch assigned to TOKEN 4.50
dispatch message and TASK 4.50 prompt
dispatch system prompt and user message
return LLM's response
return new token (TOKEN 1) agent message: recap of updated user information
persist updated user data
return new token (TOKEN 1) agent message: recap of updated user information
now hold user_data['action'] = 'TOKEN 1'
agent message: recap of user data with updated fields

An example of an actual interaction with the agent performing this task is as follows.

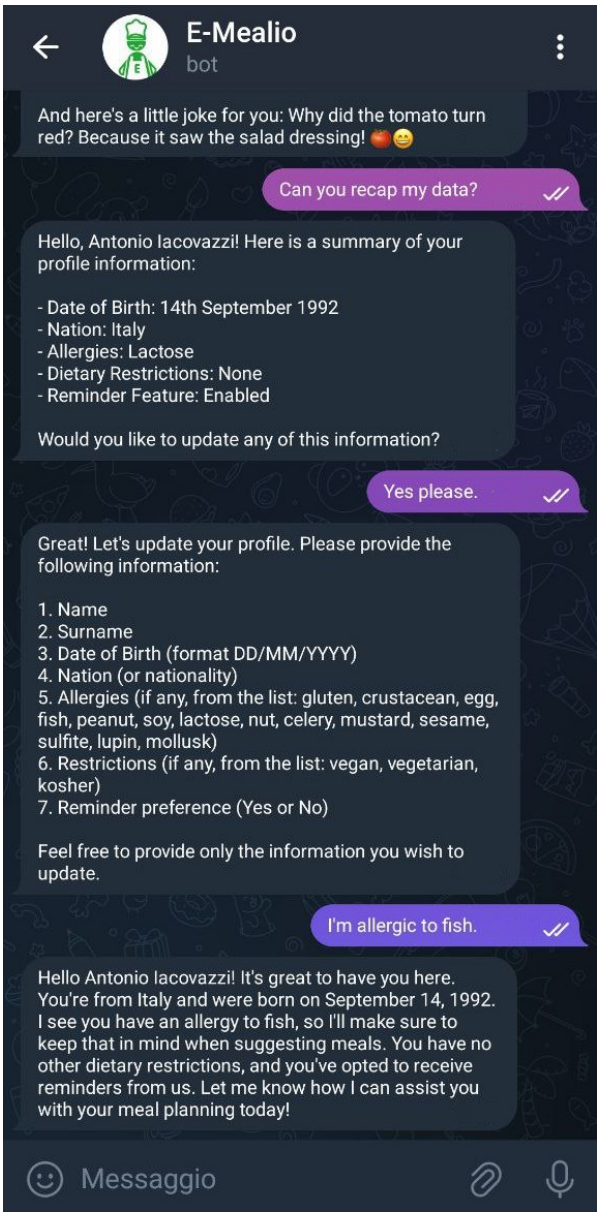


Figure 5.24: The user requests a profile recap and then agrees to update their profile by providing new allergy information. The updated information is successfully saved, and a new profile recap is presented.

5.3.7 Functionality 5: Weekly Food Diary Recap

The E-Mealio bot offers the ability to analyze the user's weekly consumption based on accepted recipes and recipes manually logged by the user through the *Food Diary* functionality.

This functionality leverages the food history service, which queries the database for the user's recorded recipe consumption over the past seven days.

Once the results are retrieved, they are provided as a JSON structure to the prompt associated with **state 5**, as shown below.

TASK 5 System Prompt: Presenting User Weekly Consumption

You are a food recommender system named E-Mealio with the role of helping users choose more environmentally sustainable foods.

You will help the user remember the food they ate in the last 7 days.

The data of the food consumed is structured as follows: {food_history}.

Follow these steps to produce the output:

- If no food history is provided:

Print the string "TOKEN 1", then inform the user that no food history is available and invite them to build it by asserting the food they ate, or accepting the suggestion provided by using the food recommendation system.

- Otherwise:

Print the string "TOKEN 5.10", then summarize the overall food history using a conversational tone. Subsequently provide a small analysis of the user's sustainability habits. Use information about the carbon footprint and water footprint of the ingredients to support your explanation, but keep it simple and understandable.

If you refer to numbers, provide an idea of whether those values are good or bad for the environment.

The sustainability score is such that the lower the value, the better the recipe is for the environment, but avoid providing it explicitly.
Provide an overall rating of the user's sustainability habits using a Likert scale from 0 to 5 stars (use ascii stars, using black stars as point and white stars as filler).
Do not write anything else.

This prompt, loaded by the controller with LangChain memory enabled, dynamically incorporates the previously retrieved weekly consumption data. If no data is available, the agent generates a message stating that it is not possible to analyze the habits since no recipes were found, and invites the user to engage with the system. It then produces the state token **1**, returning the agent to the hub.

Otherwise, it generates a sustainability-based analysis of the user's food consumption and produces the next state token **5.10**, transitioning to the state responsible for discussing the provided analysis.

The prompt associated with **state 5.10** is as follows.

TASK 5.10 System Prompt: Discussing Food Consumption Analysis

You are a food recommender system named E-Mealio with the role of helping users choose more environmentally sustainable foods. You will receive the message history about a sustainability analysis of the user's alimentary habits previously made by you.
Follow these steps to produce the output:
- If the user asks something related to the current topic, like more information about the ingredients or recipe previously mentioned: Print the string "TOKEN 5.10", answer the question.

- If the user asks about the sustainability of a recipe or ingredients not mentioned or related to the recipe in the history:
Print the string ”TOKEN 1”. Do not write anything else.
- If the user wants to terminate the conversation or asks something completely UNRELATED to the topic: Print the string ”TOKEN -1”, then write a message where you tell the user that is a question about another topic. Finally softly invite the user to start a new conversation.
Always maintain a respectful and polite tone.

This prompt instructs the LLM model to determine whether the user is asking about the previously analyzed recipes or shifting to a different topic, such as another recipe, an ingredient, or something completely unrelated.

If the user inquiry concerns the analyzed recipes, the model produces the state token **5.20**, allowing the agent to remain in the same state while tracking the conversation via LangChain memory. If, instead, the question is about an ingredient or a recipe, the model produces the state token **1** without generating a response. This causes the agent to process the question through the hub, which then forwards it to the *Sustainability Expert* functionality for further analysis.

On the other hand, if the user sends a completely unrelated message, the model provides a clarification message to remind them of the discussion's context and then produces the state token **-1**. This token triggers a memory reset and returns the agent to the hub, allowing to start new conversation.

The overall behavior of the User Weekly Food Diary Recap functionality can be summarized by the following finite state machine graph.

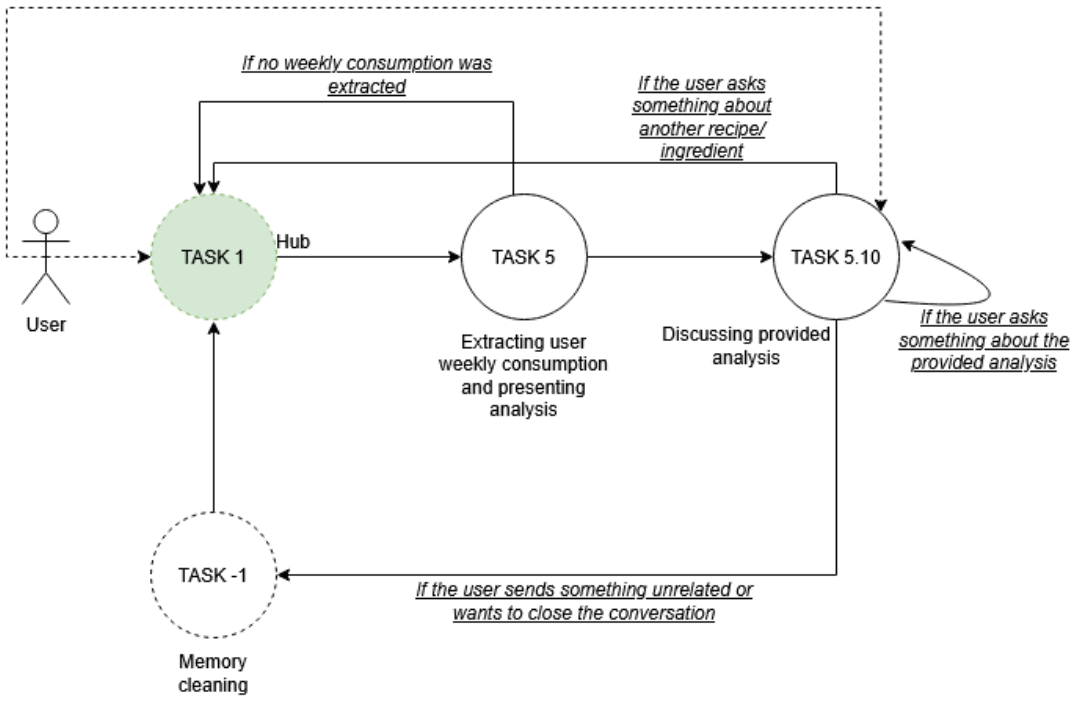
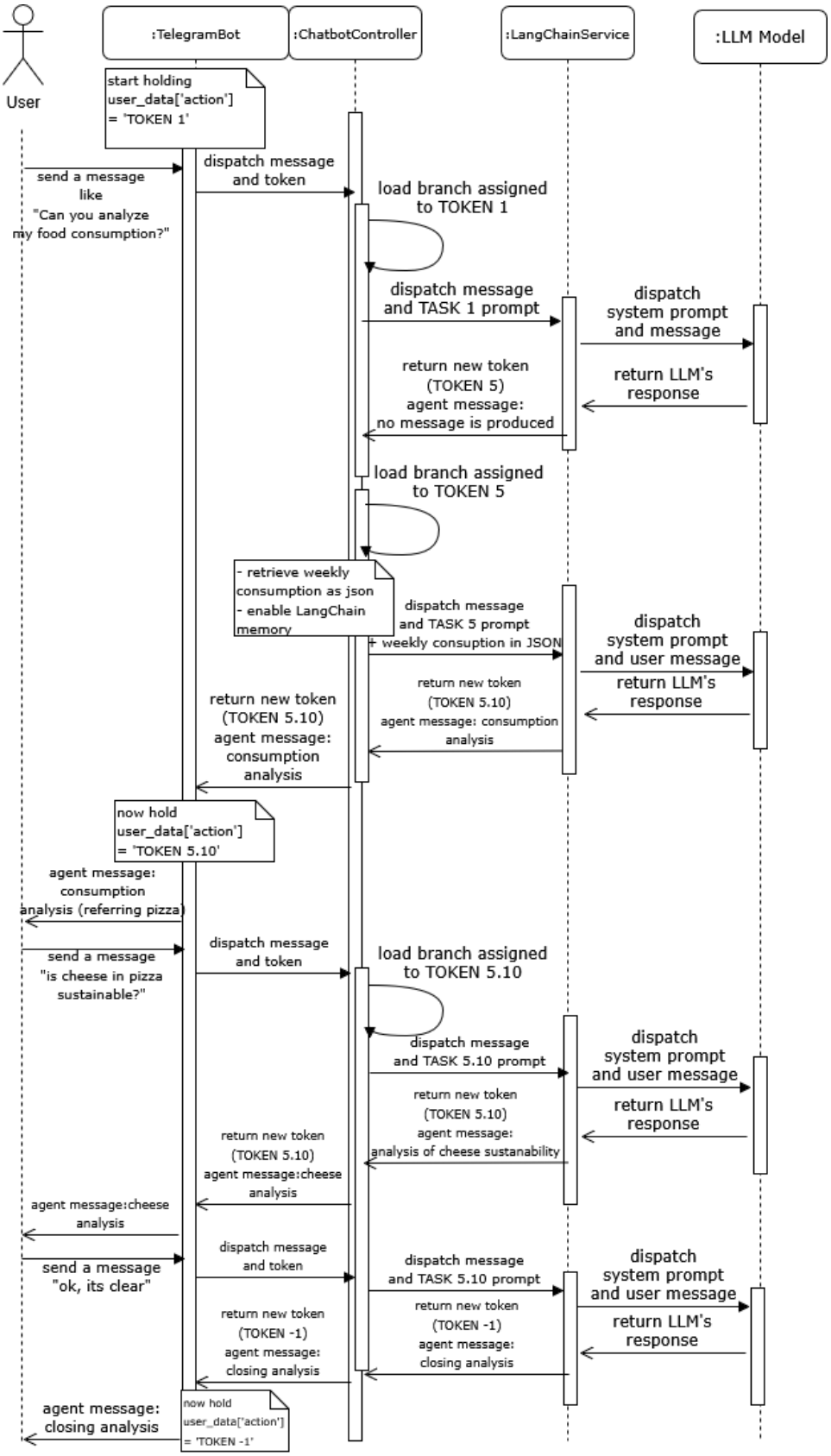


Figure 5.25: Finite State Machine representation of the Weekly Food Diary Recap functionality.

An example of interaction between the agent's components when handling this task is illustrated in the following sequence diagram. The example depicts a scenario where the user first asks to recap their weekly food consumption, the ask something about it and finally close the conversation.


User
:TelegramBot
:ChatbotController
:LangChainService
:LLM Model
start holding user_data['action'] = 'TOKEN 1'
send a message like "Can you analyze my food consumption?"
dispatch message and token
load branch assigned to TOKEN 1
dispatch message and TASK 1 prompt
dispatch system prompt and message
return LLM's response
return new token (TOKEN 5) agent message: no message is produced
load branch assigned to TOKEN 5
- retrieve weekly consumption as json
- enable LangChain memory
dispatch message and TASK 5 prompt + weekly consuption in JSON
dispatch system prompt and user message
return LLM's response
return new token (TOKEN 5.10) agent message: consumption analysis
return new token (TOKEN 5.10) agent message: consumption analysis
now hold user_data['action'] = 'TOKEN 5.10'
agent message: consumption analysis (referring pizza)
send a message "is cheese in pizza sustainable?"
dispatch message and token
load branch assigned to TOKEN 5.10
dispatch message and TASK 5.10 prompt
dispatch system prompt and user message
return LLM's response
return new token (TOKEN 5.10) agent message: analysis of cheese sustanability
return new token (TOKEN 5.10) agent message:cheese analysis
agent message:cheese analysis
send a message "ok, its clear"
dispatch message and token
dispatch message and TASK 5.10 prompt
dispatch system prompt and user message
return LLM's response
return new token (TOKEN -1) agent message: closing analysis
return new token (TOKEN -1) agent message: closing analysis
now hold user_data['action'] = 'TOKEN -1'
agent message: closing analysis

An example of an actual interaction with the agent performing this task is as follows.

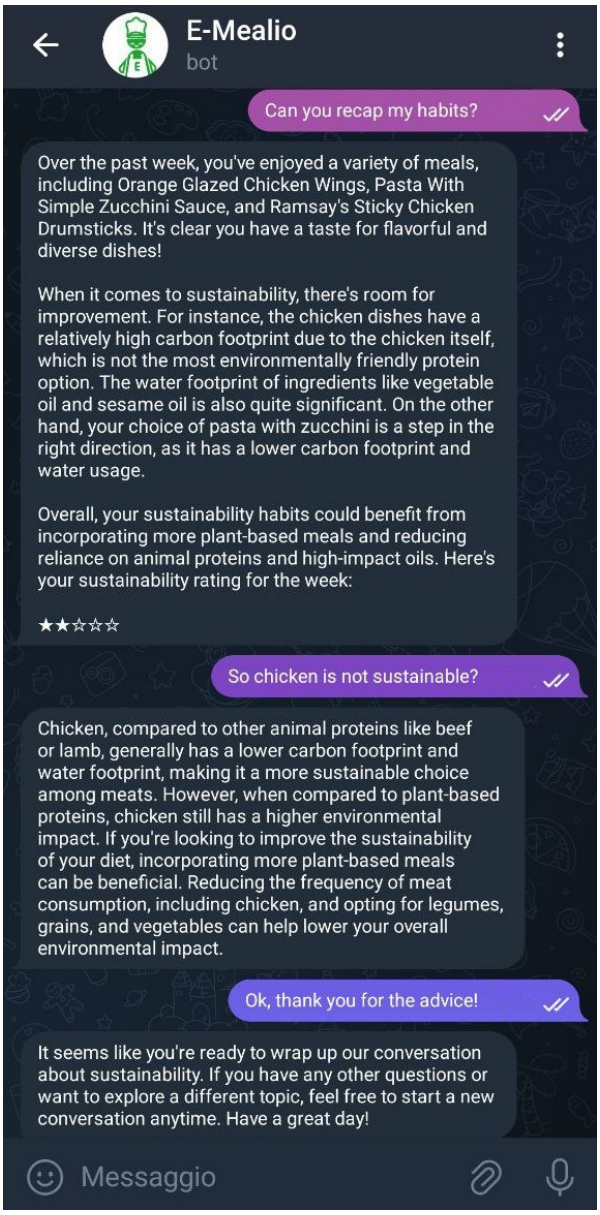


Figure 5.26: The user requests a recap of their eating habits. The agent provides the summary, after which the user asks a follow-up question. The agent responds, and the user then decides to close the conversation.

5.3.8 Functionality 6: Sustainability Expert

The E-Mealio agent also functions as a general advisor on food sustainability, allowing users to ask questions related to recipes, ingredients, or broader sustainability topics. When the inquiry concerns recipes or ingredients, the agent retrieves the most reliable information from the database using the persistence layer before generating a response.

The *Sustainability Expert* functionality is activated through the *Sustainability Expert Sub-Hub*, which responds by producing the state token **6**. Upon reaching this state, the agent loads the following prompt.

TASK 6 System Prompt: Sustainability Expert Topic Understanding

You are a food sustainability expert named E-Mealio involved in the food sector. You will help the user understand the sustainability of foods or recipes.

The user can:

1) Ask you about the sustainability of an ingredient or a list of ingredients.

2) Ask you about the sustainability of a recipe or a list of recipes. Recipes can be provided using the name or the list of ingredients.

3) Ask questions about environmental concepts like carbon footprint, water footprint, food waste, food loss, food miles, etc.

Follow these steps to produce the output:

- Based on the information provided by the user, output a json with the following structure:

recipeNames: list of the names of the recipes that the user asked about. Optional.

recipeIngredients: list of the ingredients of the recipes that the user asked about; this field must be filled only if the recipe name is not provided, otherwise, keep it empty. Optional.

ingredients: list of the ingredients that the user asked about. Optional.

concept: the environmental concept that the user asked about. Optional.

task: the type of question that the user asked. The possible values are [”recipe”, ”ingredient”, ”concept”]. Mandatory.

