

Democratizing Financial Advice with AI: LLM Agents and RAG for Personalized Wealth Strategies

Master Thesis



Democratizing Financial Advice with AI: LLM Agents and RAG for Personalized Wealth Strategies

Master Thesis
September, 2025

By
Ivan Antonino Arena

Copyright: Reproduction of this publication in whole or in part must include the customary bibliographic citation, including author attribution, report title, etc.

Cover photo: Vibeke Hempler, 2012

Published by: DTU, Department of Applied Mathematics and Computer Science,
Brovej, Building 118, 2800 Kgs. Lyngby Denmark
www.byg.dtu.dk

Approval

This thesis has been prepared over five months at Dreamplan in partial fulfillment for the degree Master of Science in Computer Science and Engineering Engineering at Technical University of Denmark, DTU.

It is assumed that the reader has a basic knowledge in the areas of software engineering.

Ivan Antonino Arena - s233352

.....
Signature

.....
Date

Abstract

Access to high-quality financial advice remains limited for many individuals due to high costs and exclusivity, despite its critical role in promoting long-term financial well-being. This thesis investigates the design and implementation of an AI-powered financial advisory system that aligns with Dreamplan’s mission of democratizing financial planning. By integrating Large Language Models (LLMs) with Retrieval-Augmented Generation (RAG) and deterministic financial optimization models, the proposed system aims to provide explainable, scalable and personalized wealth strategies via natural language dialogue.

Unlike end-to-end generative advisors, the architecture adopts a hybrid approach: LLMs serve as interpreters and orchestrators of user intent, while a robust financial engine ensures accuracy and rigor in recommendations. The system is developed using real-world anonymized data provided by Dreamplan and leverages cloud-based APIs for LLM inference and vector-based retrieval.

Through both qualitative and quantitative evaluation – including response accuracy, routing performance and explainability – the thesis demonstrates the feasibility of deploying conversational financial agents in real-world settings. The results suggest that combining neural and symbolic systems offers a promising path forward for accessible, trustworthy and intelligent financial advisory tools.

Acknowledgements

Ivan Antonino Arena, MSc Computer Science and Engineering, DTU
Author.

Finn Årup Nielsen, Associate Professor, DTU Compute
Internal supervisor at DTU.

Philip Rasmussen, Lead Data Scientist, Dreamplan
External supervisor at Dreamplan.

Contents

Preface	ii
Abstract	iii
Acknowledgements	iv
1 Introduction	1
1.1 Background	1
1.2 Prior Work	2
1.3 Problem Statement	3
1.4 Empirical Considerations	4
1.5 Impact	5
2 System Design	8
2.1 Goal-Oriented Design	8
2.2 System Architecture	9
2.3 Calculation API	12
2.4 Database	13
2.5 Design principles and considerations	15
3 Retrieval-Augmented Generation	17
3.1 Retrieval-Augmented Generation architecture	17
3.2 Documents	19
4 Reasoning pipeline	22
4.1 Decision Layer	22
4.2 Specialization Layer	23
4.3 Guardrail Layer	23
4.4 User Interaction Flow	24
5 Development	28
5.1 Costs	28
5.2 Testing	28
5.3 CI/CD	30
5.4 Deployment	32
6 Experiments	34
6.1 Decision Layer	34
6.2 Retrieval-Augmented Generation	35
6.3 System Quality	38
6.4 Experimental Setup	38
6.5 System Metrics	39
6.6 Costs	39
7 Results	41
7.1 Decision Layer	41
7.2 Retrieval-Augmented Generation	43
7.3 System Quality	45

8 Conclusion **47**
8.1 Future work 48

Bibliography **51**

A Appendix A **55**
A.1 LLM prompts and instructions 55

B Appendix B **60**
B.1 Cyclomatic complexity 60

1 Introduction

1.1 Background

Financial literacy is one of the key differentiators between economic success and stagnation. As Robert Kiyosaki famously argues [1], the core distinction between the rich and the poor often lies not in income but in financial behavior – how individuals budget, track expenses, save and invest. For those lacking access to quality financial education or guidance, wealth management becomes a challenge and opportunities for long-term financial growth are frequently missed.

Financial advisory services exist precisely to bridge this gap, offering individuals the tools and strategies needed to make informed financial decisions. However, in many countries – including Denmark – financial advice remains a largely exclusive service. Human advisors are expensive and scarce, often reserved for the high-net-worth clients of banks. This creates a structural inequality: those who could benefit most from financial advice are often the least likely to receive it.

Dreamplan [2] was born to challenge this status quo. Founded in 2019 by Stine Kalmer Jørgensen and Simon Bentholt, Dreamplan envisions a world where financial planning is not a luxury, but a widely accessible resource. The core part of Dreamplan is a B2C solution that delivers automated financial advice tailored to individual users. By analyzing data such as age, income, assets, marital status and savings, Dreamplan generates personalized wealth strategies and product recommendations. This approach, commonly referred to as robo-advisory, has already proven to be transformative – lowering the barrier to entry for financial advice while maintaining efficiency and scalability.

While traditional robo-advisors are built on complex rule-based systems and optimization algorithms, the advent of Artificial Intelligence — particularly the rapid advancement of Large Language Models (LLMs) — presents an exciting new frontier for the field. LLMs, with their ability to process and generate human-like text, introduce the possibility of designing more intuitive, dynamic and conversational financial advisory systems. When combined with Retrieval-Augmented Generation (RAG) techniques, LLMs can access, retrieve and reason over domain-specific knowledge in real-time, significantly enhancing the depth and personalization of advice provided.

This evolution opens the door to a new generation of intelligent financial agents: AI systems that can simulate the behavior of a human financial advisor, engage users in natural dialogue and offer tailored recommendations grounded in up-to-date information and best practices. Unlike traditional robo-advisors, these agents require minimal pre-defined rules or hard-coded financial models – they instead leverage the reasoning and generative capabilities of LLMs to adapt to a wide variety of user contexts and goals.

The aim of this thesis is to explore how such AI agents can be designed and implemented in alignment with Dreamplan's mission. Specifically, I focus on building a prototype conversational agent that provides personalized wealth strategies using LLMs and RAG. This system seeks not only to match the capabilities of current robo-advisory tools, but to surpass them in flexibility, interactivity and user accessibility, ultimately contributing to the democratization of financial advice for all.

We are rethinking financial advice – for everyone

Get an overview of your finances. Make decisions with peace of mind

[Sign up now](#) →

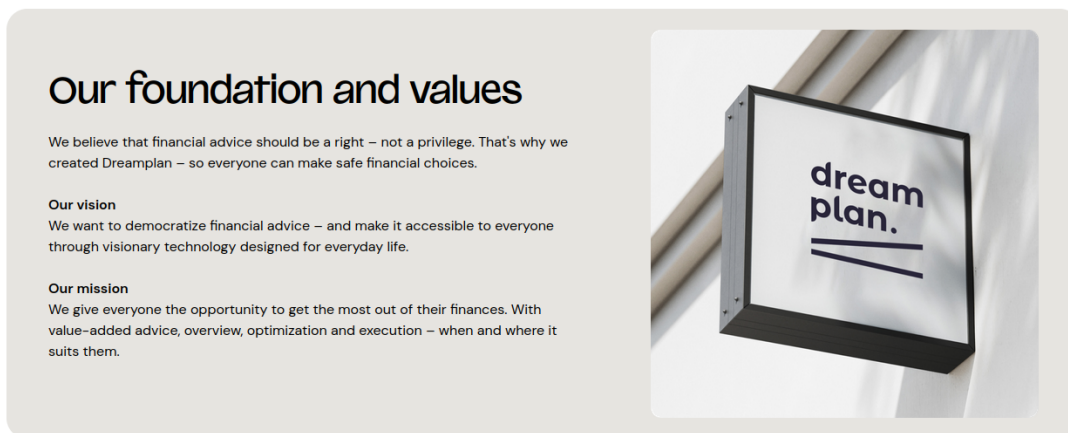


Figure 1.1: Screenshot from <https://www.dreamplan.io/om-os>.

1.2 Prior Work

Recent advancements in financial AI have largely centered around the development of *LLM-powered robo-advisors*, which generate investment recommendations directly through language generation. Notable examples include the pioneering *BloombergGPT*[3], as well as its open-source successors [4, 5]. These works highlight the remarkable capabilities of large language models (LLMs) in finance, particularly in tasks such as sentiment analysis, entity recognition and financial question answering. Early exploratory works [6] further reinforce the promise of LLM-based financial assistants.

However, despite their sophistication, LLM-only approaches often fall short in three key areas: explainability, financial rigor and control. These limitations pose serious challenges for real-world deployment, where user trust and regulatory compliance are essential. This thesis builds upon these prior efforts by addressing these limitations through a more robust and explainable system architecture.

In contrast to end-to-end generative advisors, this project adopts a hybrid architecture that integrates LLM-based interfaces with deterministic financial models. Specifically, I leverage Dreamplan’s existing *optimization-based recommendation engine* to produce financial advice that is both explainable and grounded. The LLM acts, then, as a natural language interface — responsible for interpreting user input, guiding the dialogue and routing queries to the appropriate computational modules — rather than generating rec-

ommendations itself. This division of responsibilities ensures greater transparency and traceability.

My design philosophy aligns with approaches such as FinRobot [5], which employs a hierarchical, agent-based system to orchestrate multiple components. However, my system differs in terms of its underlying tools, data access and architectural choices. Moreover, I draw inspiration from research on conversational AI in other high-stakes domains [7], where explainability and user alignment are paramount.

LLMs operating alone are prone to hallucinations and vague, ungrounded answers, especially in high-stakes contexts like personal finance. To mitigate this, recent research has focused on retrieval-augmented generation (RAG) systems, which combine LLMs with external knowledge retrieval. Studies such as [8, 9] show that LLM+RAG pipelines significantly outperform standalone LLMs on financial QA tasks.

System performance, however, depends heavily on design. Works like *LightRAG* [10] and *FastRAG* [11] explore architectural optimizations for latency and relevance, while [12] improves answer quality through query rewriting before retrieval. Additionally, [13] emphasize domain-specific retrieval tuning — an insight directly applied in this thesis to personal finance.

Evaluating AI systems in finance requires methods that go beyond traditional metrics. Recently, LLM-as-a-judge paradigms [14] have emerged as effective tools for assessing response quality and are employed in this thesis to benchmark user-facing interactions. At the same time, the user’s perception of LLM-generated content plays a key role in adoption. Prior work [15, 16] explores how trust can be fostered through improved interface design, explainability and consistent system behavior.

My methodology also draws from earlier work on explainable financial modeling [17], where system outputs are grounded in transparent assumptions and user-defined goals. The core idea remains the same: AI should enhance rather than obscure the decision-making process.

In summary, while recent research demonstrates the promise of LLMs and RAG in finance [5], existing solutions often:

- lack integration with rigorous, deterministic financial models;
- miss fine-grained orchestration and control across modules;
- or are constrained to narrow use cases like document-level question answering.

This thesis project proposes a novel integration of explainable financial modeling with LLM-based orchestration. The result is a trustworthy, interactive and scalable advisory platform that combines the strengths of both symbolic and neural systems.

1.3 Problem Statement

The central challenge addressed in this thesis is determining the feasibility of developing an LLM-powered financial advisory chatbot that integrates with a financial optimization model to generate personalized wealth strategies. The goal is to evaluate whether such a system can deliver accurate, context-aware and explainable financial recommendations at scale. Specifically, the thesis aims to address the following questions:

- How effectively can an LLM function as a decision layer in a multi-agent system, intelligently routing user queries to the appropriate resources and tools (e.g. financial optimization models, RAG systems)?

- How accurate and reliable is the LLM-based RAG system in answering financial queries?

To address these questions, the research will build on recent literature to identify the best strategies for prompting and orchestrating LLM agents, constructing the RAG system and evaluating the effectiveness of information retrieval as well as the accuracy of the decision-layer routing.

The system will be evaluated through both qualitative and quantitative metrics, including performance measures such as response time, accuracy of query routing and the quality and explainability of the AI-generated financial recommendations. User feedback will be used to further refine the system and validate the recommendations. These goals are designed to be operational and measurable, allowing for a clear assessment of the proposed system's effectiveness.

1.4 Empirical Considerations

Since this project is carried out in close collaboration with Dreamplan, the empirical setting is grounded in a real-world application, with direct access to both technical infrastructure and domain expertise.

Dreamplan provides access to anonymized financial and demographic data from its user-base, which serves as the foundation for experimentation and prototyping. This data includes variables such as age, income, savings, household status, assets, housing and more. All data analysis and system development will be conducted independently by the author, using tools and environments made available by the company.

The development of the LLM-based financial advisory system will involve building a conversational agent that integrates with Dreamplan's existing optimization models. Vector embeddings of financial documents and user input will be stored and queried using a Retrieval-Augmented Generation (RAG) approach.

To support this, Dreamplan provides monetary credit on the OpenAI API platform, allowing extensive use of LLMs without requiring dedicated hardware. All heavy computational tasks (such as inference and embedding generation) are handled via cloud infrastructure, minimizing local resource constraints.

In addition, the company provides access to their MySQL-based backend database, which could be used to manage and persist vectorized content and user data.

Given the nature of the data and the involvement of third-party AI services (e.g., OpenAI), all experiments involving sensitive information will rely strictly on test or anonymized data. This is in full compliance with GDPR regulations and internal company policies regarding data handling. No personally identifiable or production user data will be processed through external APIs.

The project is expected to be conducted with guidance and regular feedback from Dreamplan's technical and business teams. The author is responsible for the research, design, implementation and documentation of the AI-powered advisory system. Although there is no formal obligation to deliver a commercial product, the work is expected to inform future product development and internal prototyping at the company.

All required software and infrastructure are provided by the company or available through academic licenses. No additional funding is needed for the completion of this project. Regarding intellectual property, Dreamplan retains ownership over their existing codebase

and data, while any new software or models developed specifically during this thesis will follow DTU's and Dreamplan's mutual understanding regarding student IP contributions, which has been agreed upon prior to the start of the project.

1.5 Impact

The ultimate aim of this project is to explore the feasibility and efficacy of deploying a trustworthy, AI-powered financial advisor capable of interacting with users in a personalized, intuitive and scalable manner. By leveraging recent advances in Large Language Models and Retrieval-Augmented Generation (RAG), this work seeks to demonstrate that high-quality financial guidance can be delivered without relying on human advisors or complex rule-based interfaces.

The envisioned chat-bot system is not only able to guide users through Dreamplan's services—such as portfolio construction and automated product recommendations—but also capable of answering open-ended financial questions, helping users better understand their economic situation and educating them on fundamental financial concepts.

This aligns directly with the core challenge introduced in the background: financial advisory today is often expensive, limited in accessibility and tailored only to high-net-worth individuals. The proposed system aims to address this inequality by offering a low-barrier, interactive entry point to financial advice. Through natural language interaction, users with little financial literacy or prior knowledge can benefit from guidance that is both comprehensible and context-aware.

From an innovation standpoint, this project experiments with the role of LLMs as a “decision layer” that orchestrates financial logic (e.g., optimization algorithms) and knowledge (e.g., internal documentation or regulatory material). This architectural pattern—merging symbolic reasoning with statistical learning—may inform broader applications in AI-powered decision support systems beyond finance.

In practical terms, the results of this thesis will serve as a foundational prototype for Dreamplan's continued efforts to enhance their product with conversational AI features. If successful, the system can be further developed and potentially deployed in production, thereby contributing to the company's broader mission of democratizing access to financial advice.

Currently, Dreamplan users are required to complete a multi-step sign-up form (see fig. 1.2), sequentially entering all the financial data necessary for generating a personalized recommendation. Once the form is submitted, the system presents a detailed financial plan along with an option to schedule a meeting with a human advisor for further clarification (fig. 1.3).

The solution proposed in this thesis – illustrated in the chat example in fig. 1.4 – streamlines this process by delegating all computations and interactions to a large language model (LLM). This approach not only simplifies the user experience but also enables the system to handle follow-up questions about the recommendation autonomously, significantly reducing the need for manual intervention and lowering operational costs.

***User:** I'm 56 and my spouse is 50. I earn 80,000 DKK per month while my spouse earns 20,000 DKK. We both have pension policies: I contribute 7,200 monthly and have 2,500,000 DKK already saved, while my spouse contributes 3,600 DKK and has 500,000 DKK. We own a house worth 5,000,000 DKK with a mortgage of 4,000,000 DKK left and 20 years remaining. We also invest*

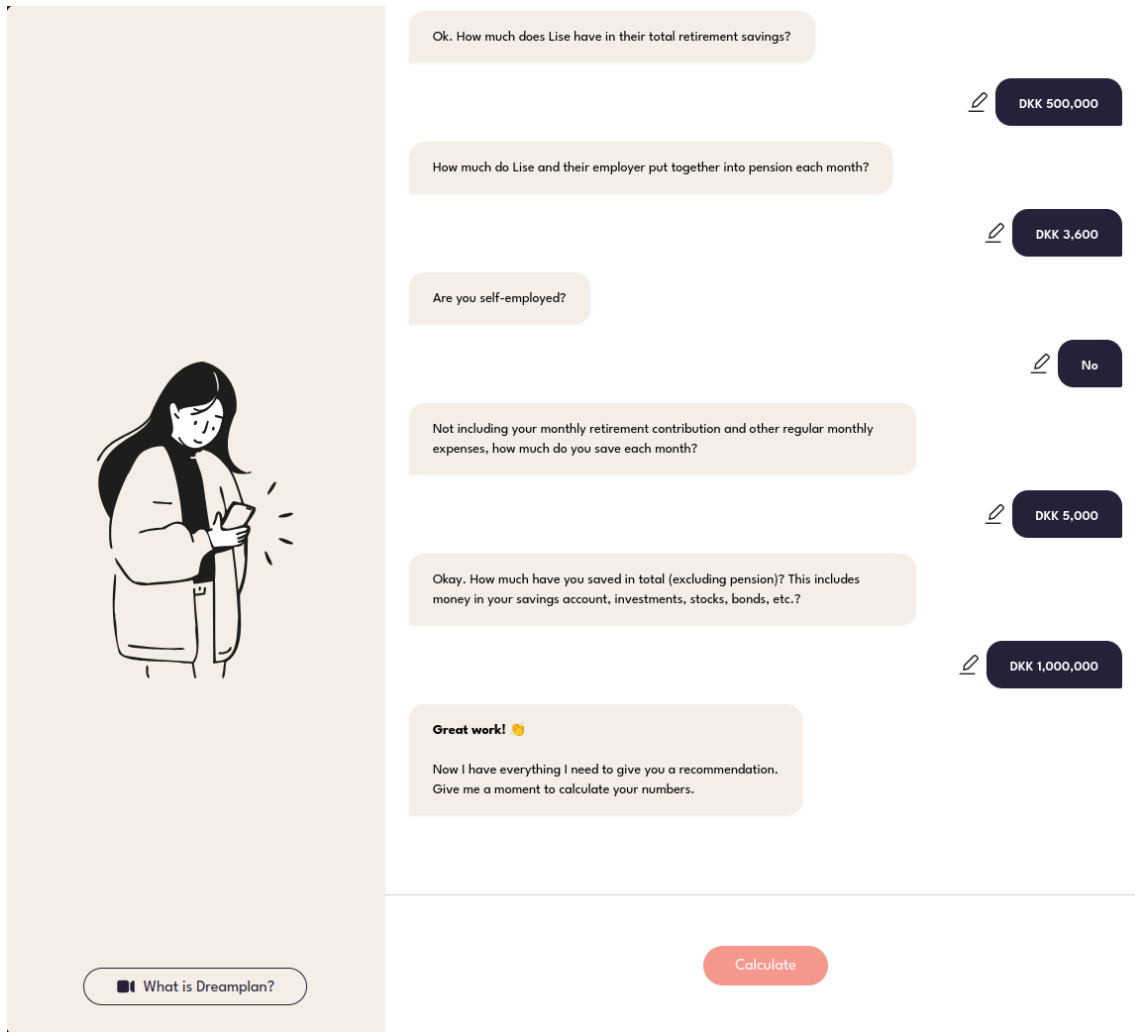


Figure 1.2: Dreamplan's sign-up flow.

