Dedication and Acknowledgements

I wish to express my sincere gratitude to all the people who supported me throughout these three years.

This journey has not always been easy. Transferring from International Economics and Management to Economics and Computer Science was both a challenge and an opportunity, requiring me to recover an entire year of courses in a very short time. Those months of intense study, pressure, and sacrifice were among the most demanding of my life, yet they proved decisive in shaping both my academic path and my personal growth.

To the friends I met in BIEM, thank you for staying close even after I switched program. More generally, I am grateful to all those who stood by me even when I disappeared for months, shut away at home to prepare for exams. Your presence and understanding truly made a difference.

To the BEMACS community, both older and younger, thank you for being a constant source of inspiration, motivation, and personal growth.

I am deeply thankful for the opportunity to study abroad at the University of Technology Sydney, an experience that enriched me both academically and personally, and one that made all the previous efforts worthwhile. I am also grateful for the recognition I received during these years, which encouraged me to continue striving for excellence and reminded me that dedication and perseverance are always rewarded.

Special thanks go to my professors for their guidance and commitment, and to the members of BAINSA, BSML, and E-Club for the inspiration, opportunities, and ideas you shared with me.

I also wish to thank my closest friends for believing in me and supporting me even when my decisions seemed overly ambitious, and for being present in both difficult and joyful moments. A heartfelt thanks to my supervisor, Professor Luca Molteni, and to all those who, in different ways, contributed to this journey.

Table of Contents

Chapter 1

Introduction

1.0.1 Background

The evolution of artificial intelligence has recently shifted from static language models to dynamic, task-oriented *agents*. Large language models (LLMs) such as GPT-4, LLaMA, and Claude, once primarily designed for single-turn text generation, are increasingly being used as the reasoning core of systems that can plan, call external tools, and sequence multiple steps to achieve complex goals. This transformation represents more than an incremental improvement in generative capability: it is a paradigm shift from producing text in isolation to orchestrating workflows and interacting with external data and computational resources.

At the center of this shift lies the ability to chain together multiple calls, integrate APIs, query databases, and adaptively select the right tools for a given sub-task. These agentic workflows extend LLMs from passive responders into active problem-solvers that can operate in dynamic environments.

1.0.2 Limitations and Enhancements

Despite these advances, LLMs remain constrained by persistent shortcomings. They are prone to **hallucinations**, confidently producing factually incorrect outputs. They face **contextual limitations**, struggling to scale to large or specialized datasets. They also suffer from **knowledge staleness**, since model parameters cannot easily incorporate new information without retraining.

Enhancements such as **Retrieval-Augmented Generation (RAG)** address these weaknesses by grounding responses in retrieved, verifiable sources of information. Similarly, techniques such as **Text-to-SQL (Text2SQL)** allow natural language inputs to be

translated into precise, executable queries over structured data, bridging the gap between flexible human expression and formal logic. These methods, combined with the agentic paradigm, substantially increase the reliability and applicability of LLM-driven systems.

1.0.3 Industrial Adoption and Hype

The potential of these systems has not gone unnoticed by industry. Frameworks such as **LangChain** have rapidly accelerated the development of LLM-powered agents by offering modular components for chaining tasks, orchestrating tools, and retaining context. As a result, agentic systems are now being marketed across domains ranging from business intelligence and customer support to finance and auditing.

However, this surge of interest has also produced a degree of hype. Increasingly, any system that makes use of LLMs is branded as "agentic," even when it lacks the adaptive planning, reasoning, and tool-use capabilities that define genuine LLM-powered agents. This creates both opportunities and challenges: while industry enthusiasm drives rapid prototyping and adoption, it also blurs the distinction between true innovation and superficial rebranding.

1.0.4 Thesis Motivation and Objectives

This thesis is motivated by the need to critically examine this evolving space, distinguishing between genuine capabilities and inflated claims, and by the opportunity to contribute both theoretically and practically. The objectives are threefold:

1. To conduct a structured literature review of evolution and of the current state of language processing, LLMs and AI agents, with some focus on modern architectures and orchestration frameworks such as LangChain.

2. To design and implement a full-stack prototype chatbot that integrates RAG, Text2SQL, and multi-tool coordination, moving from conceptual frameworks to practical deployment.

3. To adapt these techniques to different real-world contexts, with a focus on investment management and audit-related use cases, supported by synthetic data generation and scenario-specific customization.

The direction of this work is grounded in practical experience. During my internship at **Accenture**, I was exposed to the challenges faced by clients seeking to implement

AI solutions. I developed demonstrations that directly addressed these challenges and explored how emerging technologies could be applied in practice. This first-hand perspective provides a valuable lens through which to evaluate the current state of industry adoption, and it ensures that the contributions of this thesis remain both academically rigorous and industrially relevant.

Chapter 2

History and Techniques of Language Modelling: from traditional NLP to Transformers

2.1 NLP before the advent of Neural Networks

This literature review traces the evolution of natural language processing (NLP) from its early foundations to current approaches based on transformers and agentic systems. To situate recent advances, it is useful to recall how earlier paradigms framed the problem of language modelling and what challenges they encountered.

Before the rise of modern neural networks, NLP followed two main trajectories. The first was *rule-based*, grounded in theoretical linguistics and the idea that language could be formally captured through explicit grammatical rules. The second was *statistical*, rooted in information theory and probability, where models learned to predict words and sentences from observed regularities in text corpora. Rule-based approaches promised interpretability and linguistic precision [1], while statistical methods offered robustness and data-driven disambiguation [2].

These traditions shaped the foundations of the field and enabled early applications in speech recognition, machine translation, and information retrieval. Yet, they also revealed limits in scalability and adaptability. Addressing these gaps motivated the shift toward neural architectures, which in turn opened the door to the transformer models that dominate NLP today. More recently, the focus has expanded again, from powerful models to the design of agents that can act, reason, and interact using language.

2.1.1 Rule-Based Approaches

Rule-based NLP descends from Chomsky's generative grammar, which represents sentences with phrase-structure rules and makes their hierarchical organization explicit. A core claim is that natural language cannot be modeled by simple finite-state (Markov) processes: even very large n-gram models do not reach the generative capacity of a grammar [3]. Early systems—parsers and domain-specific programs such as *SHRDLU*—therefore depended on extensive sets of hand-written rules.

Machine translation followed the same path. The 1954 Georgetown–IBM demonstration translated a small set of Russian sentences into English using six grammar rules and a tiny dictionary. It was an eye-catching proof of concept, but it did not scale [4]. As researchers expanded coverage, rule inventories became complex and fragile; purely symbolic systems struggled with ambiguity and with shifting domains.

Strengths and limits. Rule-based models are transparent: a parse tree or derivation explicitly shows *why* a sentence is licensed. However, building and maintaining broad-coverage grammars requires sustained expert effort, and without probabilities there is no principled way to choose among competing analyses. These practical limits motivated the move toward probabilistic methods.

2.1.2 Statistical Language Models

A complementary viewpoint emerged from information theory. Shannon modelled English as a stochastic source and linked prediction to entropy, showing that longer context reduces uncertainty [2]. This led naturally to n-gram models with the Markov approximation

$$P(w_i \mid w_{1:i-1}) \approx P(w_i \mid w_{i-n+1:i-1}) \,,$$

estimated from corpora and made robust with smoothing to handle unseen events. The intellectual roots go back even further: Markov's 1913 analysis of *Eugene Onegin* computed by hand the transition probabilities between vowels and consonants and showed such statistics capture stylistic regularities [5].

Why they worked—and where they fell short. Statistical models learn from usage, resolve ambiguity by choosing the most probable analysis, and improve with more data. Yet fixed-length context limits long-range dependencies, and surface statistics provide little explicit structure or meaning. These weaknesses spurred methods that incorporate *latent* structure.

2.1.3 Hidden Structure: HMMs and Probabilistic Grammars

Many aspects of a sentence are not directly visible in the words themselves. To interpret language, it helps to infer *hidden structure* such as part-of-speech tags and phrase boundaries. Two classic probabilistic models make this idea operational: Hidden Markov Models (HMMs) for sequences and Probabilistic Context-Free Grammars (PCFGs) for trees. Both retain symbolic structure while using probabilities to resolve ambiguity.

Hidden Markov Models (HMMs). An HMM assumes a sequence of unobserved states (e.g., tags) that generate the observed words. The model is defined by *transition* probabilities, which state how likely one tag follows another, and *emission* probabilities, which state how likely a tag produces a particular word. A key simplification is the first-order Markov assumption: the next tag depends only on the current tag. This assumption enables efficient computation. Given a sentence, the Viterbi algorithm uses dynamic programming to find the most likely tag sequence. Parameters can be estimated from labeled data by counting, or from unlabeled data with the Expectation–Maximization procedure (Baum–Welch). In practice, this yields accurate and fast part-of-speech tagging at scale.

Probabilistic Context-Free Grammars (PCFGs). A PCFG starts from ordinary grammar rules (e.g., S $\rightarrow$ NP VP) and assigns a probability to each rule. The context-free assumption states that the choice of expansion depends only on the left-hand-side symbol, not on surrounding words or higher nodes. Chart parsers exploit this locality to consider many candidate trees while reusing intermediate results. The probability of a full parse is the product of its rule probabilities, allowing the parser to rank analyses and select the most plausible tree. This approach preserves explicit syntactic structure while adding a principled method for disambiguation.

Impact and limits. HMMs and PCFGs showed that combining simple structural assumptions with probabilities enables reliable tagging and parsing on large corpora. Their independence assumptions—Markovian transitions in HMMs and context-free expansions in PCFGs—are idealizations that ignore long-distance dependencies and fine-grained lexical effects. Nevertheless, these models established a practical baseline and provided a foundation for later extensions that relax the assumptions and incorporate richer context.

2.2 Neural Network Approaches to Language Modeling Before Transformers

Before Transformers, neural networks were developed to address limits of rule-based and purely probabilistic methods. Classical n-gram language models predicted the next word from a short window of context, while formalisms like HMMs and PCFGs introduced latent structure and principled uncertainty. These approaches supported speech recognition, machine translation, and information retrieval, but they handled long-range context and semantics only crudely. Neural models kept the predictive, data-driven spirit of earlier work while learning distributed representations that generalize across words and contexts.

2.2.1 Early Connectionist Models

Connectionist models learn from data using neuron-like units and weight updates rather than hand-written rules. Early work on Parallel Distributed Processing argued that distributed representations can capture linguistic regularities [6]. A well-known example is the learned trajectory on English verb inflection in [7]. Recurrent designs soon appeared: the Jordan network fed the previous output into context units, and Elman's Simple Recurrent Network (SRN) fed the previous hidden state back into the model [8]. Trained on next-symbol prediction, SRNs developed internal states that reflected grammatical patterns, demonstrating use of context beyond a fixed window—albeit on small datasets given the hardware of the time.

2.2.2 Feedforward Neural Language Models

The *curse of dimensionality* limits n-grams: parameters explode with larger vocabularies and longer contexts. Bengio et al. introduced a *neural probabilistic language model* that jointly learns word embeddings and a nonlinear mapping from a fixed-length context to next-word probabilities [9]. Continuous embeddings let the model share information across similar words and contexts, improving generalization over strong smoothed n-grams. These models were adopted for rescoring in Automatic Speech Recognition and Statistical Machine Translation pipelines, and evaluation commonly used corpora such as the Penn Treebank (PTB) [10].