- Then finally:

– If the detected task is ”concept,” print the string ”TOKEN 6.10”. Do not write anything else.

– If the detected task is ”ingredient,” print the string ”TOKEN 6.20”. Do not write anything else.

– If the detected task is ”recipe,” print the string ”TOKEN 6.30”. Do not write anything else.

Do not include in the JSON any markup text like ”“‘json“‘”.

This prompt instructs the LLM model to determine the topic of the request and generate a JSON structure containing both the subject and the specific task being asked about. Based on the identified task, the model assigns a corresponding state token.

The detected task can be **”concept”** for questions related to broader sustainability topics, **”ingredient”** for inquiries about the sustainability of one or more ingredients, and **”recipe”** for questions regarding the sustainability of one or more recipes. The respective state tokens generated are **6.10**, **6.20**, and **6.30**.

When executed, this prompt does not directly provide an answer to the user. Instead, it redirects the question to the appropriate branch responsible for extracting the necessary information and constructing the response.

When the produced state token is the 6.10, the loaded prompt is the following.

TASK 6.10 System Prompt: Sustainability Expert - Broad Sustanability Concepts

You are a food sustainability expert named E-Mealio involved in the food sector.

You will help the user understand the following environmental concept: {concept}.

Follow these steps to produce the output:

- Print the string ”TOKEN 6.40”, then explain the concept in detail and provide some examples related to the food sector.

Be succinct, using up to 150 words. Maintain a respectful and polite tone.

This prompt is loaded by replacing the placeholder with the concept of interest and enabling the LangChain memory. It instructs the LLM model to generate an explanation of the given concept while incorporating relevant food-related examples. Additionally, it produces the state token **6.40**, which transitions the agent into a state where the conversation can continue.

Unlike other branches of the Sustainability Expert functionality, this process does not involve querying the database, meaning no RAG techniques are applied. Instead, the response is entirely generated by the LLM, leveraging the latent knowledge stored within its model weights.

When the produced state token is the 6.20, the loaded prompt is the following.

TASK 6.20 System Prompt: Sustainability Expert - Ingredients Sustainability

You are a food sustainability expert named E-Mealio involved in the food sector.

You will help the user understand the sustainability of the following ingredients: {ingredients}.

Follow these steps to produce the output:

- Print the string "TOKEN 6.40", then explain the sustainability of the ingredients in detail, comparing their carbon footprint and water footprint if there are more than one. Keep the explanation simple and understandable. Refer to numbers like carbon footprint and water footprint, but also give an idea of whether those values are good or bad for the environment.

Be succinct, using up to 150 words. Maintain a respectful and polite tone.

This prompt is loaded by replacing the placeholder with the JSON data of the ingredients of interest extracted by the persistence layer, and enabling the LangChain memory.

In the provided JSON, sustainability information is explicitly included, such as the Carbon Footprint (CFP) and Water Footprint (WFP) of the ingredients, which assist the model in providing a well-founded analysis of them. Additionally, it produces the state token **6.40**, which transitions the agent into a state where the conversation can continue.

When the produced state token is the 6.30, the loaded prompt is the following.

TASK 6.30 System Prompt: Sustainability Expert - Recipe Sustainability

You are a food sustainability expert named E-Mealio involved in the food sector.

You will help the user understand the sustainability of the following recipes: {recipes}.

Follow these steps to produce the output:

- Print the string "TOKEN 6.40", then explain the sustainability of the recipes by comparing the carbon footprint and water footprint of the ingredients involved in the recipes. Use information about the carbon footprint and water footprint of the ingredients to support your explanation, but keep it simple and understandable. Refer to numbers of CFP and WFP, but also provide an idea of whether those values are good or bad for the environment.

The sustainability score is such that the lower the value, the better the recipe is for the environment. It ranges from 0 to 1. Do not provide it explicitly but use a Likert scale to describe it printing from 0 to 5 stars (use ascii stars, using black stars as point and white stars as filler).

Be succinct, using up to 200 words. Maintain a respectful and polite tone.

This prompt is similar to the previous one, with the only difference being that it refers to recipes. It is loaded with a JSON containing recipe data extracted by the persistence layer, which also includes their sustainability score and the ingredients list. Additionally, this prompt is loaded with LangChain memory enabled, which is leveraged by the subsequent state **6.40**.

The state **6.40** is related to the following prompt.

TASK 6.40 System Prompt: Sustainability Expert - Continuing Conversation

You are a food sustainability system named E-Mealio with the role of helping users choose more environmentally sustainable foods.

You will receive the message history about a sustainability question previously made by the user and answered by you.

Follow these steps to produce the output:

- If the user asks something related to the current topic, like more information about something already mentioned: Print the string "TOKEN 6.40", then write an answer to the user's question. If the answer refers to values like carbon footprint and water footprint, provide them explicitly but also give an idea of whether those values are good or bad for the environment.

- If the user wants to terminate the conversation or asks something unrelated to the current topic:

Print the string "TOKEN -1", then write a message where you tell the user that is a question about another topic. Finally softly invite the user to start a new conversation.

Always maintain a respectful and polite tone.

This prompt, similarly to the others provided to contextualize the LangChain memory in order to continue the conversation about the main topic, instructs the LLM to determine whether a question relates to the previously discussed topic and to respond accordingly while remaining in the same state. The prompt is also used to detect if the user is changing the topic or requesting to close the conversation, in which case it produces the state token **-1**, causing a memory reset and returning to the hub.

The overall behavior of the Sustainability Expert functionality can be summarized by the following finite state machine graph.

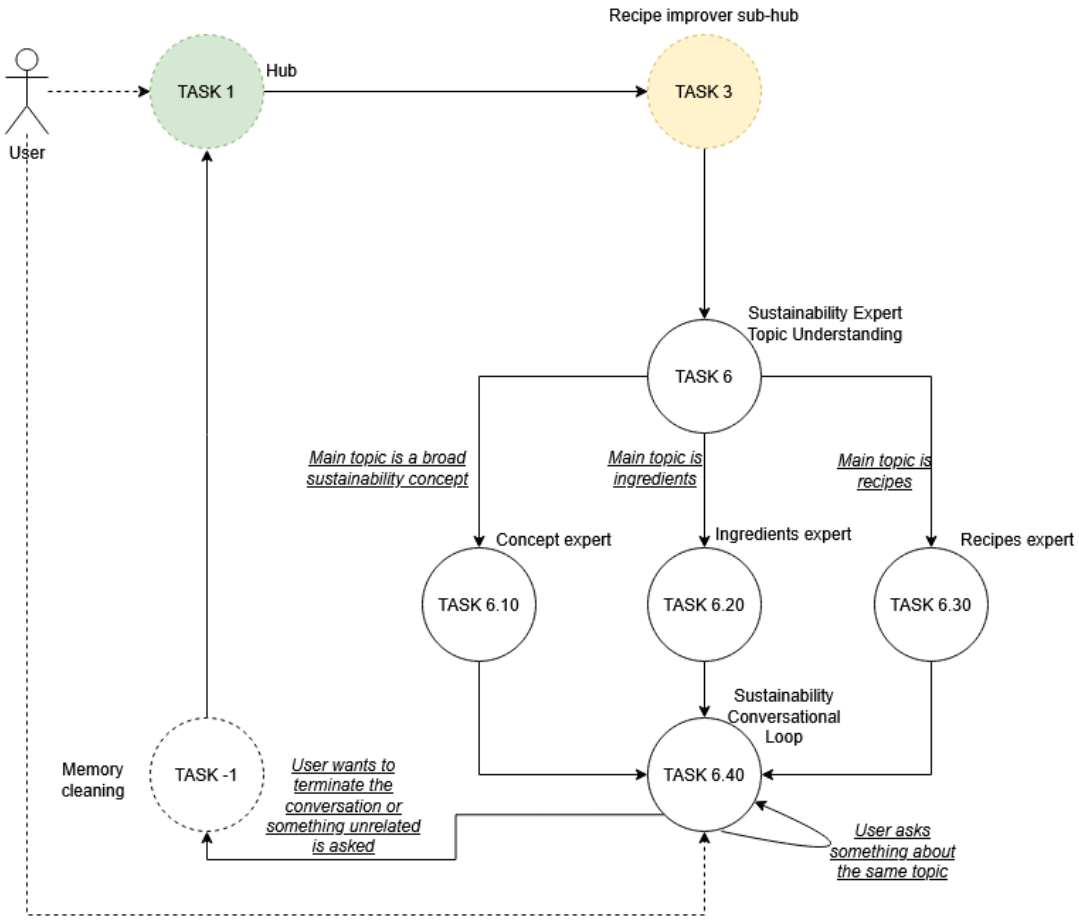


Figure 5.27: Finite State Machine representation of the Sustainability Expert functionality.

An example of interaction between the agent's components when handling this task is illustrated in the following sequence diagram. The example depicts a scenario where the user first asks a question about a couple of ingredients, the agent responds, and then the user asks something related to the provided answer. The agent responds again, and finally, the user requests to close the conversation.

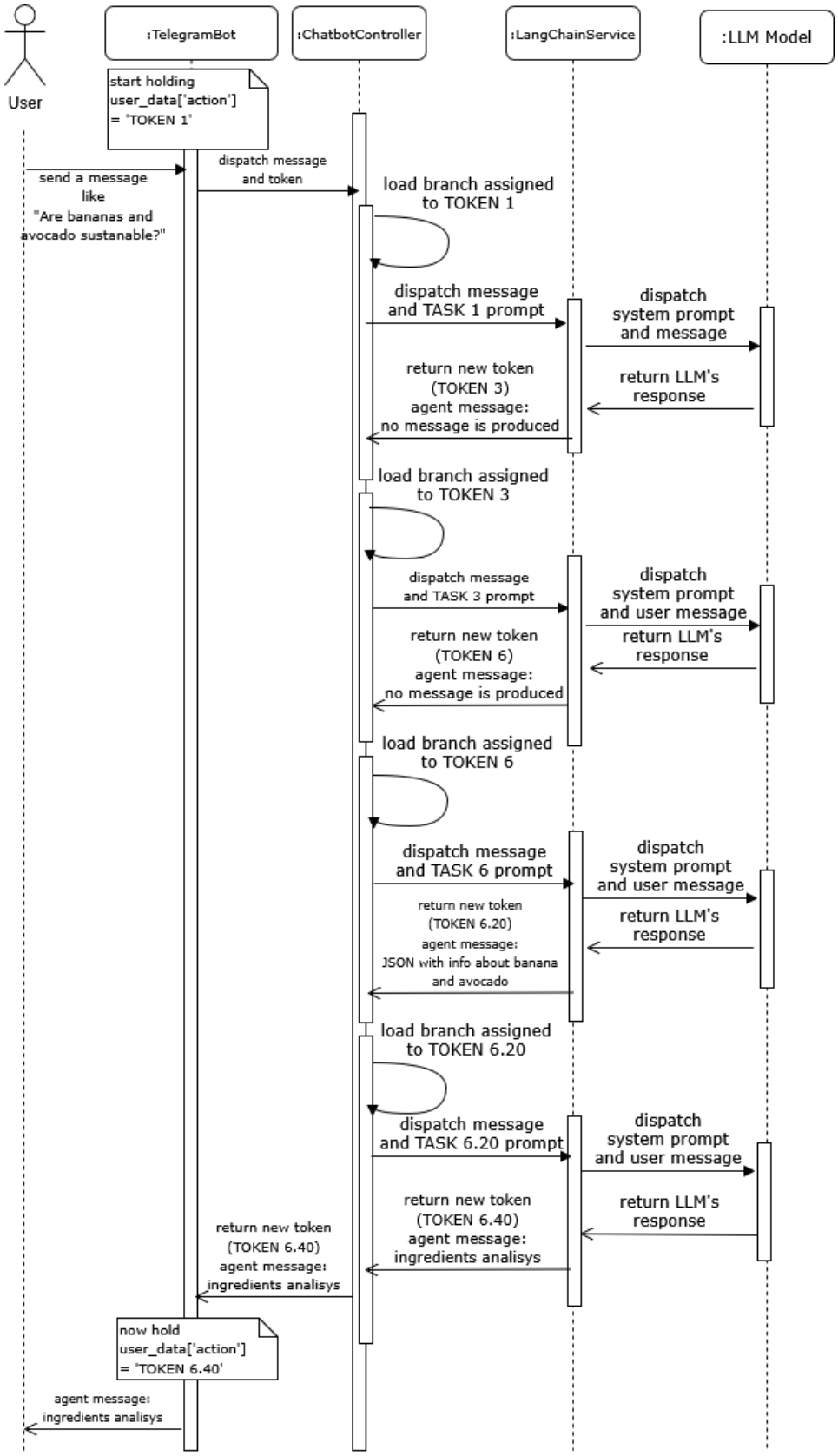
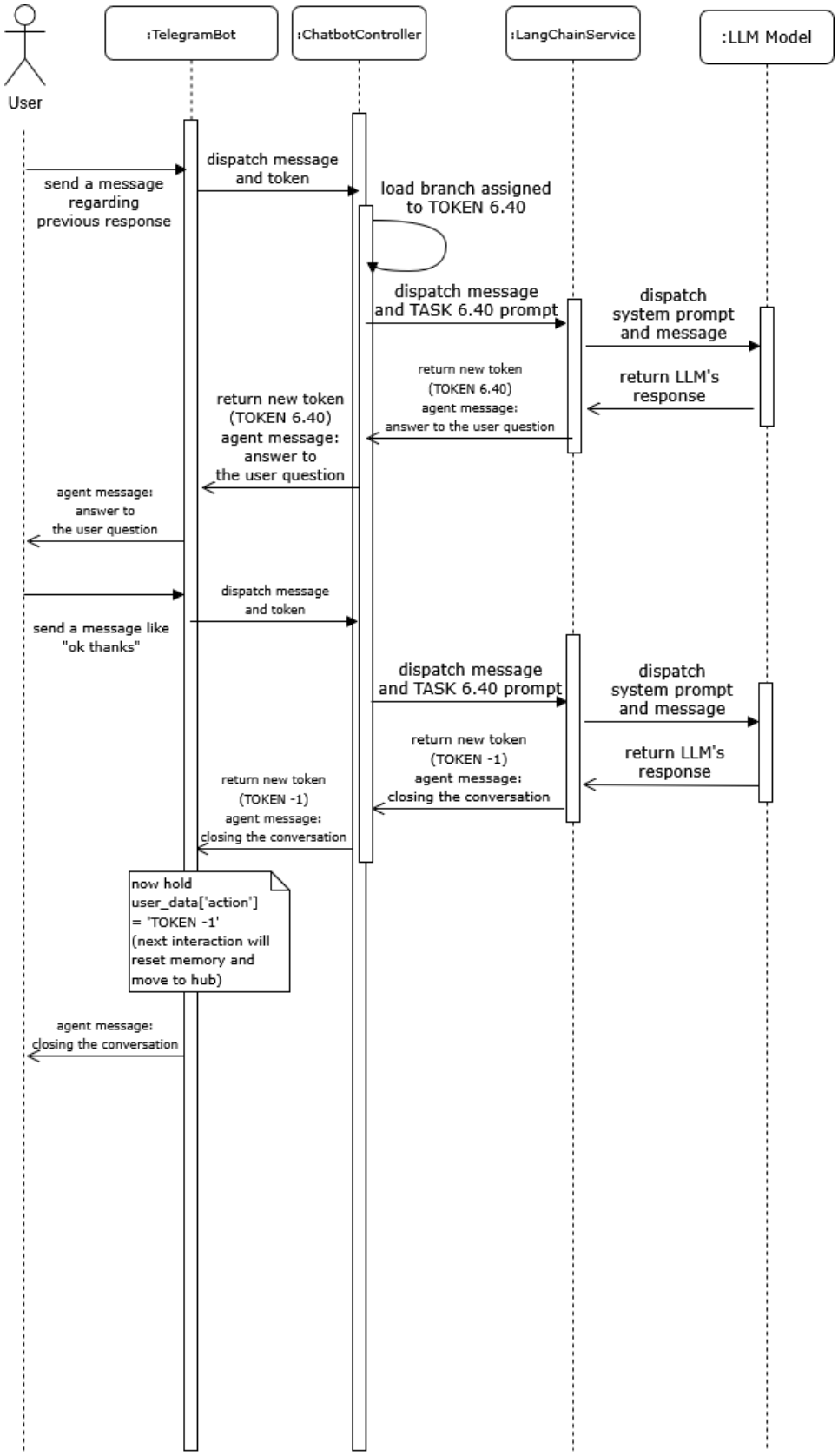


Figure 5.28: Continue in the next page.


User
:TelegramBot
:ChatbotController
:LangChainService
:LLM Model
send a message regarding previous response
dispatch message and token
load branch assigned to TOKEN 6.40
dispatch message and TASK 6.40 prompt
dispatch system prompt and message
return LLM's response
return new token (TOKEN 6.40) agent message: answer to the user question
return new token (TOKEN 6.40) agent message: answer to the user question
agent message: answer to the user question
send a message like "ok thanks"
dispatch message and token
dispatch message and TASK 6.40 prompt
dispatch system prompt and message
return LLM's response
return new token (TOKEN -1) agent message: closing the conversation
return new token (TOKEN -1) agent message: closing the conversation
now hold user_data['action'] = 'TOKEN -1' (next interaction will reset memory and move to hub)
agent message: closing the conversation

An example of an actual interaction with the agent performing this task is as follows.

