Personal Financial AI Agent

Merenda Saverio Mattia[1] and Crafa Raffaele[2]

[1] saveriomattia.merenda@studenti.unipr.it
[2] raffaele.crafa@studenti.unipr.it

December 22, 2025

Abstract

This paper presents the design, implementation, and evaluation of a Personal Financial AI Agent[1]—a machine learning-based system designed to provide intelligent, personalized financial guidance to users. The system leverages large language models (LLMs) and retrieval-augmented generation (RAG) techniques to deliver contextualized financial advice, portfolio recommendations, and analysis. The agent supports multiple LLM providers, namely Ollama for local offline inference, Google Gemini for cloud-based solutions, and OpenAI for industry-standard capabilities, enabling users to choose between privacy-first approaches and feature-rich cloud solutions. A Streamlit-based web interface facilitates interactive conversations in multiple languages, providing an accessible entry point for diverse user bases. The system demonstrates proficiency in financial profile extraction from natural language conversations, comprehensive portfolio analysis, and Monte Carlo simulations for scenario planning and risk assessment. This comprehensive study details the system's architecture, implementation techniques, evaluation methodologies, and provides clear demonstration procedures for practitioners and researchers. The project showcases the practical application of conversational AI and retrieval-augmented generation in the financial advisory domain, demonstrating how modern AI techniques can democratize access to quality financial guidance.

Keywords: Financial AI Agent, Large Language Models, Retrieval-Augmented Generation, Portfolio Analysis, Multi-provider LLM Support

1 Introduction

1.1 Motivation and Problem Statement

The democratization of financial services through digital platforms has created unprecedented opportunities for individuals to manage their wealth independently. However, effective financial planning remains a challenge for most people due to the complexity of modern financial instruments, market dynamics, and the personalized nature of financial goals. Traditional financial advisory services, while expert-driven, are often expensive and accessible only to high-net-worth individuals.

The recent advances in artificial intelligence, particularly in large language models (LLMs) and their ability to understand and generate human language, present a unique opportunity to democratize financial expertise (Brown *et al.*, 2020). LLMs can process vast amounts of financial information, understand user context through natural conversation, and provide tailored recommendations based on individual circumstances. The integration of retrieval-augmented generation (RAG) with domain-specific knowledge further enhances the quality and reliability of AI-driven advisory systems (Lewis *et al.*, 2020).

This work addresses the challenge of creating an intelligent financial advisor system that can engage users in natural language conversations to understand their financial situation, extract relevant financial information from unstructured conversations, and provide evidence-based recommendations using historical financial data. Additionally, the system must respect user privacy through optional local offline inference while adapting to different user preferences for AI providers and languages. The system must validate and present portfolio recommendations with quantitative analysis, ensuring that suggestions are grounded in financial theory and historical evidence rather than abstract or unsupported claims.

[1] Available at: https://github.com/merendamattia/personal-financial-ai-agent.

1.2 Contribution

This project contributes to the field of fintech and applied AI in several ways. First, it presents a multi-provider LLM architecture supporting Ollama, Google Gemini, and OpenAI, enabling vendor-independent deployment. Second, it integrates RAG techniques with structured financial data to improve recommendation quality and factual accuracy. Third, it develops a comprehensive financial profile extraction system that operates through conversational interfaces rather than rigid forms. Fourth, it demonstrates a production-ready web application that illustrates practical AI application in financial services. Finally, it provides detailed evaluation methodologies and comprehensive testing frameworks suitable for financial AI systems.

1.3 Document Structure

The remainder of this paper is organized as follows. Section 2 presents the overall system architecture and design principles, including the agent framework, data models, and retrieval systems. Section 3 details the technical implementation of core components, including technology stack, agent design patterns, and configuration mechanisms. Section 4 discusses the retrieval-augmented generation system specifically for financial data, covering asset organization, embedding strategies, and integration with the agent. Section 5 describes the user interface and interaction design, including the Streamlit framework, conversation flows, and visualization components. Finally, Section 6 provides comprehensive instructions for running and testing the system in various configurations, along with troubleshooting guidance.