2.2.3 Recurrent Neural Networks and BPTT

Recurrent Neural Networks (RNNs) remove the fixed-window limitation by maintaining a *hidden state* that is updated at each time step. A simple Elman-style RNN computes

$$h_t = f\big(W_{ih}x_t + W_{hh}h_{t-1} + b_h\big), \qquad y_t = \mathrm{softmax}\big(W_{ho}h_t + b_o\big),$$

with nonlinearity $f(\cdot)$ (e.g., tanh). Training uses *Backpropagation Through Time* (BPTT), which unrolls the network over the sequence and applies backpropagation to the sequence loss [11]. With improved compute and training practice, RNN language models achieved strong perplexity gains over smoothed 5-grams on PTB and other corpora [12, 13]. However, simple RNNs suffer from *vanishing* and *exploding* gradients during BPTT, which hamper learning of very long dependencies [11].

Bidirectional RNNs. A Bidirectional RNN processes text left-to-right and right-to-left, then combines the two states so each position has access to both past and future context. This design improves tagging and sequence labeling, especially when paired with LSTM units [14].

2.2.4 Long Short-Term Memory (LSTM)

LSTM networks address vanishing gradients by introducing a *cell state* c_t and gates that regulate information flow [15]. With input i_t, forget f_t [16], output o_t, and candidate $\tilde{c}_t$, the update is

$$c_t = f_t \odot c_{t-1} + i_t \odot \tilde{c}_t, \qquad h_t = o_t \odot \tanh(c_t),$$

where $\odot$ denotes element-wise multiplication. The additive path through c_t supports near-constant error flow, enabling long-range credit assignment. By 2014, LSTM-based models set state of the art on language modeling, sequence labeling, and machine translation [17]. In machine translation, encoder–decoder LSTMs and later *attention* surpassed phrase-based SMT; large-scale deployments such as GNMT used deep LSTM encoder–decoders.

2.2.5 Gated Recurrent Units (GRUs)

Gated Recurrent Units simplify the gating mechanism while preserving additive state updates [18, 19]. With update z_t and reset r_t gates,

$$\tilde{h}_t = \tanh\big(W_h x_t + U_h(r_t \odot h_{t-1})\big), \qquad h_t = z_t \odot h_{t-1} + (1 - z_t) \odot \tilde{h}_t.$$

This formulation eases gradient flow and often matches LSTM performance with fewer parameters. Early attentive neural Machine Translation systems employed GRUs effectively [20].

2.2.6 Datasets, Toolkits, and Practice

Standardized datasets made results comparable and reproducible. PTB (Penn Treebank, a standardized dataset introduced by Marcus et al. in 1993) remained a central benchmark for language modeling and tagging [10], while later corpora (e.g., WikiText and the One Billion Word Benchmark) emphasized larger vocabularies and longer contexts. Open-source frameworks reduced implementation cost: Theano and Torch7 enabled custom RNN, LSTM, and GRU layers; later, TensorFlow (often with Keras) and PyTorch provided efficient GPU training. Specialized toolkits such as *RNNLM* offered optimized training for RNN-based language models [12]. The combination of shared datasets and mature libraries accelerated adoption in research and industry.

2.3 The advent of Transformers

The *Transformer* architecture, introduced in 2017, marked a major shift in Natural Language Processing (NLP) [21]. Instead of processing tokens sequentially as in recurrent or convolutional models, Transformers use *self-attention* to let each token directly reference every other token in the sequence. This design enables highly parallel training and strong performance on sequence-to-sequence tasks

2.3.1 The 2017 Breakthrough: "Attention Is All You Need"

Vaswani et al. proposed self-attention as the central operation for sequence modeling [21]. Given token embeddings, the model forms *queries*, *keys*, and *values* via learned linear maps. Scaled dot-product similarities between queries and keys produce attention weights, which mix the values into new representations. *Multi-Head Attention* repeats this computation in parallel heads so that different heads can capture different relationships (e.g., agreement, coreference, or long-range dependencies).

Encoder–Decoder organization. For translation and other sequence-to-sequence tasks, the original Transformer uses an *encoder–decoder* layout. Each encoder block contains self-attention and a position-wise feed-forward network; each decoder block

adds *masked* self-attention (to prevent access to future targets during training) and *encoder–decoder* attention (to condition generation on the source). Because attention itself is order-agnostic, *positional encodings* inject information about token order (e.g., sinusoidal functions). *Residual connections* and *layer normalization* stabilize optimization. Even relatively shallow stacks (e.g., six encoder and six decoder layers) surpassed strong RNN baselines on WMT14 English–German and English–French [21].

Training efficiency. Removing recurrence allows attention and feed-forward layers to be computed as batched matrix operations over all positions, which maps well to GPUs and TPUs. In practice, *subword tokenization* (e.g., byte-pair encoding), warmup and inverse-square-root learning-rate schedules, and large-batch training further improve throughput and quality [21].

2.3.2 Evolution of the Paradigm: BERT, GPT, and T5

BERT (2018): *Bidirectional Encoder Representations from Transformers* is an *encoder-only* model trained with *Masked Language Modeling* (MLM) and *Next Sentence Prediction* (NSP). After pretraining on large unlabeled corpora, a small task-specific head is fine-tuned. BERT established state-of-the-art results on GLUE and other understanding benchmarks [22].

GPT series (2018 onward): The *Generative Pretrained Transformer* family uses a *decoder-only* architecture with next-token prediction. GPT-1 demonstrated effective transfer from unsupervised pretraining to supervised fine-tuning [23]. GPT-2 showed strong zero-shot behavior across diverse tasks [24]. GPT-3 highlighted the effect of scale: prompt-based *few-shot* and *in-context* learning achieved strong results without task-specific training [25]. Subsequent models (e.g., GPT-4) improved reliability and introduced multimodal inputs [26].

T5 (2019): The *Text-to-Text Transfer Transformer* frames many NLP tasks uniformly as text-to-text and trains an encoder–decoder model on the C4 corpus. Standardizing objectives and scaling data/compute produced state-of-the-art results across summarization, question answering, classification, and translation [27].

2.3.3 Why Transformers Displaced RNNs and CNNs

Transformers became dominant within a few years due to complementary advantages. First, the absence of stepwise dependencies allows efficient parallel training on modern accelerators [21]. Second, self-attention gives each position direct access to global context, which is difficult for recurrent models in practice over long sequences. Third, the same attention–feed-forward block serves encoder-only (BERT), decoder-only (GPT), and encoder–decoder (T5) designs, reducing engineering complexity. These factors, combined with consistent empirical gains on translation and language understanding benchmarks [21, 22, 27], motivated widespread adoption.

While it is true that Large Transformer models raise questions about bias and fairness (due to patterns in training data), environmental cost (from large-scale training), and potential misuse (e.g., convincing synthetic text), at the same time, they enable beneficial applications in accessibility, education, and scientific assistance. Ongoing work in data governance, evaluation, and safety aims to strengthen benefits while reducing harms.

Chapter 3

Recent Technical Evolution

3.0.1 Fine-Tuning Innovations and Instruction Alignment

The period 2022–2023 delivered major advances in how large language models (LLMs) are adapted and aligned for downstream use. Because full end-to-end fine-tuning of multi-billion-parameter models is often impractical, *parameter-efficient fine-tuning* (PEFT) methods emerged to enable specialization with far fewer trainable weights. Among these, **Low-Rank Adaptation (LoRA)** inserts small low-rank matrices into selected weight projections; during training the base model is frozen and only the adapter parameters are updated. This yields quality close to full fine-tuning while reducing memory, compute, and storage requirements by orders of magnitude, and it supports serving many task-specific variants by swapping adapters rather than duplicating the full model [28]. LoRA builds on adapter-style methods and complements techniques such as prefix/prompt-tuning and bias-only tuning (e.g., *BitFit*). Subsequent refinements (e.g., quantization-aware approaches like "QLoRA") further cut hardware requirements by storing the frozen base in low-bit precision while training a small set of adapters.

In parallel, **instruction tuning** became central to aligning models with user intent. Here, a pretrained LLM is fine-tuned on collections of task prompts and responses—sourced from human demonstrations or curated corpora—so it can generalize to novel instructions without per-task supervision. Early exemplars (e.g., FLAN and T0) demonstrated broad gains across zero- and few-shot tasks, and the approach was popularized for assistants by InstructGPT [29]. For interactive systems, instruction tuning is commonly followed by **preference optimization**: training a reward model from human (or AI-assisted) comparisons and optimizing the policy to prefer helpful, honest, and harmless outputs. Classical pipelines apply reinforcement learning from human feedback

(RLHF) with a policy-gradient optimizer, while newer direct preference methods (e.g., DPO-style objectives) optimize to favor "chosen" over "rejected" responses without an explicit RL loop. By late 2022, instruction-tuned and preference-aligned models (e.g., ChatGPT-style systems) had largely superseded raw pretrained models in user-facing applications due to their superior robustness, compliance, and conversational usability.

These advances are not without limits. PEFT reduces adaptation cost but does not eliminate the expense of pretraining; merging or composing adapters can introduce trade-offs; and license terms may restrict certain use cases. For alignment, performance depends critically on the breadth and quality of instruction and preference data; models can over-optimize to narrow preferences, exhibit jailbreak vulnerabilities, or suffer capability/harmlessness trade-offs. Ongoing work therefore couples PEFT with better data curation, safety tuning, and evaluation in order to sustain practical gains while mitigating misuse and degradation.

3.0.2 Open-Weight Models and Democratization

Early state-of-the-art LLMs were concentrated within a few research labs. In 2023, however, a surge of open-weight alternatives broadened who could study, deploy, and adapt large models. *Open-weight* means that the model parameters are available for download and local use, even if the full training data, code, or license terms are not strictly "open-source" in the OSI sense.

In February 2023, Meta released *LLaMA* (7B–65B) for research use; notably, the 13B model approached GPT-3 on several public benchmarks. In March, an unauthorized leak of the weights enabled developers to run and fine-tune a high-performing LLM locally. Instruction-tuned derivatives (e.g., Alpaca, Vicuna, Koala, Dolly) appeared within weeks, often via distillation from ChatGPT outputs. By late 2023, Meta reported over 7,000 LLaMA variants on Hugging Face, a dynamic sometimes referred to as the "LLaMA effect" [30].

Meta's subsequent *Llama 2* (July 2023) adopted a more permissive license for commercial use, reinforcing the view of foundation models as shared infrastructure. Commentators argued that Llama 2 crossed a capability threshold for broadly usable open models. In parallel, other releases—such as Falcon 40B/180B (TII), MPT-7B/30B (MosaicML), and community or startup efforts like Mistral 7B and larger Vicuna variants—expanded the ecosystem [30]. The analogy to Linux is apt: a growing "distribution" of downloadable weights that developers can adapt without depending solely on closed APIs.

What did this democratize?

1. **Experimentation:** 7B–13B models can run on a single high-end consumer GPU, and low-bit quantization enables CPU or laptop inference.
2. **Customization:** Organizations can fine-tune on private data and deploy on-premises, which is important in regulated sectors such as healthcare and law.
3. **Market entry:** Many startups now build on open-weight bases rather than training from scratch.

Limits: Most open models do not include fully open training data or recipes; pretraining still requires substantial compute; and licenses vary and are not always OSI-open-source. Policymakers also warn about misuse risks (e.g., spam, misinformation). Nevertheless, by 2025 open-weight models appear durable as a counterbalance to proprietary systems, with community iteration contributing to cumulative progress [30].

3.0.3 Retrieval-Augmented Generation and Long-Term Memory

A significant limitation of standalone LLMs is their *fixed knowledge* – they only know information present in their training data (often cut off at a certain date), and they have a limited context window for prompts. This led to the rise of **Retrieval-Augmented Generation (RAG)** techniques, which marry LLMs with external knowledge sources. In RAG, the model is augmented by a retrieval system (such as a vector database or search engine) that can fetch relevant documents, facts, or data from outside the model's parameters, based on the user query. The model then conditions its generation on both its internal knowledge and the retrieved content. This approach boosts factual accuracy and keeps responses up-to-date. NVIDIA's AI blog succinctly defines RAG as "a technique for enhancing the accuracy and reliability of generative AI models with information fetched from specific and relevant data sources." [31] By pulling in reference text at query time, an LLM can provide grounded answers with direct evidence, rather than relying purely on its internal memory (which often *hallucinates* plausible-sounding but incorrect details). Indeed, retrieval augmentation addresses the hallucination problem by allowing models to cite sources: "RAG gives models sources they can cite, like footnotes... so users can check any claims. That builds trust. It also reduces the possibility that a model will give a very plausible but incorrect answer, a phenomenon called hallucination." [31]

This capability is critical for enterprise adoption, where *factual correctness* is required. Companies began deploying RAG-based assistants that could safely answer questions using proprietary knowledge bases, without risking the model's imagination to fill gaps.