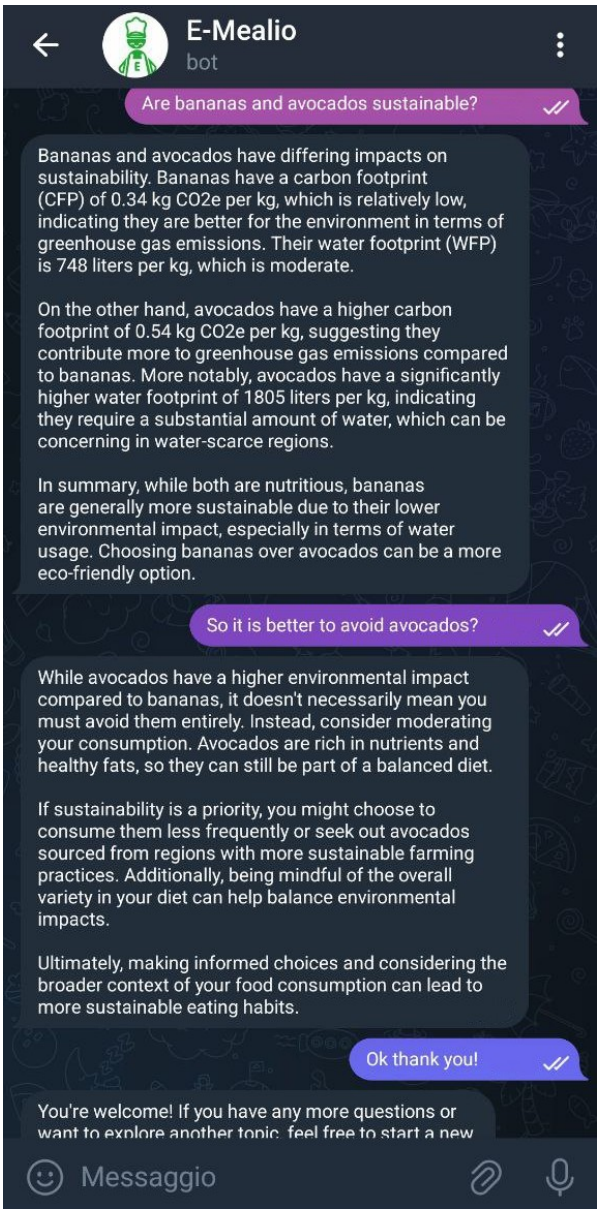


Figure 5.29: The user asks an informative question about the sustainability of bananas and avocados. The agent responds with a comparative analysis of the two. A brief discussion follows, and the user then decides to close the conversation.

5.3.9 Functionality 7: Food Diary

The final implemented functionality is the *Food Diary*, which allows users to log their consumed recipes, adding them to those they have eventually accepted through the recommendation system. This enables the agent to analyze dietary habits and provide insights through the *Weekly Food Diary Recap* function.

This functionality, associated with **state 7**, is managed by the following prompt.

TASK 7 System Prompt: Food Diary-Receiving Recipe

You are a food recommender system named E-Mealio with the role of helping users choose more environmentally sustainable foods.

The user will provide you with a sentence or a JSON containing a recipe that they assert to have eaten.

The recipe is mentioned as a list of ingredients and, eventually, the recipe name.

JSON and conversational information can also be provided together.

The meal data is structured as follows:

mealType: the type of meal. The possible values are [”Breakfast”, ”Lunch”, ”Dinner”, ”Break”]. Mandatory. Used to register the meal at the correct time of day.

ingredients: the list of ingredients of the recipe exactly as provided by the user. Do not make up any ingredient. The ingredients list is usually provided by the user as a list of ingredients separated by commas. Valorize this field as a list of strings. Mandatory.

name: the name of the recipe. Optional. The user could provide you with this information in a conversational form and also via a structured JSON.

Follow these steps to produce the output:

- If the user asks something about the constraints, explain the constraint in detail, then print the string ”TOKEN 7”.

- Otherwise: Print the string ”TOKEN 7.10”, then print a JSON with the information collected until now. Set the absent information as an empty string (for atomic fields) or an empty list (for list fields). Derive a proper recipe name from the list of ingredients provided by the user if not provided.
Do not include in the JSON any markup text like ”“‘json“‘”.
Do not make up any other question or statement that are not the previous ones.

This prompt, invoked by passing the user's request to log a recipe as the input message, instructs the LLM model to generate a JSON structure containing the recipe's ingredients list, along with the recipe name and meal type. The recipe name, being optional, can be inferred from the ingredients list if not provided. The prompt then produces the generated JSON along with the next state token **7.10**.

Given that this prompt can also be re-invoked from state **7.10**, it additionally recognizes when the user is inquiring about mandatory constraints. In such cases, it provides an appropriate response while remaining in the same state.

The prompt associated with state **7.10**, which performs recipe data validation by verifying mandatory fields, is as follows.

TASK 7.10 System Prompt: Food Diary-Data Verification

You are a food recommender system named E-Mealio with the role of helping users choose more environmentally sustainable foods.

The user will provide you with a sentence containing a recipe that they assert to have eaten. The recipe is mentioned as a list of ingredients and, eventually, the recipe name.

The recipe data is structured as follows:

mealType: the type of meal. The possible values are [”Breakfast”, ”Lunch”, ”Dinner”, ”Break”]. Mandatory.

ingredients: the list of ingredients of the recipe exactly as provided by the user. Do not make up any ingredient. The ingredients list is usually provided by the user as a list of ingredients separated by commas. Valorize this field as a list of strings. Mandatory.

name: the recipe name provided by the user. Derive it from the ingredients if not provided. Mandatory.

The user will provide you with a JSON containing some information about the meal they assert to have eaten.

Follow these steps to produce the output:

- If all the mandatory information is collected: print the string ”TOKEN 7.20” followed by the JSON provided by the user.

- If the user doesn’t provide all the mandatory information: Print the string ”TOKEN 7”, then print the JSON provided by the user. Subsequently ask them for the remaining information.

Do not include in the JSON any markup text like ”“‘json“‘“.

This prompt performs data mandatoriness verification similarly to previous tasks such as *User Data Acquisition* and *Recipe Sustainability Improvement*. The JSON to be verified is pro-

vided as the user message. If any mandatory information is missing, the prompt generates the state token **7**, prompting the user to supply the missing data. Otherwise, it produces the next state token **7.20**. In both cases, the JSON of the provided recipe is re-output to be passed to the next state.

The controller branch related to state **7.20** allows the persistence of the provided recipe in the user's consumption history. Subsequently, a new prompt is loaded to inform the user that the recipe has been successfully saved as consumed. The prompt associated with state **7.20** is as follows.

TASK 7.20 System Prompt: Food Diary Persistence

You are a food recommender system named E-Mealio with the role of helping users choose more environmentally sustainable foods.
The user will provide you with a JSON containing a meal that they assert to have eaten.
Follow these steps to produce the output:
- Print the string "TOKEN 1", then summarize the information collected in a conversational form. Finally communicate that you have saved the information in order to analyze their eating habits and refine your future suggestions.

This prompt simply instructs the LLM model to summarize the JSON recipe data provided as the user message, concluding with the production of the state token **1**, which indicates that the agent will return to the hub.

The overall behavior of the Food Diary functionality can be summarized by the following finite state machine graph.

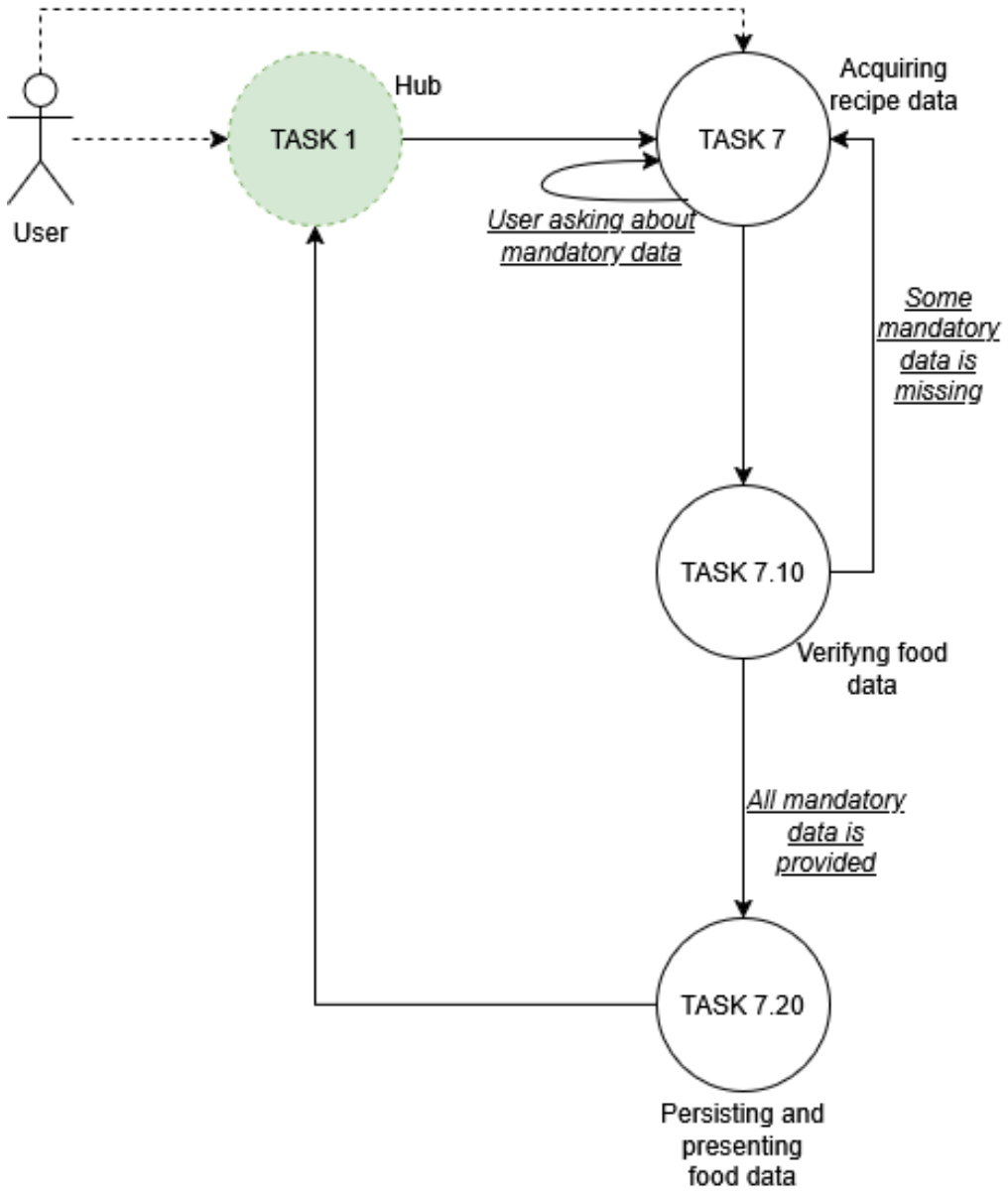


Figure 5.30: Finite State Machine representation of the Food Diary functionality.

An example of interaction between the agent's components when handling this task is illustrated in the following sequence diagram. The example depicts a scenario where the user first assert to have eaten a recipe, without providing the meal type. The agent prompts the user to provide it, then the user answer with the meal type and the agent persist the recipe providing a proper closing message.

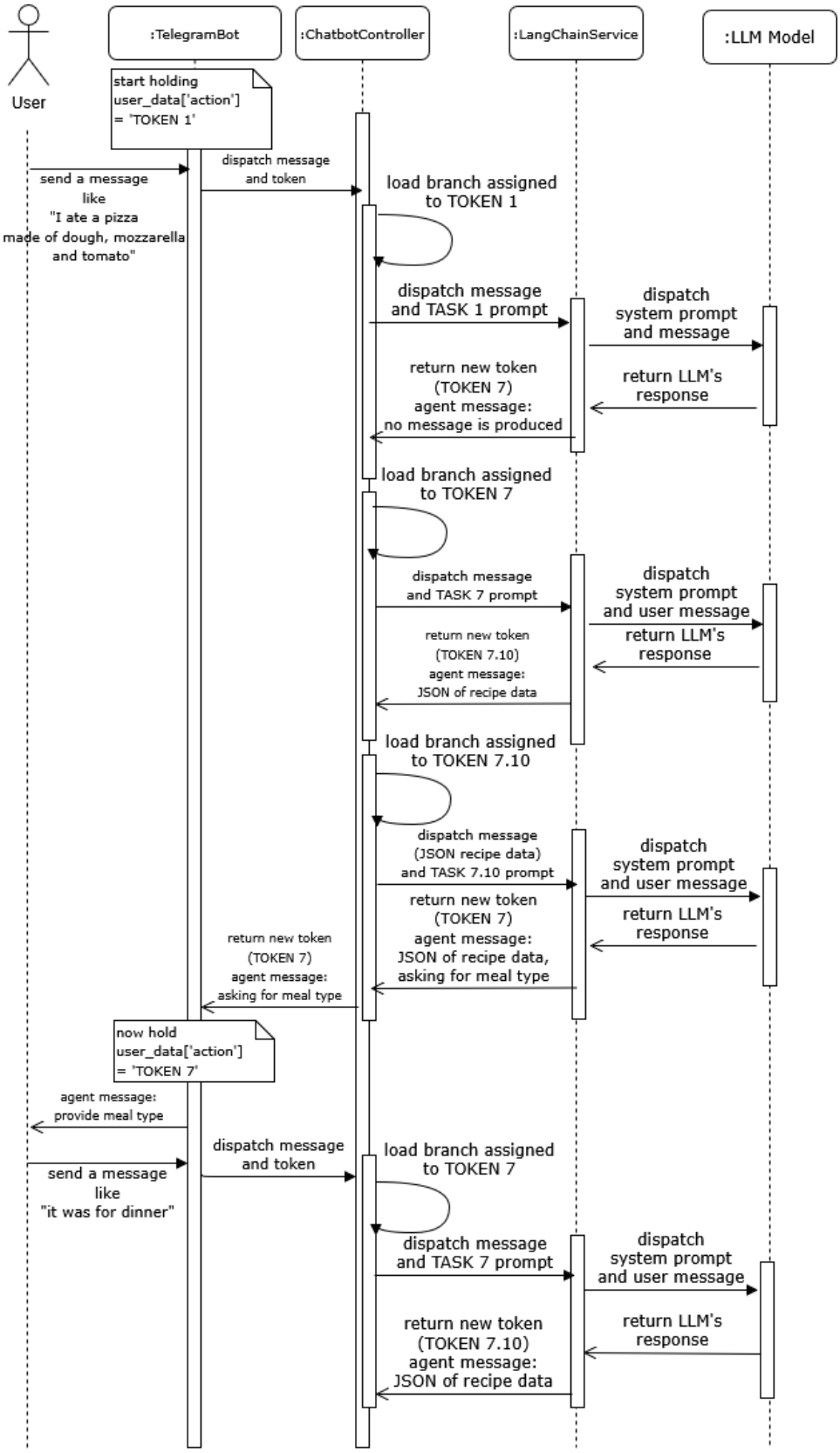
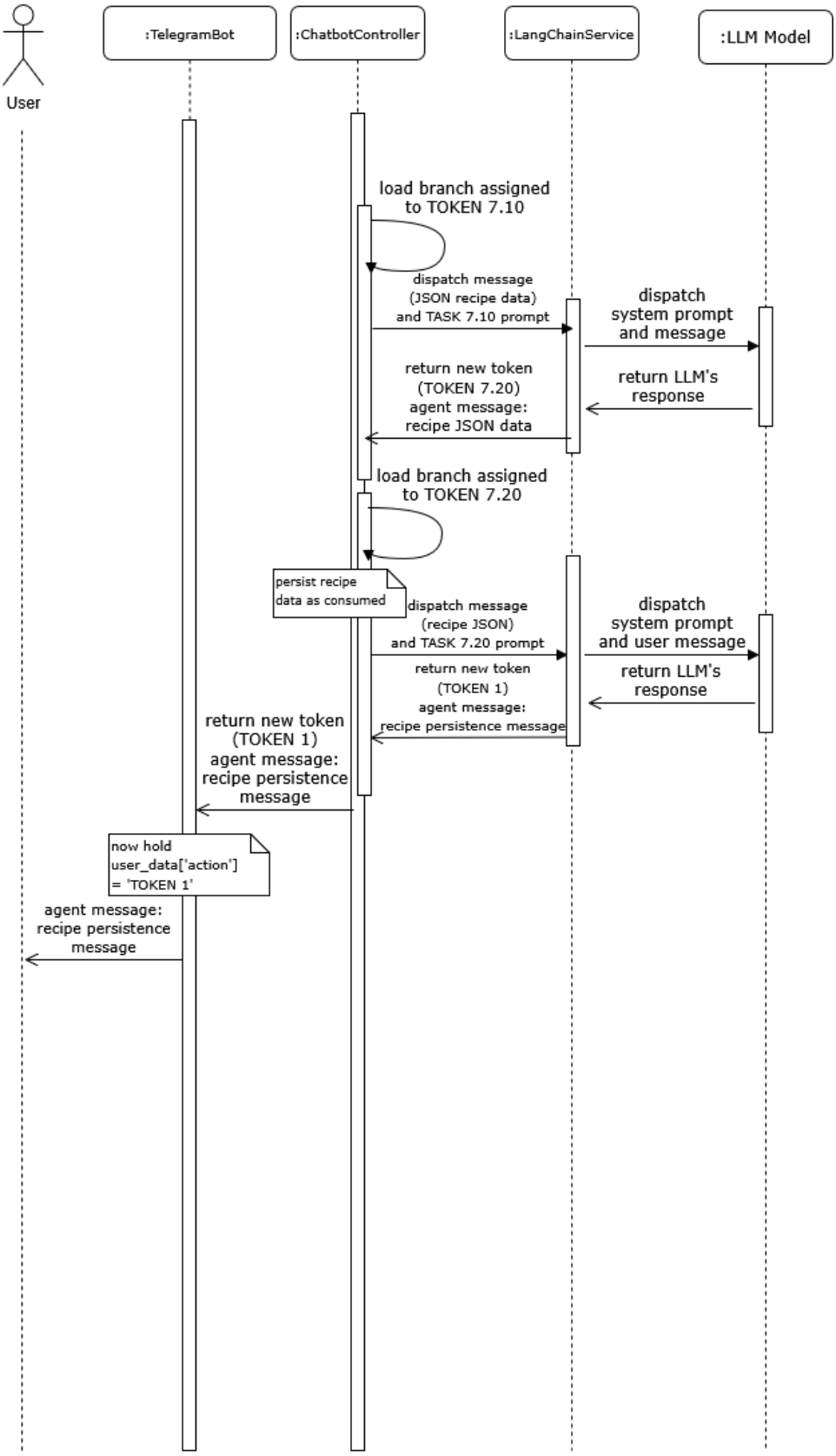


Figure 5.31: Continue in the next page.