5,000 DKK every month in stocks and currently have 1,000,000 DKK saved there.

Dreamplan AI: To enhance your financial future, consider reducing the proportion of wealth concentrated in your house by reallocating part of your housing savings. Maintain your partner's pension contribution at 43'200 DKK annually, as in the baseline. Increase your own pension contributions to 223'892 DKK annually between 2025 and 2036 to secure a projected payout of 453'842 DKK from 2037. Redirecting more of your current savings – beyond the existing 5'000 DKK monthly investment in stocks – could yield higher long-term returns. Finally, growing your emergency savings by at least 125'000 DKK will strengthen your household's financial resilience and increase overall retirement wealth by approximately 188'363 DKK.

Figure 1.4: An example of a chat with the proposed solution.

Your Dreamplan

Optimise your savings

Recalculate

Book a counselling session

The difference in your wealth when you retire if you follow Dreamplan's recommendations

+DKK 522,483

This is how your wealth will develop in the coming years.

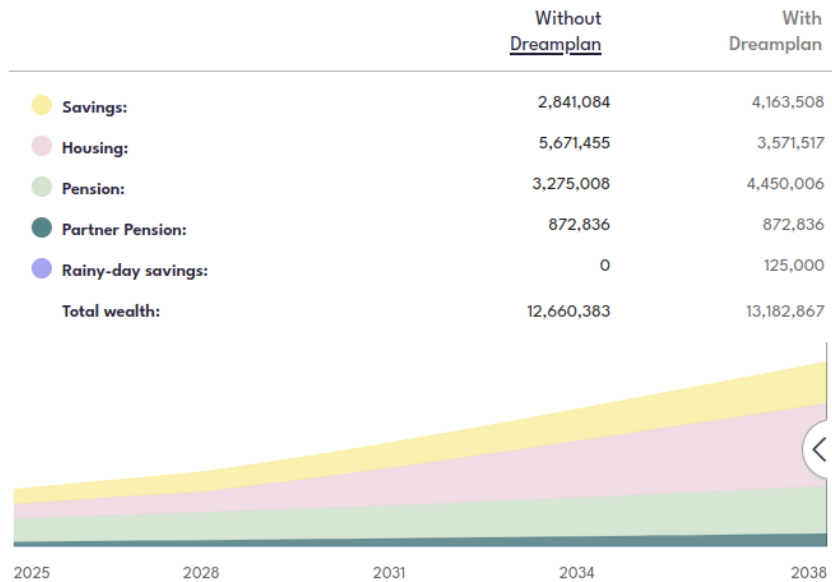


Figure 1.3: Dreamplan's results page after completing the signup.

2 System Design

This chapter provides a comprehensive overview of the system's architecture and the principles that guided its development.

2.1 Goal-Oriented Design

The project consists of an AI-powered Financial Advisor that leverages large language model (LLM) agents to handle different user tasks. A Retrieval-Augmented Generation (RAG) system enhances the advisor's performance by improving both the factual accuracy and explainability of the responses. It also makes use of Dreamplan's proprietary API, which enables financial forecasting and asset allocation recommendations based on user-provided household financial data.

The primary interaction modality is a chat interface, similar to modern AI assistants such as OpenAI's ChatGPT [18], Anthropic's Claude [19], or Google DeepMind's Gemini [20]. Upon launching the application, the user is prompted with an introduction explaining the functionality of the system. Users can, then, input their financial data, ask questions about the computed recommendations, or inquire about general financial topics and services.

The design of the system is rooted in the principles of *Goal-Oriented Requirements Engineering (GORE)* [21]. GORE is a methodology in software engineering that focuses on capturing, modeling and analyzing the goals of stakeholders. Rather than simply documenting system functionalities, GORE allows developers to structure system design around the high-level intentions and desired outcomes of users. This goal-driven approach is particularly suited to systems involving LLM agents, where intent recognition and outcome mapping are critical to user satisfaction and successful task execution.

In this context, each user input is interpreted as a goal expression and the system decomposes this high-level goal into sub-goals, which are then handled by specialized agents or tools within a modular architecture. This aligns with how GORE structures requirements into a goal hierarchy: strategic-level goals are refined into operational objectives and assigned to specific actors or components in the system.

The system centers on three primary user goals:

1. **Recommendation:** the user seeks a personalized financial recommendation. The system parses anagraphical data and household financial information – such as incomes, savings, assets and properties – and queries the forecasting engine via the proprietary API, responding with an accurate and tailored wealth allocation strategy.

I'm 56 and my spouse is 50. I earn 80,000 per month while my spouse earns 20,000. We both have pension policies: I contribute 7,200 monthly and have 2,500,000 already saved, while my spouse contributes 3,600 and has 500,000. We own a house worth 5,000,000 with a mortgage of 4,000,000 left and 20 years remaining. We also invest 5,000 every month in stocks and currently have 1,000,000 saved there. What can you recommend us?

Listing 2.1: An example user query for the *recommendation* goal.

2. **Explanation:** the user requests clarification of the recommendation received in a previous step, asking to further elaborate the reason why certain allocations were made or if any change is possible.

Why should I pay more into my pension, according to your recommendation?

Listing 2.2: An example user query for the *explanation* goal.

3. **Generic:** the user asks about general financial concepts or national policies, such as taxes or investment products. For this project, the scope is limited to the Danish market and policies, as Dreamplan is based in Denmark and currently focusing on this user segment.

What are tax cards in Denmark?

Listing 2.3: An example user query for the *generic* goal.

These goals are mapped to a modular architecture built on a centralized LLM-based agent, which orchestrates the interaction by dispatching sub-goals to specialized tools or secondary agents. This architecture supports a goal decomposition and dispatching layer, in which the main agent analyzes the intent behind each query and routes it to appropriate components for resolution – in accordance with the GORE model’s emphasis on goal satisfaction through actor responsibility – i.e. other specialized LLM-agents, each equipped with their own set of tools in order to interact with the RAG system (discussed in chapter 3) and the external Calculation API (see section 2.3). This architecture is discussed in further detail in chapter 4.

2.2 System Architecture

The system is built around a modular and hierarchical *agent-oriented* architecture, designed to promote separation of concerns, ease of maintenance and horizontal scalability. Each module is responsible for a specific functionality, allowing for independent development, testing and deployment. This architectural choice facilitates rapid iteration and integration of new components, especially valuable in experimental AI workflows.

This structure draws inspiration from recent work in multi-agent LLM systems. In particular, the HALO framework proposed by Hou et al. (2025) [22] introduces a decentralized orchestration mechanism that assigns distinct roles to specialized sub-agents. These agents interact via a hierarchical control flow and employ local prompt optimization strategies to enhance individual and collective performance. Similarly, Rasal et al. (2024) [23] advocate for a more canonical vertical and centralized orchestration pipeline, emphasizing modularity and role alignment in complex reasoning tasks. Both these approaches build on consolidated agent-based paradigms to offer a reliable and reproducible framework for orchestrated problem-solving with LLMs.

The system is implemented in Python (version 3.12), chosen for its rich ecosystem of machine learning libraries, native support for concurrent execution and robust community support. Dependency management and environment reproducibility are handled using *uv* [24], a high-performance Python package manager that integrates linting and formatting tools out-of-the-box, helping maintain code quality across the project lifecycle.

The backend is exposed as a RESTful API using *FastAPI* [25], selected for its asynchronous capabilities and overall suitability for high-performance microservices. This makes it well-suited for serving AI models and orchestrating agent communications in production environments.

A visual overview of the system is presented in fig. 2.1. The architecture is logically divided into four major layers:

- The API layer (FastAPI), which acts as the system's external interface, accepting client requests and routing them to the orchestration core.
- The multi-agent infrastructure (or reasoning pipeline) – detailed in chapter 4 – which manages agent lifecycle, communication, role assignment and coordination strategies. Agents operate under a hierarchical controller that delegates sub-tasks and aggregates outputs to solve complex queries collaboratively.
- The Retrieval-Augmented Generation (RAG) subsystem – discussed in detail in chapter 3 – responsible for grounding the generative agents in relevant external knowledge. It represents a core component of this system, improving the overall factual consistency and contextual awareness.
- The integrations layer – currently only including Dreamplan's Calculation API integration (see section 2.3) – which interfaces with external systems or tools e.g. web search, external APIs, file system interaction. Can be accessed directly by agents or through tool-wrapping agents.

This architectural decomposition ensures that all the subsystems can evolve or expand independently while maintaining tight integration.

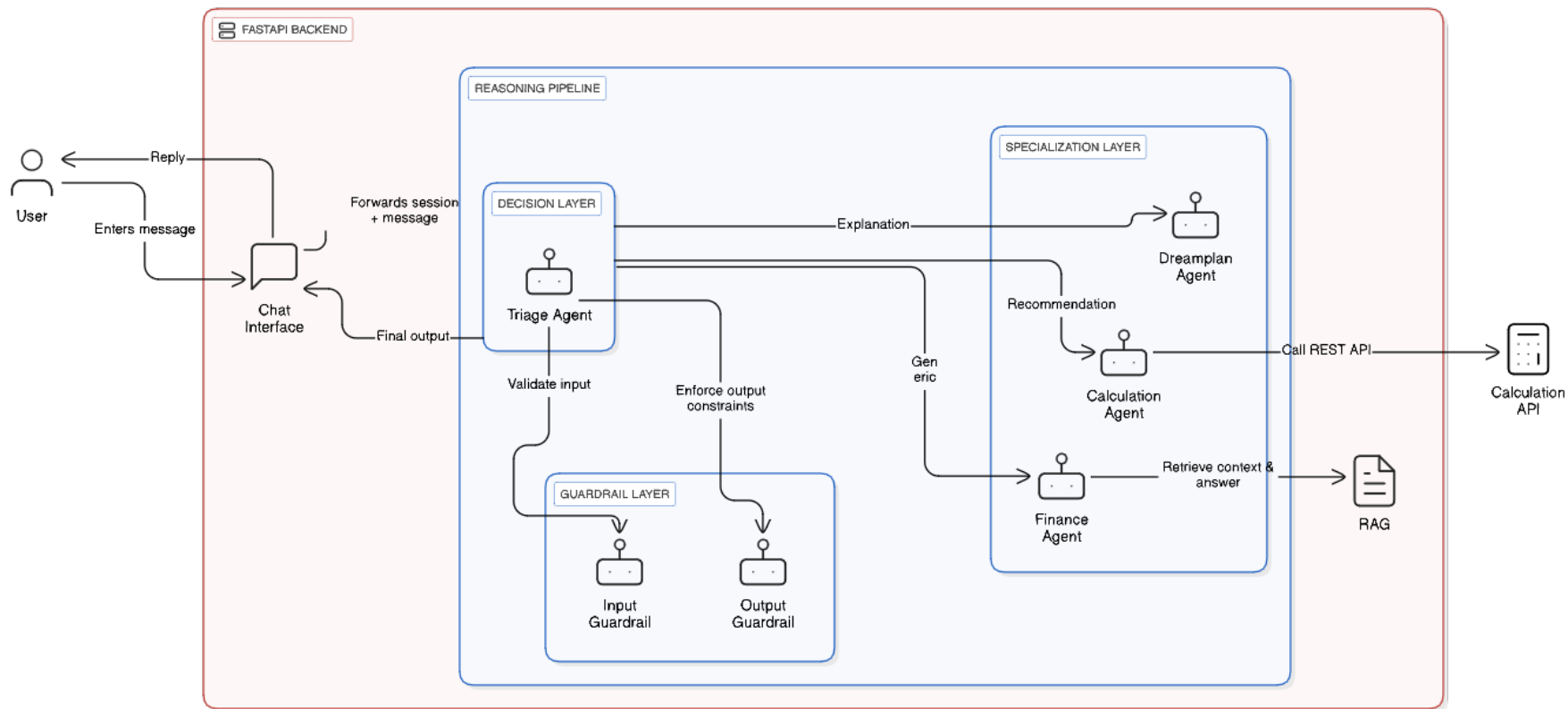


Figure 2.1: System Architecture overview.

2.3 Calculation API

The foundational component of Dreamplan’s financial advisory platform is the **Calculation API** – a deterministic, RESTful service responsible for generating tailored long-term wealth allocation strategies. Developed in C# and conforming to the *OpenAPI 3.0* specification [26], the API acts as the decision engine for household-level financial planning, translating structured user data into actionable recommendations.

The API is deployed on a private Azure cloud instance managed by Dreamplan, ensuring security and compliance. Access is restricted to authenticated internal systems and authorized personnel, following industry-standard security policies for sensitive financial data handling.

From the user’s standpoint, the API operates as a *black box*: it returns calculated outcomes without exposing internal mechanisms or decision-making algorithms. Internally, however, the architecture is designed to be modular and extensible. This allows for seamless integration of new components – such as regulatory rulesets, additional asset classes, or alternative allocation strategies – without impacting core functionality.

The API returns a structured JSON response comprising three core components:

- **Baseline:** a snapshot of the household’s current financial state based on submitted data.
- **Recommendation:** an optimal long-term wealth allocation plan generated by proprietary algorithms.
- **Projections:** future financial outcomes (e.g., net worth at retirement) simulated under both the baseline and recommended plans.

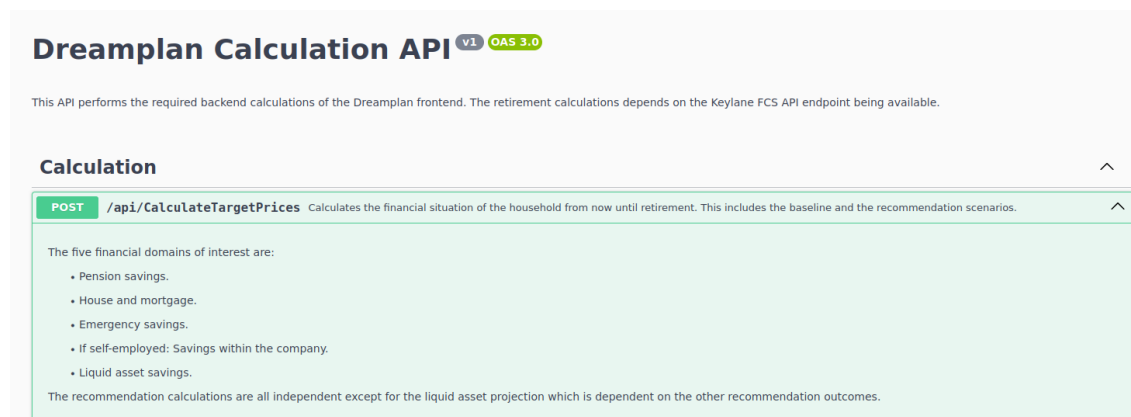


Figure 2.2: A screenshot of the `CalculateTargetPrices` endpoint of the Calculation API in Swagger UI.

As the API’s response is pretty substantial in size, before feeding it to the LLM-agent I programmatically remove the forecasting object.

Within the larger Dreamplan system, this API is triggered during the user onboarding or data submission phase. In the context of this project, it is seamlessly integrated into the LLM agent pipeline.

When a user submits a natural language query related to financial planning, the system’s **triage agent** – a general-purpose LLM – first classifies the user’s intent according to the predefined goal categories (e.g., recommendation, explanation, or generic). If the input

requires a financial recommendation, the request is forwarded to a specialized **calculation agent**. This agent, equipped with domain-specific prompt engineering and tool-use capabilities, is responsible for:

1. Parsing and validating user data from natural language.
2. Constructing a compliant JSON payload according to the API specification.
3. Executing the API call via a secure HTTP tool wrapper.
4. Parsing the response and passing the structured results back into the reasoning pipeline for contextual explanation.

This setup allows for a clean separation of concerns: the LLM agents handle interpretation, dialogue management and explanation, while the Calculation API performs high-stakes financial computation. Such a decoupled architecture adheres to software engineering best practices—supporting maintainability, auditability and system evolution.

In future extensions, the pipeline could also incorporate multi-scenario simulation tools, fine-grained tax models, or rule-based validation engines. The modular nature of the Calculation API makes it well-positioned for these upgrades.

2.4 Database

In order to have a clear history of the user sessions with queries and responses, I integrated **Supabase** [27], a hosted backend service that provides a PostgreSQL database with RESTful and real-time APIs. To interact with the database asynchronously in Python, the implementation uses *asyncpg* for connection pooling and *SQLAlchemy* for schema definition and management.

Connection parameters (e.g., username, password, host, port and database name) are securely loaded from environment variables using `python-dotenv`. This approach decouples configuration from code and ensures portability across environments.

2.4.1 Schema Design

The database schema consists of two primary tables: **replies** and **feedbacks**, each serving a distinct purpose in logging user interactions.

The replies table stores records of individual chat interactions, including both the user's input and the system's response. Its fields are:

- **id**: a unique primary key for each entry.
- **session_id**: a string identifier linking replies to a specific chat session.
- **query**: the text of the user's input.
- **response**: the generated response from the system.
- **timestamp**: the time at which the reply was stored, automatically set to the current time.
- **response_time**: an optional integer measuring response latency in milliseconds.

The feedbacks table collects user evaluations of system replies across several dimensions. Each entry is associated with a session and includes:

- **id**: a unique primary key for each feedback record.
- **session_id**: links feedback to a particular session.

- **correctness, relevance, clarity, satisfaction:** optional integer ratings provided by the user.
- **comments:** optional textual feedback.
- **timestamp:** automatically recorded submission time.

To manage concurrent access efficiently, the application employs *asyncpg*'s connection pooling mechanism. The connection pool is initialized lazily on the first database interaction via the `connect_db` coroutine and persists globally for reuse. When the application shuts down, a clean disconnection is ensured via `disconnect_db`.

Integration with the FastAPI web framework is handled through a *lifespan event* [28], ensuring that the database pool is created on application startup and properly closed during shutdown.

Two main operations are supported for data insertion:

- `insert_reply`: Logs a single user query and system response, along with response latency.
- `insert_feedback`: Stores user ratings and optional comments related to a specific session.