To enable RAG, the use of **embeddings and vector databases** became standard. Models like OpenAI's and open-source **SentenceTransformers** can encode text into high-dimensional vectors; databases such as **Pinecone, Weaviate, Chroma**, or simply FAISS indexes, allow similarity search over these embeddings. By 2023, interest in vector databases had exploded – multiple startups in this space secured major funding rounds (e.g., Pinecone $100M and Weaviate $50M in April 2023 [32]) – reflecting their newfound importance as "memory" for LLMs. The typical architecture of an LLM application in 2023 included an embedding-based retrieval layer: a user query triggers a search in a document corpus for relevant text, which is then appended to the prompt for the LLM to use in formulating its answer. This *retrieval chaining* can be iterative: the model's initial answer can spawn follow-up searches for more information, in a loop, until it has sufficient grounding to answer. Such chaining requires the model to plan what to search for – often implemented via an agent loop (described below in the context of ReAct). The result is a system that can handle queries far beyond the training data and with an effective working memory much larger than the model's context window.

Closely related are efforts to extend **long-term memory** in interactive dialogues. Techniques like *conversational memory buffers* (storing past conversation turns and summarizing older context) were integrated into chat frameworks. Some systems maintained a **rolling summary** of the conversation that gets fed to the model along with the latest query, preventing context window overflow. More advanced approaches log conversation facts into a vector store, enabling the assistant to lookup past knowledge (for example, a personal AI that remembers user preferences mentioned weeks ago). Research prototypes like *LongMem* proposed training LLMs with explicit long-term memory modules [33]. By 2025, we see specialized memory-augmented architectures and the use of databases to maintain persistent state for agents (e.g., an agent recalling what it did yesterday). This significantly improves the continuity and personalization of AI assistants. In summary, RAG and memory systems became indispensable parts of the LLM toolkit between 2022 and 2025 – mitigating hallucination, keeping answers current, and giving models something akin to an external hard drive for knowledge.

3.0.4 Reasoning and Tool-Use Techniques: The raise of AI Agents

Despite their remarkable generative abilities, base Transformer models do not consistently demonstrate *robust reasoning or planning* out of the box. To address this limitation, researchers developed a series of prompting strategies and algorithmic enhancements aimed at guiding LLMs toward more systematic, step-by-step problem solving—effectively augmenting their *cognitive toolkit*.

A pivotal early contribution was **chain-of-thought (CoT) prompting**, introduced in 2022. CoT encourages models to articulate intermediate reasoning steps before producing a final answer. Simple cues such as "Let's think step by step" significantly improved performance on math word problems and commonsense reasoning tasks [34]. This insight—that prompting a model to *externalize its reasoning* enhances accuracy—became foundational to subsequent methods.

Building upon CoT, researchers at Google proposed **ReAct** in late 2022, which integrates *Reasoning* and *Acting*. In this paradigm, the model alternates between internal reasoning (generating "thoughts") and external actions (e.g., issuing a search query or calling an API), using the resulting observations to inform subsequent reasoning. A typical ReAct prompt might generate: "I should check who won the 2010 World Cup" (thought), followed by "Search[2010 World Cup winner]" (action). This cycle continues until the task is resolved. ReAct was seminal in demonstrating that LLMs could function as **agentic systems**, interacting with external tools and environments rather than operating in isolation. By coupling reasoning traces with tool use, ReAct agents achieved more factual and verifiable outputs, particularly on *knowledge-intensive* and *decision-making* tasks [34]. Nearly all subsequent tool-using frameworks can be traced back to this formulation.

A further advance came with the introduction of **Tree-of-Thoughts (ToT)** in 2023. Whereas CoT follows a single linear reasoning trajectory, ToT enables the model to explore **multiple reasoning branches** in parallel and evaluate intermediate results before committing to a path [35]. This approach mimics human strategies such as backtracking and considering alternative hypotheses. For example, a model might generate several potential next steps, then either self-evaluate or rely on external heuristics to select the most promising branch for further exploration [36]. Variants of ToT, together with techniques like **self-consistency** (sampling multiple reasoning paths and selecting the majority answer), substantially improved performance on logically complex tasks by 2025.

Alongside these core paradigms, researchers developed a wide array of specialized prompting techniques. **Self-critique and reflection** prompts instruct the model to review and correct its outputs. **Role-playing prompts** frame the model as a domain expert, influencing its reasoning style. **Context distillation** has the model compress lengthy documents into reduced contexts for subsequent processing. Collectively, these can be understood as *reasoning tools*: structured patterns that extend the raw generative capacity of LLMs.

Importantly, these advances quickly moved from research into practice. Microsoft's 2023 **Toolformer** demonstrated that a model could be fine-tuned to autonomously decide when to invoke APIs (e.g., weather services or calculators), inserting function-calling tokens directly into its output sequence. The ReAct paradigm itself was rapidly adopted into open-source frameworks such as LangChain, and into commercial APIs such as OpenAI's function-calling interface, which internally follows a thought→action→observation loop. Community-driven methods like *Reflexion*, which prompt the model to review its own answers and revise iteratively, further exemplify the translation of academic insights into applied agent systems.

A minimal ReAct-style loop: *Controller pseudocode (single-agent):*

```
repeat until done:
  thought_t  = LLM.observe(state).think()
  if needs_tool(thought_t):
     action_t = select_tool(thought_t)
     obs_t    = tool[action_t].execute(params)
     state   += (action_t, obs_t)
  else:
     answer   = LLM.finalize(state)
     return answer
```

In summary, reasoning and tool-use techniques such as CoT, ReAct, and ToT represent a crucial layer of innovation, transforming LLMs from static text generators into interactive, problem-solving agents. These methods have established the foundation upon which modern agent frameworks and applications are built.

3.0.5 Agentic Frameworks and Tool-Use Ecosystems

The convergence of the aforementioned capabilities has given rise to a new ecosystem of **LLM-based agents and frameworks**, designed to build upon foundation models. Rather than employing an LLM as a solitary oracle, developers in 2023 increasingly orchestrated LLMs into structured pipelines and even multi-agent collaborations to support extended and open-ended tasks. Several key frameworks illustrate this evolution.

LangChain: First released in late 2022, LangChain established itself as a modular interface for chaining prompts, models, and utilities such as memory, external tools, and document retrieval. By mid-2023, it had achieved extraordinary adoption, becoming "the single fastest-growing open-source project on GitHub" [37]. LangChain introduced abstractions including *Chains* (sequences of LLM calls and logic), *Agents* (LLMs that select tools via ReAct-style reasoning), and integrations with a broad catalog of tools (e.g., web search, databases, calculators, Python REPLs). Its ability to "abstract prompt templating, document retrieval, tool calling, and agent execution" with minimal new code, combined with a large community of contributors, established it as a de facto standard for building with LLMs [38].

AutoGPT and BabyAGI: In early 2023, open-source projects such as **AutoGPT** and **BabyAGI** popularized the concept of *fully autonomous AI agents*. AutoGPT linked GPT-4 to a recursive loop for task generation, web browsing, code execution, and subtasks spawning, achieving over 100k GitHub stars within weeks of release [39]. It combined ReAct-style reasoning with long-term memory and goal-driven planning, allowing a high-level user objective to be decomposed into actionable plans. AutoGPT has been described as "the first open-source project to implement an autonomous task execution model powered by GPT-4... and inspired a wave of derivative projects" [37]. BabyAGI explored similar ideas, with emphasis on task generation and prioritization.

OpenAgents and Multi-Agent Orchestration: Building on these foundations, frameworks enabling *multi-agent collaboration* began to emerge. **OpenAgents** introduced protocols for large-scale agent communication, envisioning decentralized networks where agents discover, interact, and coordinate without central control [40]. Other initiatives adopted different approaches: **CrewAI** allows lightweight instantiation of "crews" of specialized agents with shared goals [37, 41]; Microsoft's **AutoGen** offers utilities for multi-agent conversations, reflection loops, and ReAct-style orchestration [42]; **LangGraph**

employs a graphical approach for workflow design; and **Camel** experiments with role-play between paired agents as a mechanism for collaborative problem solving. Together, these frameworks demonstrate the range of strategies available for coordinating agent interactions.

Tools, Plugins, and Infrastructure. Extensibility has also been advanced through the development of tools and plugins. For example, the open-source project often described as **HuggingGPT** positions an LLM as a coordinator of subtasks delegated to specialist machine learning models on Hugging Face, thereby expanding capabilities beyond a single foundation model [41]. In parallel, scalable inference infrastructures have gained prominence. UC Berkeley's **vLLM** introduced *PagedAttention*, delivering substantial throughput improvements compared with standard Hugging Face Transformers [43]. Hugging Face's **Text Generation Inference (TGI)** stack, together with quantization techniques such as QLoRA, has made it feasible even for individuals to deploy sophisticated LLM systems on modest hardware. Meanwhile, **OpenRouter** emerged as a unified routing service, aggregating more than 400 models from over 60 providers, and offering normalized APIs, intelligent routing, caching, and monitoring [44].

Overall, these frameworks and infrastructures reflect a broader trajectory toward openness, modularity, and scalability. The ecosystem is shifting away from monolithic deployments toward dynamic networks of cooperating agents, supported by increasingly flexible orchestration layers and optimized inference backends.

3.0.6 Design Patterns and Architectures for Agent Systems

Core components: Practical agent stacks typically decompose into several fundamental components that together enable robust decision-making and execution:

1. **Controller (Planner/Decider):** Implements reasoning paradigms such as Chain-of-Thought (CoT), ReAct, or Tree-of-Thought (ToT). It selects appropriate tools and maintains the agent's decision policy.

2. **Tools (Actuators):** Typed interfaces for performing actions (e.g., search, code execution, database queries, or API calls), frequently exposed via structured function-calling schemas.

3. **Memory:** Encompasses *short-term* scratchpads for intermediate reasoning, *episodic* logs of prior steps, and *semantic* long-term stores (often realized via vector databases in retrieval-augmented generation).

4. **Retrieval:** Document and knowledge access for grounding responses. Effective chunking, indexing, and re-ranking are essential for performance.

5. **Execution Sandbox:** Isolated runtime for code and tool invocation, typically incorporating resource limits and auditing mechanisms to manage risk.

6. **Orchestration and Observability:** Infrastructure for tracing, step-level logging, caching, and evaluation hooks, providing both reliability and cost control.

Common topologies. While implementations vary, several recurring architectural topologies can be identified:

- *Single-agent, tool-using (ReAct):* A single LLM alternates between reasoning, action, and observation steps until task completion.

- *Hierarchical planning:* A higher-level *manager* agent decomposes a goal into subgoals, which are executed by *worker* agents with access to specialized tools.

- *Multi-agent collaboration:* Multiple peer agents assume distinct roles (e.g., researcher, coder, critic) and interact via shared memory or a message bus. Coordination may emerge through reflection, voting, or consensus protocols.

- *RAG-centric pipelines:* Deterministic retrievers provide input to a generator. The agent dynamically decides whether to refine queries, branch reasoning paths (e.g., ToT), or summarize before producing an output.