User
:TelegramBot
:ChatbotController
:LangChainService
:LLM Model
load branch assigned to TOKEN 7.10
dispatch message (JSON recipe data) and TASK 7.10 prompt
dispatch system prompt and message
return LLM's response
return new token (TOKEN 7.20) agent message: recipe JSON data
load branch assigned to TOKEN 7.20
persist recipe data as consumed
dispatch message (recipe JSON) and TASK 7.20 prompt
dispatch system prompt and user message
return LLM's response
return new token (TOKEN 1) agent message: recipe persistence message
return new token (TOKEN 1) agent message: recipe persistence message
now hold user_data['action'] = 'TOKEN 1'
agent message: recipe persistence message

An example of an actual interaction with the agent performing this task is as follows.

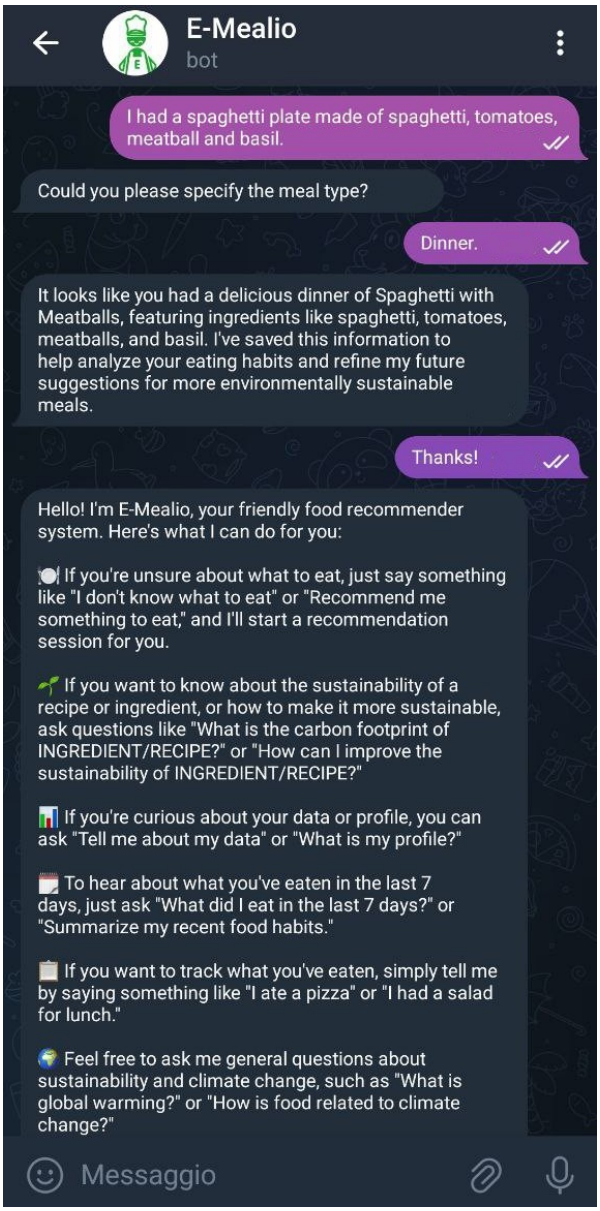


Figure 5.32: The user informs the agent that they have eaten a spaghetti dish along with its ingredients. The agent requests the meal type, which the user then provides. The agent records the consumed recipe and informs the user that it has been saved. After the user's next interaction, the agent, now in the hub state, presents the hub again to initiate a new conversation.

5.4 Testing the Agent: Unit Tests for LLM-Based Software

The functionalities described previously are based on a complex interaction between LLM-generated data, such as JSON and state tokens, and the state controller, which also calls specific services, passing the generated data as input. In this context, the consistency of the LLM model's behavior is a key factor that must be ensured. For this reason, a generally low temperature was chosen for every prompt, with minor exceptions for those designed to produce only the user response.

To ensure behavioral consistency over time and possibly between different LLM models, a battery of unit tests was built to verify, for each significant interaction, that each change of state and data production was performed as expected. Each provided test conceptually verifies that every functionality follows the correct path in the theoretical FSM models previously illustrated.

For brevity, the tests will not be described individually; however, two main categories of tests were designed.

- The first category aims to test a specific state transition, even those that are not required to produce a final state, ensuring that the correct state transition is produced.
- The second category aims to test the entire transition between the interaction and the production of a user response. In this case, only the final state is checked, implying that its presence must also depend on the correct transition between internal states required for information exchange.

In both cases, the correct production of auxiliary information and, if applicable, user data is also checked. The two approaches are used in a mixed methodology, depending on the need to focus on a specific or global transition of the functionality, also considering its criticality.

All the unit tests are provided in the file *projectRoot/tests.py*.

These tests were used to ensure the compatibility of each prompt with the Claude Sonnet 3.5 model [68], which was used as an alternative to GPT-4o in the experimental phase with users, as discussed subsequently.

Chapter 6

Experiments and Results

This chapter presents the results obtained from the experimental phase, which involved a sample of 12 users from diverse backgrounds. The experiments were conducted under the guidance of Dr. Arianna Boldi, an expert in psychology and user experience, following a mixed-methods approach that combined interviews, think-aloud protocols (were users are encouraged to verbalize their inner thought while interaction with a system), and a final questionnaire. This framework aimed to achieve the following objectives:

- Understanding users' **personal background**, including their current eating habits, needs, and their relationship with sustainability and food-related technologies.
- Gaining in-depth insights into **how users perceive their interactions with the agent**, relating their experiences to their personal background. This provides a strong qualitative analysis of the agent's strengths, weaknesses, and potential improvements.
- Summarizing **user feedback** to facilitate further quantitative analysis and better understand overall user impressions.

Each user interacted with a version of the agent, with half of the participants assigned to a system powered by *GPT-4o* and the other half to one powered by *Claude Sonnet 3.5*. This

setup was designed to assess whether the choice of LLM had a measurable impact on user experience, particularly regarding tone, informativeness, and usability.

During each session, key aspects evaluated included:

- Agent tone: How the choice of words affects user engagement, fosters a positive interaction, and encourages more sustainable dietary choices.
- Agent informativeness: The effectiveness of the agent in conveying useful information and the level of trust users place in its recommendations, particularly concerning dietary needs and preferences.
- User-friendliness: The clarity and comfort of the interaction flow, as well as the usability of the interface and the functionalities provided.

The main objectives of the evaluation phase, which were also analyzed quantitatively through survey results, were to:

- Assess whether the agent's functionalities operate correctly and coherently, aligning with user expectations while ensuring a user-friendly interaction.
- Evaluate the extent to which the provided information influences users' dietary choices and their overall awareness of food sustainability.
- Analyze whether the underlying LLM affects users' perception of the agent.

This chapter is structured as follows. First, a brief overview of the participants is presented, identifying them by user ID and summarizing their demographic data, as well as their relationship with food, sustainability, and technology. Next, a recap of the think-aloud phase is provided, analyzing each core functionality of the agent, capturing user impressions (referenced by user ID), and offering suggestions for future improvements. Finally, an analysis of the questionnaire results is presented, providing quantitative insights into the three main objectives outlined above.

6.1 User Demographics and Background

This section presents data about the 12 users involved in the trial. This is essential because the think-aloud protocol places significant emphasis on individual user experiences, where their reactions and impressions are closely linked to their personal background and preferences. Providing detailed user profiles allows for a better understanding of their opinions about the system, offering deeper insight into their thoughts.

Users were selected through purposeful sampling [69], a technique aimed at identifying individuals with specific characteristics that enable a comprehensive understanding of the studied phenomenon, in this case, the usability of the E-Mealio agent. Participants with diverse dietary choices and preferences were chosen to ensure a varied sample, allowing for a more thorough analysis of the system. Additionally, two users, **P11** and **P12**, were recruited via convenience sampling through a call to action in a food-related university course.

For each user, demographic information is presented first, followed by a brief analysis of their dietary needs and concerns, as well as their perspectives on food sustainability. Finally, an overview of their past experiences with food-related technology and chatbots is provided.

- **User P01**
  - Age: 28, Gender: Female, Educational Qualification: High School Diploma, Current Occupation: Undergraduate Student.
  - **P01** is interested in trying new, healthy recipes with fewer processed ingredients. She is also motivated to learn more about food sustainability and increase her awareness.
  - **P01** has not used any apps or bots to influence her dietary habits, except for occasional recipe searches on Google. Additionally, she reported using chatbots moderately throughout the week.

- **User P02**
  - Age: 36, Gender: Male, Educational Qualification: PhD, Current Occupation: Employee.
  - **P02** is currently following a balanced diet to manage health issues, such as hypertension and irritable bowel syndrome (IBS), with the goal of avoiding medication. He is also concerned with the aesthetic aspects of weight loss. His diet involves managing the consumption of meat and fish due to ethical concerns (animal welfare) and health reasons. However, the diet still requires significant meat consumption, which he finds challenging.
  - **P02** has not used any dietary apps before, primarily due to skepticism about their effectiveness. He downloaded an app for IBS management but was doubtful about its legitimacy, particularly because it was a paid app. He would be more likely to trust an app if it were based on scientific research and had credible sources of data. Additionally, he reported using chatbots frequently throughout the week.
- **User P03**
  - Age: 31, Gender: Male, Educational Qualification: Bachelor's Degree, Current Occupation: Freelancer.
  - **P03** aims to eat "healthier" to avoid gaining weight, as he finds it difficult to engage in much physical exercise. He supports local and organic products when possible, often buying from a friend who runs an organic store. He has educated himself on sustainability through articles, YouTube videos, and his friend's advice.
  - **P03** has never used any dietary apps and has not considered them as part of his dietary habits or lifestyle improvements. He reported being an occasional chatbot user, with only a few interactions per month.

- **User P04**
  - Age: 30, Gender: Male, Educational Qualification: Master's Degree, Current Occupation: Freelancer.
  - **P04** is particularly concerned with breakfast, as it is the first meal of the day. He would like more variety but lacks the time to research and gather information. Based on personal experience, he finds the classic Italian breakfast makes him hungry, while avoiding sugars helps him feel more energetic until lunch. He is interested in sustainability and health but has not yet started exploring these topics in depth. He feels that sustainability and health are two separate areas and is still in the discovery phase, seeking guidance.
  - **P04** has never used any food-related apps or technology. Furthermore, his usage of chatbot technology is limited to a few interactions per month.
- **User P05**
  - Age: 29, Gender: Female, Educational Qualification: Master's Degree, Current Occupation: Student.
  - **P05** follows a specific diet due to gluten and lactose intolerances, as advised by doctors. While she eats a variety of foods, she dislikes fish and has started eliminating gluten-containing ingredients for better health. Although her diet has improved her well-being, she still experiences some discomfort from certain foods. She is particularly mindful of the environmental impact of her food choices. Living abroad has made her more cautious about food quality, especially regarding non-local fruits and vegetables. She avoids certain products like avocados due to sustainability concerns and prefers seasonal produce.
  - **P05** has never used any food-related apps or technology, as she trusts her own

ability to manage her diet independently. She also reported rarely using chatbots, with only occasional interactions with this technology.

- **User P06**
  - Age: 25, Gender: Female, Educational Qualification: Bachelor's Degree, Current Occupation: Student.
  - **P06** has recently realized the need to consume more protein and fewer fats to achieve a better balance, especially since she goes to the gym. She became aware of this by tracking her food intake, counting calories, and monitoring macronutrient consumption throughout the day. For **P06**, a healthy diet means maintaining a proper macronutrient balance. She would like to prioritize local and sustainable products, but as a student living away from home, affordability is a challenge. She chooses organic products like honey and prefers foods with fewer additives and preservatives. Since moving out, she has become more flexible in her diet, occasionally incorporating plant-based protein sources like tofu and seitan, though not consistently. Economic constraints remain a key barrier.
  - **P06** has used food-tracking applications such as Lifesum and MyFitnessPal to monitor her diet and initially used them for weight loss. She also uses a Fitbit to track calories burned and macronutrient intake. Additionally, she stated that she is an occasional user of chatbots, with only a few interactions per month.
- **User P07**
  - Age: 27, Gender: Female, Educational Qualification: Master's Degree, Current Occupation: Employee.
  - **P07** wants to increase her fruit intake but struggles due to logistical issues and disorganization, often resorting to substitutes. Living alone and managing work

makes it difficult for her to integrate fresh fruit into her diet. She is also lactose intolerant and aims to incorporate more plant-based proteins. While preparing for the public competition to become a police commissioner, she needed to lose fat mass and started a diet under the guidance of a nutritionist friend. Her diet was structured by food categories (e.g., proteins, carbohydrates) rather than specific recipes, with the main difficulty being portion measurement. **P07** primarily focuses on maintaining a healthy diet.

  - She once downloaded the Yummi app to evaluate supermarket products but stopped using it because she found the ratings too polarized, as they did not consider the overall nutritional profile of the products and tended to demonize certain ingredients. She stated that she rarely uses chatbots, with no consistent monthly usage.

- **User P08**

  - Age: 25, Gender: Male, Educational Qualification: Bachelor's Degree, Current Occupation: Student.
  - **P08** has recently started paying more attention to his diet and has noticed an improvement in how he feels. With a fast metabolism, he tends to consume many sweets and would like to integrate more diverse foods into his diet. A lifestyle change and occasional fatigue triggered this shift. He avoids eating red meat too frequently. Coming from a family with an agricultural business that raises animals, he experiences a contrast between his family's dietary habits and his personal desire to transition to a vegan diet in the future. **P08** is interested in sustainability but faces logistical and cultural challenges in fully committing to a vegan lifestyle. He has a dual relationship with family influence: on one side, it provides a positive approach to food, but on the other, it makes adopting a fully vegan diet difficult.
  - He uses a fitness application for step counting but has never used food-related apps,

though he is aware of them. Additionally, he stated that he rarely uses chatbots during the month.

- **User P09**
  - Age: 26, Gender: Female, Educational Qualification: Bachelor's Degree, Current Occupation: Student.
  - **P09** has struggled with eating disorders in the past and has not yet found a balanced relationship with food. She associates certain foods with anxiety, for example, feeling comfortable eating large amounts of ice cream while experiencing stress around consuming pasta. **P09** aims for a more balanced diet and has educated herself extensively on nutrition but acknowledges that some of her beliefs may be distorted. She avoids healthy foods as a way of distancing herself from past unhealthy cycles. Additionally, she suffers from gastritis.
  - While she has used many food-related applications in the past, she currently finds them distressing, as they remind her of difficult periods in her life. She stated that she rarely uses chatbots during the month.
- **User P10**
  - Age: 31, Gender: Female, Educational Qualification: High School Diploma, Current Occupation: Employee.
  - **P10** has been vegan for 10 years and follows a well-balanced plant-based diet. She is generally satisfied with her eating habits but would like to reduce her consumption of junk food. **P10** considers her vegan diet a sustainable choice. However, she aims to be more mindful of food packaging, a habit that has become more challenging due to time constraints and limited local availability, as she does not own a car.

- In the past, she used apps like Yuka to scan barcodes and assess product suitability when transitioning to a vegan diet. Additionally, she barely uses chatbots, with almost no interaction during the month.

- **User P11**
  - Age: 41, Gender: Female, Educational Qualification: High School Diploma, Current Occupation: Student.
  - **P11** follows a predominantly vegan diet and has recently started reducing sugar and processed food intake for health reasons. She follows an anti-inflammatory diet, aiming to minimize processed foods and sugars, which she previously consumed as a form of emotional consolation. **P11** considers sustainability important but acknowledges that she has not explored the topic in depth. She perceives her mostly vegan, unprocessed diet as already sustainable.
  - She has not used specific food-related apps but has experimented with conversational AI tools like Microsoft Copilot, finding them useful. While she follows a nutritionist's guidance, she also integrates insights obtained from chatbots, which she reported using moderately during the week.

- **User P12**
  - Age: 32, Gender: Female, Educational Qualification: Master's Degree, Current Occupation: Student/Employee.
  - **P12** aims to increase her water intake and reduce meat consumption. She often forgets to drink water due to being focused on other tasks. While she finds meat consumption convenient, she does not perceive it as healthy and prefers white meats over red meats. **P12** associates sustainability with locally sourced food and being aware of product origins, viewing health and sustainability as interconnected.

– She has previously used Lifesum as a food diary but does not appreciate external suggestions, as she considers them potentially unreliable. She also reported using chatbots moderately during the week.

The following pie charts summarize the data previously presented about the users.