2 System Architecture

2.1 High-Level Overview

The Personal Financial AI Agent is built on a modular, layered architecture that separates concerns into distinct components: the core AI agents, data models, retrieval systems, and the user interface. This architectural approach follows several key design principles. Modularity ensures that components are loosely coupled and can be replaced or extended independently. Multi-provider support through a provider abstraction layer enables vendor independence, allowing users to switch between Ollama, Google Gemini, and OpenAI without modifying application code. The system prioritizes privacy through optional local inference capabilities without requiring external dependencies. An extensible tool framework allows new tools and agents to be added without modifying existing code. Finally, the stateless design of agent interactions enables horizontal scaling for enterprise deployments.

2.2 Core Components

The system is composed of several interrelated components that work together to deliver financial advisory services. The agent layer contains specialized AI agents built on top of the datapizza-ai framework.[2] Base Agent serves as an abstract foundation that encapsulates common agent functionality including initialization, configuration loading, system prompt management, and tool registration. All specialized agents inherit from Base Agent to ensure consistency and reduce code duplication. The Chatbot Agent focuses on natural conversation and financial profile extraction, maintaining conversation history and interpreting user intent while guiding users toward providing relevant financial information. Financial Advisor Agent specializes in portfolio generation and analysis, utilizing RAG to access historical financial data and providing evidence-based recommendations.

The data model layer ensures type safety and clear data contracts throughout the system. Financial Profile captures demographic and financial characteristics including age, employment status, income, expenses, debt, savings, investment experience, risk tolerance, and financial goals. Portfolio represents investment recommendations including asset allocations, expected returns, volatility metrics, and diversification ratios. PAC Metrics encodes portfolio analysis metrics including PAC (piano di accumulo) values, performance indicators, and risk assessment. These models are implemented using Pydantic, which enforces type safety at runtime and provides validation.

The retrieval layer implements retrieval-augmented generation specifically for financial data. RAG Asset Retriever maintains a vector database of historical ETF and stock data, supports

[2] `https://github.com/datapizza-labs/datapizza-ai`.

semantic search for relevant financial assets, integrates historical price data spanning ten years, and provides structured asset information for recommendations. This component is essential for grounding LLM recommendations in factual, historical information rather than allowing the model to generate potentially inaccurate figures.

The tool layer provides specialized computation capabilities that agents can invoke. Currently, the system includes financial analysis tools for computing financial metrics, historical returns, and risk indicators. Portfolio validation tools verify portfolio structure and compliance with constraints. Future extensions might include tax optimization tools, rebalancing advisors, and risk simulation engines.

2.3 Multi-Provider LLM Support

A critical design feature is the abstraction of LLM provider complexity through a unified provider layer. This enables users to select their preferred provider based on individual needs and constraints.

Ollama provides full offline inference with no data sent to external servers, making it ideal for privacy-sensitive applications. However, it requires local model download and provides variable performance depending on hardware capabilities. Google Gemini offers cloud-based API access with advanced reasoning capabilities and integration with the Google ecosystem, but requires API key authentication and incurs usage costs. OpenAI provides industry-standard LLM capabilities with the highest quality responses and fine-grained API control, but requires API key and billing setup with associated costs. The client abstraction layer in the codebase provides a unified interface for all providers, allowing seamless switching at configuration time without modifying application logic.

2.4 Data-Flow

The typical interaction flow through the system follows a well-defined sequence. A user initiates conversation with Chatbot Agent, which engages the user in dialogue to gather financial information. The user provides profile details and financial goals through natural conversation. The Financial Advisor Agent then extracts a structured Financial Profile from the conversation history using the LLM with structured output capabilities. The agent queries RAG Asset Retriever for suitable assets based on the user's profile constraints. The portfolio generation logic combines user preferences with historical data to create a diversified portfolio. The system presents recommendations to the user with comprehensive analysis including historical performance, risk metrics, and diversification ratios. The user can refine preferences and iterate through the analysis cycle, with the agent updating recommendations based on new information.