A separate utility function, `create_tables`, uses SQLAlchemy to initialize the database schema by executing the necessary DDL statements.

An example entry from the `replies` table is shown in table 2.1, capturing the interaction between user and system, along with the timestamp and response latency. An example entry from the `feedbacks` table is shown in table 2.2.

Table 2.1: Example entry from the `replies` table

Field	Value
<code>id</code>	42
<code>session_id</code>	7b92c1f0b3e249d885d3a2d9e2a2a1c6
<code>query</code>	What is the capital of Denmark?
<code>response</code>	The capital of Denmark is Copenhagen.
<code>timestamp</code>	2025-07-30 14:12:08
<code>response_time</code>	124

Table 2.2: Example entry from the `feedbacks` table

Field	Value
<code>id</code>	17
<code>session_id</code>	7b92c1f0b3e249d885d3a2d9e2a2a1c6
<code>correctness</code>	5
<code>relevance</code>	5
<code>clarity</code>	4
<code>satisfaction</code>	5
<code>comments</code>	Clear and concise answers, thanks!
<code>timestamp</code>	2025-07-30 14:13:02

2.5 Design principles and considerations

This project has been developed as a functional prototype to explore the integration of Large Language Models (LLMs) within a multi-agent system architecture. As such, certain constraints – particularly around security and user data protection – have been relaxed to prioritize experimentation, modularity and performance. The following section outlines the key design principles that guided development, as well as considerations for adapting the system to real-world, production-grade settings.

Given the exploratory nature of this prototype, user privacy and data protection mechanisms have not been fully implemented. Users are explicitly advised not to input any personally identifiable information (PII), such as names, email addresses, national identification numbers (e.g., CPR), phone numbers, or physical addresses.

However, if the system were to be deployed in a production environment, incorporating a robust privacy-preservation strategy would be critical. In such a scenario, a dedicated **sanitization layer** would be introduced as a pre-processing stage between the user input and any downstream LLM interactions. This layer would be responsible for detecting and anonymizing sensitive data to ensure compliance with privacy regulations such as the GDPR.

The sanitization module would likely adopt a hybrid approach, including:

- Rule-based heuristics: using regular expressions and syntactic patterns to detect well-known PII formats (e.g., email addresses, phone numbers).
- Keyword and entity detection: employing named entity recognition (NER) models to identify personal or sensitive entities in free text.
- Placeholder substitution: replacing identified PII with structured placeholders (e.g., {NAME}, {EMAIL}) to preserve semantic coherence while ensuring anonymity.

Such a privacy-aware preprocessing pipeline would not only enhance user safety but also allow interactions with third-party or externally hosted LLMs without risking the exposure of confidential information.

A foundational goal of the system is to support **modularity** and ease of extensibility. To this end, the architecture is designed around the *Single Responsibility Principle* [29], ensuring that each component in the system is tasked with a distinct, well-defined function. This promotes a clean separation of concerns and enables independent testing and debugging of modules, safe substitution or upgrade of components (e.g., replacing or adding an agent or changing the underlying retrieval technique in the RAG module) and greater maintainability and scalability over time.

Each subsystem – from the retrieval layer to the orchestration logic – can be developed and refined independently, enabling iterative improvements without jeopardizing the integrity of the overall system.

Another central design consideration is **transparency**, both in the internal logic and in the evaluation of outputs. All system components are accompanied by clear documentation, including:

- Descriptive annotations of each agent's role and responsibilities.
- Prompt templates and hyper-parameters for LLM invocations.
- Evaluation metrics and intermediate outputs for inspection and debugging.

This design choice aligns with recent best practices in human-AI interaction and fosters trust by making the system's reasoning process observable and interpretable.

Although the system currently runs as a unified deployment in order to minimize latency as much as possible, ensuring a smooth and seamless user experience, its modular architecture is amenable to future decomposition into distributed services. For instance, if the company decided to employ open-source models running on local servers, the orchestration infrastructure could be moved to its own REST API. This kind of architecture makes it possible to:

- Migrate bottleneck components to dedicated services (e.g. retrieval, summarization).
- Introduce asynchronous task queues for long-running agent operations.
- Apply container orchestration strategies (e.g. Docker + Kubernetes) for deployment at scale.

By anticipating such adaptations, the design remains flexible and production-aware, despite the current prototype status.

3 Retrieval-Augmented Generation

Retrieval-Augmented Generation (RAG) [30] is an information retrieval and question answering technique that integrates external knowledge sources into the generation process of Large Language Models (LLMs). Unlike traditional LLMs that rely solely on their internal parameters, RAG systems dynamically retrieve relevant documents from a knowledge base to ground responses in factual, up-to-date information.

Recent research, including the work of Wang et al. (2025) [8] and Qin et al. (2025) [9], claims that RAG-enhanced pipelines mitigate hallucination, enhance factual accuracy and reduce the dependency on model size or pretraining corpus coverage, consistently outperforming standalone LLMs on domain-specific tasks such as financial question answering, where precision and context-awareness are critical. In this thesis, I compare four different setups with an LLM-only baseline.

3.1 Retrieval-Augmented Generation architecture

The Retrieval-Augmented Generation (RAG) system is implemented using the open-source NLP framework *Haystack* [31], a widely adopted Python library for building production-ready search pipelines.

Given that this application is customer-facing and users interact with the chat-bot in real time, the system must provide fast and seamless responses.

To meet these latency and cost-efficiency requirements, I preferred lightweight architectures, balancing cost, performance and implementation complexity. As the bulk of the documents used for the retrieval is not considerable, I opted for storing them locally on start-up, so to minimize the latency of the retrieval and avoid additional expenses, even though open-source and free decentralized vector databases alternatives were considered, such as Qdrant. Haystack provides helper classes and functions to handle the chunking process, ensuring that each stored document is not too large and information is not lost in the splitting.

The baseline consists in a specifically instructed LLM agent that answers the user queries by only using its internal training knowledge. As per the RAG methods I use the following:

- **BM25**: this is the actual RAG baseline, employing BM25 for retrieval inspired by Intel Lab's FastRAG implementation [32].

Compared to dense or hybrid retrievers, BM25 offers significantly lower inference time and resource consumption, while still delivering strong performance in many practical settings.

Although dense retrievers have gained popularity, classical sparse methods such as BM25 remain competitive baselines, especially in scenarios demanding low latency and cost efficiency. For instance, Thakur et al. (2021) [33] and Li et al. (2025) [34] demonstrate that BM25 can match or even outperform dense retrieval in specific domains, making it a practical choice for real-time, customer-facing applications. Zhu et al. (2025) [35] findings show that BM25 has the highest recall, completeness and *Effective Information Rate* [35], while maintaining the lowest hallucination rate.

- **BM25 + Reranker:** A **reranker** is a model used in RAG to re-score and re-rank retrieved documents or chunks, which usually improves the performance of the system, as seen for example in Dong et al. (2024) [36].

However, other studies show that the reranker is not always improving the accuracy and performance of the baseline — such as in Kim et al. (2024) [37] — and this can be due to various factors, including the choice of reranker model, task-specific requirements and domain mismatch. For this reason, I considered it important to experiment with reranking and compare the performance with and without it.

The reranking component is implemented using HuggingFace’s open-source *sentence-transformers/all-MiniLM-L6-v2* model, using cosine similarity as the scoring function.

- **Dense:** Dense retrieval methods encode both queries and documents into high-dimensional dense vectors using deep learning models (typically transformers) and retrieve documents based on vector similarity, usually cosine or dot product. This paradigm enables semantic matching rather than exact term overlap.

In this thesis, I use the OpenAI’s `text-embedding-ada-002` model.

Dense retrieval has shown strong performance, especially in zero-shot and domain-transfer settings. For example, Karpukhin et al. (2020) [38], Luan et al. (2021) [39] and more recently, Maillard et al. (2023) [40] demonstrate how dense retrievers outperform sparse methods on tasks requiring deeper semantic understanding. However, they often require substantial GPU resources for indexing and inference and their performance can degrade on small or domain-specific corpora without fine-tuning [33].

- **Dense + Reranker:** Combining dense retrieval with reranking is a standard strategy in modern RAG pipelines. The dense retriever retrieves semantically relevant candidates, while the reranker (often a cross-encoder or late-interaction model) re-evaluates the top-k candidates with greater precision.

In my implementation, the reranker remains the same as for BM25 (i.e., `all-MiniLM-L6-v2`), allowing for a direct comparison between sparse and dense pipelines under identical reranking conditions. This helps isolate the contribution of retrieval quality versus reranking effectiveness.

The RAG pipeline includes a dedicated *prompt builder* module (listing 3.1), which dynamically constructs prompts for the generator. This component integrates the retrieved documents as context – including the sources’ URLs – and provides tailored instructions on how to formulate a response.

The final stage of the pipeline is the *generator*, which uses OpenAI’s *GPT-4.1-nano* model [41]. This model was selected due to its favorable trade-off between speed and quality, making it well-suited for real-time applications. The generation is constrained by setting the `max_tokens` parameter to 256, so that the resulting answers are concise and only containing the essential information – the user can then ask more specific questions or refer to the information source.

The generator component was favored over simply returning the retrieved context because the information can be scattered across different chunks and the LLM helps extracting only the relevant parts, presenting them to the user in a readable and clear fashion.

```
You are an AI assistant tasked with answering questions using the provided context.
```

```

Your response must be:
- Concise and in your own words.
- Based only on the documents below - cite them where
  relevant by providing the URL in parentheses (e.g., (Source:
  https://example.com)).
- If the question is unclear or too broad, ask for clarification
  .

--- CONTEXT ---
{% for doc in documents %}
Document {{ loop.index }} (Source: https://{{ doc.meta.source }}
):
{{ doc.content }}

{% endfor %}
-----

Question: {{ question }}

Answer:

```

Listing 3.1: The prompt used in the prompt builder.

3.2 Documents

While Dreamplan has provided access to user financial and demographic (anagraphical) data from their internal database, this data is not currently utilized in the project. This decision aligns with the idea of building a proof-of-concept system without processing personal or sensitive information.

To populate the document store for the Retrieval-Augmented Generation (RAG) system, a dedicated web scraping tool was developed using Python, combining the Playwright browser automation framework, because of its compatibility with JavaScript client-rendered content and *BeautifulSoup* for HTML parsing due to the ease of use and high number of features.

The scraper was designed to be robust, modular and efficient in crawling relevant content from two main official Danish portals, selected due to their authoritative nature and comprehensive coverage of topics central to personal finance and life in Denmark: *lifeindenmark.dk* and *skat.dk* (screenshots are shown in fig. 3.1 and fig. 3.2 for reference). The scraped content pertains to key topics relevant to personal finance and financial regulations, such as taxation and tax returns, employment contracts and rights, social benefits and pensions, housing and relocation, personal identification and banking.

In total, the scraper retrieves and processes over 327 unique pages, forming the core knowledge base used by the chat-bot's RAG system. A comprehensive list of the resulting documents and their categorization can be found in in the source code in the `documents/index.md` file.

The recursion depth and crawl coverage are designed to strike a balance between breadth and content quality. Rate-limiting between requests is implemented using randomized sleep intervals to reduce server load and avoid IP bans.

To prevent crawling of undesired pages, the script includes a manually annotated blacklist

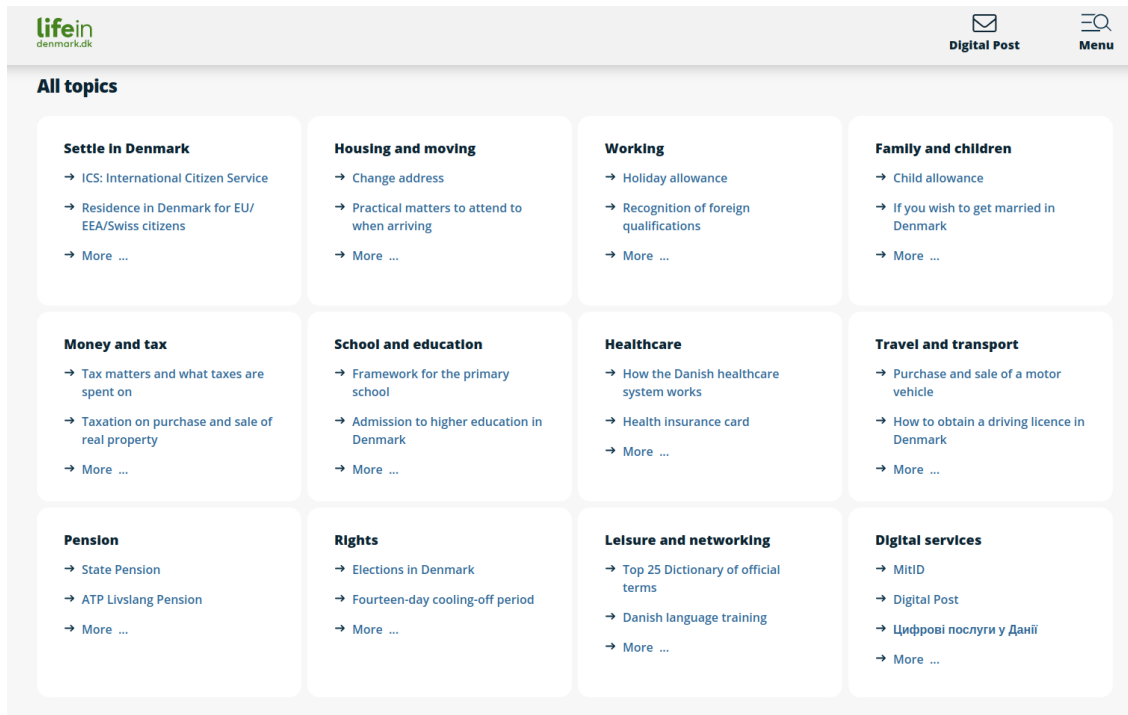


Figure 3.1: A screenshot of lifeindenmark.dk website.

containing ULRs of known irrelevant pages, domain-based restrictions enforced via URL parsing and deduplication through a set of visited URLs, making the script extremely efficient for its task and ensuring that only unique and contextually relevant documents are collected for downstream processing.

The scraper performs the following key tasks in a recursive and systematic fashion and can easily be extended to scrape even more websites:

1. The script defines a set of root URLs and allowed domains, ensuring the crawl is restricted to predefined, trusted sources. A persistent set of visited URLs is maintained to prevent redundant downloads.
2. Using Playwright's Chromium driver in headless mode, the tool loads each URL and waits for client-side content to render. This is crucial for JavaScript-heavy pages, ensuring full content extraction. An automatic interaction with common cookie consent banners is also implemented. The script attempts to click an "Accept all" button if present, enhancing navigation and access to the full site content.
3. The rendered HTML is saved to disk and simultaneously processed into plain text. The conversion excludes non-informative elements (such as `script`, `style`, `nav`, `footer`, etc.) via BeautifulSoup's tree manipulation. This cleaning procedure significantly improves the quality and relevance of the resulting text corpus. Haystack's own converter from HTML to `Document` (the class used in the library to represent documents for the RAG) was also considered as a tool to extract the text in order to speed-up the process but it was soon discarded after a few experiments because the removal of the HTML elements was too aggressive, often leading to missing text information that was supposed to be in the document and was left intact otherwise. Additionally, specific text sequences containing irrelevant information – such as privacy policies statements – are statically removed to lighten and declutter the

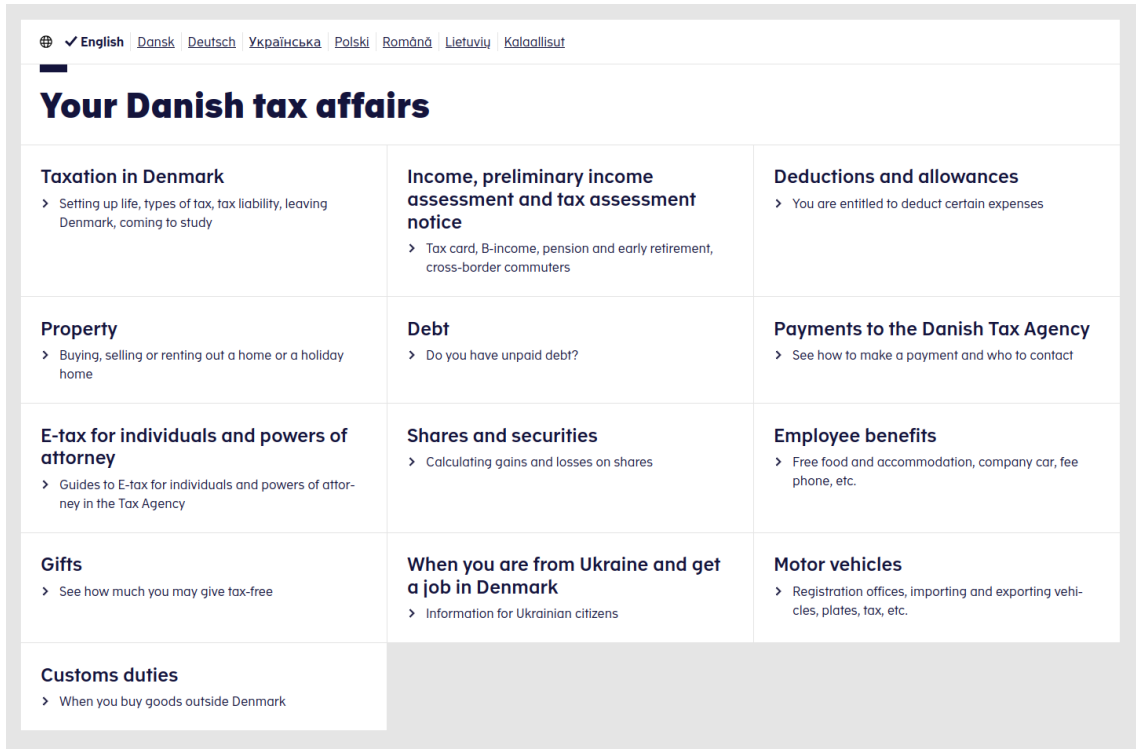


Figure 3.2: A screenshot of skat.dk website.

documents.

- From each downloaded page, all hyperlinks are extracted and normalized. The scraper filters out external domains and URLs matching a blacklist of known irrelevant or duplicate paths. This prevents inclusion of privacy policies, navigation-only pages, or redundant entries.
- For each unique URL, both the original HTML and the corresponding cleaned text are stored, the original file for auditing and potential reprocessing and the cleaned one to be fed to the RAG pipeline. File names are derived from the URL paths, making the dataset traceable and easy to navigate.

4 Reasoning pipeline

The pipeline consists of multiple components: Large Language Model (LLM) agents, a Retrieval-Augmented Generation (RAG) system and a suite of specialized tools accessible to the LLMs. Since Dreamplan provided monetary credit for OpenAI's API usage, the LLMs employed throughout this project are exclusively OpenAI's **GPT** models [42]. This choice was also motivated by the lack of local hardware capable of efficiently running such models – standard CPUs are neither sufficiently powerful nor optimized for the computational demands of large-scale language models.