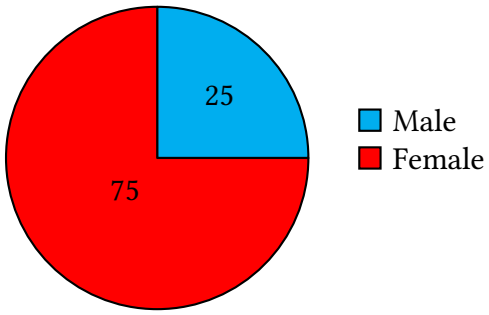


Figure 6.1: Gender Distribution

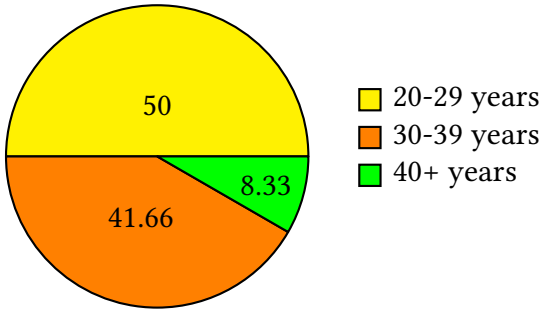


Figure 6.2: Age Distribution

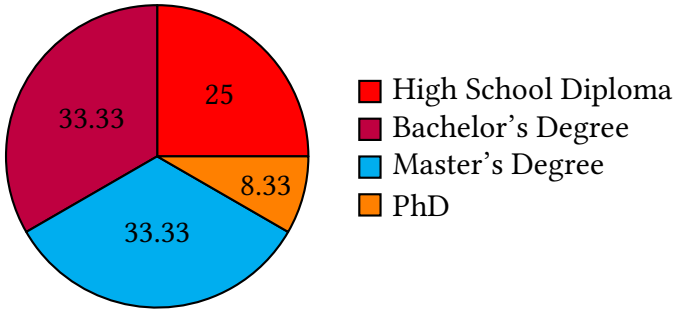


Figure 6.3: Educational Qualification

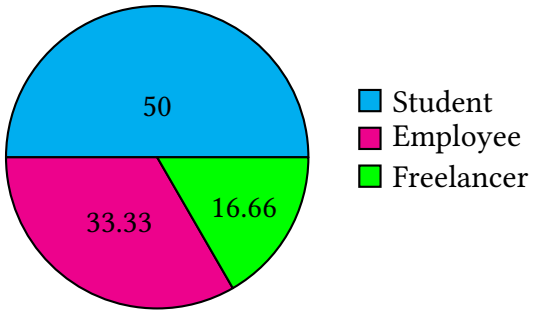


Figure 6.4: Occupation Distribution

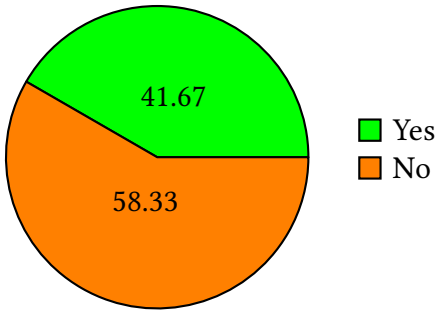


Figure 6.5: Previous Use of Food-Related Technologies

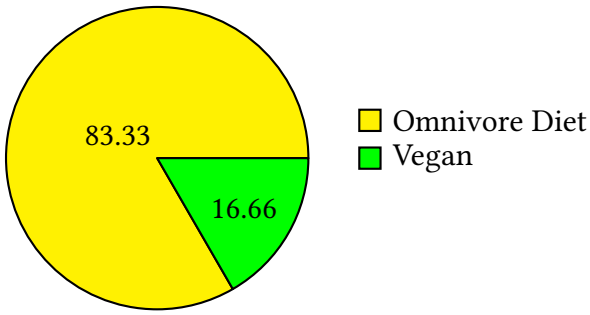


Figure 6.6: Diet Type Distribution

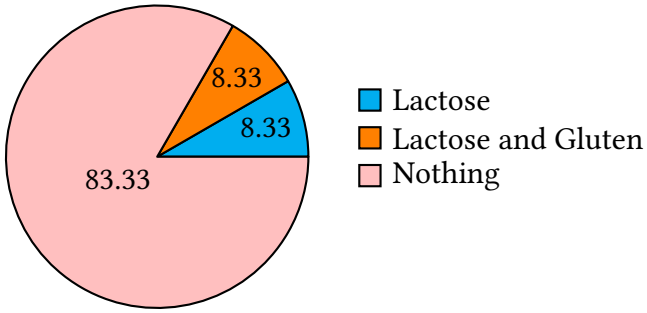


Figure 6.7: Reported Allergies or Intolerances

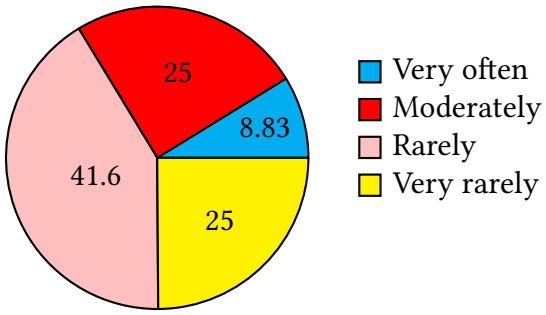


Figure 6.8: Chatbot Technology Usage Frequency

6.2 Agent Usability Analysis via Think Aloud Protocol

In this section, the insights obtained from the think-aloud interviews are presented. The Think Aloud Protocol [70] is a method in which users are asked to interact with a system while verbalizing their thoughts and reflections. This approach allows researchers to access users' cognitive processes, providing a clearer understanding of any struggles or challenges they may face while using the system.

For this study, participants were encouraged to freely interact with the agent, focusing on the functionalities they found most interesting.

Each session was conducted online using the Cisco Webex platform. This choice was made

due to its accessibility, allowing users to join the session without needing to download any additional software. Cisco Webex also offers a session-recording function, which was utilized to record each interview for further analysis. Before starting, each user was asked to share their screen while interacting with the agent, after providing their informed consent regarding screen sharing and the recording of the session for documentation purposes. Dr. Boldi conducted the user background interviews, along with more general questions about usability, while I focused on asking the more technical questions and providing support to users whenever they encountered difficulties. Each session was followed by a debriefing to identify key insights that emerged, as well as to evaluate how to structure subsequent sessions in order to address any remaining un-explored aspects of the agent's usability.

The analysis is organized by functionality, with a summary of each user's experience for every feature they interacted with.

6.2.0.1 User Data Acquisition

The *User Data Acquisition* functionality was tested by all 12 participants, as it is automatically invoked when a user interacts with the agent for the first time.

The agent's request for personal data was clearly understood by all users; however, some participants, namely **P06**, **P10**, and **P11**, expressed reluctance to share certain personal details, such as their surname or full birthdate, due to privacy concerns. Additionally, **P09** observed that a key piece of information, such as gender, was not requested. These observations highlight the importance of privacy for some users and suggest that the necessity of collecting certain data should always be justified with compelling examples.

Half of the users provided their personal information in list form rather than engaging in a conversational manner, despite the agent's message indicating that data could be communicated naturally. While this behavior did not affect data acquisition, it suggests that the format

of the request influences the way users respond, sometimes leading to less natural interactions.

In one case, **P12** sent both her personal data and a separate request for the bot in the same message. This triggered a response where the agent summarized her data and asked for confirmation. However, after confirmation, the data was not actually saved, and the agent requested it again, indicating a flaw in data processing.

Regarding the ability to communicate allergies and dietary restrictions, all users appreciated this functionality and provided suggestions for improvement. These included the option to specify generally disliked ingredients, as well as the ability to report medical conditions that could affect dietary recommendations, such as irritable bowel syndrome or gastritis. Additionally, user **P08** suggested allowing the agent to acknowledge dietary preferences that they aspire to adopt, in order to receive nudges toward those choices.

With regard to disliked ingredients, **P06** stated that she dislikes fish, but the agent treated this information as an allergy. While this is not entirely incorrect, a dedicated mechanism for managing food preferences separately from allergies could enhance the agent's reliability.

The agent's recap message was generally perceived as clear and understandable. However, **P02** noted that the agent referred to his Telegram nickname, which was not appreciated. In a couple of cases, the bot miscalculated users' ages when summarizing their information.

The reminder question was always understood, and all users appreciated that it was an optional feature. While some consented to reminders, others felt that the default frequency was too high and suggested the ability to customize the frequency and timing of reminders, a feature that is not yet implemented.

Suggestions for Improvements:

- Clearly explain why certain information is requested and avoid collecting data that is not actively used by the system.

- Enable users to briefly discuss the necessity of providing specific information with the agent.
- Improve the mechanism that tracks user information even when unrelated messages are sent.
- Implement a dedicated system for tracking disliked ingredients, medical conditions that influence diet, and users' willingness to transition to a new diet.
- Ensure that the agent does not communicate incorrect or unnecessary personal information, such as age or Telegram usernames, to improve perceived reliability.
- Allow users to customize reminders via the profile update function and inform them of this possibility from the start.

6.2.0.2 The Main Hub

The *Main Hub*, also referred to as the "menu," was tested by all users since it is mandatorily displayed immediately after the user data acquisition phase. The message introducing the agent's capabilities was clear to all users. However, some participants (**P02**, **P06**, and **P08**) found it too lengthy and verbose. **P02** suggested that a button-oriented interface would be clearer and easier to navigate.

The use of emojis at the start of each functionality was generally well received, as they helped break up the block of text. Additionally, the final food-related joke was consistently appreciated, being interpreted as a sort of "reward" for reading through all the options.

Regarding navigation, the ability to return to the menu using the */start* command was functional, but **P07** found it difficult to associate the term "starting point" with the menu. She suggested that this functionality should be described more explicitly.

All available options were understood by users; however, **P06** attempted to ask a question about the meal-tracking function. While the system's response and explanation were clear,

she noted that a more detailed description of this feature in the menu itself would have been helpful from the start.

Suggestions for Improvements:

- Simplify the menu text or introduce a more button-oriented interface (which is possible for Telegram bots).
- Clearly indicate that the ”starting point” refers to the menu.
- Provide a more detailed description of the meal-tracking functionality within the menu.
- Improve intent recognition to ensure that requests phrased in a more complex way are properly understood and not overlooked.

6.2.0.3 Recipe Recommendation

The *Recipe Recommendation* function was tested by all users except for **P08**, who preferred to ask the agent more general informational questions. This functionality was evaluated under different conditions: users **P05** and **P07** tested it with their specified allergies, while users **P10** and **P11** tested it with their dietary restrictions. User satisfaction varied depending on the presence of strict constraints, with those having more specific requirements showing different levels of contentment.

Users who explicitly requested ingredient-based recommendations had mixed experiences. While **P01** (who requested a vegetable-based recipe) and **P10** (who asked for a tofu-based recipe) were satisfied, others encountered issues. For example, **P05** (who requested a noodle-based dish), **P06** (who asked for a recipe containing eggplant, seitan, and tomato), **P07** (who requested a rice and carrot dish after rejecting a previous suggestion) and **P11**(who requested a recipe containing pumpkin and potato) were unable to receive recipes matching their requests, even after multiple attempts. This resulted in a lower overall appreciation of the functionality.

The system managed vegan restrictions effectively, making users **P10** and **P11** (when she asked a second suggestion) generally satisfied with their recommendations. However, allergies were not consistently accounted for, as users **P05** and **P07** were sometimes given recipes containing cheese or milk related products, despite having provided lactose allergy in the user data acquisition phase.

Users **P02** and **P04**, who did not specify any constraints, were dissatisfied with the recommendations, as their unstated preferences were not met. The remaining users either received satisfactory recommendations on the first or second attempt.

All users understood the role of meal types (e.g., breakfast, lunch, break, or dinner). Some provided this information intuitively, while others were prompted to specify it.

User **P04** suggested that the agent should ask more questions before providing a recommendation, guiding the user through a structured, tree-based selection process. This approach would help clarify which constraints are being considered, ensuring that users understand how their inputs influence the final suggestion. Currently, the agent's constraint-handling process is not transparent, making it unclear which factors are being taken into account.

The response format was clear, and the presence of a recipe URL was generally appreciated. However, opinions on sustainability data were mixed. While the 5-star Likert scale representing overall sustainability was understood and appreciated, the raw data on carbon footprint and water footprint were perceived as too technical. Even though the agent attempted to contextualize these values with qualitative descriptions, they were still seen as overly complex. **P02** pointed out that the recommendations could benefit from additional health-related information, making them more appealing to users who are not solely focused on sustainability.

User **P11** noted that the agent frequently compared ingredients to meat, even though she is vegetarian. She found this type of comparison both irrelevant and trivial, suggesting that more meaningful comparisons using ingredients relevant to the user's diet would be more

useful.

Finally, **P12** observed that the suggested recipes were often from a pool of options that did not align with Italian cuisine, despite her nationality being explicitly stated.

Regarding interactions following the recommendation, most users found it unclear that they needed to explicitly accept or reject the suggested recipe. **P09** suggested separating this confirmation request from the recipe recommendation itself to emphasize its importance. Additionally, the way the agent communicated the outcome of user feedback was unclear for **P10** that did not realize that accepting a suggestion meant saving it as a consumed recipe.

Only one user, **P09**, asked follow-up questions about the suggested recipe. The responses were informative and clear, and this feature was well received.

Suggestions for Improvements:

- Relying solely on semantic similarity between ingredients sometimes leads to irrelevant recipes. More precise embedding models should be tested, and/or semantic similarity should be used only as a fallback, prioritizing a more structured query-based approach for greater accuracy.
- Allergy tags in the recipe database should be reviewed and corrected, as some recipes may have incorrect labels.
- When users request a new recipe with additional constraints, these should be added to the previously provided ones or replace them if incompatible.
- Consider asking the user additional questions before providing a recommendation to clarify which constraints are being applied, rather than relying solely on the information they initially provide.
- Sustainability data should be presented in a more user-friendly way.
- Recommendations should consider factors beyond just ingredient sustainability to be

more effective.

- Ingredient comparisons should be tailored to the user's dietary preferences (e.g., warning a vegan about the sustainability of red meat is unnecessary).
- The recipe pool should be expanded to better align with users' cultural and national preferences.
- The user acceptance request should be presented in a separate message and its outcome should be clearly communicated.
- Take into account the number of portions when recommending a recipe.
- Consider the required kitchen tools for a recipe, explicitly stating in advance if a necessary tool is unavailable to avoid recommending recipes that require it.

6.2.0.4 Recipe Sustainability Improver

The *Recipe Sustainability Improver* function was tested by only two users, **P07** and **P09**. This limited engagement is likely due to the feature being somewhat hidden within the broader presentation of the agent's sustainability expertise in the menu message, making it less distinguishable as a separate function.

Users found it difficult to invoke the function naturally, as it required a comma-separated list of ingredients, which did not align with typical conversational interactions. The sustainability improvement suggestions received mixed reactions.

User **P07** requested an improved version of a recipe containing pasta, tuna, oil, and tomatoes. The agent suggested a more sustainable alternative, Penne Arrabbiata, which was perceived as too different due to the removal of protein. This led to a follow-up conversation where the user inquired about suitable protein alternatives. Eventually, they reached a satisfactory modification, prompting **P07** to accept the suggestion. However, the accepted version was not correctly saved with the modified ingredients, frustrating the user.

User **P09** attempted to improve a recipe previously recommended by the agent. However, the system was unable to process cross-function requests, requiring the user to manually re-enter the ingredient list. The first suggested improvement was only loosely related to the original dish, causing discomfort and dissatisfaction. However, upon requesting another improvement, the user received a more relevant suggestion, which was ultimately accepted.

Both users understood the distinction between the *base recipe* and the *improved recipe*, but they found the agent's suggestions overly rigid and unnatural.

Additionally, **P06** attempted to invoke this functionality while engaged in another conversational loop. The response they received, though clear, was not generated by the intended sustainability improvement process. This highlights a broader issue with the agent's interaction flow, which will be discussed later in this section.

Suggestions for Improvement:

- Make this function more distinct from the *expert sustainability consultation* feature to improve its discoverability.
- Improve the ingredient recognition mechanism to allow users to input ingredients in a more natural way, rather than requiring a strict comma-separated format.
- Use a more nuanced substitution mechanism for impactful ingredients instead of simply recommending a more sustainable recipe, to make suggestions feel more natural.
- Ensure that any modifications discussed during the improvement process are properly tracked so that the final saved recipe reflects the accepted version.

6.2.0.5 Sustainability Expert Chat

The *Sustainability Expert Chat* function was tested by the majority of users, specifically 8 out of 12. User satisfaction varied significantly depending on the nature of their inquiries. Since

each user engaged with the function in a unique way, asking about different aspects of sustainability, their experiences will be analyzed individually.

- **P02** inquired about the carbon and water footprint of an entire meal, providing a recipe along with its ingredients and two additional servings (a couple of fruits). The agent understood the request but responded with a general explanation of carbon and water footprint concepts before relating them to the mentioned ingredients.

  The user found the response too vague, as it lacked the expected quantitative details about the ingredients. This behavior likely resulted from a misinterpretation of the request[1]. As a result, the user was somewhat dissatisfied with the response.

- **P03** began interacting with the agent by asking a broad question about how to combat climate change through food choices. The response was perceived as complete and informative but lacked external references or bibliographic sources. The user then inquired about media content and books that could help build awareness on food sustainability. The agent recommended well-known books such as *We Are The Weather* [6] and *The Omnivore's Dilemma* [71], along with documentaries like *Cowspiracy* [72] and *Kiss The Ground* [73][2]. The user verified these references online to confirm their existence and ensure they were not hallucinated responses. Encouraged by the recommendations, the user requested further details about the suggested documentaries, which were perceived as informative and comprehensive.

  Following this, the user asked whether dairy and meat could be eaten together at dinner.

[1] A log analysis revealed that the agent invoked the "concept" expert instead of the ingredient or recipe specific expert. This was likely triggered by the wording of the question, which led the system to prioritize a conceptual explanation over a direct footprint analysis.

[2] This response was surprising, given that the sustainability expert function at the "concept" level relies exclusively on the model's latent knowledge.

While their intention was to understand the health implications, the agent initially responded from a sustainability perspective. This led to a follow-up question about protein intake and, finally, a specific inquiry regarding the healthiness of combining dairy and meat. The agent's response was clear and informative, also advising the user to consult a professional nutritionist if necessary. The user appreciated the completeness of the answer and the fact that the agent did not attempt to replace human expertise.

The conversation continued with a question about substituting pasta with legumes. The agent provided an informative and satisfactory response[3].

Finally, the user concluded the interaction with a farewell message, returning to the main hub. Despite needing to rephrase some questions, the overall experience was positive, as the received information was ultimately clear and complete.

- **P05** inquired about the sustainability of the Pak Choi ingredient. The provided answer[4] was perceived as informative and clear. However, the typical comparison with animal products was considered too trivial, with a preference for a more compelling comparison. The user also noted the absence of information regarding potential regions of cultivation, suggesting that the agent does not account for the environmental impact of importing such an ingredient. Additionally, the user pointed out that the methodology behind the CFP and WFP values was not provided, nor were any bibliographic references included. They emphasized that this aspect should be addressed to allow interested users to access reliable sources for such data.

[3]This question was answered within the same conversational loop, without invoking any retrieval-augmented generation (RAG) mechanisms. While the response was still relevant, it did not leverage the agent's capability to retrieve ingredient data from its database. This suggests that the conversational loop prompt should be refined to better recognize different contexts.

[4]Correctly generated by invoking the sustainability expert at the ingredient level.

- **P06** asked about the sustainability of consuming fish and potatoes in the same meal. While this request should have triggered the sustainability expert at the recipe or ingredient level, the agent failed to retrieve the correct information due to an occasional bug in ingredient extraction that requires further analysis. As a result, the response was unrelated, significantly lowering the perceived reliability of the agent.

  The user then inquired about making a more sustainable version of pasta with tuna. However, since the agent did not recognize the context change, it provided a general response instead of invoking the correct service for recipe improvement. While the answer was perceived as informative, it did not follow the intended procedure.

  Additionally, the user suggested that the agent should be able to recommend specific brands of products while considering their cost. The conversation continued with the agent providing a related recipe, which was appreciated[5].

  Finally, the user also recommended that the number of portions should be taken into account when providing recipe suggestions. Despite the errors and the unintended use of internal functions, the overall conversation was perceived as interesting.