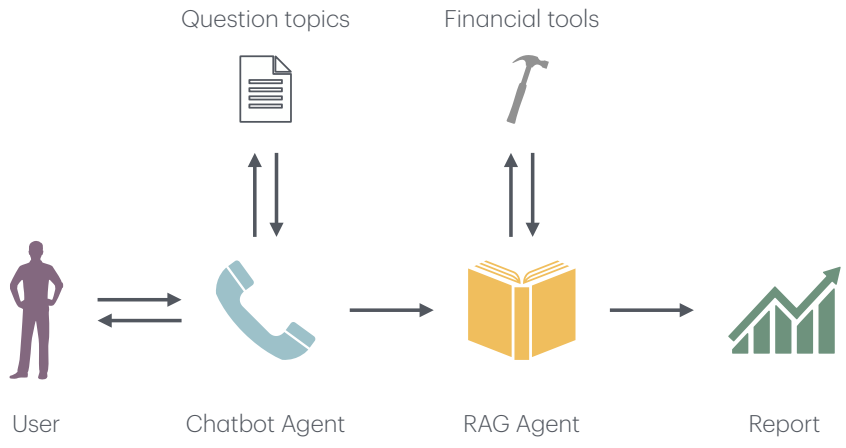

Figure 1: High-level execution flow of the Personal Financial AI Agent.

3 Technical Implementation

3.1 Technology Stack

The implementation leverages a carefully selected technology stack optimized for AI, data processing, and web deployment. The core framework is datapizza-ai, a modern Italian AI framework specifically designed for building conversational agents with complex orchestration requirements. The datapizza-ai framework provides comprehensive abstractions for agent lifecycle management, message routing, and tool invocation. It abstracts away the complexities of different LLM provider APIs and authentication mechanisms, allowing the system to seamlessly switch between Ollama for local inference, Google Gemini for cloud-based capabilities, or OpenAI's models, without requiring changes to the agent implementation. The framework manages conversation context and memory through a unified interface, handling both immediate conversation history for contextual understanding and persistent memory for long-term user profile management. Additionally, datapizza-ai provides sophisticated tool management capabilities that enable agents to register and invoke external functions as part of their decision-making process, facilitating integration with specialized services like the RAG retriever.

The web framework is Streamlit, a Python-based framework enabling rapid development of data applications without requiring JavaScript expertise. Streamlit handles reactive UI updates and state management automatically, greatly reducing development overhead. Pydantic provides runtime type validation and serialization for data models, used extensively for Financial Profile, Portfolio, and related models, ensuring data consistency throughout the system. Qdrant serves as a vector database engine for semantic search over financial documents and embeddings. Plotly enables interactive, publication-quality financial visualizations including candlestick charts, return distributions, and allocation pie charts.

The data processing layer relies on Pandas for tabular data manipulation and financial time-series analysis, NumPy for numerical computations and matrix operations, and scikit-learn for statistical analysis and machine learning utilities.

The infrastructure components include Docker for container orchestration, Docker Compose for multi-container application management, and Ollama as a local LLM runtime environment. This comprehensive technology stack provides all necessary capabilities for building a production-quality financial AI system.

3.2 Agent Implementation

The agent architecture, detailed in Section 2, is built on the datapizza-ai framework and implements a layered approach with base agent functionality and specialized implementations for both conversational and financial-domain interactions. Agents manage multi-turn conversations with memory management capabilities and orchestrate complex workflows combining language model reasoning with tool invocation and financial calculations.

3.3 Data Model Validation

Pydantic models ensure data integrity throughout the system. Financial Profile enforces type safety for all financial attributes, provides default values for missing information, includes field descriptions for API documentation, and supports optional fields for incomplete data. Similar validation is applied to Portfolio and other data structures. This approach catches data errors early and provides clear error messages when data does not conform to expected schemas.

3.4 DevOps and Continuous Integration

The project implements a comprehensive continuous integration and continuous deployment pipeline leveraging GitHub Actions to ensure code quality and reliable releases. The automated testing pipeline executes the full test suite on every push to main branches and on pull requests, validating that changes do not introduce regressions. The test environment is configured with Python 3.11 to match production requirements, and critical API keys are provided through secure secrets management to enable testing of cloud-based LLM providers.