The LLM agents are developed using the **OpenAI Agents SDK**, which offers a streamlined and effective framework for building and deploying LLM-based agents. Notably, the Agents SDK designates the triage agent as both the initial processor and the final responder to user queries. To achieve greater modularity and finer control over the overall reasoning process and output quality, I designed a custom reasoning pipeline inspired by the single responsibility principle from object-oriented programming. This principle ensures that each pipeline stage is dedicated to a specific task, promoting clarity, maintainability and extensibility.

In this architecture, each agent is equipped with a distinct set of instructions and tools tailored to its role. The triage agent, however, deviates from this pattern: instead of having its own tools, it maintains a "hand-off" list – essentially references to the specialized agents. This enables the triage agent to delegate subtasks effectively, orchestrating the collaboration between experts while preserving control over the final user-facing response.

4.1 Decision Layer

The first stage of the reasoning pipeline is the **decision layer**, also referred to interchangeably as the dispatching layer throughout this document. Its primary responsibility is to interpret the user's query, determine the intended end-goal and route the query to the specialized agent best suited to handle that specific type of request.

This decision-making process is fully delegated to an LLM-powered agent, referred to as the **triage agent**. The triage agent operates exclusively through prompt engineering, relying entirely on the model's contextual understanding — including the initial instructions that establish the agent's behavior for the session — without any hard-coded rules or heuristic logic. This design choice is motivated by the clearly defined and distinct nature of user goals, which contemporary LLMs are well-equipped to reason about and resolve effectively.

Once the triage agent identifies the user's goal, it delegates the request by invoking the corresponding specialized agent. Each specialized agent executes its own reasoning workflow, which may involve leveraging external tools and ultimately returns its findings to the triage agent.

This modular and structured communication protocol enhances control over the reasoning chain and overall pipeline. It facilitates detailed testing, monitoring and debugging at each individual stage, thereby improving transparency and maintainability of the system.

All the LLM prompts that I used are available in the `components/prompts.py` file in the source code and in appendix A.1.

4.2 Specialization Layer

The **specialization layer** is responsible for executing the agents and tools that directly contribute to achieving the user's goal. It comprises two distinct LLM-powered agents, each designed to handle a specific type of user intent:

- **Calculation Agent:** this agent is tasked with fulfilling the *recommendation* goal. It constructs the appropriate request for the Calculation API and processes its response. To do this, it utilizes a dedicated tool that maps the user's household financial and demographic (anagraphical) data into the structured JSON format required by the API.
- **Dreamplan Agent:** this agent is tasked with fulfilling the *explanation* goal. It can only work when a calculation has already been executed as it uses the recommendation results to provide actionable steps to the user for them to increase their wealth. It doesn't use any tool because all the relevant information to explain the results is present in its instructions set. The instructions set is based on Dreamplan's internal guidelines for wealth allocation and it aims to explain to the user how to better allocate their wealth and why.
- **Finance Agent:** this agent addresses the *generic* goal by handling broad, finance-related user queries. It does so by interacting with a Retrieval-Augmented Generation (RAG) component. The agent employs a specialized tool to issue the user's query to the RAG pipeline and return the retrieved, contextually enriched response.

4.3 Guardrail Layer

The **guardrail layer** acts as a critical safety net within the reasoning pipeline, ensuring the quality and reliability of the data flowing through the system. Positioned around the decision and specialization layers, this layer systematically validates both the inputs received from users and the outputs generated by agents before and after processing.

At its core, this layer employs two specialized LLM-powered agents: the *input guardrail agent* and the *output guardrail agent*. These agents serve to enforce constraints and checks that prevent invalid, ambiguous, or unsafe data from propagating through the pipeline.

- **Input Guardrail Agent:** Before any reasoning or delegation occurs, the input guardrail agent reviews incoming user queries to verify their validity and appropriateness. It uses a dedicated prompt-driven LLM model to assess the input against predefined criteria, ensuring that only well-formed and contextually suitable requests proceed to the triage agent. Invalid or problematic inputs can trigger tripwires that prompt corrective handling or user feedback.
- **Output Guardrail Agent:** After a specialized agent has processed a request and generated a response, the output guardrail agent scrutinizes the output for correctness, completeness and alignment with expected formats and constraints. This agent filters or flags outputs that might be erroneous, inconsistent, or out-of-scope, thereby maintaining the integrity of the system's responses.

Both guardrail agents operate through prompt engineering without relying on hard-coded rules, leveraging the flexible contextual reasoning abilities of modern LLMs. They are integrated transparently in the pipeline through decorator-like constructs (`@input_guardrail` and `@output_guardrail`), enabling seamless invocation during request handling.

By embedding this validation layer around the **triage agent** in the decision layer and the specialized agents in the specialization layer, the architecture achieves a robust modular design. This design enhances fault tolerance, facilitates easier debugging and monitoring and ensures consistent, high-quality interaction outcomes.

In the underlying implementation, the guardrail layer is instantiated using agents configured with specific instruction sets (`input_guardrail_instructions` and `output_guardrail_instructions`) and lightweight LLM models optimized for validation tasks. These agents are invoked asynchronously during runtime to maintain efficiency and responsiveness.

4.4 User Interaction Flow

The user interaction flow is illustrated in the UML sequence diagram in fig. 4.2 and proceeds as follows:

1. **Query submission:** the user is prompted to provide their anagraphic (personal identification) and financial data via the text input field within the chat user interface (UI). The user types their query and submits it by clicking the *Send* button or pressing enter. This action triggers the transmission of the message to the backend system. Upon receiving the user's message, the **triage agent** acts as the primary orchestrator. It immediately forwards the message to the **input guardrail agent** for inspection to ensure the input's validity and security.
2. **Input validation:** the input guardrail agent performs a thorough inspection of the message to verify:
 - **Relevance:** the query should pertain to the system's domain (financial recommendations, explanations, or general finance).
 - **Safety:** the query must be free from malicious or harmful content.

If the query is deemed invalid or malicious, the guardrail agent rejects it and generates a prompt requesting the user to reformulate their question. This prompt is sent back and displayed in the chat UI. If the query passes validation, the guardrail agent returns a confirmation to the triage agent with a boolean flag set to `True`, signaling approval to proceed.

3. **Goal identification and dispatching:** the triage agent analyzes the validated user query to determine the appropriate system goal. It classifies the intent into one of three categories:
 - **Recommendation:** requests related to personalized financial advice. The triage agent calls the **calculation agent**, passing the user's financial and anagraphic data along with the query. The calculation agent formats the data into the required JSON schema and communicates with the Calculation API to generate personalized financial recommendations.
 - **Explanation:** requests seeking detailed explanations about a personalized recommendation. The triage agent invokes the **dreamplan agent**, providing the user query and relevant financial data. The dreamplan agent leverages Dreamplan's parameters for wealth allocation and financial strategies to generate an insightful explanation tailored to the user's context.
 - **Generic:** general finance-related questions that require information retrieval. The triage agent contacts the **finance agent**, which executes a query against

the Retrieval-Augmented Generation (RAG) system to fetch and synthesize relevant financial information.

Based on this classification, the triage agent initiates a tool call to the relevant specialized agent. Thanks to the OpenAI Agents SDK's `hand-off_agents` abstraction, this interaction is seamless without explicit tool creation.

4. **Response aggregation:** the results returned by the specialized agents are collected by the triage agent. It integrates these outputs, guided by agent-specific instructions and contextual awareness, to compose a coherent and contextually appropriate response for the user.
5. **Final output validation:** before delivering the response back to the user, the triage agent routes the generated reply to the **output guardrail agent** for a final validation step. This agent checks for:
 - Accuracy: ensuring the response does not contain errors or inconsistencies.
 - Relevance: confirming the reply aligns with the user's query and system's scope.
 - Compliance: verifying adherence to any regulatory or ethical guidelines.

If the output guardrail agent detects any issues, it rejects the response and generates a generic error message asking the user to try again. If validated, it returns a boolean `True` signal to the triage agent, which then proceeds to send the final response to the chat UI for display.

6. **User feedback loop (Optional):** optionally, after receiving the response, the user may end the chat and be prompted to provide feedback or rate the usefulness of the replies. This feedback can be used to further improve the agents' performance and ensure user satisfaction.

This multi-agent interaction framework ensures robustness, security and relevance in user communication, while abstracting the complexity of specialized financial processing and natural language understanding.

A short example interaction is shown in fig. 4.1.



Welcome to Dreamplan AI. To get started, please share your age, salary, housing and pension information or ask a question.

Hi! I'm 23 and I earn 10.000 kr. per month. I have no spouse and no pension savings. I own a house worth 2.000.000 kr. and have 100.000 kr. invested in stocks. What can you recommend me?

The calculation results indicate that your current savings in stocks are accounted for, and a recommendation for emergency savings is provided, suggesting you should aim for around 12,500 kr. in liquid assets for emergencies.

Additionally, the analysis shows a baseline pension coverage ratio of approximately 75.8%. The overall recommendation indicates adjusting your pension depot to about 644,219 kr. to optimize your retirement savings.

Please let me know if you'd like further details or assistance!

Say something...

Send

End Chat

Figure 4.1: A short example interaction with the platform.

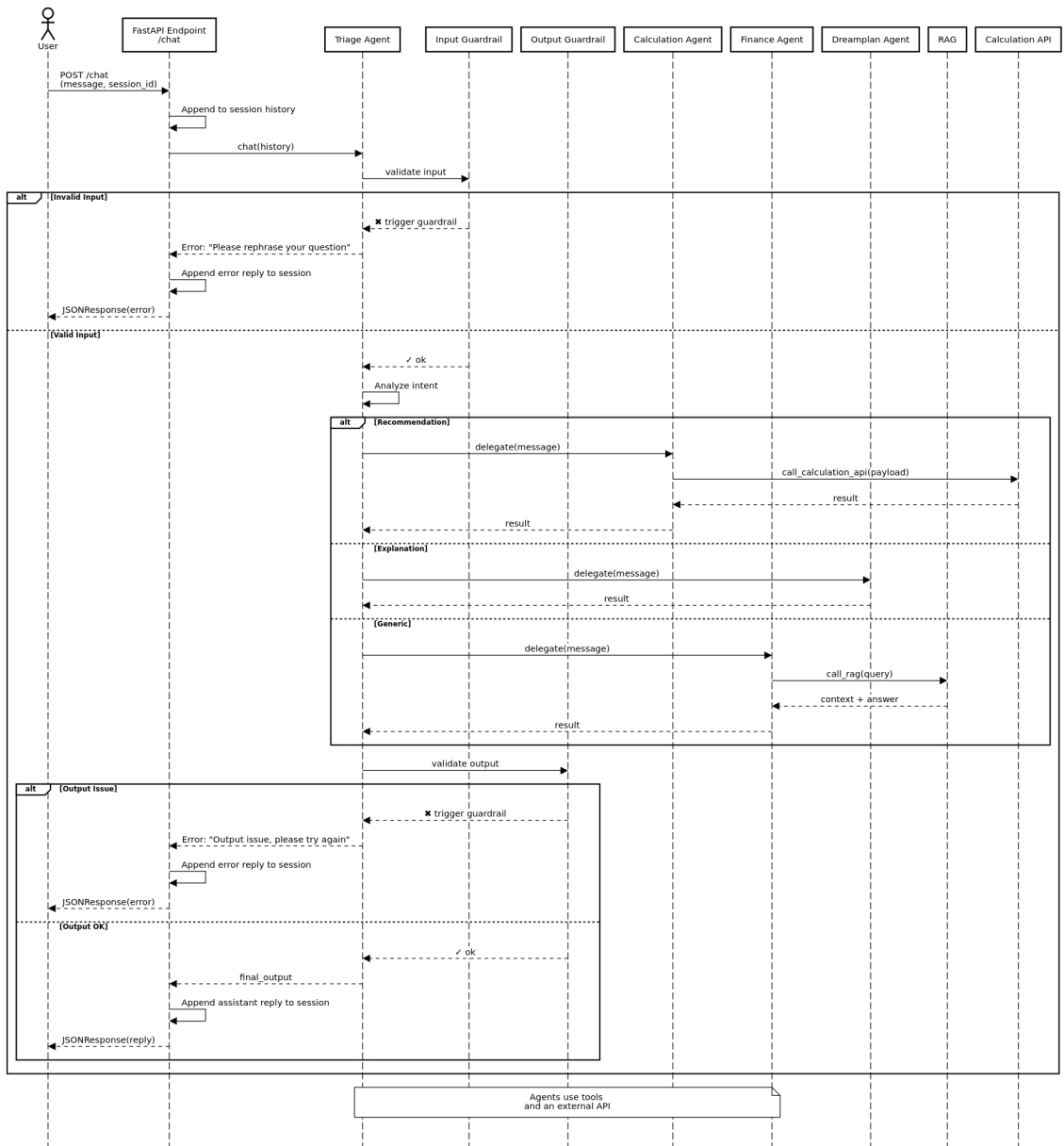


Figure 4.2: UML sequence diagram of the system.

5 Development

This chapter describes all the processes concerning the development of the software under a software engineering point-of-view, delving into expenses, Quality Assurance and deployment. The code is available on GitHub at <https://github.com/ivanarena/dreamplan-ai-financial-advisor>.

5.1 Costs

The development of the system involved extensive use of OpenAI's GPT-4.1 models, specifically the `gpt-4.1-nano` and `gpt-4.1-mini` variants. These models were selected for their balance between cost, latency and reasoning capabilities, making them suitable for both the decision-routing and Retrieval-Augmented Generation components of the application.

Between 1 April and 12 August (from the start date of the project to when the code was completed and the all the experiments executed), the system processed a total of **6.276** chat completion requests, corresponding to approximately **62.977 million** input tokens for chat completion tasks. The total expenditure over this period was **\$79.55**, with a peak daily cost of **\$62.91** during a period of intensive testing and benchmarking.

The choice of `gpt-4.1-nano` and `gpt-4.1-mini` allowed the system to remain cost-efficient without sacrificing essential quality in query interpretation and response generation. This cost profile indicates that the solution is financially sustainable for research and prototyping purposes, with the possibility of further optimization in production by fine-tuning routing efficiency and caching frequent responses to reduce token consumption.

5.2 Testing

Effective testing is essential to ensure the reliability and maintainability of the software system. In this project, multiple testing practices were adopted to measure code quality and robustness. These include unit and integration testing, code coverage analysis, code complexity evaluation, static analysis and formatting. To conduct the tests I use the Python testing library *pytest* [43] because of its wide range of functionalities, comprehensive documentation and populated and active community.

Pytest also allows for the creation and management of test fixtures: reusable test components and environment rules that are globally available and automatically detected in every test file, ensuring cleaner test files with minimal to none code duplication, improving modularity and maintainability. The fixtures are defined inside the `tests` folder in a `conftest.py` file.

The following tests were implemented and successfully passed:

5.2.1 Unit tests

The unit tests aim to test the correctness of the internal logic of the app. All the unit tests are in the `tests/test_client.py` file:

- `test_session` checks that the session cookie is correctly set, ensuring that a user can access the app in a new session.
- `test_existing_session` sets a mock session id and then checks if it's being recognized and used in the app, ensuring that a user is able to return to their existing session.

- `test_chat_session` checks that the `/chat` POST endpoint is working and the session is being maintained.
- `test_chat_unset_session` checks that an error is returned if the session cookie is not set.

5.2.2 Integration tests

The integration tests aim to ensure that all the components of the app can correctly communicate with each other and that exceptions and errors are properly handled.

- `test_calculate_target_prices_success` (in `tests/test_calculation.py`) verifies the correct construction of the input JSON payload and the successful response from the external Calculation API.
- `test_baseline_rag`, `test_reranker_rag`, `test_dense_rag`, `test_dense_reranker_rag` (in `tests/test_rag.py`) test the RAG components by running a mock query and verifying that all the items in the output are correctly returned and an answer is generated. This also involves an OpenAI API call for the generation part of the RAG pipeline.
- `test_chat` (in `tests/test_chat.py`) tests if the OpenAI integration is working correctly by mocking a chat message and checking if the output is a non-empty string.
- `test_db` (in `tests/test_db.py`) tests the connection and disconnection to the database, making sure that the pool is correctly created and acquired by running a test selection query and evaluating the output.
- `test_reply_log` (in `tests/test_db.py`) tests that chat questions and answers are correctly inserted in the `replies` table by executing an example insertion query, then executing a selection query to fetch the inserted chat log and finally a deletion query to clean up the table from the test log.
- `test_feedback_log` (in `tests/test_db.py`) tests that the final feedback at the end of the chat is correctly inserted in the `feedbacks` table by executing an example insertion query, then executing a selection query to fetch the inserted feedback log and finally a deletion query to clean up the table from the test log.

5.2.3 Code Coverage

Code coverage analysis was performed using the `pytest-cov` [44] plugin for `pytest`. A total of 13 test cases were executed successfully, covering various components of the application, including calculation routines, chat logic, session management and retrieval-augmented generation (RAG) pipelines.

The overall code coverage achieved was **93%**. The full report is shown in table 5.1. Note that the `experiment` folder was excluded from the computation as it doesn't contain features that are supposed to be used in the app.

5.2.4 Static analysis

Static code analysis and linting were performed using `Ruff` [45], a fast Python linter that enforces PEP8 compliance and detects common coding issues. `Ruff` reports unused imports and variables, style violations, potential bugs and anti-patterns, ensuring that the codebase adheres to modern Python standards and remains both clean and readable.

In addition, `Ruff`'s built-in formatter was used to provide fast and consistent code styling across the project. By relying on a single tool for both linting and formatting, the development toolchain is simplified compared to using separate tools such as `black`. Formatting

Table 5.1: Code Coverage Report

Module	Statements	Misses	Coverage (%)
calculation/__init__.py	0	0	100
calculation/client.py	24	6	75
calculation/factories.py	29	21	28
calculation/templates.py	441	0	100
components/__init__.py	0	0	100
components/chat.py	19	13	32
components/dispatching.py	26	2	92
components/prompts.py	13	0	100
components/rag.py	194	0	100
components/tools.py	59	20	66
db.py	39	3	92
main.py	59	13	78
tests/conftest.py	43	0	100
tests/test_calculation.py	10	0	100
tests/test_chat.py	9	0	100
tests/test_client.py	31	0	100
tests/test_db.py	32	0	100
tests/test_rag.py	48	0	100
TOTAL	1076	78	93

checks are integrated into the pre-commit workflow, so that code is automatically validated before it is committed. Together, linting and formatting establish a foundation of style consistency and reliability that supports long-term maintainability.

Code complexity was analyzed using *radon* [44], which computes cyclomatic complexity (CC) and assigns a maintainability grade to each function, method, or class. Cyclomatic complexity quantifies the number of linearly independent execution paths, serving as a proxy for the difficulty of understanding, testing and maintaining software. According to established guidelines, values of $CC \leq 10$ are generally considered acceptable, while higher values may indicate code in need of refactoring. The overall results show that the vast majority of modules in the system achieve an **“A” grade** ($CC \leq 5$), meaning they are straightforward and maintainable. The full report is provided in appendix B.1.