- **P08** asked about the sustainability of milk. However, the initial request was not recognized, causing the agent to loop back to the hub, which highlights the need for improvements in intent recognition. Once the correct service was invoked, the response was perceived as overly technical and dense, making it difficult for users to understand. The user suggested presenting the information in a more accessible way, such as through infographics, or offering a function that would allow users to invoke a more in-depth analysis in a different context (such as a separate tab, which is currently not possible

[5]Notably, this recipe was not provided by the recipe suggestion function, as the agent was still within the expert conversational loop. Addressing this issue is one of the key improvements needed.

on Telegram bots). Finally, the user pointed out that providing bibliographic references or links to external resources could enhance the perceived reliability of the information, which aligns with feedback from other users emphasizing the importance of citing sources for increased credibility.

- **P09** did not ask anything that would trigger the sustainability expert chat. However, she asked the bot for suggestions to improve her mood, citing some reluctance about food. The agent recognized that the request was outside its scope and responded with a simple message suggesting contacting a psychology expert. It then presented the menu options again, ensuring that the interaction remained within the bot's capabilities. This behavior was appreciated by the user, who acknowledged the bot's effort to avoid becoming an uncanny conversational agent that could manipulate users on an emotional level.

- **P10** first inquired about the sustainability of milk. The response provided by the appropriate service was perceived as informative, reinforcing the user's vegan preferences. The user then asked for the recipe for oat milk, which triggered a context change. The agent replied by notifying the user that it was exiting the current topic and then presented the menu again. This behavior was perceived as annoying, though understandable. After re-entering the request, the agent provided a sustainability analysis of oat milk, which was informative but not entirely related to the original question. This led the user to request the actual recipe for oat milk, which triggered another context change. This was perceived as particularly frustrating[6], causing the user to express dissatisfaction with the rigidity of the interaction. Overall, while the conversation was informative, it was perceived as too "on rails" to be fully enjoyable.

[6]In this instance, the agent was expected to provide the ingredient list given that it is injected into the prompt in order to analyze its sustainability. More development of the prompts generating this kind of response is needed to ensure the agent can handle this capability without exiting the context.

- **P11** inquired about the sustainability of potatoes and received an informative and comprehensible response. However, the user pointed out that comparing the sustainability of potatoes with beef was too trivial and irrelevant to her, given her vegan diet. She suggested that the agent should offer more diversified comparisons tailored to different dietary preferences. At the same time, she appreciated the comparison between potatoes and other plant-based options, such as rice and almonds.

  In a separate session, the user posed the same question about potatoes, but this time specifically asked for more sustainable alternatives. The agent responded that while potatoes are among the most sustainable options available, sweet potatoes and cassava are also viable alternatives. The user found this response helpful and appreciated the comparison, resulting in a positive overall impression of the functionality.

The reported user experiences highlight the *Sustainability Expert Chat* as an interesting and useful feature with significant room for improvement. Enhancements should focus on refining the agent's ability to recognize context shifts, personalizing responses to better suit individual users, and incorporating external resources to support its recommendations.

Suggestions for Improvement:

- Adjust the sustainability expert prompt to better interpret user intent and minimize misunderstandings.
- Reinforce the capability of recognizing context changes within conversational loops, allowing the system to return to the hub and load RAG-based functions when needed, rather than relying solely on the model's latent knowledge for responses.
- Enhance the ability to mention studies and bibliographic references through RAG-based queries to increase the perceived reliability of the provided information.

- Analyze the ingredient extraction error, which is likely due to some contingencies related to the phrasing of the request and the prompt used for the extraction process.
- Enhance the recipe sustainability expert prompt to also respond to questions about the recipe itself, not just its sustainability.
- Enhance the agent's ability to provide analyses and comparisons that consider the user's actual diet and habits, avoiding trivial or irrelevant comparisons.

6.2.0.6 User Profile Recap and Update

The *User Profile Recap and Update* function was tested by only a couple of users. This low number is likely due to the fact that most users provided their data during the initial interaction and did not feel the need to review or update them. The users who did test this function were primarily interested in exploring the type of interaction it offered.

User **P07** requested to review her personal data and received them in a complete and correct format. While she appreciated the ability to update her information, she did not test this functionality, as she subsequently used the "*/start*" command to explore another function.

User **P11** also requested to review her data and received a comprehensive profile summary. However, when prompted about updating her profile, she directly provided a new piece of information (disliking tofu) instead of responding with the expected yes/no answer. This led to the agent asking again for confirmation before updating the data. Besides the current limitation of not having a dedicated field for tracking disliked ingredients, the strict expectation of a yes/no response before processing an update was seen as an unnecessary constraint. The agent should be more flexible in interpreting user intentions.

Despite the limited number of interactions, these tests provided valuable insights for improvement.

Suggestions for Improvement:

- Improve flexibility in recognizing when a user wants to update their data, without strictly requiring a yes/no response.
- Consider adding functionality to track user preferences, including liked and disliked ingredients, as previously mentioned in the user data acquisition section.

6.2.0.7 Food Diary (Meal Tracker)

The *Food Diary* functionality (also referred to as the *Meal Tracker* during the interviews) was tested by 4 out of 12 users.

User **P07** reported eating a roast beef for dinner. However, the agent's response was perceived as too neutral and unengaging, as it neither provided any analysis or information about the meal nor initiated a discussion about it. A similar concern was raised by user **P08**, who logged a breakfast of milk and biscuits and found the agent's response lacking in depth.

User **P09** used the function to log a meal of pizza with tuna and onion. In this case, the system correctly prompted the user to specify the meal type, and the request was successfully processed. Similarly, user **P12** tracked the consumption of a toast. However, in this last instance, the agent did not ask for an ingredient list, despite "toast" being a recipe rather than a single ingredient. Ideally, this should have triggered a request for the ingredient list. Nonetheless, the user did not perceive this omission as an issue.

Both **P09** and **P12** found the recap message at the end of the meal tracking interaction to be satisfactory. They did not expect an immediate analysis at that stage and considered the overall behavior of the functionality acceptable.

Suggestions for Improvement:

- Consider providing an initial brief analysis of the logged meal to encourage user engagement and discussion.

- Improve the function's ability to recognize when a user is referring to a recipe and prompt them to provide an ingredient list if necessary.

6.2.0.8 Weekly Food Diary Recap

The *Weekly Food Diary Recap* function was tested by 4 out of 12 users. Users **P07**, **P08**, and **P09** tried it immediately after using the meal tracking function, recognizing their strong correlation.

User **P07** anticipated receiving a somewhat negative evaluation, given that the only logged meal included beef. However, she expressed that the agent should be capable of contextualizing such information more comprehensively. She emphasized that meat should not be demonized but rather consumed in moderation. Additionally, she noted that certain individuals, such as teenagers, could benefit from the iron provided by red meat. Therefore, she suggested that the bot should acknowledge not only the environmental impact but also the nutritional benefits of such meals. Furthermore, she observed that while the agent had the capability to discuss the provided analysis, this feature was not sufficiently highlighted. As a result, she were not encouraged to ask follow-up questions or engage in deeper discussions.

Users **P08** and **P09** generally appreciated the analysis and its star-based Likert scale. Their critiques aligned with those mentioned in previous discussions regarding sustainability analysis, particularly the lack of external references for sustainability data.

User **P10** tested the function to review her accepted meal suggestions and found that the provided analysis was consistent with the previous evaluations of the recommended meals.

Additionally, users **P05** and **P06** noticed the function in the menu but did not test it. Both suggested that the functionality should allow customization of the analysis period, enabling users to review data for different time spans (e.g., an entire month or just a few days) or analyze a specific timeframe. User **P05** also proposed integrating a more health-oriented assessment

alongside the sustainability analysis to provide a more holistic dietary evaluation.

Suggestions for Improvement:

- Allow users to customize the time span for the analysis.
- Improve the visibility of the discussion feature, encouraging users to engage with the agent for further insights.
- Enhance the agent's capability to evaluate meals from a health-oriented perspective, complementing the sustainability assessment.

6.2.0.9 Tone and Informativeness

Beyond the agent's core functionality, users were asked to share their impressions regarding its overall tone and its ability to provide reliable information.

Almost all users agreed that the agent maintained a friendly tone without compromising reliability. This aspect is crucial, as it ensures that the interaction remains both pleasant and informative.

Regarding the informativeness of the agent, most users identified areas for improvement. Users such as **P02**, **P06**, **P08**, and **P09** expressed a preference for a more visual approach to data presentation, suggesting the inclusion of graphical elements alongside references to external sources. Others, including **P05**, **P07**, **P11**, and **P12**, indicated a desire for more geographic information and diversified comparisons when analyzing the sustainability of ingredients. They also noted that comparisons should go beyond the default contrast with meat to provide more meaningful insights.

Overall, while the agent's tone is well-received, there remains significant room for improvement in terms of informativeness. However, the current implementation is already perceived as a strong foundation for further enhancement.

Suggestions for Improvement:

- Incorporate references to external sources to enhance the credibility of the provided information.
- Introduce graphical representations to present sustainability data more intuitively and facilitate quick understanding.
- Expand the analysis by integrating more diverse data, such as geographic insights, health-related information, and a wider range of ingredient comparisons.

6.2.0.10 General Flow of the Agent and User Friendliness

All users were asked to evaluate the conversational flow and overall user-friendliness of the agent. This was one of the most critical aspects of this work. Several users, including **P05**, **P06**, **P10**, and **P11**, noted that the interaction process, specifically the need to enter and exit each function following a rigid conversational path, sometimes induced frustration. While some users, such as **P09**, accepted this structure as a natural consequence of the bot's function-based design, others found the interaction unnatural. Some suggested implementing all functions in a dedicated app while using the chatbot as contextual support for each function. Alternatively, a more fluid chatbot experience, akin to ChatGPT (which was widely recognized as the gold standard for this kind of technology), was preferred by many.

Interestingly, **P09** appreciated the restricted flexibility, as it prevented over-humanization of the agent, aligning with the intended goal of this project. Despite its limitations, all users managed to navigate the agent's functionalities and resolve occasional interaction stalls, often with the help of the "/start" command, which resets the agent to the main menu.

Other users, such as **P02**, **P03**, and **P07**, appreciated that the agent was implemented as a Telegram bot since they were already familiar with the platform. This eliminated the need to download a separate app. While they acknowledged that a more fluid conversational structure

would improve usability, they perceived the current implementation as a strong foundation.

Regarding potential additional functionalities, **P05** suggested introducing an expert function dedicated to discussing health-related aspects of food, which could also be integrated into recipe recommendations.

In conclusion, the current conversational flow should be refined to balance its role as a functional tool with improved responsiveness to contextual changes. A redesign of the underlying finite state machine (FSM) could be beneficial, potentially incorporating a ”pre-hub” mechanism to detect context changes dynamically. Alternatively, a major redesign could transform the agent into a more traditional UI-based application, with the chatbot serving as an assistive tool rather than the primary interface.

Suggestions for Improvement:

- Reduce the rigidity of the conversational path to enhance user interaction.
- Alternatively, consider implementing the agent as a more conventional UI-based application, with the chatbot serving as a support tool.
- Introduce a dedicated function for discussing health-related topics concerning food.

6.2.0.11 Untested Functionality

Certain functionalities, namely the reminder reception and the agent's taste adaptability, were not tested as it was impossible to evaluate them within the current experiment setting. Fully experiencing these features would have required users to engage with the agent over an extended period, which was not feasible in this study. Future experiments involving long-term user interactions should be conducted to assess these aspects. However, these functionalities are relatively less critical compared to those already tested, and their absence from this evaluation does not compromise the overall findings.

6.2.0.12 E-Mealio Agent Usability Recap

As a conclusion to this in-depth analysis of the user experience with the E-Mealio agent, the following table provides a summary of the pros, cons, and key improvements for each functionality.

Functionality	Pros	Cons	Key Improvements
User Data Acquisition	• Clearly understood by all users. • Users appreciated the ability to specify allergies and dietary restrictions.	• Privacy concerns about sharing personal information. • Users provided data in an unnatural format (lists). • System failed to track data correctly in some cases.	• Justify why certain data is requested. • Improve tracking mechanisms to ensure data is stored correctly. • Include fields for disliked ingredients and evolving dietary preferences.
Main Hub (Menu)	• All users understood the agent's capabilities. • Emojis and food-related jokes were appreciated.	• Some users found the menu too verbose. • "/start" command was not intuitively linked to returning to the menu.	• Simplify the menu text or introduce a button-based interface. • Clarify that "/start" returns users to the main menu.

Recipe Recommendation	• Users who specified simple constraints (e.g., vegetable-based recipes) received satisfactory recommendations. • Meal-type specification (breakfast, lunch, break, dinner) was well understood.	• Some ingredient-specific requests were not met. • The system failed to consistently account for allergies. • Sustainability data was too technical.	• Improve allergy tracking to ensure restricted ingredients are never suggested. • Make sustainability data more user-friendly, possibly with visual aids.
Recipe Sustainability Improver	• Users understood the difference between base and improved recipes. • Users appreciated discussions about modifying recipes.	• Function was difficult to invoke naturally. • Improved recipes were sometimes too different from the original. • Users' accepted modifications were not always saved.	• Improve ingredient recognition to allow more natural user input. • Ensure modifications made during interactions are correctly saved.

Sustainability Expert Chat	• Most users found it informative. • Provided well-known book and documentary recommendations. • Users appreciated that the agent did not overstep into psychological/emotional advice.	• Some explanations were too technical. • Some responses lacked external references for credibility. • Comparisons defaulted to ”meat vs. plant-based,” which was irrelevant to some users.	• Provide external sources and citations for sustainability information. • Personalize comparisons based on the user’s diet.
User Profile Recap and Update	• Users who tested it received correct data. • Users appreciated the ability to update their profiles.	• The agent strictly required a yes/no response before updates, making it rigid.	• Allow users to update information without needing to confirm with yes/no.
Food Diary (Meal Tracker)	• Users successfully logged meals.	• Agent’s response to logged meals was too neutral. • The system didn’t always recognize full recipes vs. single ingredients.	• Provide a brief initial analysis or acknowledgment of the logged meal. • Improve recognition of full recipes vs. single ingredients.

Weekly Food Diary Recap	• Users understood the correlation with the Meal Tracker. • The star-based Likert scale was well received.	• Some users felt the analysis lacked context (e.g., meat shouldn't be demonized outright). • Users didn't realize they could ask follow-up questions.	• Allow customization of the analysis period (e.g., full month, custom dates). • Highlight that users can discuss their results with the agent.
Tone and Informativeness	• The friendly tone was well received. • Users appreciated the balance between approachability and reliability.	• Data presentation was often too technical. • Comparisons were sometimes irrelevant to the user's diet.	• Introduce graphical elements to simplify data interpretation. • Expand sustainability comparisons beyond default "meat vs. plant-based" contrasts.
General Flow of the Agent	• The structured interaction helped users navigate functionalities. • Some users appreciated the restriction, preventing over-humanization of the agent.	• The rigid conversation structure frustrated some users. • Users wanted either a more flexible chatbot or a traditional UI-based app.	• Introduce a more flexible conversation flow to allow easier context changes. • Consider a UI-based implementation where the chatbot serves as a supporting tool.

Table 6.1: Pros, Cons, and Key Improvements for Each Functionality

6.3 Final Questionnaire Results

While the think-aloud sessions provided valuable insights into real user interactions with the system, a structured questionnaire was administered to collect more quantitative data about each user's experience. This questionnaire was designed to complement the qualitative observations by offering a more systematic evaluation of user perceptions.

Excluding personal information, each question was answered using a 5-point Likert scale, which enabled the collection of a quantitatively analyzable dataset. The questions were phrased as self-statements rather than direct inquiries to encourage more reflective responses. Since the questionnaire was primarily administered to Italian users, **all questions were presented in Italian**.

The questionnaire aimed to achieve three primary objectives: first, to gather additional personal data that may have been overlooked during the interviews; second, to assess the agent's usability, with a focus on both user-friendliness and its alignment with user expectations; and third, to evaluate the agent's potential impact on users' food sustainability awareness and future dietary choices. The additional personal data, such as chatbot usage frequency, were already reported in the user background section, and will not be repeated here.

Regarding the evaluative aspect, although some topics overlapped with those explored in the think-aloud sessions, collecting user opinions in a quantitative format allowed for a more structured comparison between the experiences of users who interacted with the two different versions of the agent. It is important to note that these data are based on a small sample size and should always be interpreted alongside the insights from the think-aloud sessions to provide proper context. A strictly quantitative evaluation would require a larger sample size, which was not feasible within the scope of this work.

6.3.1 Usability and User Friendliness

The evaluation of usability and user-friendliness is divided into two key aspects: **usability**, which assesses how technically easy the agent is to interact with, and **dialog quality** evaluation, which examines the naturalness and engagement of the conversation.

To assess usability, the following questions were included:

1. Penso che vorrei usare questo sistema frequentemente.
2. Ho trovato il sistema inutilmente complesso.
3. Ho trovato il sistema facile da usare.
4. Penso che avrei bisogno del supporto di una persona tecnica per poter usare questo sistema.
5. Ho trovato che le varie funzioni di questo sistema fossero ben integrate.
6. Ho pensato che ci fosse troppa incoerenza in questo sistema.
7. Immagino che la maggior parte delle persone imparerebbe a usare questo sistema molto rapidamente.
8. Ho trovato il sistema molto macchinoso da usare.
9. Mi sono sentito/a molto sicuro/a nell'usare il sistema.
10. Ho dovuto imparare molte cose prima di poter iniziare a usare questo sistema.
11. Ho acquisito familiarità con il chatbot molto rapidamente.

To evaluate the dialog quality, the following questions were included:

1. Nel complesso, sono soddisfatto/a della qualità del dialogo con il chatbot.
2. Il chatbot era reattivo e tempestivo durante la conversazione.

3. L'interazione con il chatbot era naturale.

4. Mi sono sentito/a supportato/a e incoraggiato/a durante l'interazione con il chatbot.

For each user, the statistics for both usability and dialog evaluation are reported, including the average and standard deviation of their responses, along with the LLM model that powered their session. Additionally, a model-wise average is provided to highlight any potential differences in user perception between the two models.

For questions posed in a negative way, specifically, questions 2, 4, 6, 8, and 10 in the usability group, the values reported by the users have been normalized using the formula $6 - \text{vote}$. This transformation ensures that all responses are consistently reported on a 1-5 scale, where higher values always correspond to positive aspects of usability.