These design patterns illustrate the diversity of strategies for structuring agentic systems. Each topology embodies different trade-offs in transparency, scalability, and control, reflecting the broader evolution of agent architectures from simple tool use toward increasingly modular and collaborative paradigms.

3.0.7 Integration into Products and Systems

By 2023, large language models (LLMs) transitioned from research prototypes to widely adopted applications, often branded as “AI assistants” or “copilots.” A prominent early

domain was **web search**. Microsoft's Bing integrated a GPT-4-powered chatbot with citations, while Google introduced **Bard** and subsequently the **Search Generative Experience (SGE)**. Despite early challenges—such as the widely publicized "eat one rock per day" hallucination [45, 46]—providers iterated rapidly on factuality, safety guardrails, and user experience. By 2025, conversational AI had become a standard component of search interfaces.

Adoption soon extended beyond search into productivity software. Microsoft launched **Copilot for Microsoft 365**; Google embedded AI-driven writing support; and **GitHub Copilot** matured into a staple for software development. A broader wave of "copilot for X" products followed, spanning creative tools (Adobe Firefly), knowledge management (Notion AI), communication (Slack AI), and language support (GrammarlyGO). Enterprises similarly integrated LLMs, exemplified by Salesforce's **Einstein GPT** [47].

Supporting these deployments required substantial infrastructure. Early analyses estimated that serving ChatGPT cost approximately $700k per day in early 2023 [48]. Cloud providers soon introduced specialized hosting options, and efficiency gains enabled significant cost reductions (OpenAI, for instance, announced major price cuts by late 2024 [49]). Concurrently, serving ecosystems such as **vLLM**, **Text Generation Inference (TGI)**, and **OpenRouter** made it feasible to deliver multi-tool, multi-model agent systems at scale [43, 44].

Taken together, these developments underscore how modern agent systems are enabled by the synthesis of (i) *reasoning scaffolds* (e.g., CoT, ReAct, ToT, reflection), (ii) *tool interfaces* with typed contracts and retrieval for grounding, and (iii) *serving infrastructure* that ensures fast, reliable, and cost-effective operation. This layered stack—algorithms, tools, and systems—underpins today's LLM applications and delineates the design space for future research and productization.

3.0.8 Generative AI in Enterprise and SaaS Platforms

By this period, generative AI had permeated nearly every software-as-a-service (SaaS) product aimed at knowledge workers. Application domains spanned customer service (e.g., Zoom AI Companion), marketing (Canva, HubSpot), software development and DevOps (Atlassian AI, CodeWhisperer), education (Duolingo Max, Khanmigo), and numerous domain-specific copilots in law, finance, and healthcare. These integrations typically relied on fine-tuning or prompting with proprietary domain data. Regulatory and compliance requirements simultaneously drove demand for deployment models that ensured data isolation, including *on-premises* open-weight stacks and cloud-based offerings

such as Azure OpenAI and Anthropic on AWS. Hugging Face Hub emerged as a central model repository, while containerized serving and routing became standard operational practices. Efficiency remained paramount: enterprises prioritized lower latency and higher throughput, while steadily declining costs were described as *LLMflation* [49, 50].

At the same time, the discourse around "agents" grew increasingly ambiguous. The label was widely applied to products that lacked the defining characteristics of autonomous or semi-autonomous agents. Marketing campaigns amplified this ambiguity, branding simple automation or templated workflows as "agentic" systems. As a result, customers, particularly those without technical expertise, often assumed that "agentic AI" implied fully autonomous capabilities. Practitioners, by contrast, recognized the substantial technical and safety constraints. This divergence created an expectation gap: a marketplace characterized by genuine innovation, but also by oversold promises and frequent misalignment between user expectations and delivered functionality.

In summary, enterprise adoption of generative AI combined real advances in capability and infrastructure with hype-driven market dynamics. While the integration of LLMs into SaaS platforms has reshaped software and business processes, it has also highlighted the importance of precision in terminology and transparency in communicating system limitations.

3.0.9 Evaluation, Safety, and Limitations

Measuring performance: Evaluation of agentic systems extends beyond static benchmarks to encompass metrics that more faithfully capture real-world task performance. At the *task level*, researchers often measure exact-match accuracy or overall success rate on end-to-end objectives. Complementary *process metrics*—such as the number of tool invocations or the breadth of search branches explored—provide insight into agent reasoning strategies. In addition, *efficiency metrics* (e.g., latency, token consumption, and compute expenditure) allow comparison across system designs. Ablation studies, in which specific components such as reflection, tree-of-thought (ToT) exploration, or retrieval augmentation are selectively disabled, are particularly valuable for attributing performance gains to specific architectural choices.

Reliability and security: Despite their capabilities, agentic systems remain vulnerable to distinct classes of risk. These include the generation of *hallucinated tools or outputs*, *prompt injection* from untrusted inputs, *excessive tool use* leading to increased costs and an expanded error surface, and *insufficient sandboxing* when executing generated code.

Common mitigations span both technical and governance strategies: typed tool contracts and pre-execution policy checks ensure adherence to specification; rate limits and least-privilege credentials constrain exposure; reproducible execution traces facilitate auditing; and post-hoc validation (e.g., automated unit testing of generated code) provides an additional safeguard against unsafe or erroneous outputs.

Human factors: The design of human–AI interaction also shapes safety outcomes. For instance, exposing the agent's internal reasoning can enhance transparency and facilitate debugging, yet such verbosity may simultaneously introduce privacy or safety risks if sensitive intermediate details are revealed. As a result, many production deployments conceal detailed reasoning traces, instead returning concise, sourced, and user-oriented answers. This trade-off is particularly evident in consumer-facing domains such as search, productivity, and coding assistance, where usability and safety must be carefully balanced

In summary, the evaluation and governance of agentic systems must be understood as multidimensional, integrating empirical measurement, technical safeguards, and careful attention to human factors. Only through such a holistic approach can these systems be deployed reliably and responsibly in enterprise and consumer settings alike[45–49].

3.0.10 Privacy

The deployment of intelligent agents and large language models (LLMs) in cloud environments introduces significant and multifaceted privacy risks, spanning both training and inference phases. In practical use, such systems *process* user prompts and often integrate enterprise-specific data through retrieval augmentation or fine-tuning. These workflows generate exposure risks if inputs, contextual augmentations, or outputs are logged, cached, or persisted without robust controls. Beyond operational leakage, large models have been empirically shown to memorize and reproduce rare training examples, including sensitive personal data [45]. This dual vector—operational handling of sensitive inputs and structural risks of memorization—renders privacy a primary concern in the adoption of LLM-driven systems.

Cloud service defaults in enterprise offerings. Major cloud providers have articulated distinct privacy commitments and technical controls within their managed LLM platforms:

- **Microsoft Azure OpenAI.** Azure OpenAI operates entirely within Microsoft's Azure infrastructure. Prompts and outputs are *not* shared with OpenAI and are *not* incorporated into either Microsoft's or OpenAI's foundation models. Customers may constrain processing to specific geographic regions (Global or DataZone). The service integrates abuse monitoring and content filtering; logging configurations can be modified by application. Persistent data is encrypted at rest by default (AES-256), with support for customer-managed keys (CMK) [51].

- **AWS Bedrock.** By default, Bedrock does *not* log prompts or completions, and such data is neither shared with model providers nor repurposed for training. Providers are segregated via deployment accounts. Encryption is enforced in transit and at rest, with additional protections available through AWS KMS and private networking.

- **Google Cloud Vertex AI (Gemini).** Google prohibits the use of customer data for training without explicit permission (the "Training Restriction"). For Google-managed models, input caching is enabled by default (to optimize latency) and may retain data for up to 24 hours unless Zero Data Retention (ZDR) is activated. Certain grounding features (e.g., Search or Maps integrations) store prompts and context for up to 30 days. Vertex AI resources also support CMEK for encryption.

Mitigations and governance. At the application layer, enterprises must align cloud-native controls (data residency, CMEK/KMS, private networking, configurable logging) with internal governance practices, including least-privilege access, classification and data loss prevention (DLP) filters on prompts and retrieved content, and rigorously defined retention and deletion policies. For customization scenarios, privacy-preserving learning approaches such as differential privacy (e.g., DP-SGD) and federated learning provide additional protection, albeit with trade-offs in utility and system complexity [51, 52]. At an organizational level, regulatory frameworks increasingly mandate structured governance of data flows, retention periods, and incident response processes.

In summary, privacy emerges not as a peripheral concern but as a structural dimension of deploying agentic systems in enterprise SaaS. Ensuring privacy requires a defense-in-depth strategy, combining technical safeguards, carefully tuned cloud configurations, and rigorous compliance and policy frameworks.

Chapter 4

Presentation of the Full-Stack Application

This chapter presents the architecture and implementation of the developed full-stack application. The system integrates a modern *FastAPI*-based backend with a *React*-based frontend and includes dedicated data pipelines. The application is designed for AI-assisted workflows in the domains of *Audit* and *Investments*, making it a suitable case study for demonstrating agentic LLM architectures integration within enterprise applications. It is important to note that these domains are not the main focus in themselves—their choice stems from my own professional experience. They serve primarily as a concrete context and an illustrative excuse to showcase the broader potential of AI and its integration into real-world systems.

4.1 General Overview

The application adopts a classical **frontend–backend** separation:

- The **backend** is implemented using *FastAPI*. It exposes RESTful APIs, integrates with external services such as Azure OpenAI and LangChain, and manages persistent storage through *SQLite* databases and *ChromaDB* vector indices.
- The **frontend** is developed in *React* with *Vite*, *TypeScript*, and *TailwindCSS* (extended with the `shadcn/ui` library). It provides interactive user interfaces for querying the backend and visualising retrieved data.

- A set of **data generation pipelines** support the creation of synthetic datasets, indexing of documents, population of the investment database, and structured ingestion of complaint data.

The combination of these components enables the construction of intelligent, data-driven interfaces supported by LLM reasoning.

4.2 System Architecture

The backend is structured into several modules:

- **Entrypoint:** `backend/main.py`, which creates the FastAPI app, registers routers, and configures CORS policies for frontend access.
- **Routers:** Located in `src/router/`, these define the HTTP endpoints. They include:
  - `chat.py` (general chat interface),
  - `audit.py` and `audit_documents.py` (Audit domain),
  - `investments.py` and `investment_chat.py` (Investments domain),
  - `graph_chat.py` (graph-based reasoning for investments),
  - `complaint_processing.py` (complaints domain).
- **Services and Agents:** Implemented in `src/services/`, these provide the business logic. They interact with databases, vector stores, or LLMs (via Azure OpenAI).
- **Core Integration:** The module `src/core/Azure/` defines connections to Azure services for embeddings, parsing, and LLM access.
- **Data Management:** The `backend/data/` folder stores generated markdown/PDFs, SQLite databases, and ChromaDB indices for each domain.

A typical backend workflow proceeds as follows: an HTTP request arrives at a router $\rightarrow$ the router delegates to a service/agent $\rightarrow$ the service queries databases, embeddings, or LLMs $\rightarrow$ the response is returned as JSON.

The frontend is implemented with the following structure:

- **API Layer:** `src/services/api.ts` acts as a central client for REST API calls. It maps frontend methods to backend endpoints, handling requests, responses, and errors.

- **Pages and Components:** Organised by domain:
  - Audit: dedicated pages and components for chat, filters, and document inspection.
  - Investments: database viewers, upload forms, and investment chat.
- **UI Toolkit:** TailwindCSS and `shadcn/ui` provide reusable, responsive design elements.

The frontend flow can be summarised as follows: user interaction → API service call → HTTP request to backend → backend processing → JSON response → UI state update.

To support the application's three domains, specific pipelines were developed:

- **Audit:** Synthetic reports are generated in Markdown, converted into PDF, and indexed in ChromaDB for semantic retrieval.
- **Investments:** Client and product data are stored in SQLite, enriched with additional documents, and indexed for RAG (retrieval-augmented generation).