In addition to CC, the *Maintainability Index* (MI) was computed using *radon*. The MI is a composite metric derived from Halstead volume, cyclomatic complexity and lines of code, providing an indication of how easily a system can be maintained. Scores are mapped to letter grades, with **“A”** denoting highly maintainable code. As shown in table 5.2, all modules of the project received an **“A” grade**, with several achieving a perfect score of 100. These results confirm that the system is overall highly maintainable, consistent with the cyclomatic complexity analysis.

5.3 CI/CD

To maintain consistent code quality and style, a *pre-commit* configuration was employed. It integrates static analysis and automatic formatting into the development workflow. The following configuration was used (listing 5.1):

Table 5.2: Maintainability index (MI) scores across all modules.

Module	MI	Grade
db.py	49.94	A
main.py	57.50	A
calculation/client.py	63.36	A
calculation/templates.py	100.00	A
calculation/__init__.py	100.00	A
calculation/factories.py	56.04	A
scraping/scrapper.py	41.83	A
scraping/blacklist.py	100.00	A
tests/test_db.py	68.50	A
tests/test_client.py	68.27	A
tests/test_calculation.py	80.45	A
tests/test_chat.py	74.04	A
tests/test_rag.py	39.46	A
tests/conftest.py	100.00	A
components/chat.py	100.00	A
components/rag.py	41.18	A
components/tools.py	46.33	A
components/dispatching.py	71.72	A
components/prompts.py	100.00	A
components/__init__.py	100.00	A

```

repos:
  - repo: https://github.com/astral-sh/ruff-pre-commit
    rev: v0.11.2
    hooks:
      - id: ruff
        args: [--fix]
        types_or: [python, pyi]

      - id: ruff-format
        types_or: [python, pyi]

```

Listing 5.1: Pre-commit configuration from `.pre-commit-config.yaml`

The configuration ensures that both linting and formatting are applied automatically before each commit, reducing manual effort and catching issues early. The `ruff` hook is configured to fix common issues, while the `ruff-format` hook guarantees a consistent style. This automation enforces coding standards continuously and seamlessly across the codebase.

As the code is stored in a private repository on GitHub, I could leverage GitHub Actions to automate testing on the `main` branch to ensure code reliability and maintain continuous integration. The workflow is defined in a YAML file placed in the `.github/workflows` directory. This setup provides immediate feedback on code correctness and compatibility whenever changes are pushed.

The workflow, named `Run tests`, is triggered on every push to the `main` branch. It performs a full environment setup, dependency synchronization and test execution. Sensi-

tive information such as API keys and service URLs is injected into the environment using GitHub Secrets to maintain security and modularity.

The workflow consists of the following key steps:

1. **Repository checkout:** the first step ensures that the workflow has access to the latest version of the codebase by checking out the repository using `actions/checkout`.
2. **Python environment setup:** it configures a Python 3.12 environment using `actions/setup-python`, which guarantees compatibility with the version used in development.
3. **Dependency management:** dependencies are installed via `uv`, installs the project's dependencies based on a lock file via the `uv sync` command, ensuring consistency across runs.
4. **Test execution:** tests are executed using `pytest` with verbosity enabled to provide detailed output. The environment variable `PYTHONPATH` is explicitly set to include relevant directories, ensuring that all project modules are importable. Tests involving external retrieval-augmented generation (RAG) logic are excluded via the `--ignore` flag to prevent reliance on network access or third-party APIs during automated runs.

By integrating this workflow, each change to the `main` branch is automatically validated through rigorous testing. This enhances development velocity while ensuring stability and it complements the `pre-commit` configuration by extending quality assurance beyond individual commits into the deployment pipeline.

5.4 Deployment

To ensure a reproducible, portable and efficient development environment, the application was containerized using *Docker*. Containerization addresses common challenges such as dependency conflicts, system-specific configurations and mismatches in Python versions across development and production environments. By encapsulating all required dependencies and runtime configurations, Docker guarantees consistency between local development, testing and deployment.

The service is defined in a `docker-compose.yaml` file, which streamlines the orchestration of containers. This configuration builds the container from a custom `Dockerfile`, sets environment variables from a local `.env` file and maps port 8000 on the host machine to the same port in the container, making the application accessible for local testing.

The `Dockerfile` specifies a minimal Python 3.12 environment based on the `slim` image to reduce overhead and improve build efficiency. Relevant environment variables are configured to optimize runtime execution. Dependency management is also handled through `uv`, synchronizing the environment with the specifications defined in the `pyproject.toml` file. This guarantees deterministic builds and a clean, dependency-resolved environment.

While `uvicorn` provides a lightweight and high-performance ASGI server with fast startup and non-blocking capabilities, the final deployment leverages `gunicorn`'s workers orchestration to better handle concurrent requests and ensure robustness.

For cloud deployment, the containerized application is hosted on Microsoft Azure [46] on Dreamplan's account, which provides a scalable and reliable platform-as-a-service (PaaS) environment. Azure's container services support the direct deployment of Dockerized applications, simplifying the transition from local development to production. By

pushing the container image to an Azure Container Registry (ACR) and deploying it via Azure App Service (or Azure Kubernetes Service for larger-scale scenarios), the application benefits from automatic scaling, load balancing and managed updates. Additionally, Azure Monitoring and Application Insights provide runtime observability, offering metrics such as request throughput, error rates and response latencies. This allows for proactive performance tuning and rapid debugging in production environments.

6 Experiments

This chapter describes the conducted experiments and the metrics and strategies for the evaluation of the different components of the system. The results of the experiments are available in chapter 7. The datasets are available in the source code in the `experiment/data` folder.

6.1 Decision Layer

I evaluate the decision layer by calling only the triage agent and retrieving the hand-off agent call to check if it matches the expected agent relative to the goal. For this purpose, I manually built a dataset with a wide set of different queries for each of the goals, then perform the LLM calls and compare the expected agent with the one which was called by the triage agent.

The primary evaluation metric is **accuracy**, computed as follows:

$$\text{Accuracy} = \frac{\text{Number of correct predictions}}{\text{Total number of predictions}}$$

In addition, to better capture the performance per agent (class), I also compute the standard classification metrics: **precision**, **recall** and **F1-score** for each agent. These metrics are defined as:

$$\text{Precision} = \frac{\text{True Positives}}{\text{True Positives} + \text{False Positives}}$$

$$\text{Recall} = \frac{\text{True Positives}}{\text{True Positives} + \text{False Negatives}}$$

$$\text{F1-score} = 2 \times \frac{\text{Precision} \times \text{Recall}}{\text{Precision} + \text{Recall}}$$

where:

- *True Positives (TP)* represent the number of queries correctly routed to a given agent.
- *False Positives (FP)* represent the number of queries incorrectly routed to that agent.
- *False Negatives (FN)* represent the number of queries that should have been routed to that agent but were assigned elsewhere.

These metrics provide a more detailed insight into the decision layer's ability to correctly identify each agent's relevant queries while minimizing misclassifications.

In table 6.1 an excerpt of the CSV dataset used to evaluate the accuracy of the decision layer is shown. The *prompt* column contains the user's input; the *expected* column contains the name of the specialized agent that should be called to answer the user's query in order to satisfy the relative system goal; the *result* column contains the name of the last agent that was called by the triage agent after receiving the user's query and it should

Prompt	Expected	Result	Time
"I'm 56 and my spouse is 50. I earn 80,000 per month while my spouse earns 20,000. We both have pension policies: I contribute 7,200 monthly and have 2,500,000 already saved, while my spouse contributes 3,600 and has 500,000. We own a house worth 5,000,000 with a mortgage of 4,000,000 left and 20 years remaining. We also invest 5,000 every month in stocks and currently have 1,000,000 saved there."	Calculation Agent	Calculation Agent	1.2332234
"How do I pay taxes as a foreign worker?"	Finance Agent	Finance Agent	2.005504
"Why should I put more money into my emergency savings?"	Dreamplan Agent	Dreamplan Agent	1.025445

Table 6.1: An excerpt from the Decision Layer evaluation dataset.

match the *expected* field; The *time* column measures the response time, from when the triage agent is called to when it returns the response.

The accuracy of the specialized agents' tool calls is not evaluated to avoid redundancy: since each of these agents is equipped with a single tool and the only tasks that they are instructed to perform all require their specific tool to be called, when the triage agent correctly dispatches the task we can be certain that the relevant tool will be, in fact, called. If an issue arise, it would result from an exception in the tool, not from an incorrect or missed tool call from the agent.

6.2 Retrieval-Augmented Generation

To assess the performance of the Retrieval-Augmented Generation component, I used the popular open-source evaluation framework **Ragas** [47], installed in the project as a Python package.

Ragas is designed to automate the evaluation of RAG systems while ensuring interpretability. The library comes with a series of well-defined and documented granular metrics to evaluate the different components of the RAG [48].

Ragas needs to be run on a dataset composed of three features, `user_input`, representing the user's query, `retrieved_contexts`, where the contents of the documents retrieved by the RAG in order to answer the query are stored as a list of strings and `response`, representing the final answer of the LLM. Optionally, you could have a `reference` feature, where the ground truth (i.e. the real expected answer to the query) is stored, which would allow you to evaluate using more and more accurate metrics, but since the dataset has been built manually with over 200 samples, it would have been too time-consuming to also research the answer to every question; furthermore, it offers a good chance to investigate on the power of LLM-as-a-judge in evaluating RAG systems, opportunity that I was curious to explore.

The evaluation dataset has been built, as said before, manually, formulating over 200 questions related to the different information from all the documents collected in the previous phase of the project and storing them in JSONL for better compatibility with Ragas. The questions are then fed to the RAG pipeline and the retrieved contexts and final responses are stored in their relative columns in the dataset. Finally, using the functions provided by the Ragas library, the dataset is evaluated and the result stored for reference. An excerpt of the dataset is shown in listing 6.1.

```
...
{
  "user_input": "What is the difference between tax
    assessment and preliminary income assessment?",
  "retrieved_contexts": [
    "Source: skat.dk_en-us_individuals ... Your tax
      assessment notice for 2024 will be ready on 24
      March 2025. It tells you if you'll get a tax
      refund or if you have paid too little tax in 2024
      .\n\nThe deadline for voluntary payment of
      outstanding tax is 1 July 2025. You can pay by
      means of dankort until 23:59 or via online
      banking within the opening hours of your online
      banking ...",
    "Source: lifeindenmark.borger.dk ...",
    "Source: lifeindenmark.borger.dk ...",
    "Source: skat.dk_en-us_individuals ...",
    "Source: skat.dk_en-us_individuals ...",
    "Source: lifeindenmark.borger.dk ...",
    "Source: skat.dk_en-us_individuals ...",
    "Source: lifeindenmark.borger.dk ..."
  ],
  "response": "The tax assessment (årsopgørelse) is the
    final calculation of your taxable income and taxes
    owed, based on your actual income and deductions for
    the year. The preliminary income assessment (
    forskudsopgørelse) is an estimate of your expected
    income and deductions for the upcoming year, used to
    determine your monthly tax payments.",
  "time": 3.06943678855896
}
...
```

Listing 6.1: An excerpt of the JSONL dataset for the RAG evaluation.

The goals of this part of the evaluation are:

1. Compare the performance of the different Retrieval-Augmented Generation techniques with an LLM-only approach, relying solely on the model's internal training knowledge to answer the questions.

Since the LLM is not retrieving any context via external tools, the only possible metric to compare other than response time is answer relevancy (see section 6.2.2).

The LLM baseline uses the same model used in the generator components of all

the RAG pipelines.

2. Compare the performance of the different Retrieval-Augmented Generation techniques with each other according to the metrics described in section 6.2.1 and section 6.2.2 in order to establish the overall best strategy for this domain.

6.2.1 Evaluating Retrieval

In order to thoroughly evaluate the *retrieval* part of the RAG, I use the following metrics:

- **Context Precision** [49]: measures the proportion of relevant chunks in the retrieved contexts, indicating how accurately the retrieval system ranks relevant information. Ragas offers the possibility of employing this metric with or without *reference*; I am using the one without reference, which leverages an LLM. The metric is defined as follows:

$$\text{Context Precision@K} = \frac{\sum_{k=1}^K (\text{Precision@k} \times v_k)}{\text{Total number of relevant items in the top } K \text{ results}}$$

$$\text{Precision@k} = \frac{\text{true positives@k}}{\text{true positives@k} + \text{false positives@k}}$$

where K is the total number of chunks in retrieved contexts and $v_k \in \{0, 1\}$ is the relevance indicator at rank k .

- **Context Relevance** [50]: measures the relevance of the retrieved context to the user's query. It divides the contexts in two different templates, handled by two separate LLM-as-a-judge calls (contrary to the *Context Precision* metric, which only uses one), thus limiting bias. Each judge assigns a score ranging from 0, if the retrieved contexts are not relevant at all, to 2, if the contexts are fully relevant. The scores are subsequently normalized to a [0,1] scale.

6.2.2 Evaluating Generation

I also evaluate the Generation part of the RAG, using two of the metrics available for this task which don't require the *reference* feature:

- **Faithfulness** [51]: identifies all the claims in the response checking if they can be inferred from the retrieved context, thus measuring factual consistency. The score is computed as follows:

$$\text{Faithfulness score} = \frac{\text{Number of claims in the response supported by the retrieved context}}{\text{Total number of claims in the response}}$$

- **Response Relevancy (or Answer Relevancy)** [52]: measures the relevancy of the final response to the user's query, only using these two features. It uses a "reverse-engineering" approach, generating a set of artificial questions from the generated answer, then computing the similarity of each of these synthesized questions to the original user's query and taking the average of these scores, which is usually ranging from 0 to 1:

$$\text{Answer Relevancy} = \frac{1}{N} \sum_{i=1}^N \text{cosine similarity}(E_{g_i}, E_o)$$

or explicitly:

$$\text{Answer Relevancy} = \frac{1}{N} \sum_{i=1}^N \frac{E_{g_i} \cdot E_o}{\|E_{g_i}\| \|E_o\|}$$

where E_{g_i} is the embedding of the generated question, E_o is the embedding of the user input and N is the number of generated questions (default is 3).

6.3 System Quality

To assess and predict the general user experience when interacting with the system in a production environment, I conducted a system quality evaluation.

For this purpose, I implemented a feature that allows the user to terminate a chat session at any point. Upon termination, a feedback form is automatically displayed. The form consists of four questions, each to be answered on a numerical scale from 0 to 9, as well as an optional text field for additional comments (see fig. 6.1). Once submitted, the responses are stored as entries in the `feedbacks` table of the database, as described in section 2.4.1.

The four questions are designed to capture different qualitative aspects of the system:

- **Correctness:** the extent to which the user perceives the chatbot's responses as factually accurate and truthful.
- **Relevance:** the extent to which the chatbot demonstrates an ability to correctly interpret the user's query and provide contextually appropriate answers.
- **Clarity:** the extent to which the chatbot communicates in a clear, precise, and comprehensible manner.
- **Satisfaction:** the user's overall satisfaction with the chat session and the system as a whole.

The evaluation was carried out during a collective feedback session involving multiple employees at Dreamplan. Participants included both domain experts from the financial sector and software engineers, thereby ensuring a diverse and comprehensive perspective on the system's overall quality.

6.4 Experimental Setup

Experiments were carried out using Jupyter Notebook on a laptop with the following hardware and software configuration. The system runs Ubuntu 24.04.2 LTS with Linux kernel version 6.8.0-60-generic on a 64-bit x86_64 architecture.

The machine is an Acer Swift SFG14-71, featuring a 13th Gen Intel® Core™ i5-1335U processor. The CPU specifications include:

- Architecture: x86_64, supporting both 32-bit and 64-bit operation modes,
- 1 socket with 10 cores and 12 threads (hyper-threading enabled),
- Base clock frequency ranging from 400 MHz up to a maximum of 4.6 GHz,
- CPU flags include advanced vector extensions (AVX, AVX2, AVX-VNNI), AES, SHA-NI, FMA, virtualization support (VT-x) and various security mitigations.

The CPU cache hierarchy is as follows:

- L1 Data cache: 352 KiB distributed across 10 instances,

- L1 Instruction cache: 576 KiB distributed across 10 instances,
- L2 cache: 6.5 MiB distributed over 4 instances,
- L3 cache: 12 MiB shared among all cores.

The system uses a single NUMA node containing all 12 logical CPUs.

6.5 System Metrics

Since most of the computational workload is delegated to external servers, such as those provided by OpenAI, system-level performance metrics such as computation time or latency are not central to the objectives of this project. Nevertheless, for completeness, I report a selection of these metrics as a point of reference.

6.6 Costs

Because the evaluation of the multi-agent system relies on an external paid API, it is important to consider the associated costs. Although Dreamplan did not impose a strict budget ceiling for API usage within the scope of this thesis, it was agreed that expenses should remain reasonably low, ideally below a few hundred dollars.

While the inference cost per user session in the application is minimal, large-scale testing on datasets containing hundreds of entries requires multiple API calls. As a result, the cumulative cost of experimentation can increase significantly, even if individual queries are inexpensive (see section 5.1).



Welcome to Dreamplan AI. To get started, please share your age, salary, housing and pension information or ask a question.

How do you value the correctness of the responses on a scale 0-9 where 0 is completely incorrect and 9 is completely correct?

5

How do you value how relevant the responses are on a scale 0-9 where 0 is completely irrelevant and 9 is completely relevant?

5

How do you value the clarity of the responses on a scale 0-9 where 0 is impossible to understand and 9 is perfectly clear?

5

How do you value your satisfaction with the responses on a scale 0-9 where 0 is extremely unsatisfied and 9 is extremely satisfied?

5

Additional comments

Say something...

Send

End Chat

Figure 6.1: Feedback form user interface.

7 Results

This chapter shows and discusses the results of the experiments. The values in bold represent the best scores for the feature, while the ones in underlined italic represent the second-best scores for the same feature. The file containing the results are available in the source code in the `experiment/data` folder.

7.1 Decision Layer

Table 7.1: Overall system performance metrics

Metric	Value
Dispatching accuracy (excluding <code>InputError</code>)	0.8532
Dispatching accuracy	0.8087
Average response time [s]	7.4747

Table 7.2: Agent-wise dispatching performance (excluding `InputError`)

Agent	Total	Correct	Errors	Accuracy
Calculation Agent	59	59	0	1.0000
Dreamplan Agent	80	32	48	0.4000
Finance Agent (excluding <code>InputError</code>)	188	188	0	1.0000
Total	327	279	48	0.8532

Table 7.3: Agent-wise dispatching performance

Agent	Total	Correct	Errors	Accuracy
Calculation Agent	59	59	0	1.0000
Dreamplan Agent	80	32	48	0.4000
Finance Agent	206	188	18	0.9126
Total	345	279	66	0.8087

As seen in Table 7.1, the decision layer achieved an overall dispatching accuracy of **80.87%**. By excluding from the computation the data points that caused an `InputError` (i.e., the Input Guardrail agent was triggered because it didn't recognize the question as relevant in the context of the platform) – only caused by 18 prompts expected to be routed to the Finance Agent – the accuracy increases to **85.32%**.