Code quality enforcement is implemented through multiple mechanisms. Conventional commits validation ensures that all commit messages follow established standards for clarity and automated changelog generation, improving project history readability and enabling semantic versioning. Pre-commit hooks are automatically executed before commits are recorded, running code formatters

and linters to maintain consistent style. The black code formatter ensures uniform code formatting throughout the project, while import sorting tools organize dependencies in a canonical order. These automated checks prevent many common code quality issues from entering the codebase.

Docker image validation is performed on pull requests and pushes to main branches, testing that the containerized application builds successfully and runs without errors. This early detection of deployment issues prevents broken container images from being pushed to registries. When code is merged to the main branch and all validation checks pass, a semantic release process automatically analyzes commit messages to determine version numbers and generate release notes, maintaining a changelog without manual intervention. Upon successful semantic release, the Docker image is automatically built with the new version tag and pushed to Docker Hub, making the latest build immediately available for deployment.

The infrastructure as code approach, enabled through Docker Compose configuration files and environment-based configuration, allows reproducible deployments across different environments. The combination of automated testing, code quality enforcement, and container validation creates a robust pipeline that catches issues early while enabling rapid iteration and reliable production deployments.

4 Retrieval-Augmented Generation for Financial Data

4.1 Motivation for RAG in Financial Advisory

Retrieval-augmented generation enhances LLM capabilities by grounding responses in retrieved factual information.

This approach overcomes the intrinsic limitation of static knowledge in LLMs, whose internal representations are fixed at training time and cannot reflect newly available or frequently changing information. This is particularly useful since pre-trained models were used for this project. By dynamically retrieving relevant documents at inference time, RAG enables the model to operate on up-to-date, domain-specific, and potentially proprietary knowledge without requiring modifications to the underlying parameters.

Moreover, grounding generation in retrieved factual sources significantly mitigates the risk of hallucinations, as the model is encouraged to base its responses on explicit evidence rather than relying solely on learned statistical patterns. This characteristic is particularly important in scenarios where factual accuracy, reliability, and consistency are critical, such as in the financial sector. Using real historical data rather than hallucinated figures ensures that recommendations are based on verifiable information. Relevance is equally important, as finding assets that match user criteria from available options requires domain knowledge that the system must possess. Compliance considerations mean providing factual, verifiable information that can withstand scrutiny from regulators. Interpretability allows showing users which data informed recommendations, building trust in the advice-giving process. The system retrieves relevant Exchange Traded Funds (ETFs) based on user preferences and financial profiles, providing 10-year historical return data for each recommendation.

Finally, RAG-based techniques are very versatile because they allow you to change the knowledge base on which the retrieval is performed, if necessary, and are much more efficient and scalable because they avoid the excessive cost of having to retrain the model.

4.2 Asset Data Organization

Financial assets are organized in a structured directory hierarchy. The dataset directory contains subdirectories for different asset types. ETFs are further organized by asset class, with bonds and stocks as primary categories. Within each category, assets are organized by sector or type, such as corporate bonds, government bonds, tech sector ETFs, healthcare sector ETFs, and diversified index ETFs. Each asset includes asset name and ticker symbol, asset class and sector classification, risk level assessment, 10-year historical return data, expense ratios and fees, and diversification characteristics. This structured organization enables efficient retrieval and more accurate semantic search.

4.3 Knowledge Base Composition and Dataset

The primary knowledge source for the RAG system consists of a curated collection of ETF factsheets, sourced from the justETF platform.[3] These documents represent the *ground truth* for the agent, providing standardized, up-to-date financial data that would otherwise be subject to hallucinations if left to the LLM's internal weights.

Each document in the dataset follows a consistent semi-structured format, typically organized into several key informational blocks:

- Core Identifiers: Metadata such as the ISIN (International Securities Identification Number), ticker symbol, and fund name (e.g., iShares Core S&P 500 UCITS ETF).
- Investment Objective: A qualitative description of the fund's strategy, such as replicating the performance of a specific index or targeting corporate bonds within a certain maturity range.
- Financial Metrics: Quantitative data including the Total Expense Ratio (TER), fund size (AUM), dividend policy (Accumulating vs. Distributing), and the replication methodology (Physical vs. Optimized).
- Risk Profile: The synthetic risk and reward indicator (SRRI), which provides a standardized score from 1 to 7, essential for the agent to align recommendations with the user's extracted risk tolerance.