4.3 Deployment and Setup

The project requires:

- Python 3.11+ (with FastAPI and related dependencies),
- Node.js 18+ (for React frontend),
- Azure OpenAI and Cognitive Services accounts (for LLMs, embeddings, and document parsing).

Both backend and frontend can be started independently in development mode. By default:

- **Backend:** can be launched by navigating to the `backend/` directory, synchronising the environment using `uv sync`, and executing

```
uv run uvicorn main:app --reload
```

- **Frontend:** can be started from the `frontend/` directory with the command

```
npm run dev
```

Then, by opening `http://localhost:8080/`, the landing page is displayed:

Figure 4.1: Landing Page

From the landing page, it is possible to choose between the two different applications that will be described in the following sections.

Chapter 5

Audit-Inspired Application: Agentic RAG for Multi-Dimensional Document Information Retrieval

5.1 Introduction to Audit-related application and Problem Setting

Audit-related examinations often require employees to find very specific details hidden within large collections of reports and documents. In an ideal setup, this information would be stored in structured databases, making it easy to retrieve with a simple SQL query. In reality, much of it remains buried in long, unstructured reports that mix narrative text with tables, figures, and raw data originally meant for structured storage. Extracting what is needed from these sources becomes a slow, inefficient, and often frustrating task. At the same time, many sales employees still have limited data literacy, and not everyone is comfortable working with SQL, which adds to the challenge.

Large Language Models (LLMs) offer a promising solution: they can process unstructured text and assist in retrieving relevant information quickly and intelligently. Yet, this approach is not without challenges. Two of the most important difficulties concern the temporal nature of documents and the different perspectives of departments.

One of the central challenges in this context is **temporal ambiguity**. Audit documents are typically produced on a recurring basis, such as annually or quarterly, often by the same department. Although the textual content may address similar themes, such as a company's financial standing or reliability, the underlying figures and commentary evolve

over time. Standard RAG-enhanced LLMs generally retrieve information by focusing on semantic similarity between a query and the documents. However, this becomes problematic with time-dependent data. For example, years like 2023, 2024, and 2025 carry almost identical semantic weight, even though they refer to distinct periods. As a result, the system may conflate reports across different years, retrieving outdated or irrelevant documents alongside the intended ones. Consequently, when a user asks about the financial situation in Fall 2025, the model may surface older reports as well, leading to cluttered retrieval, outdated evidence, and ultimately inaccurate or confusing answers.

A second challenge relates to **departmental perspective**. Different departments can analyze the same topic from different angles. For instance, the Audit Department may focus on compliance and accuracy, while the Strategy Division may examine the same financial data for forecasting and growth opportunities. If the system cannot distinguish between these perspectives, retrieval may mix documents from different divisions, producing answers that are inconsistent or misleading from an audit standpoint.

To address these challenges, preprocessing of the data is essential before building the index database. Each document should be enriched with metadata such as the year (e.g., 2025), the semester or period (e.g., Fall, Q1), the department (e.g., Audit, Strategy), and the topic (e.g., Financial Reliability, Risk Assessment). This metadata allows the retrieval system to distinguish not only what the document discusses, but also when it was written and from whose perspective. By leveraging metadata during retrieval, agents can focus on the right subset of documents, drastically improving both accuracy and usefulness.

The project presented in this thesis simulates a simplified setting (without recreating the complexity of a real audit database) to demonstrate how these challenges can be overcome. The focus is on showing how different agentic approaches, coupled with metadata-aware indexing, can transform document retrieval in audit scenarios. Three complementary strategies are explored: a **Brute Force Approach**, with one function per retrieval topic creating a large cross-product of possibilities; a **Black Box Approach**, a fully agentic "self-query" strategy in which the agent determines how to query; and finally, a **Balanced Approach**, which combines structure and flexibility through dynamic function generation.

5.2 Frontend Notes

Upon selecting the Audit option from the landing page of the application, the following interface is displayed (Figure 5.1).

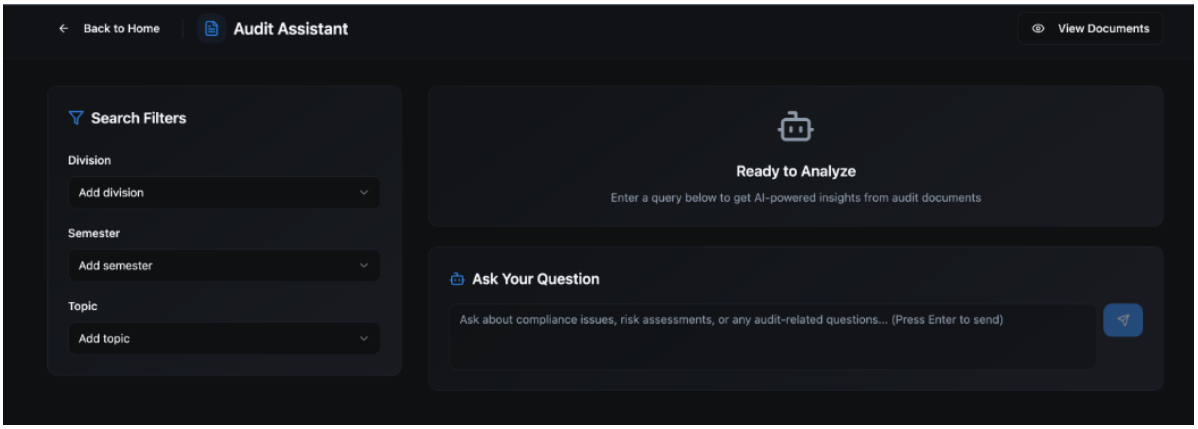


Figure 5.1: Audit application interface displayed after selection from the landing page

On the left-hand side of the interface, it is possible to select filters. These filters are used by the intelligent agent to retrieve the documents most relevant to answering the user's query. Alternatively, the same filters can be described directly in natural language as part of the question.

For example, the user may explicitly set:

- `division = X`
- `semester = Y`
- `topic = Z`

and then request: *"Summarize the report."* Alternatively, the same intent can be expressed without selecting filters, for instance: *"Summarize the report written by Division X during Semester Y about Topic Z."* It is also allowed to mix the 2 selecting some pre-determined filters and adding other in the question for the agent to determine Finally, on the top-right corner of the interface, there is a link that allows the user to directly access and view the underlying documents.

5.3 Detailed Explanation of the Data Pipeline

By clicking on the *View Documents* button in the Audit application, the user is redirected to the dedicated document page (Figure 5.2).

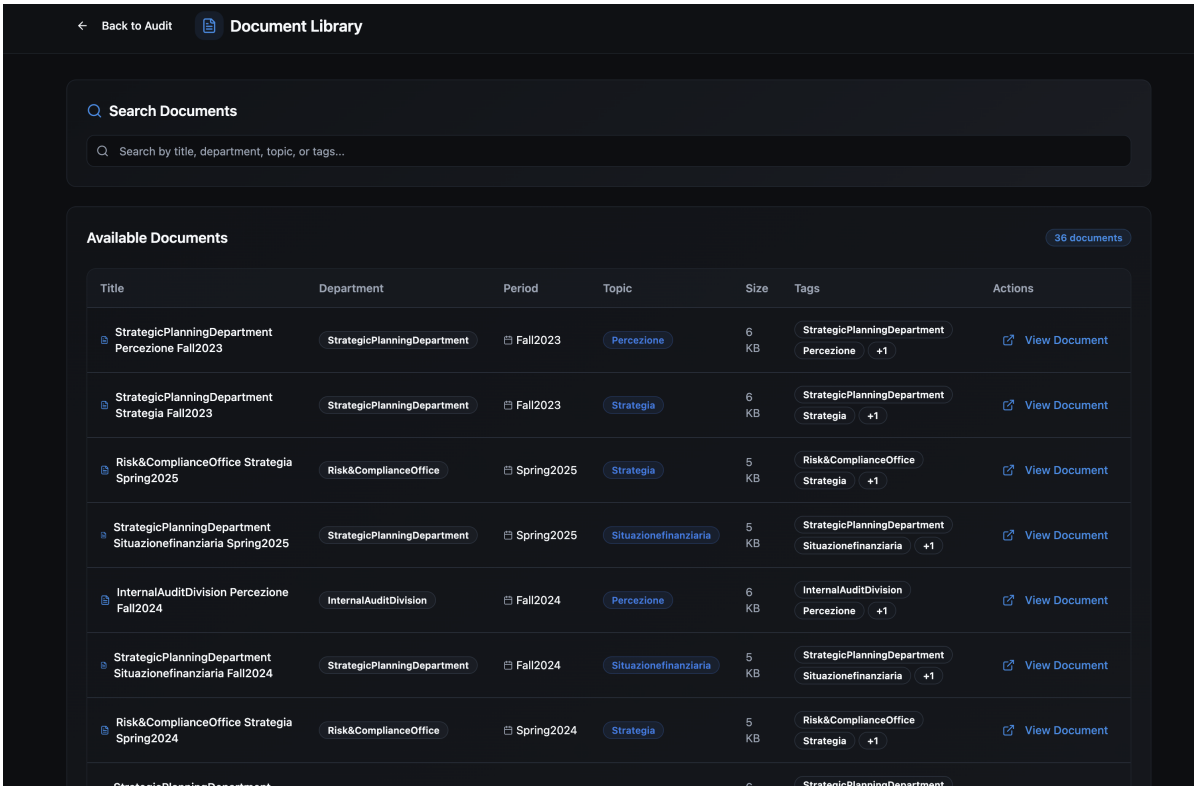


Figure 5.2: Audit documents page within the application

As described previously, all documents available in this section are artificially generated. The use of synthetic data serves two purposes. First, it prevents the inclusion of sensitive organisational material while still simulating realistic retrieval challenges such as temporal drift and department-specific perspectives. Second, it ensures that generated documents include numerical tables and KPIs, which are essential for stress-testing table handling and chunking procedures.

This section provides a detailed account of the two main components of the pipeline: first, the **data generation process**, which produces synthetic but realistic audit reports; and second, the **indexing and retrieval process**, which prepares those reports for metadata-aware retrieval-augmented generation (RAG), with particular attention given to tabular data.