Usability:

User ID	LLM Model	Score AVG	Score STD
P01	GPT4-o	3.4545	0.4979
P02	GPT4-o	4.0909	0.7925
P03	GPT4-o	4.5455	0.4979
P04	Sonnet 3.5	4.7273	0.4454
P05	Sonnet 3.5	4.8182	0.3857
P06	Sonnet 3.5	3	1.2792
P07	GPT4-o	3.6364	0.7714
P08	GPT4-o	4.0909	0.6680
P09	GPT4-o	4.2727	0.8624
P10	Sonnet 3.5	3.7273	0.6166
P11	Sonnet 3.5	3	0.8528
P12	Sonnet 3.5	3.7273	0.7497

Table 6.2: Usability Evaluation by User

LLM Model	Score AVG	Score STD
GPT-4-o	4.0151	0.3689
Sonnet 3.5	3.961	0.7431

Table 6.3: Comparison of Usability Evaluation Averaged by LLM Model

Based on this data, it's possible to see that users generally evaluated the usability of the agent positively, with a few minor exceptions, such as users **P06** and **P11**, who reported lower scores. These lower ratings are consistent with the feedback they provided during the think-aloud session. Regarding the comparison between the two models, the results were largely comparable, with no significant differences between them.

Dialog Quality Evaluation:

User ID	LLM Model	Score AVG	Score STD
P01	GPT4-o	3.25	0.8292
P02	GPT4-o	4	0.7071
P03	GPT4-o	4.5	0.5
P04	Sonnet 3.5	4.75	0.4330
P05	Sonnet 3.5	4.5	0.5
P06	Sonnet 3.5	2.25	0.8292
P07	GPT4-o	3	0
P08	GPT4-o	3.75	0.4330
P09	GPT4-o	4.5	0.5
P10	Sonnet 3.5	3.75	0.8292
P11	Sonnet 3.5	2.75	0.8292
P12	Sonnet 3.5	4	0.7071

Table 6.4: Dialog Quality Evaluation by User

LLM Model	Score AVG	Score STD
GPT4-o	3.8333	0.5713
Sonnet 3.5	3.8214	0.9133

Table 6.5: Comparison of Dialog Quality Evaluation Averaged by LLM Model

This data aligns with the usability results, demonstrating how the two aspects are related. Dialog evaluations were generally positive for the majority of users, with a few exceptions for those who encountered more issues during their interactions, leading to a lower appreciation of the dialog experience. Once again, the two models did not show significant differences. However, for Sonnet 3.5, a larger standard deviation was reported, indicating slightly higher variability in user appreciation.

To summarize the results regarding usability and user friendliness, both aspects were generally perceived as above the sufficiency threshold but still leave significant room for improvement. Enhancing the flow of interactions could help deliver a more compelling and engaging experience, which would ultimately enhance the overall use of the agent.

6.3.2 Impact on Food Sustainability Awareness and Dietary Choices

The evaluation of the impact on sustainability awareness and dietary choices is divided into five key aspects: **user profiling**, which assesses how well the agent adapts its responses and suggestions to user needs; **suggestion efficacy**, which evaluates whether the proposed suggestions effectively influence users' dietary choices; **persuasion**, which measures the agent's ability to convince users to try new recipes; **transparency**, which examines the clarity and reliability of the agent's responses and recommendations; and **explanation quality**, which assesses whether the agent provides clear and concise explanations tailored to the user.

To assess **user profiling**, the following questions were included:

1. Era facile comunicare al chatbot le mie preferenze alimentari (es. restrizioni dietetiche, intolleranze, allergie).
2. Le alternative suggerite dal chatbot tenevano conto delle mie preferenze alimentari (es. restrizioni dietetiche, intolleranze, allergie).
3. Il chatbot ricordava e teneva conto delle mie preferenze nel corso dell'interazione.
4. Il chatbot era in grado di offrire suggerimenti pertinenti considerando il contesto in cui mi trovavo (es. momento, ingredienti, utensili, luogo).
5. I suggerimenti forniti dal chatbot si adattavano alle mie necessità.
6. Il chatbot mi aiutava a trovare soluzioni alternative quando non potevo seguire i suoi suggerimenti iniziali.
7. Il chatbot era in grado di anticipare correttamente i miei bisogni alimentari.

To assess **suggestion efficacy**, the following questions were included:

1. Sono soddisfatto/a delle ricette alternative proposte dal chatbot.
2. Il chatbot era in grado di proporre ricette alternative più sane.
3. Il chatbot era in grado di proporre ricette alternative più sostenibili.

To assess **persuasion**, the following questions were included:

1. Il chatbot mi ha convinto a provare nuove ricette.
2. Il chatbot potrebbe influire sul modo in cui pianificherò i pasti o sceglierò gli alimenti in futuro.

To assess **transparency**, the following questions were included:

1. Era chiaro quali criteri il chatbot utilizzava per suggerire ricette alternative.

2. Sentivo di potermi fidare delle proposte del chatbot.
3. Il chatbot spiegava su quali criteri si basavano i suoi suggerimenti.
4. Le informazioni fornite dal chatbot erano trasparenti e facili da verificare.

To assess **explanation quality**, the following questions were included:

1. Le spiegazioni fornite dal chatbot erano dettagliate senza essere troppo complesse.
2. Il chatbot comprendeva facilmente le mie richieste.
3. Le risposte del chatbot alle mie richieste erano accurate e pertinenti.

For each user, the statistics for these evaluations are reported, including the average and standard deviation of their responses, along with the LLM model that powered their session. Additionally, a model-wise average is provided to highlight any potential differences in user perception between the two models.

User Profiling Evaluation:

User ID	LLM Model	Score AVG	Score STD
P01	GPT4-o	3.2857	0.6999
P02	GPT4-o	2.2857	0.6999
P03	GPT4-o	3.8571	0.3499
P04	Sonnet 3.5	4.2857	0.8806
P05	Sonnet 3.5	3.8571	0.9897
P06	Sonnet 3.5	3.1429	1.2454
P07	GPT4-o	1.8571	0.6389
P08	GPT4-o	3.2857	0.4518
P09	GPT4-o	4.1429	0.3499
P10	Sonnet 3.5	4.5714	0.7284
P11	Sonnet 3.5	1.4286	1.0498
P12	Sonnet 3.5	3.1429	0.8330

Table 6.6: User Profiling Evaluation for Different Users and LLM Models

LLM Model	Score AVG	Score STD
GPT4-o	3.1190	0.8099
Sonnet 3.5	3.5306	1.0035

Table 6.7: Comparison of User Profiling Evaluation Averaged by LLM Model

Based on these results, user profiling was generally perceived positively, except for users **P02**, **P07** and **P11**, whose average scores fell below the sufficiency threshold of 3. This is likely due to the system's limited adaptability to specific dietary needs, as these users pointed out during the think-aloud session. Other users reported a slightly better experience, though user profiling was not perceived as a completely solid aspect of the agent. Regarding the difference between the two LLM models, Claude Sonnet 3.5 had a slightly higher average score, but not significantly enough to be considered superior to GPT-4o.

Suggestion Efficacy:

User ID	LLM Model	Score AVG	Score STD
P01	GPT4-o	4.3333	0.4714
P02	GPT4-o	3	1.4142
P03	GPT4-o	5	0
P04	Sonnet 3.5	5	0
P05	Sonnet 3.5	4	0
P06	Sonnet 3.5	3.6667	0.4714
P07	GPT4-o	3	0
P08	GPT4-o	4.3333	0.4714
P09	GPT4-o	5	0
P10	Sonnet 3.5	5	0
P11	Sonnet 3.5	4	0
P12	Sonnet 3.5	3.3333	0.9428

Table 6.8: Suggestion Efficacy Scores for Different Users and LLM Models

LLM Model	Score AVG	Score STD
GPT4-o	4.1111	0.8315
Sonnet 3.5	4.2857	0.6529

Table 6.9: Comparison of Suggestion Efficacy Scores Averaged by LLM Model

Regarding the effectiveness of the suggestions, these data show that this aspect was strongly perceived as positive, highlighting that almost all users found the proposed recipe suggestions valid and informative. Even the more critical users, such as **P06** and **P07**, reported an average score that was sufficient or higher. In this scenario as well, the two LLM models did not show significant differences, indicating a comparable level of perceived quality.

Persuasion Evaluation:

User ID	LLM Model	Score AVG	Score STD
P01	GPT4-o	3.5	0.5
P02	GPT4-o	3	1
P03	GPT4-o	4.5	0.5
P04	Sonnet 3.5	4.5	0.5
P05	Sonnet 3.5	4	0
P06	Sonnet 3.5	3.5	0.5
P07	GPT4-o	3	0
P08	GPT4-o	4	0
P09	GPT4-o	5	0
P10	Sonnet 3.5	4.5	0.5
P11	Sonnet 3.5	3	0
P12	Sonnet 3.5	3.5	0.5

Table 6.10: Persuasion Evaluation for Different Users and LLM Models

LLM Model	Score AVG	Score STD
GPT4-o	3.8333	0.7454
Sonnet 3.5	3.9286	0.5624

Table 6.11: Comparison of Persuasion Evaluation Averaged by LLM Model

Regarding persuasion, the agent was generally perceived as convincing and capable of potentially influencing users' dietary choices. Even in this case, more demanding users still reported scores at or above the sufficiency threshold. As for the difference between the two LLM models, the evaluations on this aspect were nearly identical, confirming both as solid options for delivering persuasive messages effectively.

Transparency Evaluation:

User ID	LLM Model	Score AVG	Score STD
P01	GPT4-o	4	0
P02	GPT4-o	3.25	0.8292
P03	GPT4-o	4.5	0.5
P04	Sonnet 3.5	4.75	0.433
P05	Sonnet 3.5	3.75	0.433
P06	Sonnet 3.5	2.75	0.8292
P07	GPT4-o	3.75	0.433
P08	GPT4-o	4.75	0.433
P09	GPT4-o	4.5	0.5
P10	Sonnet 3.5	4.25	0.433
P11	Sonnet 3.5	2.25	0.433
P12	Sonnet 3.5	4.25	0.433

Table 6.12: Transparency Evaluation for Different Users and LLM Models

LLM Model	Score AVG	Score STD
GPT4-o	4.125	0.5153
Sonnet 3.5	3.8214	0.9035

Table 6.13: Comparison of Transparency Evaluation Averaged by LLM Model

The transparency aspect emerged as another generally appreciated feature of the agent, with the exception of users like **P06** and **P11**, who had already expressed dissatisfaction with other aspects. Nine out of twelve users evaluated this aspect well above the sufficiency threshold, suggesting that the quality of the proposed data is solid enough to be perceived as reliable. However, there was a general suggestion, noted in the think-aloud sessions, that more effort should be made to cite sources in support of the agent's statements.

Regarding the two LLM models, both showed similar average results on this aspect, indicating that they provide comparable outcomes in terms of perceived transparency.

Explanation Quality:

User ID	LLM Model	Score AVG	Score STD
P01	GPT4-o	3.6667	0.4714
P02	GPT4-o	2.6667	0.4714
P03	GPT4-o	4.3333	0.4714
P04	Sonnet 3.5	5	0
P05	Sonnet 3.5	4	0
P06	Sonnet 3.5	3	0
P07	GPT4-o	2.3333	0.4714
P08	GPT4-o	3.3333	0.4714
P09	GPT4-o	3.6667	0.9428
P10	Sonnet 3.5	4.3333	0.4714
P11	Sonnet 3.5	3.6667	0.9428
P12	Sonnet 3.5	4	0

Table 6.14: Explanation Quality Scores for Different Users and LLM Models

LLM Model	Score AVG	Score STD
GPT4-o	3.3333	0.6667
Claude Sonnet 3.5	4.1429	0.6633

Table 6.15: Comparison of Explanation Quality Scores Averaged by LLM Model

The final aspect evaluated was the quality of the explanations. This aspect was generally perceived positively, with a couple of users, **P02** and **P07**, rating it just below the sufficiency threshold. These data suggest that the agent is capable of delivering compelling explanations, which aligns with the positive feedback on the transparency aspect.

Regarding the comparison between the two LLM models, Claude Sonnet 3.5 appears to be slightly better, indicating that the content of its proposed messages may be structured in a way that makes its responses seem more reliable. However, the differences are not substantial

enough to draw solid conclusions.

To summarize the results regarding the impact on food sustainability awareness and dietary choices, the agent was generally evaluated positively, with the strongest appreciation for the informative aspect. It was seen as strong enough to potentially influence users' dietary choices. The data also indicates that more effort should be put into user profiling. Additionally, as mentioned in the think-aloud sessions, providing resources to support the agent's claims would enhance its perceived reliability and trustworthiness among users.

6.3.2.1 Summarizing Agent LLM Models Comparison

Averaging the results across both LLM models, the data indicate that user experiences were largely similar between them. The reported issues were consistent across both versions, primarily stemming from the agent's internal design rather than the specific LLM used to generate responses. Likewise, the positive aspects of the interaction were observed regardless of the underlying model. While minor differences emerged in a few evaluated aspects, they were not substantial enough to be considered significant.

This conclusion is reinforced by the following summary table, which presents the average scores for each evaluated aspect across both models.

Area	Aspect	LLM	Avg	Std
Usability and User Friendliness	Usability	GPT-4o	4.0151	0.3689
		Sonnet 3.5	3.961	0.7431
	Dialog Quality	GPT-4o	3.8333	0.5713
		Sonnet 3.5	3.8214	0.9133
Impact on Food Sustainability Awareness and Dietary Choices	User Profiling	GPT-4o	3.119	0.8099
		Sonnet 3.5	**3.5306**	1.0035
	Suggestion Efficacy	GPT-4o	4.1111	0.8315
		Sonnet 3.5	4.2857	0.6529
	Persuasion	GPT-4o	3.8333	0.7454
		Sonnet 3.5	3.9286	0.5624
	Transparency	GPT-4o	**4.125**	0.5153
		Sonnet 3.5	3.8214	0.9035
	Explanation Quality	GPT-4o	3.3333	0.6667
		Sonnet 3.5	**4.1429**	0.6633

Table 6.16: Summary of evaluations across the main assessed areas. The **bolded** values indicate cases where one model showed a slight better average over the other.

Aside from a few highlighted differences in certain cases, the overall gap is not significant enough to clearly favor one model over the other. This becomes even more apparent when the reported values are averaged by evaluation area.

Area	LLM	Avg	Std
Usability and User Friendliness	GPT-4o	3.9242	0.0909
	Sonnet 3.5	3.8912	0.0698
Impact on Food Sustainability Awareness and Dietary Choices	GPT-4o	3.7043	0.4097
	Sonnet 3.5	3.9418	0.2617

Table 6.17: Average Scores for each main area evaluated.

Finally, a global average, calculated from the values of each evaluation area for both models, is provided, showing that both achieved nearly identical results.

LLM	Avg	Std
GPT-4o	3.81427	0.10993
Sonnet 3.5	3.91652	0.02532

Table 6.18: Global average scores.

These findings suggest that, when using a sufficiently capable language model, user satisfaction is more strongly influenced by the quality of the provided data and the fluidity of the interaction, both of which are shaped by the agent's overall design rather than the specific LLM in use.

Chapter 7

Conclusions and Future Developments

This work has presented all the fundamental steps in the development and evaluation of the *E-Mealio* conversational agent, from dataset enhancement to the technical design choices underlying the system, culminating in its evaluation phase. Through interactions with real users from diverse backgrounds, the system demonstrated both its strengths and areas for improvement.

7.1 Areas of Improvement

While this serves as a solid starting point, both in terms of dataset quality and agent design, the evaluation phase has highlighted several aspects that could be refined. Efforts should focus on making the conversational flow more natural and engaging, alongside improving data quality. For instance, providing explicit sources when reporting sustainability data, enhancing allergy management through a more detailed ingredient mapping for each recipe, and increasing the quantity, quality, and localization of recipes. The latter could be addressed by incorporating recipes from external sources and adapting them to the current dataset structure, enabling the agent to tailor suggestions based on a user's nationality and cultural preferences.

Additionally, the sustainability scoring system could be enhanced by scaling the carbon footprint of each ingredient based on actual quantities used in recipes. This was previously unfeasible due to many recipes listing ingredients as generic units (e.g., "one carrot") rather than precise weights. Addressing this limitation would require an in-depth study of the average weight of such ingredients.

Despite its limitations, this work provides a solid approach for constructing a conversational agent that leverages a multi-prompt, FSM-like structure. This framework could serve as inspiration for other domains requiring conversational agents with sequential functionality and multiple interaction layers.

Moving forward, several enhancements could be explored:

- **Gamification Features** – Introducing challenges, sustainability streaks, quizzes, and badges to improve user engagement.
- **Social Integration** – Developing a social component where users can maintain a friend list, share consumed recipes, and compare challenge results. This would require advancements in both usability and privacy, as well as a dedicated user interface beyond the current Telegram bot.
- **Recommendation System Integration** – Exploring external recommendation services to refine recipe suggestions.
- **Embedding Model Optimization** – Conducting a thorough study on embedding models used to compute vector representations of ingredients and recipes, exploring potential alternatives to the currently used one.
- **Environmental Impact Assessment** – Investigating the CO_2 emissions of the underlying LLMs, evaluating different models to minimize environmental impact [74] and align the agent's functionality with its sustainability-driven mission.

By addressing these aspects, *E-Mealio* could evolve into a more intuitive, engaging, and environmentally responsible assistant, further supporting users in making sustainable dietary choices.

7.2 Closing Thoughts

We are currently witnessing a rapidly changing world, where technology is evolving at an unprecedented pace, developing capabilities that were once confined to the realm of science fiction. However, as society navigates the challenges of this rapid progress, technology can either serve as a powerful tool for good or become a source of disruption, depending on how it is directed.

In this work, a modest example of how these new technologies can be leveraged for the common good has been presented, with the hope that it can serve as a foundation for further development, both in the specific domain of sustainability, and as a broader example of how artificial intelligence can assist in addressing complex problems without overriding human experience and capabilities.

We have the power to tackle some of the greatest challenges humanity has ever faced, from climate change to resource allocation aimed at reducing social inequality. Like any tool, the remarkable potential of today's artificial intelligence can be directed either toward benefiting or harming human well-being. I hope that the current generation of students, researcher and professionals recognizes this potential and strives to build a better and brighter future for everyone.

We will see our children growing [75].