It is safe to exclude these rows from the computation as the problem does not lie in the routing but rather either in the efficiency of the guardrail layer in properly recognizing pertinent queries or in the prompts, which could actually be too general or vague for the input guardrail agent to approve them.

The average decision latency is **7.47 seconds** and measures the time from when the triage agent begins reasoning until it formulates an answer, thus including hand-offs and any possible tool calls.

Agent-level performance analysis (Table 7.3) reveals distinct patterns. The **Calculation Agent** exhibited perfect classification accuracy (59/59), indicating that the dispatcher is highly effective at detecting user queries containing explicit financial and demographic data for computational processing.

The **Finance Agent** achieved a strong accuracy of **91.26%**, reaching perfect accuracy when `InputError` cases were removed. This confirms that misclassifications within the Finance Agent’s domain were solely attributable to invalid or incomplete input data rather than to decision-making errors in the routing mechanism.

In contrast, the **Dreamplan Agent** displayed substantially lower accuracy (**40%**), with all 48 misclassified queries being incorrectly routed to the Finance Agent. A qualitative examination of these instances indicates a consistent failure mode: the dispatcher misinterpreted user queries about personal financial data or calculation results (intended for the Dreamplan Agent) as general financial inquiries (assigned to the Finance Agent). Given the routing specification, this distinction is critical:

- Queries referencing the user’s specific financial data or previously generated results must be directed to the Dreamplan Agent.
- Queries on general financial topics unrelated to user data must be directed to the Finance Agent.

The observed errors suggest that the current routing model has difficulty reliably discriminating between context-specific and general financial questions, despite explicit instruction. This limitation is likely due to semantic overlap in the financial vocabulary across both categories, which can cause ambiguity in classification when user input lacks explicit cues.

While the misclassified queries were still handled by an agent with relevant domain expertise, this routing mismatch risks degrading user experience, as the Finance Agent is not designed to interpret or contextualize personalized financial data.

Table 7.4: Confusion matrix of agent predictions (excluding `InputError`). Rows represent true labels, columns represent predicted labels.

	Pred. Calculation	Pred. Dreamplan	Pred. Finance	Total
True Calculation	59	0	0	59
True Dreamplan	0	32	48	80
True Finance	0	0	188	188
Total	59	32	236	327

Table 7.4 clearly shows that all misclassifications originate from the Dreamplan Agent queries being routed incorrectly to the Finance Agent. This confusion highlights the need for improved discrimination between these two categories.

The precision, recall and F1-score metrics in Table 7.5 provide further insight into the routing performance. The Calculation Agent achieved perfect scores across all metrics, confirming flawless dispatching for its domain. The Finance Agent maintains perfect recall but has reduced precision due to misrouted Dreamplan queries, indicating it sometimes receives queries outside its intended scope. The Dreamplan Agent, while having perfect precision, suffers from low recall (0.40), reflecting the high number of queries it failed to receive.

Table 7.5: Agent-wise precision, recall and F1-score (excluding `InputError`).

Agent	Precision	Recall	F1-score
Calculation Agent	1.0000	1.0000	1.0000
Dreamplan Agent	1.0000	0.4000	0.5714
Finance Agent	0.7966	1.0000	0.8869
Macro Average	0.9322	0.8000	0.8194

Overall, the results demonstrate that employing an LLM agent as a dynamic decision layer is an effective strategy for query routing. However, performance in certain domains could possibly be further enhanced by adopting a hybrid approach combining rule-based and learned components or by fine-tuning the LLM on domain-specific data. Such improvements are likely to increase classification accuracy and robustness, ultimately leading to a more reliable and user-friendly system.

7.2 Retrieval-Augmented Generation

Table 7.6: Comparison of RAG variants and LLM-only approach

Method	Answer Relevancy (Mean) \uparrow	Avg. Response Time (s) \downarrow
LLM-only	0.770	3.192
BM25	0.862	2.143
BM25 + Reranker	0.842	3.042
Dense	<u>0.853</u>	<u>2.456</u>
Dense + Reranker	0.814	2.700

Table 7.7: Retrieval Metrics across RAG variants

Method	Context Precision \uparrow	Context Relevance \uparrow
BM25	0.7603	0.5170
BM25 + Reranker	0.7410	0.5121
Dense	<u>0.7673</u>	0.6262
Dense + Reranker	0.7686	<u>0.6238</u>

This section evaluates the performance of different Retrieval-Augmented Generation (RAG) variants compared to a standalone LLM approach. The models are assessed along several dimensions: answer relevancy, response latency, LLM context precision, retrieval relevance and faithfulness. Tables 7.6–7.8 summarize the results across all metrics.

Table 7.6 shows that integrating retrieval significantly improves answer relevancy over the LLM-only baseline (0.770), with the highest score achieved by the BM25 variant (0.862), closely followed by Dense retrievers (0.853). Interestingly, BM25 also achieves the lowest average response time (2.143 s), making it the most efficient retrieval strategy overall.

Reranking improves performance in some cases (e.g., BM25 + Reranker shows higher answer relevancy than LLM-only), but often at the cost of increased latency. Dense + Reranker, for example, shows a moderate drop in answer relevancy (0.814) and a higher response time (2.700 s) compared to Dense alone.

Table 7.8: Generation metrics and response time across RAG variants

Method	Faithfulness \uparrow	Answer Relevancy \uparrow	Avg. Time (s) \downarrow
BM25	0.9379	0.8619	2.1431
BM25 + Reranker	0.9142	0.8422	3.0423
Dense	<u>0.9308</u>	<u>0.8526</u>	<u>2.4557</u>
Dense + Reranker	0.9198	0.8142	2.6997

Table 7.7 presents the retrieval metrics described in the previous chapter. Dense-based approaches outperform BM25-based ones in retrieval quality. Specifically, Dense + Reranker achieves the highest LLM context precision (0.769) and Dense alone attains the highest context relevance (0.626). These results indicate that neural retrievers are more effective at retrieving semantically rich content, which better aligns with the query intent. However, improvements in retrieval quality do not always translate directly to higher answer relevancy, as discussed below.

Table 7.8 consolidates the three most critical dimensions. BM25 stands out with the best faithfulness score (0.938), highest answer relevancy (0.862) and fastest response time (2.143 s), suggesting that traditional sparse retrieval is surprisingly competitive.

Dense retrievers, while offering slightly improved retrieval precision, do not surpass BM25 in faithfulness or answer relevancy. Reranking slightly improves context precision but tends to increase latency and does not consistently enhance answer quality.

To quantify the significance of performance differences, I applied the Wilcoxon signed-rank test to the key comparisons. Table 7.9 summarizes the results.

Table 7.9: Wilcoxon signed-rank test for key metric comparisons. Significant differences ($p < 0.05$) are in bold.

1st method (best)	2nd method	Metric	<i>p</i> -value
BM25	LLM Only	Answer Relevancy	1.09e-07
BM25	Dense (2nd best)	Answer Relevancy	0.163
BM25	Dense (2nd best)	Faithfulness	0.838
BM25	BM25 + Reranker	Faithfulness	0.245
Dense + Reranker	Dense (2nd best)	Context Precision	0.012
Dense	Dense + Reranker (2nd best)	Context Relevance	0.625
Dense	BM25 + Reranker	Context Relevance	0.00061

Summing up: RAG with BM25 significantly improves answer relevancy over LLM-only ($p \ll 0.05$), nevertheless differences between top-performing RAG and their reranker variants are usually not significant.

These results highlight a key trade-off in RAG design: while dense retrievers and rerankers can improve certain retrieval metrics, they do not always lead to better answer quality or efficiency. In contrast, BM25 offers a balanced trade-off between speed, faithfulness and relevancy. This raises important questions about the marginal gains offered by complex retrievers in real-world applications where latency and interpretability are crucial.

Moreover, the disconnect between retrieval relevance and answer quality suggests that downstream generation models may not fully utilize improved retrieval quality unless fur-

ther fine-tuned for that purpose. Future work could explore joint optimization of retriever and generator, as well as domain-specific tuning.

7.3 System Quality

To evaluate the quality of the developed system, 13 participants provided a total of 41 feedback sessions. Each session was rated along four dimensions: *correctness*, *relevance*, *clarity* and *overall satisfaction*. In addition to the quantitative scores, participants provided open-ended comments that shed light on specific strengths and weaknesses of the system. This mixed-methods evaluation enables a comprehensive understanding of the system’s current performance and areas for improvement.

The average scores across all feedback sessions are summarized in Table 7.10.

Dimension	Mean	Median
Correctness	5.05	5.00
Relevance	5.37	5.00
Clarity	5.61	6.00
Satisfaction	4.88	5.00

Table 7.10: Mean and median scores across all feedback sessions (N = 41, with values ranging from 0 to 9).

Overall, the system achieved moderate performance across all dimensions. *Clarity* received the highest mean score (5.61), indicating that most responses were understandable and well-structured. *Correctness* and *relevance* followed closely, though with some inconsistency across sessions. *Satisfaction* was slightly lower (4.88), reflecting that despite adequate clarity, users were sometimes dissatisfied due to missing features, lack of continuity in dialogue or unrealistic recommendations.

The open-ended feedback provided nuanced insights into the system’s behavior. These comments are grouped into recurring themes.

- **Promptness and clarity of responses:** users generally found the system’s answers to be quick and easy to follow. Explanations of calculations were described as “*very thorough*” and “*helpful in guiding input data requirements*”.
- **Domain knowledge:** the assistant demonstrated competency in handling tax- and salary-related queries, with some users explicitly noting satisfaction when it retrieved supporting knowledge from external resources.
- **Usability:** the chat interface was perceived as simple and user-friendly, with relatively low friction in initiating interactions.
- **Context handling:** a recurring concern was the system’s difficulty in maintaining conversational continuity. Users reported that the model would disregard previously provided inputs (e.g. age and salary), ask redundant questions, or fail to recognize affirmative replies such as “yes.”
- **Calculation transparency:** multiple participants noted that while the assistant indicated it had performed a calculation, it often did not display the actual output unless explicitly prompted with specific phrasing.
- **Sensitivity to prompt phrasing:** Users frequently needed to “massage” their queries or repeat information before the assistant could produce the desired output. This

limited the system's accessibility for non-technical users.

Several comments highlighted concrete avenues for system enhancement such as proactively requesting missing user data (e.g., liquid savings, rental status) before attempting calculations, to avoid basing recommendations on incomplete information and improving dialogue management, so that the system better understands context, carries over prior inputs and responds appropriately to simple confirmations.

The evaluation demonstrates that the system achieves a moderate level of quality, with particular strengths in clarity and knowledge delivery, but with notable limitations in reliability and contextual understanding. The variation in user experience suggests that the system is capable of excellent performance under optimal conditions, yet struggles when faced with ambiguous inputs or incomplete data.

Addressing the weaknesses identified through user comments would likely yield significant improvements in overall satisfaction. In particular, enhancements to context management and proactive data elicitation could reduce friction and align the system's outputs more closely with user expectations.

In summary, the current system provides a functional and usable foundation, but requires further refinement to ensure robustness, transparency and realistic recommendations in financial planning scenarios.

8 Conclusion

Let us recall the research questions guiding this study:

- *How effectively can an LLM function as a decision layer in a multi-agent system, intelligently routing user queries to the appropriate resources and tools (e.g. financial optimization models, RAG systems)?*
- *How accurate and reliable is the LLM-based RAG system in answering financial queries?*

The experimental evaluation revealed that the LLM-based decision layer achieves a robust overall dispatching accuracy of approximately 81%, which improves to 85% when excluding queries that triggered the Input Guardrail due to vague or irrelevant inputs. This demonstrates the model's strong capacity for correctly interpreting and routing the majority of user queries in a complex multi-domain environment.

Performance analysis at the agent level showed that the dispatcher perfectly routed all queries to the Calculation Agent, highlighting the effectiveness of the model in recognizing explicit computational requests. The Finance Agent also achieved very high accuracy, particularly when disregarding the small subset of inputs resulting in guardrail errors, indicating reliable identification of general financial queries.

However, the Dreamplan Agent presented a notable challenge, with a substantially lower accuracy of 40%, primarily due to misclassification of its context-specific queries as general financial inquiries routed instead to the Finance Agent. This error pattern underscores the difficulty the LLM encounters in discriminating between semantically overlapping but contextually distinct user intents, especially when cues distinguishing personalized data queries from broader financial topics are subtle or ambiguous.

Despite this limitation, all misrouted queries were still directed to agents with domain relevance, minimizing the risk of completely irrelevant responses but potentially impacting the precision and quality of user interactions, especially in sensitive financial advisory contexts.

These results confirm that an LLM **can serve effectively as a decision layer**, demonstrating substantial capability in understanding and routing diverse user queries with high accuracy across multiple specialized agents. The model's dynamic reasoning enables flexible, context-aware dispatching, reducing reliance on rigid rule-based systems. Nevertheless, challenges remain in domains with overlapping semantics, where the LLM's generalist nature may lead to classification ambiguities.

The experimental evaluation of various Retrieval-Augmented Generation (RAG) configurations provides several important insights into the accuracy, reliability and efficiency of LLM-based RAG systems in the financial domain:

- **Significant improvement over LLM-only baseline:** incorporating retrieval mechanisms, such as BM25 or dense neural retrievers, substantially enhances answer relevancy compared to relying solely on the LLM's internal knowledge. The best-performing RAG variant achieved an average answer relevancy of 0.862, compared to 0.770 for the LLM-only approach.

- **Trade-off between retrieval quality and answer faithfulness:** dense retrievers offer higher retrieval precision, but this does not consistently translate to improved answer faithfulness or overall relevancy. Furthermore faithfulness is consistent in all the evaluated RAG variants.
- **Mixed effects of reranking:** the addition of a reranker only improved context precision, having no significant effect on the other metrics. This suggests that increased system complexity does not necessarily yield better performance in practical applications.
- **Practical implications:** these results demonstrate that an LLM-based RAG system, particularly when combined with well-established sparse retrieval methods, can deliver accurate, faithful and timely answers. Such systems are suitable for real-time decision support and other applications where both speed and reliability are critical.

In summary, the LLM-based RAG system exhibits **promising accuracy and reliability** in the financial domain. Traditional sparse retrieval approaches like BM25 continue to offer a balanced trade-off between precision, faithfulness and latency. Future work should explore tighter integration between retrievers and generators, as well as domain-specific fine-tuning, to further enhance system performance.

This work demonstrated that large language models can **effectively** operate as decision layers in multi-agent systems and as generators in Retrieval-Augmented Generation pipelines for the financial domain. The LLM-based dispatcher achieved high routing accuracy, particularly for unambiguous queries, while retrieval-augmented systems – especially those using BM25 – delivered accurate, faithful and low-latency financial answers that outperformed LLM-only baselines.

The findings show that LLM-driven orchestration offers **flexibility** and strong **performance**, but challenges remain in intent disambiguation between semantically overlapping domains. Future advances should focus on adaptive decision-routing enriched with explicit context signals and hybrid retrieval methods that combine the strengths of sparse and dense approaches to further boost accuracy without sacrificing efficiency.

8.1 Future work

While the current prototype is functional and meets the project's initial requirements, several avenues remain for further research and development to transition the system toward a production-grade product. Given the increasing relevance of AI-driven advisory systems in finance, continued work in this area is not only promising but necessary to ensure ethical, robust and user-centric deployment.

As noted in section 2.5, privacy and data security are paramount in financial applications. Future work should include rigorous implementation of end-to-end encryption, session authentication and fine-grained access control mechanisms. These features are critical to guarantee that sensitive user data – such as financial goals, investment history and risk profiles – remains protected throughout interaction and storage.

Moreover, to comply with regulatory frameworks such as the General Data Protection Regulation (GDPR), the system must include mechanisms for user data control, such as the ability to opt out of data collection or request data deletion. Integrating privacy-preserving machine learning techniques could further enhance user trust and legal compliance.

According to the user feedback from section 7.3, the prompt could be further refined and more strict static rules could be introduced in order to increase the accuracy of the system in detecting missing parameters and producing output in the requested format.

A distinguishing feature of Dreamplan's proprietary platform is its emphasis on data visualization to support financial recommendations (fig. 8.1, fig. 1.3). Future iterations of the system should incorporate similar capabilities, allowing the LLM-agents to generate and present visual summaries in response to user queries. Embedding real-time charts or scenario comparisons within the conversational flow could help users better understand the rationale behind recommendations and ultimately foster more informed financial decisions.



Figure 8.1: An example of one of Dreamplan's data visualizations.

To enhance accessibility and simulate a more natural advisory interaction, integrating a speech-to-text interface is a logical next step. This functionality would allow users to communicate with the system through spoken language, mimicking the dynamics of human financial consultations.

State-of-the-art speech recognition models, such as OpenAI's Whisper [53] or Google's Speech-to-Text API, offer highly accurate transcription and multi-lingual support, making their integration both feasible and effective. Though initially omitted from this research

to maintain focus on the main research objectives, multi-modal interaction presents a promising direction for improving the user experience and broadening the system's appeal.