5.3.1 Data Generation

The data generation script programmatically creates a corpus of audit-like reports that vary along three controlled axes:

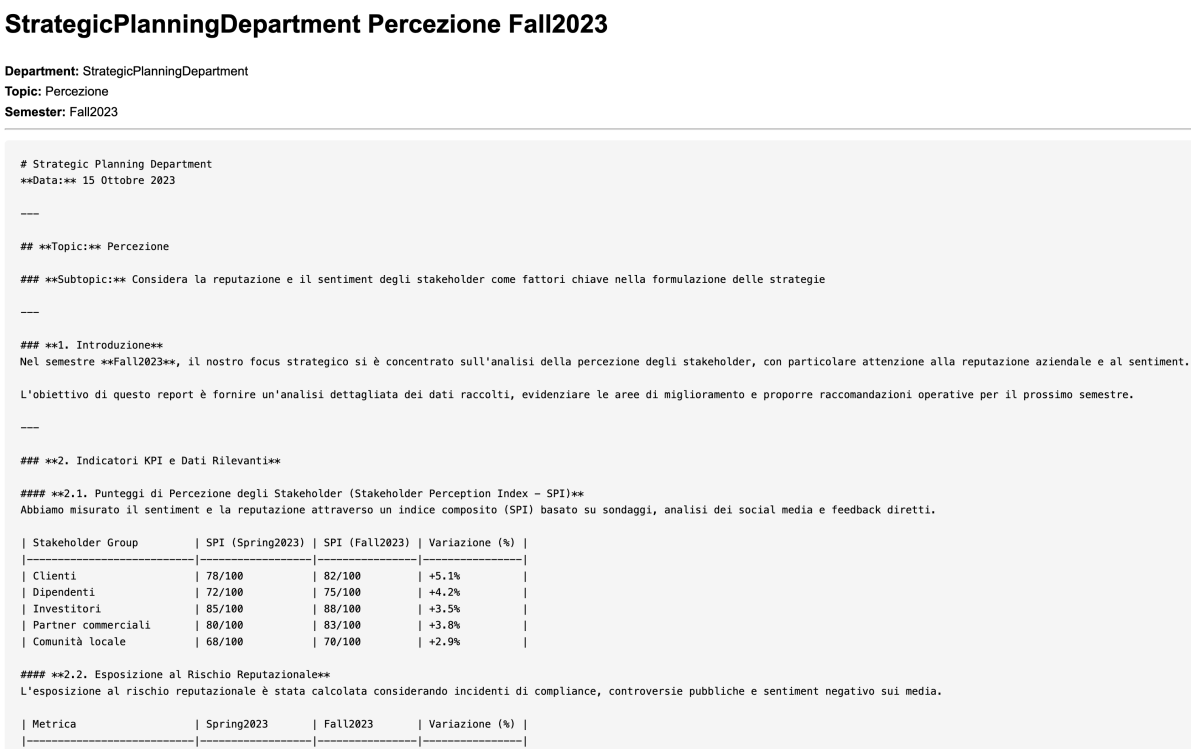
- **Semester:** e.g., Fall2023, Spring2024.
- **Department:** Internal Audit Division, Risk & Compliance Office, Strategic Planning Department.
- **Topic:** *Situazione finanziaria*, *Percezione*, *Strategia*.

For each possible combination, the script synthesises a report through an LLM using a domain-specific prompt. The prompt is instantiated with department, semester, topic, and subtopic strings, and instructs the model to generate professional, structured reports. The requirements include:

- a formal and realistic internal reporting tone,
- the inclusion of KPI-style numerical indicators,
- the presence of tables or structured lists,
- analytical commentary and actionable recommendations,
- a strict Markdown structure.

Each generated file is saved under `reports/` with the naming convention `Department_Topic_Semester.md`, which encodes key metadata directly in the filename for later parsing during indexing.

In the documents page, it is also possible to inspect the generated reports individually. By clicking on *View Document* for a single entry, the system displays the full content of that report (Figure 5.3).



StrategicPlanningDepartment Percezione Fall2023

Department: StrategicPlanningDepartment
Topic: Percezione
Semester: Fall2023

```
# Strategic Planning Department
**Data:** 15 Ottobre 2023

---

## **Topic:** Percezione

### **Subtopic:** Considera la reputazione e il sentiment degli stakeholder come fattori chiave nella formulazione delle strategie

---

### **1. Introduzione**
Nel semestre **Fall2023**, il nostro focus strategico si è concentrato sull'analisi della percezione degli stakeholder, con particolare attenzione alla reputazione aziendale e al sentiment.

L'obiettivo di questo report è fornire un'analisi dettagliata dei dati raccolti, evidenziare le aree di miglioramento e proporre raccomandazioni operative per il prossimo semestre.

---

### **2. Indicatori KPI e Dati Rilevanti**

#### **2.1. Punteggi di Percezione degli Stakeholder (Stakeholder Perception Index - SPI)**
Abbiamo misurato il sentiment e la reputazione attraverso un indice composito (SPI) basato su sondaggi, analisi dei social media e feedback diretti.

| Stakeholder Group         | SPI (Spring2023) | SPI (Fall2023) | Variazione (%) |
|---------------------------|------------------|----------------|----------------|
| Clienti                   | 78/100           | 82/100         | +5.1%          |
| Dipendenti                | 72/100           | 75/100         | +4.2%          |
| Investitori               | 85/100           | 88/100         | +3.5%          |
| Partner commerciali       | 80/100           | 83/100         | +3.8%          |
| Comunità locale           | 68/100           | 70/100         | +2.9%          |

#### **2.2. Esposizione al Rischio Reputazionale**
L'esposizione al rischio reputazionale è stata calcolata considerando incidenti di compliance, controversie pubbliche e sentiment negativo sui media.

| Metrica                   | Spring2023       | Fall2023       | Variazione (%) |
|---------------------------|------------------|----------------|----------------|
```

Figure 5.3: Example of a single audit document instance displayed within the application

It can be observed that the generated files successfully incorporate tables with numeric data, structured sections, and credible textual content, making them well-suited for our simulation scenario.

5.3.2 Indexing and retrieval

The second component is the **indexing and retrieval script**. Its role is to prepare the corpus for RAG through several steps: splitting Markdown files into semantically manageable chunks, detecting likely table chunks and summarizing them into natural language, attaching rich metadata such as division, topic, semester, and table flags, building and persisting a vector index using embeddings, and finally providing a retrieval function that returns the most relevant results.

For document splitting, the `RecursiveCharacterTextSplitter` divides each report into overlapping chunks of text (with a size of 1000 characters and an overlap of 100). The overlap preserves continuity between chunks, ensuring that concepts introduced at the end of one segment remain available at the start of the next. Importantly, the splitter respects natural boundaries such as paragraphs or headings, which maintains semantic coherence. Filename-derived metadata is then attached to every chunk, recording division,

topic, and semester. In cases where parsing fails, the values default to "Unknown," which mirrors the robustness needed in real-world corporate settings where documents are often heterogeneous.

Tables are given special treatment. A chunk is heuristically classified as a table if it contains at least six vertical bars and three or more newline characters. When detected, the table is passed to the LLM before indexing in order to generate a concise and informative natural-language summary. This enrichment step is crucial: embeddings derived directly from numeric tables are semantically poor, which makes retrieval ineffective. By contrast, the generated summaries describe what the table measures, the timeframe, entities involved, and any notable patterns or KPIs. The semantic summary is indexed as the chunk content, while the original table text is preserved in metadata and passed to the agent which crafts the final answer. This ensures that no quantitative detail is lost, while still making the table retrievable by meaning.

The metadata schema for each document chunk includes the file source, the division, topic, and semester, as well as flags for whether the chunk is a table and, if so, the full table text. This schema supports structured filtering such as "Audit reports only, Fall 2025." Both table and non-table chunks are embedded and indexed in *Chroma*, with persistence to allow reuse without reprocessing.

The retrieval flow is straightforward. A query loads the persisted index and performs similarity search, typically returning five results. Previews are shown: table content for table chunks and text snippets for others. Metadata is also displayed, allowing downstream logic to filter or re-rank results, and ensuring that queries can respect temporal and departmental constraints. If a file does not conform to naming conventions, metadata defaults to "Unknown" rather than causing failure, which provides graceful degradation.

Overall, the system combines vector similarity with metadata filtering to address the problems of temporal ambiguity and departmental perspective identified in the introduction. Possible extensions include finer-grained metadata fields, re-rankers prioritizing exact matches, improved table parsing, and mechanisms to always return the original tables for citation-ready answers.

5.4 Agentic Approaches to Retrieval

The agent for this project has been created using the LangChain framework, specifically by adopting the ReAct (Reasoning + Acting) agent architecture. This architecture allows the agent to reason step by step, make decisions on which tool to call, and integrate

retrieved information into its final response. In this section, three different approaches to answer generation are described and compared.

5.4.1 Brute Force Approach

The first strategy consists in defining a function for every possible filtering combination and providing these functions to the agent as tools. Given a user question, the agent determines which function to call in order to perform the retrieval. This method is highly effective in terms of explainability, since the reasoning process of the agent is constrained to use only the predefined filters engineered upfront. The only freedom left to the agent is the choice among these functions.

This approach is particularly useful when the scope of the search is potentially messy or confusing and we want to limit the agent's freedom in order to prevent errors. It is also effective if the queries are often similar and the filtering criteria are extremely important, because it minimizes the chance of the agent misinterpreting or neglecting them. However, the rigidity of this approach may also be a limitation in more open-ended retrieval scenarios.

5.4.2 Self-Query Approach

A second approach relies on LangChain's *self-query retriever*. This method allows the agent to interpret a natural language query and automatically construct structured metadata filters before performing the similarity search. In practice, the developer defines the metadata schema by specifying which fields are available (for example, `semester`, `division`, or `topic`) together with their possible values. Once this schema is given, the retriever uses an LLM to translate the user's question into a combination of similarity search and metadata filtering.

For example, when given a query such as:

> "Retrieve information on financial reliability in Spring 2025 from the Risk & Compliance Office."

the retriever will attempt to generate a structured filter that restricts the search to documents with metadata corresponding to
`semester=Spring2025`, `topic=FinancialReliability`, and
`division=Risk&ComplianceOffice`. This filter is then applied to the vector store, after which a similarity search is executed on the filtered subset of documents.

The advantage of this method lies in its flexibility: the user does not need to specify filters manually, and the agent autonomously decides which conditions to apply. This generally leads to strong retrieval performance and makes the approach convenient when queries vary widely.

However, a critical drawback is the lack of interpretability. Although one can usually see which documents have been retrieved, the exact filter generated by the retriever is not always visible. In addition, we cannot fully verify how the query was transformed into filtering conditions. For audit contexts, where reliability and transparency are key, this opacity reduces trust in the system and limits its applicability despite its technical efficiency.

5.4.3 Few-shots Approach (Chosen Method)

The final approach, and the one adopted in this project, strikes a balance between flexibility and explainability. Here, the agent is explicitly instructed on how to construct correct filters for the Chroma index using a few-shots approach (giving the models some examples on how to construct the filter given a question), and then allowed to generate the filter dynamically based on the user's question. In this way, it is possible to 'teach'the agent exactly how we want it to generate the filters, to inspect which filters were applied and which documents were retrieved, achieving both transparency and generalization.

Unlike the brute force approach, there is no need to define a separate function for each possible filter combination. At the same time, unlike the self-query approach, the filters used are visible and controllable. This balance makes the method particularly well-suited for audit-related applications, where traceability is essential.

5.4.4 Applications

The retrieval framework developed in this project can be applied to a variety of audit-related queries. Three common application scenarios are outlined below.

Comparing departments in a given semester and topic. One useful query is to compare reports on a specific topic in a given semester, but across different departments. For example, the question could be:

> "How did the Internal Audit Division, the Risk & Compliance Office, and the Strategic Planning Department each evaluate the topic of *Percezione* in Spring 2025?"

Such a comparison highlights differences in departmental perspectives on the same issue, which is often valuable for cross-functional analysis.