By indexing these specific PDF files, the RAG architecture enables the Financial Advisor Agent to perform *source-grounded* reasoning. When a user asks for a tech-heavy or low-cost portfolio, the retriever can pinpoint exact segments within these factsheets—such as a TER of 0.07% for CSSPX or the Asia-Pacific focus of CSEMAS—ensuring that every financial advice is backed by official fund documentation.

4.4 RAG Architecture

The RAG architecture is centered around the RAG Asset Retriever class, which implements a streamlined pipeline for document ingestion, embedding generation, and semantic retrieval. Unlike traditional database-heavy approaches, the system utilizes a high-performance, in-memory vector indexing strategy based on NumPy and Pickle for persistence, optimized for the specific scale of the ETF dataset.

The pipeline begins with the ingest_pdfs method, which recursively scans the data directory to extract text from PDF documents using the pypdf library. To ensure context preservation while maintaining granularity, the extracted text is processed into overlapping chunks of 800 characters with a 120-character overlap. This character-based chunking strategy ensures that semantic boundaries are respected during the retrieval phase.

For the embedding stage, the system employs the Sentence Transformer framework, specifically utilizing the `all-roberta-large-v1` model. This model transforms text chunks into high-dimensional dense vectors (1024 dimensions), capturing deep semantic relationships that simple keyword searches would miss. To optimize performance and reduce computational overhead, the system implements a caching mechanism: once generated, the embeddings and associated metadata are serialized into an embeddings `.pkl` file. Subsequent initializations of the agent load this index directly from disk, eliminating the need for re-processing the entire dataset.

The retrieval logic follows a *top-K* similarity approach. When a query is received, it is encoded into the same vector space as the document chunks. The system then computes the cosine similarity between the query vector and the entire embedding matrix using scikit-learn's optimized routines. The k most relevant segments (with a default of `k = 15`) are returned to the Financial Advisor Agent, ranked by their similarity score. This architecture ensures that the LLM is provided with the most contextually relevant financial data to ground its recommendations in factual evidence.

4.5 Integration with Agent

The Financial Advisor Agent leverages the RAG system through a structured integration layer designed to ground portfolio recommendations in real-world data. The process begins with the

[3] `https://www.justetf.com/it`

RAG Query Builder, which transforms the user's structured Financial Profile into a natural language search query. This query is designed to capture the essence of the user's risk tolerance and financial goals, which are then passed to the RAG Asset Retriever.

The portfolio generation workflow follows a rigorous sequential process. First, the agent extracts the client's profile from the conversation history using structured output validation. Next, it invokes the retriever to identify the top 15 most relevant asset documents. To maintain efficiency within the LLM context window, the agent extracts the most significant metadata and descriptions from these documents, creating a rich *asset context*.

Finally, the agent utilizes a structured response mechanism to generate a Portfolio recommendation. By providing both the client's financial profile and the retrieved asset context to the LLM, the system ensures that the suggested allocation percentages are informed by actual historical data found in the dataset. This architectural choice minimizes hallucinations and ensures that the risk level of the generated portfolio is strictly aligned with the user's extracted profile. Analysis of specific assets can further be performed through direct tool invocation, providing detailed historical return metrics and quantitative analysis to the end-user.

4.6 Data Processing Pipeline

The data processing pipeline is designed to transform unstructured PDF prospectuses into a searchable knowledge base. The process begins by scanning the dataset directory for PDF files using a recursive globbing strategy. Text extraction is performed page-by-page, and the resulting strings are partitioned into chunks of 800 characters with a 120-character overlap to maintain semantic continuity.