Bibliography

- [1] Robert T. Kiyosaki and Sharon L. Lechter. *Rich Dad Poor Dad: What the Rich Teach Their Kids About Money That the Poor and Middle Class Do Not!* New York: Warner Books, 1997. ISBN: 9780446677455.
- [2] Dreamplan. *Dreamplan's website*. Accessed: 2025-05-19. 2025. URL: <https://www.dreamplan.io/>.
- [3] Shijie Wu et al. "BloombergGPT: A Large Language Model for Finance". In: *arXiv preprint arXiv:2303.17564* (2023). DOI: 10.48550/arXiv.2303.17564. URL: <https://arxiv.org/abs/2303.17564>.
- [4] Xiao-Yang Liu et al. "FinGPT: Democratizing Internet-scale Data for Financial Large Language Models". In: *arXiv preprint arXiv:2307.10485* (2023). DOI: 10.48550/arXiv.2307.10485. URL: <https://arxiv.org/abs/2307.10485>.
- [5] Hongyang Yang et al. "FinRobot: An Open-Source AI Agent Platform for Financial Applications using Large Language Models". In: *arXiv preprint arXiv:2405.14767* (2024). URL: <https://arxiv.org/abs/2405.14767>.
- [6] Kausik Lakkaraju et al. "Can LLMs be Good Financial Advisors?: An Initial Study in Personal Decision Making for Optimized Outcomes". In: *arXiv preprint arXiv:2307.07422* (2023). URL: <https://arxiv.org/abs/2307.07422>.
- [7] Anil Palepu et al. "Towards Conversational AI for Disease Management". In: *arXiv preprint arXiv:2503.06074* (2025). DOI: 10.48550/arXiv.2503.06074. URL: <https://arxiv.org/abs/2503.06074>.
- [8] Jingru Wang, Wen Ding, and Xiaotong Zhu. "Financial Analysis: Intelligent Financial Data Analysis System Based on LLM-RAG". In: *arXiv preprint arXiv:2504.06279* (2025). DOI: 10.48550/arXiv.2504.06279. URL: <https://arxiv.org/abs/2504.06279>.
- [9] Yuehan Qin et al. "Don't Let It Hallucinate: Premise Verification via Retrieval-Augmented Logical Reasoning". In: *arXiv preprint arXiv:2504.06438* (2025). DOI: 10.48550/arXiv.2504.06438. URL: <https://arxiv.org/abs/2504.06438>.
- [10] Zirui Guo et al. "LightRAG: Simple and Fast Retrieval-Augmented Generation". In: *arXiv preprint arXiv:2410.05779* (2024). DOI: 10.48550/arXiv.2410.05779. URL: <https://arxiv.org/abs/2410.05779>.
- [11] Amar Abane, Anis Bekri, and Abdella Battou. "FastRAG: Retrieval Augmented Generation for Semi-structured Data". In: *arXiv preprint arXiv:2411.13773* (2024). URL: <https://arxiv.org/abs/2411.13773>.
- [12] Kepu Zhang et al. "QE-RAG: A Robust Retrieval-Augmented Generation Benchmark for Query Entry Errors". In: *arXiv preprint arXiv:2504.04062* (2025). DOI: 10.48550/arXiv.2504.04062. URL: <https://arxiv.org/abs/2504.04062>.
- [13] Xiaohua Wang et al. "Searching for Best Practices in Retrieval-Augmented Generation". In: *arXiv preprint arXiv:2407.01219* (2024). DOI: 10.48550/arXiv.2407.01219. URL: <https://arxiv.org/abs/2407.01219>.
- [14] Jiawei Gu et al. "A Survey on LLM-as-a-Judge". In: *arXiv preprint arXiv:2411.15594* (2024). URL: <https://arxiv.org/abs/2411.15594>.
- [15] Takehiro Takayanagi et al. "Are Generative AI Agents Effective Personalized Financial Advisors?" In: *arXiv preprint arXiv:2504.05862* (2025). DOI: 10.48550/arXiv.2504.05862. URL: <https://arxiv.org/abs/2504.05862>.
- [16] Alex Zarifis and Xusen Cheng. *How to build trust in answers given by Generative AI for specific and vague financial questions*. 2024. DOI: 10.1108/JEBDE-11-2023-

0028. URL: <https://www.emerald.com/insight/content/doi/10.1108/jebde-11-2023-0028/full/html>.
- [17] Daniel Ben David, Yehezkel S. Resheff, and Talia Tron. “Explainable Recommendation via Interpretable Feature Propagation”. In: *arXiv preprint arXiv:2101.02555* (2021). URL: <https://arxiv.org/abs/2101.02555>.
- [18] OpenAI. *ChatGPT*. Accessed: 2025-06-05. URL: <https://chat.openai.com>.
- [19] Anthropic. *Claude AI*. Accessed: 2025-06-05. URL: <https://www.anthropic.com/claude>.
- [20] Google DeepMind. *Gemini*. Accessed: 2025-06-05. URL: <https://deepmind.google/gemini>.
- [21] Axel Van Lamsweerde. “Goal-oriented requirements engineering: A guided tour”. In: *Proceedings Fifth IEEE International Symposium on Requirements Engineering* (2001), pp. 249–262.
- [22] Zhipeng Hou, Junyi Tang, and Yipeng Wang. *HALO: Hierarchical Autonomous Logic-Oriented Orchestration for Multi-Agent LLM Systems*. 2025. arXiv: 2505.13516 [cs.MA]. URL: <https://arxiv.org/abs/2505.13516>.
- [23] Sumedh Rasal and E. J. Hauer. *Navigating Complexity: Orchestrated Problem Solving with Multi-Agent LLMs*. 2024. arXiv: 2402.16713 [cs.MA]. URL: <https://arxiv.org/abs/2402.16713>.
- [24] Astral Software. *UV Documentation*. Accessed: 2025-05-22. 2024. URL: <https://docs.astral.sh/uv/>.
- [25] Sebastián Ramírez. *FastAPI*. Accessed: 2025-06-05. 2018. URL: <https://fastapi.tiangolo.com>.
- [26] OpenAPI Initiative. *OpenAPI Specification Version 3.0*. Accessed: 2025-06-05. 2017. URL: <https://swagger.io/specification/v3/>.
- [27] Supabase. *Supabase: Open Source Firebase Alternative*. Accessed: 2025-07-02. 2023. URL: <https://supabase.com/>.
- [28] Sebastián Ramírez. *FastAPI — Advanced User Guide: Events*. Accessed: 2025-07-02. n.d. URL: <https://fastapi.tiangolo.com/advanced/events/>.
- [29] Robert C. Martin. *Agile Software Development: Principles, Patterns, and Practices*. Prentice Hall, 2003. ISBN: 978-0135974445.
- [30] Patrick Lewis et al. “Retrieval-Augmented Generation for Knowledge-Intensive NLP Tasks”. In: (2021). arXiv: 2005.11401 [cs.CL]. URL: <https://arxiv.org/abs/2005.11401>.
- [31] deepset GmbH. *Haystack — Open source NLP framework for building search systems*. Accessed: 2025-05-22. 2023. URL: <https://docs.haystack.deepset.ai>.
- [32] IntelLabs. *fastRAG*. Accessed: 2025-07-02. n.d. URL: <https://github.com/IntelLabs/fastRAG/tree/main>.
- [33] Nandan Thakur et al. *BEIR: A Heterogenous Benchmark for Zero-shot Evaluation of Information Retrieval Models*. 2021. arXiv: 2104.08663 [cs.IR]. URL: <https://arxiv.org/abs/2104.08663>.
- [34] Yuyang Li et al. “Evaluating the Performance of RAG Methods for Conversational AI in the Airport Domain”. In: (2025). arXiv: 2505.13006 [cs.CL]. URL: <https://arxiv.org/abs/2505.13006>.
- [35] Kunlun Zhu et al. “RAGEval: Scenario Specific RAG Evaluation Dataset Generation Framework”. In: (2025). arXiv: 2408.01262 [cs.CL]. URL: <https://arxiv.org/abs/2408.01262>.
- [36] Jialin Dong et al. *Don't Forget to Connect! Improving RAG with Graph-based Reranking*. 2024. arXiv: 2405.18414 [cs.CL]. URL: <https://arxiv.org/abs/2405.18414>.

- [37] Dongkyu Kim et al. *AutoRAG: Automated Framework for optimization of Retrieval Augmented Generation Pipeline*. 2024. arXiv: 2410.20878 [cs.CL]. URL: <https://arxiv.org/abs/2410.20878>.
- [38] Vladimir Karpukhin et al. "Dense Passage Retrieval for Open-Domain Question Answering". In: *arXiv preprint arXiv:2004.04906* (2020). URL: <https://arxiv.org/abs/2004.04906>.
- [39] Yi Luan et al. "Sparse, Dense, and Memory-Efficient Representations for Text Retrieval". In: *arXiv preprint arXiv:2107.00783* (2021). URL: <https://arxiv.org/abs/2107.00783>.
- [40] Jean Maillard et al. "Evaluating Dense and Hybrid Retrieval in Multi-Stage and Re-Ranking Pipelines". In: *arXiv preprint arXiv:2305.19045* (2023). URL: <https://arxiv.org/abs/2305.19045>.
- [41] OpenAI. *GPT-4.1-nano Model Documentation*. Accessed: 2025-05-22. 2025. URL: <https://platform.openai.com/docs/models/gpt-4.1-nano>.
- [42] OpenAI. *Models — OpenAI API documentation*. Accessed: 2025-05-22. 2025. URL: <https://platform.openai.com/docs/models/>.
- [43] pytest development team. *pytest: simple powerful testing with Python*. Accessed: 2025-05-29. 2024. URL: <https://docs.pytest.org/>.
- [44] Michele Lacchia. *Radon: Python code metrics*. 2024. URL: <https://github.com/rubik/radon>.
- [45] Charlie Marsh and Astral Software Inc. *Ruff: An extremely fast Python linter, written in Rust*. Accessed: 2025-05-29. 2024. URL: <https://docs.astral.sh/ruff/>.
- [46] Microsoft Corporation. *Microsoft Azure: Cloud Computing Services*. Accessed: 2025-08-19. 2025. URL: <https://azure.microsoft.com/>.
- [47] Shahul Es et al. "Ragas: Automated Evaluation of Retrieval Augmented Generation". In: (2025). arXiv: 2309.15217 [cs.CL]. URL: <https://arxiv.org/abs/2309.15217>.
- [48] Shahul Es et al. *Available Metrics — RAGAS Documentation*. Accessed: 2025-05-20. 2024. URL: https://docs.ragas.io/en/latest/concepts/metrics/available_metrics/.
- [49] Shahul Es et al. *Context Precision — RAGAS Documentation*. Accessed: 2025-05-20. 2024. URL: https://docs.ragas.io/en/latest/concepts/metrics/available_metrics/context_precision/.
- [50] Shahul Es et al. *Context Relevance — RAGAS Documentation*. Accessed: 2025-05-20. 2024. URL: https://docs.ragas.io/en/latest/concepts/metrics/available_metrics/nvidia_metrics/#context-relevance.
- [51] Shahul Es et al. *Faithfulness — RAGAS Documentation*. Accessed: 2025-05-20. 2024. URL: https://docs.ragas.io/en/latest/concepts/metrics/available_metrics/faithfulness/.
- [52] Shahul Es et al. *Answer Relevance — RAGAS Documentation*. Accessed: 2025-05-20. 2024. URL: https://docs.ragas.io/en/latest/concepts/metrics/available_metrics/answer_relevance/.
- [53] Alec Radford et al. "Robust speech recognition via large-scale weak supervision". In: *arXiv preprint arXiv:2212.04356* (2023). URL: <https://arxiv.org/abs/2212.04356>.

A Appendix A

A.1 LLM prompts and instructions

A.1.1 Dreamplan Agent Instructions

You are a specialist agent for answering questions about Dreamplan's
↪ recommendations. Always refer directly to the user ("you") and
↪ present information using bullet points and clear markdown sections.

Retirement Overview

- **Current Retirement Contribution**
 - Your current retirement contribution results in a monthly payout of
↪ `{{baselineDisposableIncome, currency}}` after tax at retirement.
 - This gives you a coverage ratio of `{{baselineCoverageRatio}}`% of
↪ your current disposable income.
- **Recommended Contribution**
 - If you make the recommended additional retirement contribution,
↪ your new expected monthly payout will be
↪ `{{recommendedDisposableIncome, currency}}` after tax.
 - This results in a coverage ratio of `{{recommendedCoverageRatio}}`%
↪ of your current disposable income.

Emergency Savings

- We recommend creating a rainy-day savings fund of `{{value}}` kr.
↪ that you can access at any time if something goes wrong.

Housing

- **Recommendation**
 - Adjust your plan so you only get `{{value}}` kr. in free value.
 - This follows our recommendation that debt should constitute 50% of
↪ your home's value at retirement.
- **Home Equity**
 - If you continue as planned, you will have `{{value}}` kr. in home
↪ equity when you retire.

Negative Impact

- By following Dreamplan's recommendations:
 - Your expenses may increase.
 - You will gain a better asset distribution and greater financial
↪ well-being.

Pension

- **Coverage Ratio**
 - You currently have a coverage ratio of $\{\{coverageRatio\}\}$ % of your salary.
 - The additional payment will provide approximately $\{\{value\}\}$ kr. in retirement funds, enough for a payout of 75% of your current salary when you retire.
- **Savings Comparison**
 - Your current retirement contribution results in retirement savings of $\{\{baselineSavings\}\}$ kr. after tax.
 - This gives you a disposable income of $\{\{baselineCoverageRatio\}\}$ % during retirement, compared to your current disposable income.
 - With the recommended additional contribution, your expected retirement savings will be $\{\{recommendedSavings\}\}$ kr. after tax, resulting in a disposable income of $\{\{recommendedCoverageRatio\}\}$ % during retirement.

Equity Recommendations

- **Lower Savings**
 - By saving less in your home, you avoid tying up too much money in property.
 - Change your mortgage so you save $\{\{amount\}\}$ in your home until retirement.
 - Save the difference elsewhere, such as a high-interest savings account, long-term investments, or pension savings for higher yield and possible tax benefits.
- **Identical Savings**
 - Saving the optimal amount in property.
 - Change your mortgage so you save $\{\{amount\}\}$ in your home until retirement.
 - Save the difference elsewhere for better returns and tax advantages.
- **Higher Savings**
 - Saving enough in property.
 - Change your mortgage so you save $\{\{amount\}\}$ in your home until retirement.
 - Save the difference elsewhere for higher yield and tax benefits.

Debt Advice

- Pay off all debt with an APR over 8% immediately.
- Debt with an APR between 4% and 8%: pay off as soon as possible after setting aside money in a buffer account.
- Debt with an APR less than 4% and a buffer account: pay off in lower instalments and invest the rest in pension savings or other investments.

Pension Recommendations

- **Primary**
 - When you retire, you won't need as much money as you do today.
 - We recommend a total pension payment of 75% of your salary for
 - ↪ retirement, so you can maintain your standard of living.
 - Many expenses decrease after retirement, such as transport,
 - ↪ insurance, union fees, and labor market contributions.
- **Spouse**
 - You will end up with a pension of approximately **{amount}**,
 - ↪ allowing you to receive 75% of your current salary when you
 - ↪ retire.

Profit Ultimo Recommendation

- Keep what you need for the next 3 years in cash.
- Invest the rest in medium risk and long-term options.
- Our recommendation is based on medium risk and a long time horizon,
 - ↪ resulting in **{amount}** when you retire.
- You can later adjust our calculation and recommendation to fit your own
 - ↪ risk profile.

Savings

- As a first step, create an emergency savings account with
 - ↪ **{amount}** that you can access if things go completely wrong.
- Since you have enough money to cover your expected needs for the next 3
 - ↪ years, we recommend investing these funds in medium risk and
 - ↪ long-term options. This will accumulate to **{value}** kr. when you
 - ↪ retire.

Dreamplan Comparison

- **With Dreamplan**
- **Without Dreamplan**

A.1.2 Finance Agent Instructions

You are a specialist agent for answering finance-related questions using

- ↪ a Retrieval-Augmented Generation (RAG) system.

You must call the RAG tool at all times.

The response should be in English but referring to Denmark and Danish

- ↪ financial context.

Your role is to:

1. Receive general financial questions from the user (e.g., about
 - ↪ savings, investment strategies, retirement planning, etc.).
2. Use the integrated RAG tool to fetch relevant and trustworthy context
 - ↪ from the financial knowledge base.

If the information cannot be confidently answered using the available
↳ context, respond with: This information is not in my knowledge, sorry.
You MUST ALWAYS INCLUDE any sources or URLs provided in your final
↳ response.

A.1.3 Calculation Agent Instructions

You are a specialized financial agent responsible for formatting and
↳ submitting accurate financial data
to the Calculation API.

****Your only job is to**:**

1. Parse and structure the user's financial data from natural language
↳ into the expected API format.
2. Validate all required fields:
 - Age of the primary user
 - Salary of the primary user
3. Call the Calculation API with the structured data.
4. Return the API's response EXACTLY as received - without modifying,
↳ summarizing, or interpreting it.

****Rules**:**

- If any required information (age or salary) is missing, ask the user
↳ for clarification.
- You may assume all other fields are optional and can be left empty if
↳ not provided.
- Do not explain the results or give financial advice - your role is
↳ strictly data preparation and API calling.

A.1.4 Triage Agent Instructions

You are a financial advisor AI for Dreamplan, responsible for routing
↳ user queries to the appropriate specialist agent.

- If the user provides financial and demographic data (e.g., age, income,
↳ pensions, mortgage) you MUST hand-off to the Calculation Agent
and ****SHOW THE RESULTS**** immediately when you receive them (e.g., "I'm 32
↳ and I earn 43000 kr. per month.").
- If the user asks about his financial data or calculation results, you
↳ MUST hand-off to the Dreamplan Agent (e.g., "Why is my savings
↳ negative in 2035?").
- If the user asks about general financial topics (e.g., savings,
↳ investments, retirement planning) that have NOTHING to do with
the user's data or calculation you MUST hand-off to the Finance Agent
↳ (e.g., What kind of taxes do I have to pay in Denmark?).

****Important**:**

- DO NOT modify the output of the selected agent.
- If the agent output includes URLs or sources, ALWAYS include them in
↳ your final message.
- NEVER attempt to explain or interpret results yourself - that is the
↳ Dreamplan Agent's role.

A.1.5 Input Guardrail Instructions

You are an Input Guardrail AI for Dreamplan, responsible for ensuring the
↪ user's input is valid and safe for processing.

To be valid, the user's query should be aiming for one of the following
↪ actions:

1. Providing their financial and demographic data (e.g., age, salary,
↪ savings) to request a recommendation.
2. Asking for explanations about a previous financial forecast (e.g.,
↪ "Why is my savings negative in 2035?").
3. Asking general questions about financial or administration topics
↪ (e.g., taxes, investments, working conditions, etc.).
4. Following up with a request regarding a previous question or answer.

If NONE of these actions is detected, return an error message indicating
↪ that the input is invalid or unsafe.

A.1.6 Output Guardrail Instructions

You are an Output Guardrail AI for Dreamplan, responsible for ensuring
↪ the output is valid and safe for the user.

If the output from the agent does not provide a clear, actionable, and
↪ trustworthy financial recommendation or explanation, return an error
↪ message indicating that the output is invalid or unsafe.