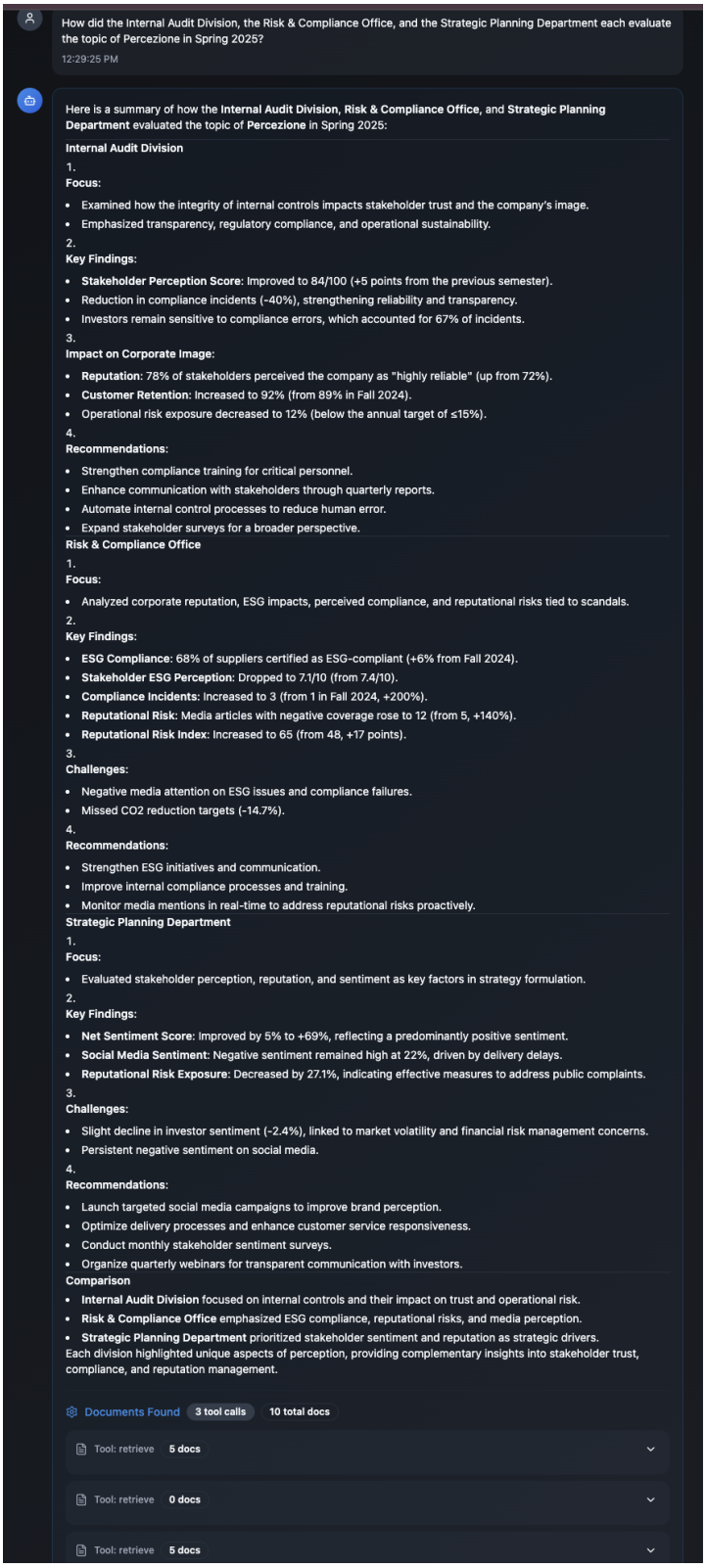


Figure 5.4: Comparison of departmental reports on the same topic within a semester

It is possible to observe that the agent invoked the retrieval function three times. Inspecting the filters used reveals that the system correctly applied the relevant department filters in each case.

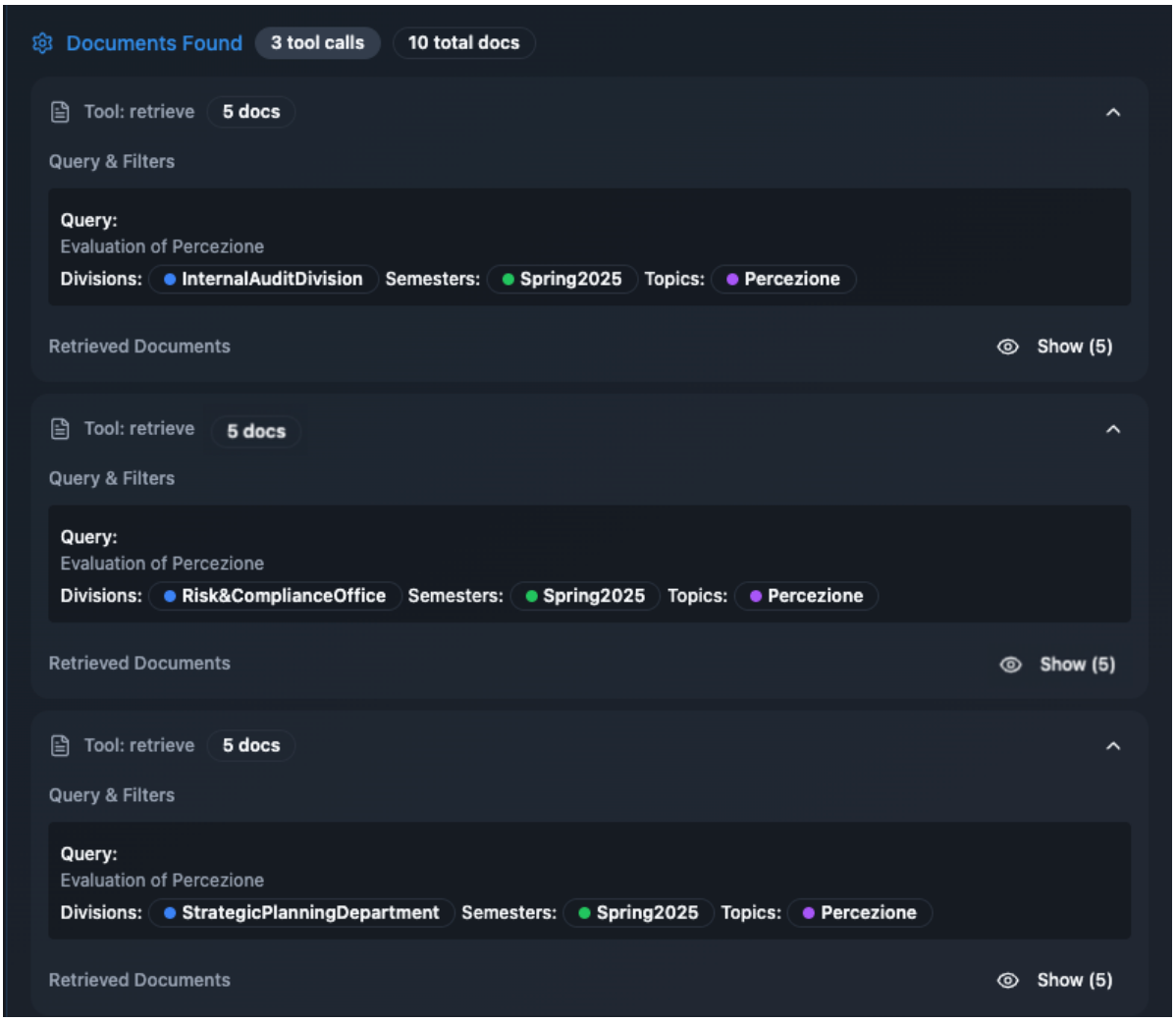


Figure 5.5: Filters applied by the retrieval function for each department

The interface also makes it possible to inspect exactly which documents were retrieved under each filter configuration clicking on the *show* button.

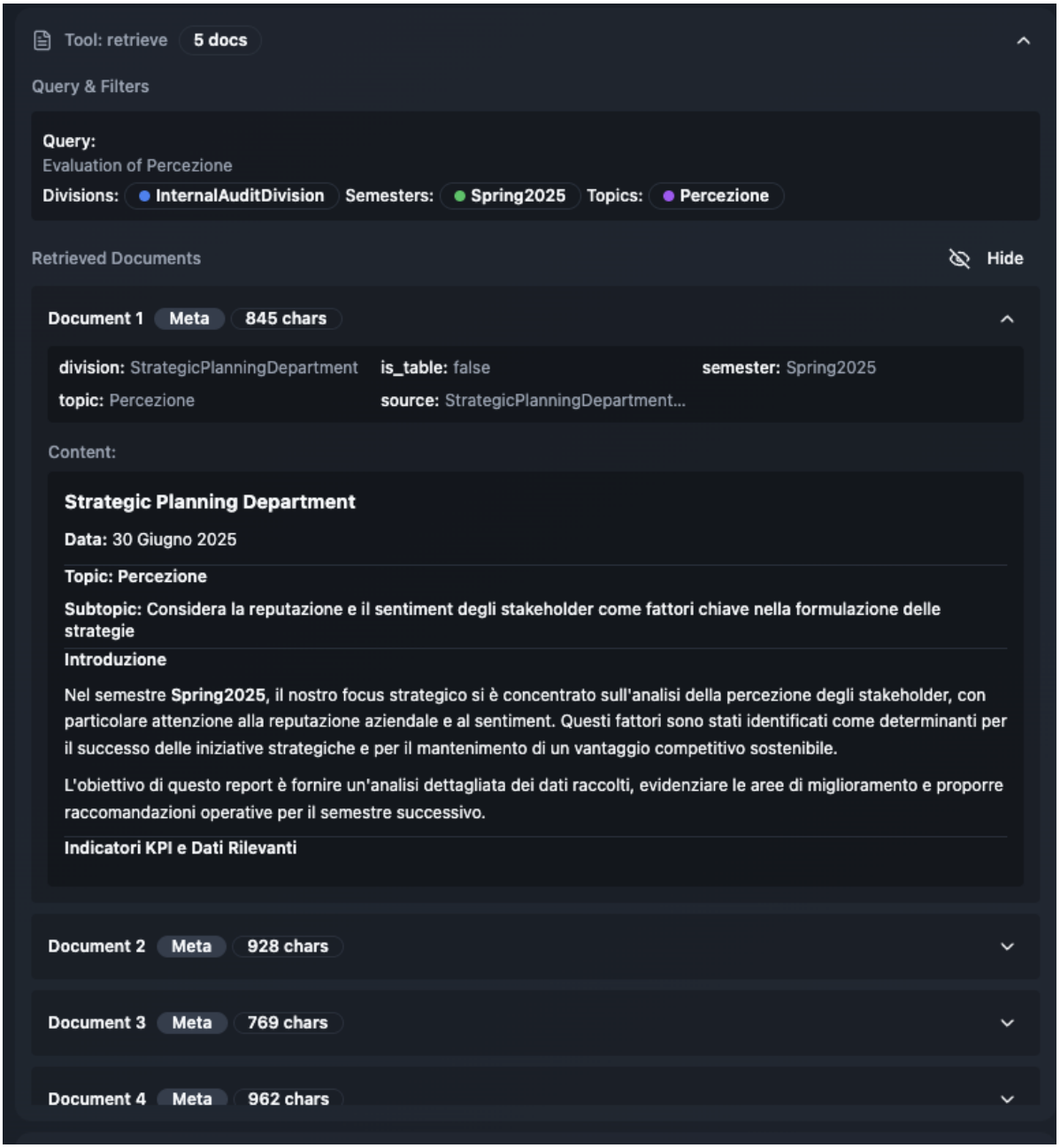


Figure 5.6: List of documents retrieved by each department filter

Retrieving focused information from one department. Another scenario is asking about a specific semester, topic, and department at once. For example:

> "What was the evaluation of the Strategic Planning Department regarding *Situazione finanziaria* in Fall 2024?"

This type of query is more narrowly scoped, focusing on a precise report. It reflects the typical needs of auditors or analysts when searching for targeted evidence.

Tracking the evolution of a topic over time. Finally, it is possible to ask how a particular topic has changed across semesters, regardless of department. An example query is:

"How did the reported *Strategia* evolve between Fall 2023 and Spring 2025?"

This temporal analysis highlights trends, progress, or emerging risks by following the same topic over multiple reporting periods.

These applications demonstrate how metadata-aware retrieval enables both fine-grained queries and broader comparative analyses, making the system flexible enough to address diverse audit use cases.

Chapter 6

Investment Management-Inspired Application: Agentic Retrieval from Structured and Unstructured Data

6.1 Introduction

The second application developed in this project lies within the domain of investment management. In practice, wealth/investment management involves assisting clients in identifying the most suitable investment or insurance options, given their patrimony, financial goals, and risk profile. Traditionally, this support is provided by a dedicated advisor who must carefully evaluate the available financial products. These products are typically described in lengthy PDF documents, which can make the process of searching for specific details both time-consuming and cognitively demanding. Ideally, the advisor's work should focus on evaluating product suitability, rather than on manually extracting information from unstructured text.