Each chunk is then transformed into a dense vector representation using the Sentence Transformer model. Unlike generic text processing, this pipeline focuses on capturing the specific terminology found in financial asset descriptions. The system computes these embeddings during the initial setup and stores the entire payload—consisting of the original text, metadata (such as the source file name and chunk ID), and the numerical vectors—into a serialized Pickle file for efficient subsequent loading.

4.7 Performance Optimization

To ensure a responsive user experience, the system implements several optimization strategies at the retrieval level. Instead of relying on external database calls, the RAG Asset Retriever loads the entire embedding index into memory as a NumPy array. This allows for near-instantaneous similarity computations using optimized matrix operations.

The use of a global cache for the embedding model ensures that the Sentence Transformer is loaded into memory only once, significantly reducing the latency of subsequent user queries. Furthermore, by utilizing the all-roberta-large-v1 model, the system achieves a balance between high-dimensional semantic accuracy (1024 dimensions) and computational speed, even on hardware without dedicated GPU acceleration.

5 User Interface and Interaction Design

5.1 Streamlit-Based Web Application

The user interface is implemented using Streamlit, a Python framework for building data applications. This choice enables rapid development without requiring JavaScript expertise. The framework provides automatic state management and reactive UI updates, handling much of the complexity of building interactive web applications. Built-in support for Markdown, charts, and interactive components allows developers to focus on business logic rather than UI infrastructure. Seamless integration with Python data science libraries like Pandas, NumPy, and Plotly enables rapid visualization of financial data.

5.2 Application Structure and Management

The Streamlit application is organized into several logical sections. Session management uses Streamlit's built-in session state to maintain current conversation history, extracted financial profiles, generated portfolios, user settings including language and provider preference, and analysis results with associated charts. The application is structured as a multi-page application with several distinct pages. The Chat Page provides the main conversational interface where users interact

with the Chatbot Agent. The Portfolio Analysis Page displays generated portfolios with detailed metrics and visualizations including allocation charts and historical performance. The Settings Page allows users to configure LLM provider, language, and financial parameters including Monte Carlo simulation settings. The Profile Management Page enables users to load, edit, and save financial profiles as JSON files.

5.3 Conversation Flow and Profile Interaction

When a new user arrives at the application, the system displays a welcome message in the user's preferred language. The Chatbot Agent initiates a greeting and explains available capabilities, helping the user understand what the system can do. The system requests selection of the LLM provider if not previously configured, allowing users to choose based on their privacy preferences and available resources.

The agent guides users through information collection using a conversational approach. Rather than presenting a form, it asks questions about demographics such as age and employment status. Income and expenses are explored to understand the user's financial capacity. Existing assets and investments are examined to assess current portfolio composition. Risk tolerance is evaluated through behavioral questions rather than abstract risk rating scales. Finally, financial goals and time horizons are discussed to set appropriate expectations. The conversational approach is natural and adaptive, with questions varying based on previous answers, making the interaction feel like speaking with a knowledgeable advisor.

Once sufficient information is gathered, the portfolio generation process begins. The user requests a portfolio recommendation either explicitly or implicitly through the conversation. The Financial Advisor Agent receives the full conversation context. The agent extracts a Financial Profile using the LLM with structured output to ensure data quality. Then, it builds a RAG query based on profile constraints. Relevant assets are retrieved from the vector database. The agent generates a portfolio with specific allocation percentages. The system validates the portfolio structure to ensure allocations sum to 100 percent and comply with constraints. The portfolio is presented to the user with comprehensive analysis including expected return, volatility, and diversification metrics.

5.4 Visualization and Display Components

The interface includes rich visualizations that help users understand portfolio concepts. Portfolio composition is shown through a pie chart displaying allocation percentages for each asset class or security. Historical returns are visualized as line charts showing 10-year performance of recommended assets, allowing users to see how the suggested portfolio components have performed historically. Risk-return scatter plots show an efficient frontier with assets plotted by volatility on the x-axis and expected return on the y-axis, helping users visualize the risk-return trade-off. Monte Carlo simulation results are displayed as distribution graphs showing possible portfolio values at future time horizons, illustrating the range of outcomes under different market scenarios.