Bibliography

[1] World Meteorological Organization. (2025) Wmo confirms 2024 as warmest year on record at about 1.55°c above pre-industrial level. [Online]. Available: https://wmo.int/news/media-centre/wmo-confirms-2024-warmest-year-record-about-155degc-above-pre-industrial-level

[2] M. Bearak. (2025) Trump orders a u.s. exit from the world's main climate pact. [Online]. Available: https://www.nytimes.com/2025/01/20/climate/trump-paris-agreement-climate.html

[3] H. Steinfeld, P. Gerber, T. Wassenaar, F. Nations, V. Castel, E. Livestock, and C. de Haan, *Livestock's Long Shadow: Environmental Issues and Options*. Food and Agriculture Organization of the United Nations, 2006. [Online]. Available: https://books.google.it/books?id=1B9LQQkm_qMC

[4] R. Goodland and J. Anhang, "Livestock and climate change: What if the key actors in climate change were pigs, chickens and cows?" November 2009. [Online]. Available: https://awellfedworld.org/wp-content/uploads/Livestock-Climate-Change-Anhang-Goodland.pdf

[5] Food and A. O. of the United Nations (FAO), "Greenhouse gas emissions from agrifood systems: Global, regional and country trends (2000-2022)," 2024. [Online]. Available: https://www.fao.org/statistics/highlights-archive/highlights-detail/greenhouse-gas-emissions-from-agrifood-systems.-global--regional-and-country-trends--2000-2022/en

[6] J. S. Foer, *We Are the Weather: Saving the Planet Begins at Breakfast*. New York: Farrar, Straus and Giroux, 2019.

[7] P. Lally, C. H. M. Van Jaarsveld, H. W. W. Potts, and J. Wardle, "How are habits formed: Modelling habit formation in the real world," *European Journal of Social Psychology*, vol. 40, no. 6, pp. 998–1009, 2009. [Online]. Available: https://onlinelibrary.wiley.com/doi/abs/10.1002/ejsp.674

[8] R. H. Thaler and C. R. Sunstein, *Nudge: Improving Decisions About Health, Wealth, and Happiness*. New York: Penguin Books, 2008.

[9] D. Kahneman, *Thinking, Fast and Slow*. Farrar, Straus and Giroux, 2011.

[10] S.-C. Wee, W.-W. Choong, and S.-T. Low, "Can 'nudging' play a role to promote pro-environmental behaviour?" *Environmental Challenges*, vol. 5, p. 100364, 2021. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S2667010021003383

[11] M. J. Nelson, "Soviet and american precursors to the gamification of work," in *Proceedings of the 16th International Academic MindTrek Conference*, 2012, pp. 23–26. [Online]. Available: https://dl.acm.org/doi/10.1145/2393132.2393138

[12] Wikipedia contributors, "Analysis Paralysis — Wikipedia, The Free Encyclopedia." [Online]. Available: https://en.wikipedia.org/wiki/Analysis_paralysis

[13] ——, "Decision Fatigue — Wikipedia, The Free Encyclopedia." [Online]. Available: https://en.wikipedia.org/wiki/Decision_fatigue

[14] H. e. a. Zhou, "A comprehensive survey on multimodal recommender systems: Taxonomy, evaluation, and future directions," 2023. [Online]. Available: https://arxiv.org/abs/2302.04473

[15] Wikipedia contributors, "Yummly," 2024. [Online]. Available: https://en.wikipedia.org/wiki/Yummly

[16] M. Wolf, "Whirlpool lays off entire team for cooking and recipe app yummly," 2024. [Online]. Available: https://thespoon.tech/whirlpool-lays-off-entire-team-for-cooking-and-recipe-app-yummly/

[17] Flavorish, "Flavorish: AI-Powered Recipe Management and Meal Planning," 2024. [Online]. Available: https://www.flavorish.ai/

[18] Z. Yang, E. Khatibi, N. Nagesh, M. Abbasian, I. Azimi, R. Jain, and A. Rahmani, "Chatdiet: Empowering personalized nutrition-oriented food recommender chatbots through an llm-augmented framework," *Smart Health*, vol. 32, p. 100465, 06 2024.

[19] D. Sambola, M. Rodríguez-García, and F. Garcia-Sanchez, "Ontology-based nutritional recommender system," *Applied Sciences*, vol. 12, p. 143, 12 2021.

[20] N. Adila and Z. Baizal, "Ontology-based food menu recommender system for patients with coronary heart disease," *sinkron*, vol. 8, pp. 2363–2371, 10 2023.

[21] S. Haussmann, O. Seneviratne, Y. Chen, Y. Ne'eman, J. Codella, C.-H. Chen, D. Mcguinness, and M. Zaki, *FoodKG: A Semantics-Driven Knowledge Graph for Food Recommendation*, 10 2019, pp. 146–162.

[22] S. Yadav, M. Powers, E. Vakaj, S. Mishra Tiwari, and F. Ortiz-Rodríguez, "Semantic carbon footprint of food supply chain management," 11 2023, pp. 202–216.

[23] A. Petruzzelli, C. Musto, M. C. Di Carlo, G. Tempesta, and G. Semeraro, "Recommending healthy and sustainable meals exploiting food retrieval and large language models," in *Proceedings of the 18th ACM Conference on Recommender Systems*, ser. RecSys '24. New York, NY, USA: Association for Computing Machinery, 2024, p. 1057–1061. [Online]. Available: https://doi.org/10.1145/3640457.3688193

[24] J. N. Bondevik, K. E. Bennin, Babur, and C. Ersch, "A systematic review on food recommender systems," *Expert Systems with Applications*, vol. 238, p. 122166, 2024. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0957417423026684

[25] A. Felfernig, M. Wundara, T. N. T. Tran, S. Polat-Erdeniz, S. Lubos, M. El Mansi, D. Garber, and V.-M. Le, "Recommender systems for sustainability: Overview and research issues," *Frontiers in*

Big Data, vol. 7, p. 1284511, 2024. [Online]. Available: https://www.frontiersin.org/articles/10.3389/fdata.2023.1284511/full

[26] J. Mendoza-Bernal, A. González-Vidal, and A. F. Skarmeta, "A convolutional neural network approach for image-based anomaly detection in smart agriculture," *Expert Systems with Applications*, vol. 247, p. 123210, 2024. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0957417424000757

[27] E. A. Abioye, O. Hensel, T. J. Esau, O. Elijah, M. S. Z. Abidin, A. S. Ayobami, O. Yerima, and A. Nasirahmadi, "Precision irrigation management using machine learning and digital farming solutions," *AgriEngineering*, vol. 4, no. 1, pp. 70–103, 2022. [Online]. Available: https://www.mdpi.com/2624-7402/4/1/6

[28] T.-W. Yoo and I.-S. Oh, "Time series forecasting of agricultural products' sales volumes based on seasonal long short-term memory," *Applied Sciences*, vol. 10, no. 22, 2020. [Online]. Available: https://www.mdpi.com/2076-3417/10/22/8169

[29] A. Taneja, G. Nair, M. Joshi, S. Sharma, S. Sharma, A. R. Jambrak, E. Roselló-Soto, F. J. Barba, J. M. Castagnini, N. Leksawasdi, and Y. Phimolsiripol, "Artificial intelligence: Implications for the agri-food sector," *Agronomy*, vol. 13, no. 5, 2023. [Online]. Available: https://www.mdpi.com/2073-4395/13/5/1397

[30] D. K. Pandey and R. Mishra, "Towards sustainable agriculture: Harnessing ai for global food security," *Artificial Intelligence in Agriculture*, vol. 12, pp. 72–84, 2024. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S2589721724000151

[31] Y. Long, L. Huang, R. Fujie, P. He, Z. Chen, X. Xu, and Y. Yoshida, "Carbon footprint and embodied nutrition evaluation of 388 recipes," *Scientific Data*, vol. 10, no. 1, p. 794, 2023. [Online]. Available: https://doi.org/10.1038/s41597-023-02702-1

[32] G. Simsek-Senel, H. Rijgersberg, B. Öztürk, J. Weits, and A. Fensel, "I-know-foo: Interlinking and creating knowledge graphs for near-zero co2 emission diets and sustainable food production," in

AI, Data, and Digitalization, ser. Communications in Computer and Information Science (CCIS), R. Akerkar, Ed. Germany: Springer, Mar. 2024, pp. 106–119, 1st International Symposium on AI, Data and Digitalization, SAIDD 2023 ; Conference date: 09-05-2023 Through 10-05-2023.

[33] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, L. Kaiser, and I. Polosukhin, "Attention is all you need," 2023. [Online]. Available: https://arxiv.org/abs/1706.03762

[34] T. Mikolov, K. Chen, G. Corrado, and J. Dean, "Efficient estimation of word representations in vector space," 2013. [Online]. Available: https://arxiv.org/abs/1301.3781

[35] M. E. Peters, M. Neumann, M. Iyyer, M. Gardner, C. Clark, K. Lee, and L. Zettlemoyer, "Deep contextualized word representations," *CoRR*, vol. abs/1802.05365, 2018. [Online]. Available: http://arxiv.org/abs/1802.05365

[36] J. Devlin, M. Chang, K. Lee, and K. Toutanova, "BERT: pre-training of deep bidirectional transformers for language understanding," *CoRR*, vol. abs/1810.04805, 2018. [Online]. Available: http://arxiv.org/abs/1810.04805

[37] J. Starmer, "Transformer neural networks, chatgpt's foundation, clearly explained," YouTube video, 2023. [Online]. Available: https://www.youtube.com/watch?v=zxQyTK8quyY

[38] A. Radford, K. Narasimhan, T. Salimans, and I. Sutskever, "Improving language understanding by generative pre-training," 2018.

[39] H. Touvron, T. Lavril, G. Izacard, X. Martinet, M.-A. Lachaux, T. Lacroix, B. Rozière, N. Goyal, E. Hambro, F. Azhar, A. Rodriguez, A. Joulin, E. Grave, and G. Lample, "Llama: Open and efficient foundation language models," 2023. [Online]. Available: https://arxiv.org/abs/2302.13971

[40] T. B. Brown, B. Mann, N. Ryder, M. Subbiah, J. Kaplan, P. Dhariwal, A. Neelakantan, P. Shyam, G. Sastry, A. Askell, S. Agarwal, A. Herbert-Voss, G. Krueger, T. Henighan, R. Child, A. Ramesh, D. M. Ziegler, J. Wu, C. Winter, C. Hesse, M. Chen, E. Sigler, M. Litwin, S. Gray, B. Chess,

J. Clark, C. Berner, S. McCandlish, A. Radford, I. Sutskever, and D. Amodei, "Language models are few-shot learners," 2020. [Online]. Available: https://arxiv.org/abs/2005.14165

[41] J. M. J. et al., "Highly accurate protein structure prediction with alphafold," *Nature*, vol. 596, pp. 583 – 589, 2021. [Online]. Available: https://api.semanticscholar.org/CorpusID:235959867

[42] P. H. et al., "Embed-search-align: Dna sequence alignment using transformer models," 2024. [Online]. Available: https://arxiv.org/abs/2309.11087

[43] E. Saravia, "Prompt engineering guide," Available at: https://www.promptingguide.ai/.

[44] S. Vatsal and H. Dubey, "A survey of prompt engineering methods in large language models for different nlp tasks," 2024. [Online]. Available: https://arxiv.org/abs/2407.12994

[45] J. Wei, M. Bosma, V. Y. Zhao, K. Guu, A. W. Yu, B. Lester, N. Du, A. M. Dai, and Q. V. Le, "Finetuned language models are zero-shot learners," 2022. [Online]. Available: https://arxiv.org/abs/2109.01652

[46] S. Min, X. Lyu, A. Holtzman, M. Artetxe, M. Lewis, H. Hajishirzi, and L. Zettlemoyer, "Rethinking the role of demonstrations: What makes in-context learning work?" 2022. [Online]. Available: https://arxiv.org/abs/2202.12837

[47] J. Wei, X. Wang, D. Schuurmans, M. Bosma, B. Ichter, F. Xia, E. Chi, Q. Le, and D. Zhou, "Chain-of-thought prompting elicits reasoning in large language models," 2023. [Online]. Available: https://arxiv.org/abs/2201.11903

[48] T. Kojima, S. S. Gu, M. Reid, Y. Matsuo, and Y. Iwasawa, "Large language models are zero-shot reasoners," 2023. [Online]. Available: https://arxiv.org/abs/2205.11916

[49] OpenAI, "OpenAI O1 System Card," 2024. [Online]. Available: https://openai.com/index/openai-o1-system-card/

[50] DeepSeek-AI and D. G. et al., "Deepseek-r1: Incentivizing reasoning capability in llms via reinforcement learning," 2025. [Online]. Available: https://arxiv.org/abs/2501.12948

[51] S. R. et al., "Retrieval-augmented generation: Streamlining the creation of intelligent natural language processing models," 2020. [Online]. Available: https://tinyurl.com/55ydzvc6

[52] OpenAI, "Introducing chatgpt search," *OpenAI*, 2024. [Online]. Available: https://openai.com/index/introducing-chatgpt-search/

[53] R. Claypool, "Chatbots are not people: The designed-in dangers of human-like a.i. systems," Sep. 2023. [Online]. Available: https://www.citizen.org/article/chatbots-are-not-people-dangerous-human-like-anthropomorphic-ai-report/

[54] D. C. Dennett, "The problem with counterfeit people," *The Atlantic*, May 2023, accessed: 2025-03-18. [Online]. Available: https://www.theatlantic.com/technology/archive/2023/05/problem-counterfeit-people/674075/

[55] G. Tempesta and M. Di Carlo, "Sustainameal: Ai-powered recipe recommendations for sustainable eating," 2024.

[56] F. Bölz, D. Nurbakova, S. Calabretto, A. Gerl, L. Brunie, and H. Kosch, "Hummus: A linked, healthiness-aware, user-centered and argument-enabling recipe data set for recommendation," in *Proceedings of the 17th ACM Conference on Recommender Systems*, ser. RecSys '23. New York, NY, USA: Association for Computing Machinery, 2023, p. 1–11. [Online]. Available: https://doi.org/10.1145/3604915.3609491

[57] T. Petersson, L. Secondi, A. Magnani, M. Antonelli, K. Dembska, R. Valentini, A. Varotto, and S. Castaldi, "SU-EATABLE LIFE: a comprehensive database of carbon and water footprints of food commodities," 4 2021. [Online]. Available: https://figshare.com/articles/dataset/SU-EATABLE_LIFE_a_comprehensive_database_of_carbon_and_water_footprints_of_food_commodities/13271111

[58] A. Gigantelli and A. R. Iacovazzi, "Foodprintdb: an extensive database for recipes sustainability estimation," 2023.

[59] OpenAI, "Gpt-4o mini." [Online]. Available: https://openai.com/index/gpt-4o-mini-advancing-cost-efficient-intelligence/

[60] "all-mpnet-base-v2." [Online]. Available: https://huggingface.co/sentence-transformers/all-mpnet-base-v2

[61] OpenAI, "Gpt-4o." [Online]. Available: https://openai.com/index/hello-gpt-4o/

[62] "Open food facts." [Online]. Available: https://it.openfoodfacts.org/scopri

[63] "Agribalyse." [Online]. Available: https://agribalyse.ademe.fr/

[64] "Carboncloud." [Online]. Available: https://apps.carboncloud.com/

[65] M. The and D. Juhl, "Livelca." [Online]. Available: https://livelca.com/

[66] E. Parliament and C. of the European Union, "Regulation (eu) no 1169/2011 of the european parliament and of the council of 25 october 2011 on the provision of food information to consumers," 2011. [Online]. Available: https://eur-lex.europa.eu/legal-content/EN/ALL/?uri=CELEX%3A32011R1169

[67] LangChain, "Langchain - applications that can reason. powered by langchain." 2024, accessed: 2024-06-10. [Online]. Available: https://www.langchain.com/

[68] Anthropic, "Introducing claude 3.5 sonnet," 2024. [Online]. Available: https://www.anthropic.com/news/claude-3-5-sonnet

[69] M. N. Marshall, "Sampling for qualitative research." *Family practice*, vol. 13 6, pp. 522–5, 1996. [Online]. Available: https://api.semanticscholar.org/CorpusID:3039466

[70] W. contributors, "Think-aloud protocol," 2025. [Online]. Available: https://en.wikipedia.org/wiki/Think_aloud_protocol

[71] M. Pollan, *The Omnivore's Dilemma: A Natural History of Four Meals*. New York: Penguin Press, 2006.

[72] K. Andersen and K. Kuhn, “Cowspiracy: The sustainability secret,” Documentary film, 2014, produced by A.U.M. Films and First Spark Media.

[73] J. Tickell and R. Tickell, “Kiss the ground,” Documentary film, 2020. [Online]. Available: https://kissthegroundmovie.com/

[74] G. Smith, M. Bateman, R. Gillet, and E. Thanisch, “Environmental impact of large language models,” *Cutter Consortium*, 2023. [Online]. Available: https://www.cutter.com/article/environmental-impact-large-language-models

[75] Gojira, “Global warming.” [Online]. Available: https://www.youtube.com/watch?v=VB5QY8aW4PI

Dedication and Acknowledgments

This work is dedicated to the memory of my father, Filippo, whose inventiveness and intelligence will forever remain a personal myth to me. He continues to inspire me every day in what I do, and in the pursuit of my ideas and dreams.

A special thank you goes to my supervisor, Professor Cataldo Musto, whose professionalism, combined with his incredible ability to listen, has made these past months of work both stimulating and enjoyable. I would also like to express my gratitude to my co-supervisor, Dr. Arianna Boldi, for her support during the evaluation phase and for the many valuable suggestions that helped shape the final phase of this work, making it truly worthy of attention.

A heartfelt thank you goes to my partner, Arianna, who, a few years ago, encouraged me to reconsider the idea of dedicating myself fully to studying again, with the necessary focus and effort. Without her love and support, I would probably not be writing these words right now.

Of course, I must also thank my mother, Nunzia, and my brother, Raffaello, for their unwavering support and immense patience in enduring my most challenging moments. I know I can have a bad temper, and I sincerely apologize for that. I hope to find some inner peace moving forward.

Finally, I am deeply grateful to all my past and present work colleagues for their constant support throughout this journey. Your help has been invaluable and truly appreciated.

I would like to conclude by thanking all those I have met and reconnected with during this second university journey, especially Emanuele, Andrea, Roberto, Alberto, Antonio and Gianfranco, with whom I shared the challenges of several exams, experiencing both the difficulties and joys of a path that was far from easy but incredibly fascinating. I truly hope to make the most of it in the future.

Once again, thank you all.