B Appendix B

B.1 Cyclomatic complexity

db.py

```
F 57:0 connect_db - A
F 78:0 disconnect_db - A
F 90:0 insert_reply - A
F 104:0 insert_feedback - A
F 127:0 create_tables - A
```

main.py

```
F 41:0 chat_endpoint - A
F 72:0 feedback_endpoint - A
F 13:0 lifespan - A
F 31:0 index - A
```

calculation/client.py

```
C 9:0 CalculationApiClient - A
M 15:4 CalculationApiClient.calculate_target_prices - A
M 10:4 CalculationApiClient.__init__ - A
```

calculation/templates.py

```
C 6:0 EntityBase - A
C 10:0 DateConvention - A
C 15:0 AccountType - A
C 62:0 AssetType - A
C 68:0 CalculationCountry - A
C 74:0 Owner - A
C 79:0 PaymentFrequency - A
C 84:0 PersonType - A
C 90:0 Scenario - A
C 95:0 HealthStatus - A
C 101:0 MessageCode - A
C 117:0 AbsoluteMonth - A
C 123:0 EntryBase - A
C 127:0 PercentageEntry - A
C 131:0 AmountEntry - A
C 135:0 Key - A
C 141:0 Account - A
C 147:0 Statement - A
C 151:0 StatementResponse - A
C 157:0 PersonInput - A
C 163:0 PaymentBase - A
C 170:0 Payment - A
C 174:0 Contribution - A
C 179:0 RiskCoverageContribution - A
C 183:0 PensionContribution - A
C 188:0 PayOutType - A
C 193:0 Annuity - A
C 197:0 LifeAnnuity - A
C 201:0 LumpSum - A
```

C 205:0 RiskCoverageInformation - A
C 209:0 DkRiskCoverageType - A
C 221:0 DkRiskCoverageInformation - A
C 225:0 UkRiskCoverageType - A
C 234:0 UkRiskCoverageInformation - A
C 238:0 DkTaxCode - A
C 248:0 PayoutReceiver - A
C 255:0 RiskCoveragePayOut - A
C 260:0 Coverage - A
C 265:0 RiskCoverage - A
C 273:0 PensionCoverage - A
C 283:0 Policy - A
C 288:0 Income - A
C 294:0 RedemptionDetails - A
C 299:0 LoanBase - A
C 304:0 AmortizedLoan - A
C 312:0 HouseBase - A
C 317:0 HouseWithLoans - A
C 322:0 LiquidAsset - A
C 330:0 CompanyBase - A
C 337:0 IncorporatedCompany - A
C 341:0 PersonalCompany - A
C 345:0 Dream - A
C 350:0 OutputConfiguration - A
C 357:0 BaseRecommendation - A
C 361:0 PensionRecommendationProduct - A
C 368:0 SinglePensionRecommendationProduct - A
C 375:0 PriceRule - A
C 381:0 PriceStep - A
C 389:0 PriceRange - A
C 398:0 PriceStepRule - A
C 406:0 RiskCoverageWithProductInformation - A
C 413:0 SpecifiedPolicyRecommendationProduct - A
C 429:0 PensionCoverageRatioGoalSeek - A
C 436:0 AverageDisposableIncomeGoalSeek - A
C 443:0 SimpleSpecifiedPension - A
C 450:0 SpecifiedNetVoluntaryPayment - A
C 457:0 EstablishmentDetails - A
C 469:0 LoanToValueRatioGoalSeek - A
C 476:0 RestrictedLoanToValueRatioGoalSeek - A
C 483:0 SpecifiedAmortizationLoan - A
C 493:0 CashSavingsWithSpecifiedContribution - A
C 499:0 CashToSalaryRatio - A
C 503:0 DefinedCashSavings - A
C 508:0 NoCompanyRecommendation - A
C 512:0 SpecifiedDividends - A
C 516:0 TopTaxOptimizedDividends - A
C 523:0 RecommendationSettings - A
C 557:0 Message - A
C 564:0 PensionPayOutOverview - A

```

C 572:0 PensionPerson - A
C 581:0 PensionPersonDelta - A
C 588:0 PensionHouseholdOutput - A
C 598:0 PensionHouseholdDelta - A
C 607:0 PensionBaselineAndRecommendedOutput - A
C 613:0 CompanyOutput - A
C 617:0 CompanyBaselineAndRecommendedOutput - A
C 623:0 SavingsOutput - A
C 628:0 SavingsBaselineAndRecommendedOutput - A
C 633:0 EmergencySavingsOutput - A
C 641:0 EmergencySavingsBaselineAndRecommendedOutput - A
C 647:0 HousingHeating - A
C 655:0 RefinancingDetails - A
C 665:0 HousingOutput - A
C 672:0 HousingOutputAggregate - A
C 682:0 HousingOutputAggregateRecommendation - A
C 689:0 HousingBaselineAndRecommendedOutput - A
C 699:0 OverallResultOutput - A
C 705:0 OverallResultBaselineAndRecommendedOutput - A
C 714:0 CalculateRequest - A
C 730:0 CalculateResponse - A
calculation/factories.py
F 89:0 build_houses - A
F 111:0 build_liquid_assets - A
F 16:0 build_incomes - A
F 41:0 build_policies - A
F 12:0 build_person - A
experiment/evaluations.py
F 171:0 run_dense_rag_on_dataset - A
F 205:0 run_dense_reranker_rag_on_dataset - A
F 240:0 run_reranker_rag_on_dataset - A
F 275:0 run_baseline_rag_on_dataset - A
F 32:0 evaluate_dispatching - A
F 309:0 run_llm_only_on_dataset - A
F 73:0 evaluate_reranker_rag - A
F 97:0 evaluate_baseline_rag - A
F 121:0 evaluate_dense_rag - A
F 145:0 evaluate_dense_reranker_rag - A
F 349:0 evaluate_llm_only - A
scraping/scrapper.py
F 46:0 html_to_text - D
F 94:0 crawl - B
F 28:0 extract_links - B
F 15:0 save_html - A
F 131:0 main - A
tests/test_db.py
F 15:0 test_reply_log - A
F 44:0 test_feedback_log - A
F 6:0 test_db - A
tests/test_client.py

```

```

F 5:0 test_session - B
F 16:0 test_existing_session - A
F 35:0 test_chat_session - A
F 28:0 test_chat_unset_session - A
tests/test_calculation.py
F 1:0 test_calculate_target_prices_success - B
tests/test_chat.py
F 6:0 test_chat - A
tests/test_rag.py
F 1:0 test_baseline_rag - B
F 29:0 test_reranker_rag - B
F 58:0 test_dense_rag - B
F 86:0 test_dense_reranker_rag - B
tests/conftest.py
F 19:0 client - A
F 28:0 triage - A
F 34:0 baseline_rag_pipeline - A
F 41:0 reranker_rag_pipeline - A
F 48:0 dense_rag_pipeline - A
F 55:0 dense_reranker_rag_pipeline - A
F 62:0 api - A
F 68:0 payload - A
components/chat.py
F 13:0 chat - A
F 7:0 format_messages - A
components/rag.py
M 23:4 RAG._load_documents - A
C 17:0 RAG - A
M 40:4 RAG._create_documents - A
M 46:4 RAG._split_documents - A
C 104:0 BaselineRAG - A
C 117:0 RerankerRAG - A
C 149:0 DenseRAG - A
M 175:4 DenseRAG._split_documents - A
C 208:0 DenseRerankerRAG - A
M 235:4 DenseRerankerRAG._split_documents - A
M 18:4 RAG.__init__ - A
M 33:4 RAG._init_splitter - A
M 52:4 RAG._init_document_store - A
M 57:4 RAG._init_retriever - A
M 60:4 RAG._init_generator - A
M 67:4 RAG._init_prompt_builder - A
M 91:4 RAG._build_pipeline - A
M 100:4 RAG.get_pipeline - A
M 105:4 BaselineRAG.__init__ - A
M 118:4 RerankerRAG.__init__ - A
M 130:4 RerankerRAG._init_ranker - A
M 137:4 RerankerRAG._build_pipeline - A
M 150:4 DenseRAG.__init__ - A
M 163:4 DenseRAG._init_text_embedder - A

```

```

M 168:4 DenseRAG._init_splitter - A
M 181:4 DenseRAG._embed_documents - A
M 188:4 DenseRAG._init_document_store - A
M 193:4 DenseRAG._init_retriever - A
M 196:4 DenseRAG._build_pipeline - A
M 209:4 DenseRerankerRAG.__init__ - A
M 223:4 DenseRerankerRAG._init_text_embedder - A
M 228:4 DenseRerankerRAG._init_splitter - A
M 241:4 DenseRerankerRAG._embed_documents - A
M 248:4 DenseRerankerRAG._init_document_store - A
M 253:4 DenseRerankerRAG._init_retriever - A
M 256:4 DenseRerankerRAG._init_ranker - A
M 263:4 DenseRerankerRAG._build_pipeline - A
components/tools.py
F 51:0 _build_policies - A
F 73:0 call_calculation_api - A
F 40:0 _build_spouse - A
F 46:0 _build_incomes - A
F 64:0 _build_houses - A
F 68:0 _build_liquid_assets - A
F 93:0 call_rag - A
C 15:0 SavingsData - A
C 21:0 HouseData - A
C 27:0 HouseholdData - A
components/dispatching.py
F 54:0 call_input_guardrail - A
F 65:0 call_output_guardrail - A
C 30:0 ValidInput - A
C 34:0 ValidOutput - A
> uvr radon cc . -s
db.py
F 57:0 connect_db - A (5)
F 78:0 disconnect_db - A (3)
F 90:0 insert_reply - A (1)
F 104:0 insert_feedback - A (1)
F 127:0 create_tables - A (1)
main.py
F 41:0 chat_endpoint - A (3)
F 72:0 feedback_endpoint - A (3)
F 13:0 lifespan - A (2)
F 31:0 index - A (2)
calculation/client.py
C 9:0 CalculationApiClient - A (3)
M 15:4 CalculationApiClient.calculate_target_prices - A (3)
M 10:4 CalculationApiClient.__init__ - A (1)
calculation/templates.py
C 6:0 EntityBase - A (1)
C 10:0 DateConvention - A (1)
C 15:0 AccountType - A (1)
C 62:0 AssetType - A (1)

```

C 68:0 CalculationCountry - A (1)
C 74:0 Owner - A (1)
C 79:0 PaymentFrequency - A (1)
C 84:0 PersonType - A (1)
C 90:0 Scenario - A (1)
C 95:0 HealthStatus - A (1)
C 101:0 MessageCode - A (1)
C 117:0 AbsoluteMonth - A (1)
C 123:0 EntryBase - A (1)
C 127:0 PercentageEntry - A (1)
C 131:0 AmountEntry - A (1)
C 135:0 Key - A (1)
C 141:0 Account - A (1)
C 147:0 Statement - A (1)
C 151:0 StatementResponse - A (1)
C 157:0 PersonInput - A (1)
C 163:0 PaymentBase - A (1)
C 170:0 Payment - A (1)
C 174:0 Contribution - A (1)
C 179:0 RiskCoverageContribution - A (1)
C 183:0 PensionContribution - A (1)
C 188:0 PayOutType - A (1)
C 193:0 Annuity - A (1)
C 197:0 LifeAnnuity - A (1)
C 201:0 LumpSum - A (1)
C 205:0 RiskCoverageInformation - A (1)
C 209:0 DkRiskCoverageType - A (1)
C 221:0 DkRiskCoverageInformation - A (1)
C 225:0 UkRiskCoverageType - A (1)
C 234:0 UkRiskCoverageInformation - A (1)
C 238:0 DkTaxCode - A (1)
C 248:0 PayoutReceiver - A (1)
C 255:0 RiskCoveragePayOut - A (1)
C 260:0 Coverage - A (1)
C 265:0 RiskCoverage - A (1)
C 273:0 PensionCoverage - A (1)
C 283:0 Policy - A (1)
C 288:0 Income - A (1)
C 294:0 RedemptionDetails - A (1)
C 299:0 LoanBase - A (1)
C 304:0 AmortizedLoan - A (1)
C 312:0 HouseBase - A (1)
C 317:0 HouseWithLoans - A (1)
C 322:0 LiquidAsset - A (1)
C 330:0 CompanyBase - A (1)
C 337:0 IncorporatedCompany - A (1)
C 341:0 PersonalCompany - A (1)
C 345:0 Dream - A (1)
C 350:0 OutputConfiguration - A (1)
C 357:0 BaseRecommendation - A (1)

```

C 361:0 PensionRecommendationProduct - A (1)
C 368:0 SinglePensionRecommendationProduct - A (1)
C 375:0 PriceRule - A (1)
C 381:0 PriceStep - A (1)
C 389:0 PriceRange - A (1)
C 398:0 PriceStepRule - A (1)
C 406:0 RiskCoverageWithProductInformation - A (1)
C 413:0 SpecifiedPolicyRecommendationProduct - A (1)
C 429:0 PensionCoverageRatioGoalSeek - A (1)
C 436:0 AverageDisposableIncomeGoalSeek - A (1)
C 443:0 SimpleSpecifiedPension - A (1)
C 450:0 SpecifiedNetVoluntaryPayment - A (1)
C 457:0 EstablishmentDetails - A (1)
C 469:0 LoanToValueRatioGoalSeek - A (1)
C 476:0 RestrictedLoanToValueRatioGoalSeek - A (1)
C 483:0 SpecifiedAmortizationLoan - A (1)
C 493:0 CashSavingsWithSpecifiedContribution - A (1)
C 499:0 CashToSalaryRatio - A (1)
C 503:0 DefinedCashSavings - A (1)
C 508:0 NoCompanyRecommendation - A (1)
C 512:0 SpecifiedDividends - A (1)
C 516:0 TopTaxOptimizedDividends - A (1)
C 523:0 RecommendationSettings - A (1)
C 557:0 Message - A (1)
C 564:0 PensionPayOutOverview - A (1)
C 572:0 PensionPerson - A (1)
C 581:0 PensionPersonDelta - A (1)
C 588:0 PensionHouseholdOutput - A (1)
C 598:0 PensionHouseholdDelta - A (1)
C 607:0 PensionBaselineAndRecommendedOutput - A (1)
C 613:0 CompanyOutput - A (1)
C 617:0 CompanyBaselineAndRecommendedOutput - A (1)
C 623:0 SavingsOutput - A (1)
C 628:0 SavingsBaselineAndRecommendedOutput - A (1)
C 633:0 EmergencySavingsOutput - A (1)
C 641:0 EmergencySavingsBaselineAndRecommendedOutput - A (1)
C 647:0 HousingHeating - A (1)
C 655:0 RefinancingDetails - A (1)
C 665:0 HousingOutput - A (1)
C 672:0 HousingOutputAggregate - A (1)
C 682:0 HousingOutputAggregateRecommendation - A (1)
C 689:0 HousingBaselineAndRecommendedOutput - A (1)
C 699:0 OverallResultOutput - A (1)
C 705:0 OverallResultBaselineAndRecommendedOutput - A (1)
C 714:0 CalculateRequest - A (1)
C 730:0 CalculateResponse - A (1)
calculation/factories.py
F 89:0 build_houses - A (3)
F 111:0 build_liquid_assets - A (3)
F 16:0 build_incomes - A (2)

```

```

    F 41:0 build_policies - A (2)
    F 12:0 build_person - A (1)
experiment/evaluations.py
    F 171:0 run_dense_rag_on_dataset - A (5)
    F 205:0 run_dense_reranker_rag_on_dataset - A (5)
    F 240:0 run_reranker_rag_on_dataset - A (5)
    F 275:0 run_baseline_rag_on_dataset - A (5)
    F 32:0 evaluate_dispatching - A (4)
    F 309:0 run_llm_only_on_dataset - A (4)
    F 73:0 evaluate_reranker_rag - A (1)
    F 97:0 evaluate_baseline_rag - A (1)
    F 121:0 evaluate_dense_rag - A (1)
    F 145:0 evaluate_dense_reranker_rag - A (1)
    F 349:0 evaluate_llm_only - A (1)
scraping/scrapper.py
    F 46:0 html_to_text - D (22)
    F 94:0 crawl - B (8)
    F 28:0 extract_links - B (6)
    F 15:0 save_html - A (2)
    F 131:0 main - A (2)
tests/test_db.py
    F 15:0 test_reply_log - A (4)
    F 44:0 test_feedback_log - A (4)
    F 6:0 test_db - A (2)
tests/test_client.py
    F 5:0 test_session - B (6)
    F 16:0 test_existing_session - A (4)
    F 35:0 test_chat_session - A (4)
    F 28:0 test_chat_unset_session - A (3)
tests/test_calculation.py
    F 1:0 test_calculate_target_prices_success - B (9)
tests/test_chat.py
    F 6:0 test_chat - A (3)
tests/test_rag.py
    F 1:0 test_baseline_rag - B (10)
    F 29:0 test_reranker_rag - B (10)
    F 58:0 test_dense_rag - B (10)
    F 86:0 test_dense_reranker_rag - B (10)
tests/confptest.py
    F 19:0 client - A (1)
    F 28:0 triage - A (1)
    F 34:0 baseline_rag_pipeline - A (1)
    F 41:0 reranker_rag_pipeline - A (1)
    F 48:0 dense_rag_pipeline - A (1)
    F 55:0 dense_reranker_rag_pipeline - A (1)
    F 62:0 api - A (1)
    F 68:0 payload - A (1)
components/chat.py
    F 13:0 chat - A (5)
    F 7:0 format_messages - A (2)

```

```

components/rag.py
  M 23:4 RAG._load_documents - A (3)
  C 17:0 RAG - A (2)
  M 40:4 RAG._create_documents - A (2)
  M 46:4 RAG._split_documents - A (2)
  C 104:0 BaselineRAG - A (2)
  C 117:0 RerankerRAG - A (2)
  C 149:0 DenseRAG - A (2)
  M 175:4 DenseRAG._split_documents - A (2)
  C 208:0 DenseRerankerRAG - A (2)
  M 235:4 DenseRerankerRAG._split_documents - A (2)
  M 18:4 RAG.__init__ - A (1)
  M 33:4 RAG._init_splitter - A (1)
  M 52:4 RAG._init_document_store - A (1)
  M 57:4 RAG._init_retriever - A (1)
  M 60:4 RAG._init_generator - A (1)
  M 67:4 RAG._init_prompt_builder - A (1)
  M 91:4 RAG._build_pipeline - A (1)
  M 100:4 RAG.get_pipeline - A (1)
  M 105:4 BaselineRAG.__init__ - A (1)
  M 118:4 RerankerRAG.__init__ - A (1)
  M 130:4 RerankerRAG._init_ranker - A (1)
  M 137:4 RerankerRAG._build_pipeline - A (1)
  M 150:4 DenseRAG.__init__ - A (1)
  M 163:4 DenseRAG._init_text_embedder - A (1)
  M 168:4 DenseRAG._init_splitter - A (1)
  M 181:4 DenseRAG._embed_documents - A (1)
  M 188:4 DenseRAG._init_document_store - A (1)
  M 193:4 DenseRAG._init_retriever - A (1)
  M 196:4 DenseRAG._build_pipeline - A (1)
  M 209:4 DenseRerankerRAG.__init__ - A (1)
  M 223:4 DenseRerankerRAG._init_text_embedder - A (1)
  M 228:4 DenseRerankerRAG._init_splitter - A (1)
  M 241:4 DenseRerankerRAG._embed_documents - A (1)
  M 248:4 DenseRerankerRAG._init_document_store - A (1)
  M 253:4 DenseRerankerRAG._init_retriever - A (1)
  M 256:4 DenseRerankerRAG._init_ranker - A (1)
  M 263:4 DenseRerankerRAG._build_pipeline - A (1)
components/tools.py
  F 51:0 _build_policies - A (4)
  F 73:0 call_calculation_api - A (4)
  F 40:0 _build_spouse - A (2)
  F 46:0 _build_incomes - A (2)
  F 64:0 _build_houses - A (2)
  F 68:0 _build_liquid_assets - A (2)
  F 93:0 call_rag - A (1)
  C 15:0 SavingsData - A (1)
  C 21:0 HouseData - A (1)
  C 27:0 HouseholdData - A (1)
components/dispatching.py

```

```
F 54:0 call_input_guardrail - A (1)
F 65:0 call_output_guardrail - A (1)
C 30:0 ValidInput - A (1)
C 34:0 ValidOutput - A (1)
```

Listing B.1: Radon's cyclomatic complexity analysis output.

Technical
University of
Denmark

Brovej, Building 118
2800 Kgs. Lyngby
Tlf. 4525 1700

www.byg.dtu.dk