5.5 Settings, Configuration and Profile Management

The settings page empowers users to customize their experience. For LLM provider selection, users can choose between Ollama for local privacy-preserving inference, Google Gemini for advanced cloud-based reasoning, or OpenAI for highest-quality responses. Users can configure API keys for cloud providers, select specific models within each provider, and test connectivity to verify configuration before proceeding. Financial parameters can be customized including Monte Carlo simulation parameters such as the number of scenarios and projection years, initial investment amounts, monthly contribution levels, risk tolerance overrides, and asset class preferences. This customization allows users to tailor the analysis to their specific circumstances.

Users can export their extracted financial profile as JSON, enabling them to download a structured record of their financial situation for record-keeping or sharing with professional advisors. Users can also import previously saved profiles to resume analysis or continue work from a previous session. Manual editing of extracted profiles is supported, allowing users to correct information or update their circumstances. The ability to manage profiles increases user confidence in data privacy and gives users a sense of control over their information.

6 Demonstration and Usage

This section outlines how to deploy and use the Personal Financial AI Agent, including system requirements, installation procedures, and the integrated testing infrastructure.

6.1 Software Requirements

You will need **Python 3.11** or higher installed on your system. For containerized deployment, **Docker version 20.10+** is required, along with **Docker Compose version 1.29+** for managing multi-container setups. Finally, **Git** is necessary for cloning the repository from GitHub.

6.2 Installation

For detailed installation instructions, please refer to the comprehensive **README.md** file in the repository at `https://github.com/merendamattia/personal-financial-ai-agent`. The README contains step-by-step, up-to-date guidance for cloning the repository, setting up Python environments, installing dependencies, and configuring environment variables for all supported LLM providers.

We **strongly recommend** using Docker Compose for deployment as it provides a consistent, isolated, and production-ready environment that works seamlessly across all platforms without local dependency management hassles.

6.3 Testing and Validation

The project includes a comprehensive test suite covering unit tests for core components (financial profile, portfolio management, asset retrieval), integration tests for complete workflows, and tool-specific tests for financial analysis features. All tests are automatically executed in the CI/CD pipeline to ensure code quality and prevent regressions.

7 Conclusion

The Personal Financial AI Agent demonstrates that artificial intelligence can effectively augment human financial decision-making. By combining conversational AI, retrieval-augmented generation, and modern web technologies, the system creates an offering that is simultaneously intelligent, accessible, and trustworthy. The modular architecture and emphasis on provider flexibility ensure longevity as the AI landscape continues to evolve. The comprehensive testing and documentation enable both researchers and practitioners to build upon this foundation.

As AI increasingly shapes financial services, maintaining focus on user trust, transparency, and privacy will be crucial. This project provides a template for building AI systems in regulated domains where accuracy and trustworthiness are paramount. Financial advisory is a domain where AI's promise to democratize expertise must be balanced with responsibility to protect users from misinformation and poor recommendations. By emphasizing grounding in real data through RAG, supporting multiple inference options for privacy, and maintaining transparency about limitations, this system demonstrates an approach to trustworthy AI in high-stakes domains.

Despite its strengths, the system has several inherent limitations that should be acknowledged. Model dependency means system quality is fundamentally bounded by underlying LLM capabilities, with smaller models showing limitations in financial reasoning compared to larger models. Data currency presents a challenge since historical financial data has a fixed lookback window, and real-time market data integration would improve recommendations. Limited risk modeling is employed, with simplified volatility metrics; advanced portfolio construction techniques such as mean-variance optimization and factor models could enhance recommendations. The system provides recommendations only and cannot execute trades, so integration with trading platforms would require additional compliance and security measures. Scalability is limited since the current architecture uses in-memory processing, and distributed systems would be needed for enterprise-scale deployment.

The complete source code for the Personal Financial AI Agent is available at `https://github.com/merendamattia/personal-financial-ai-agent`.

References

Brown, Tom B *et al.*, (2020). "Language Models are Few-Shot Learners", *arXiv preprint arXiv:2005.14165*,
Lewis, Patrick *et al.*, (2020). "Retrieval-Augmented Generation for Knowledge-Intensive NLP Tasks", *arXiv preprint arXiv:2005.11401*,