The need for streamlined information retrieval extends beyond advisors. Other stakeholders, such as data analysts and marketing professionals, can also benefit from rapid access to information. For instance, being able to quickly identify the profiles of clients who have purchased a given product, or to monitor adoption rates across the client base, can reveal important market trends. A poorly performing product may signal the need for a redesign, while identifying similar products adopted by comparable client profiles can guide personalized recommendations.

An integrated system capable of instantly retrieving this information therefore offers substantial efficiency gains. However, the challenge lies in the fact that relevant infor-

mation is often fragmented across structured databases and unstructured documents. Furthermore, the overlap between these sources is inconsistent, requiring a system that can flexibly access and synthesize both. The objective of this application is thus to deploy an intelligent assistant able to answer queries by retrieving information from structured databases and unstructured documents, while generating interpretable answers in real time. Such a system can provide insights that would otherwise require hours of manual analysis.

To achieve this, a solution was engineered based on a chain of LLMs (Large Language Models), enriched with agentic steps that minimize response time while maintaining accuracy and interpretability. This architecture ensures that recommendations are both reliable and transparent—qualities that are essential in the highly regulated domain of investment management.

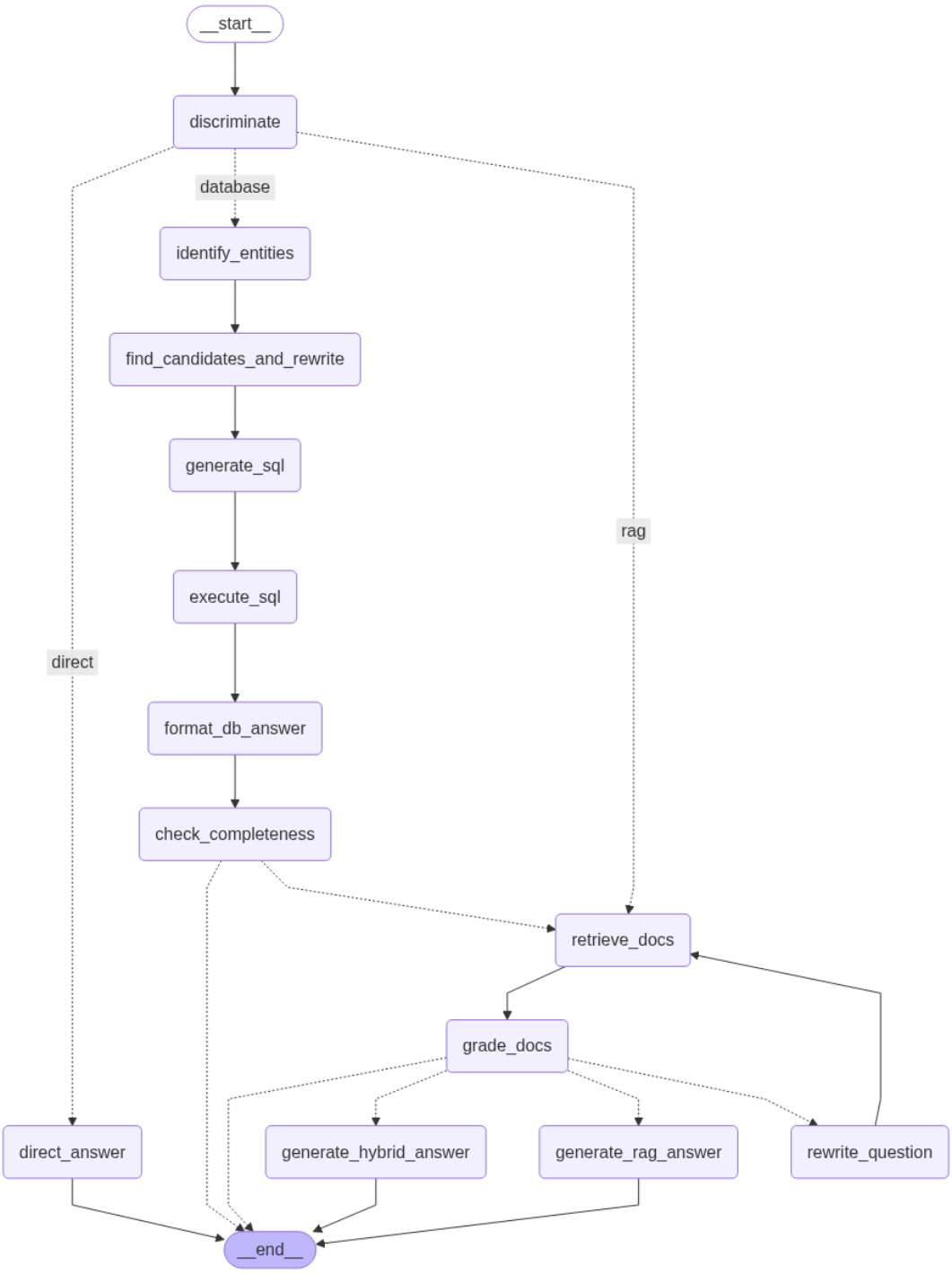


Figure 6.1: Graph representation of the agentic chain for structured and unstructured data retrieval.

6.2 Data Sources

The assistant developed in this project integrates two main categories of data sources: structured databases and unstructured product documentation. Structured data includes transactional records, client demographics, portfolio allocations, and aggregated statistics stored in relational databases. Unstructured data refers to descriptive documents, primarily PDFs, containing detailed product specifications, fees, risk profiles, and additional information.

The coexistence of these heterogeneous data sources creates the core challenge addressed in this chapter. Certain information, such as the number of clients who purchased a product, can only be found in the database, while qualitative descriptions of the product itself are only available in the PDFs. In some cases, information that would intuitively belong to a structured database (such as product fees) is instead embedded in a table inside a document. The assistant must therefore flexibly determine whether to access the database, the documents, or both.

6.3 Frontend Note

When clicking on the investment application link from the landing page, we arrive at the following page:

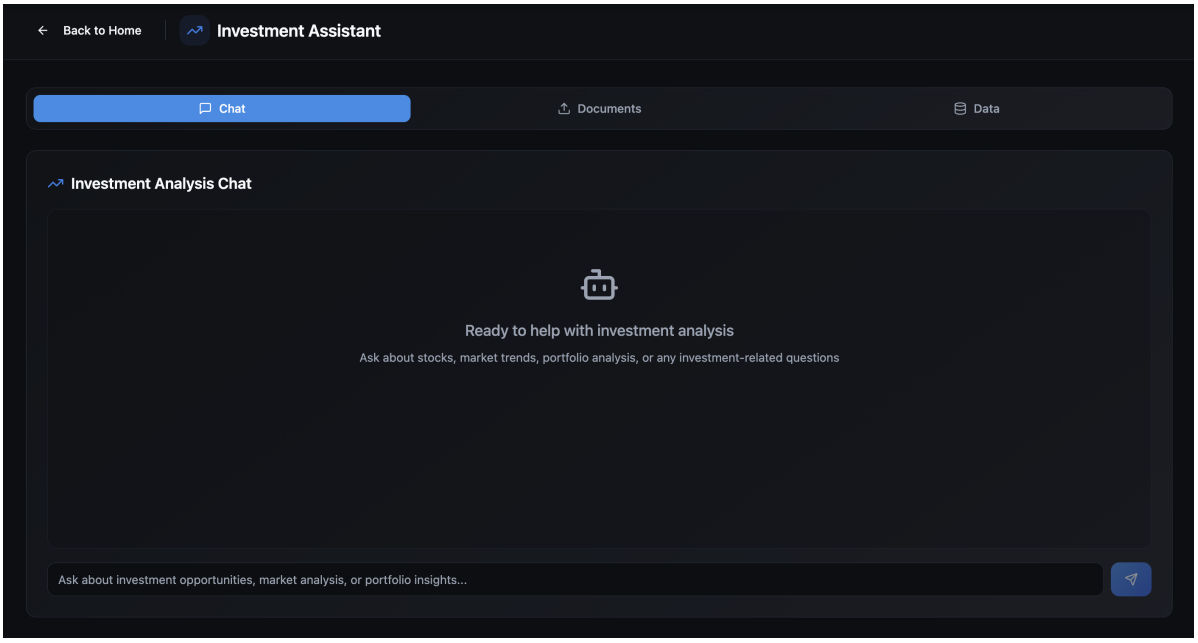


Figure 6.2: Investment application landing page

This interface opens in the chat environment, where it is possible to interact with the agent.

There is then an upload page, where we can add new documents, either as PDF files or as URL links. These documents get saved, parsed and indexed,in order to become part of the agent's knowledge base and are therefore accessible through RAG (Retrieval-Augmented Generation).

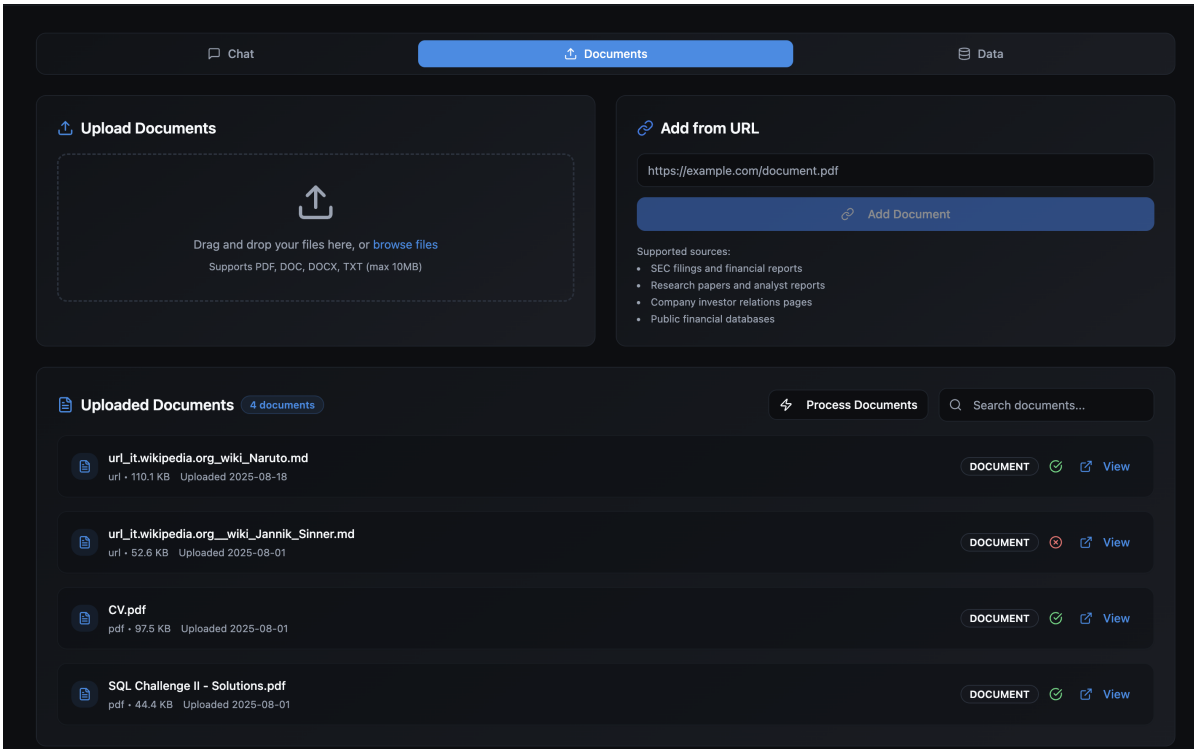


Figure 6.3: Upload page for adding documents (PDF or URL)

The data page allows us to inspect both structured and unstructured data. It is divided into:

- **Datasets and SQL:** where we can preview structured tables and their schema.
- **Products and Documents:** where we can access investment and insurance product PDFs, as well as manually uploaded documents.

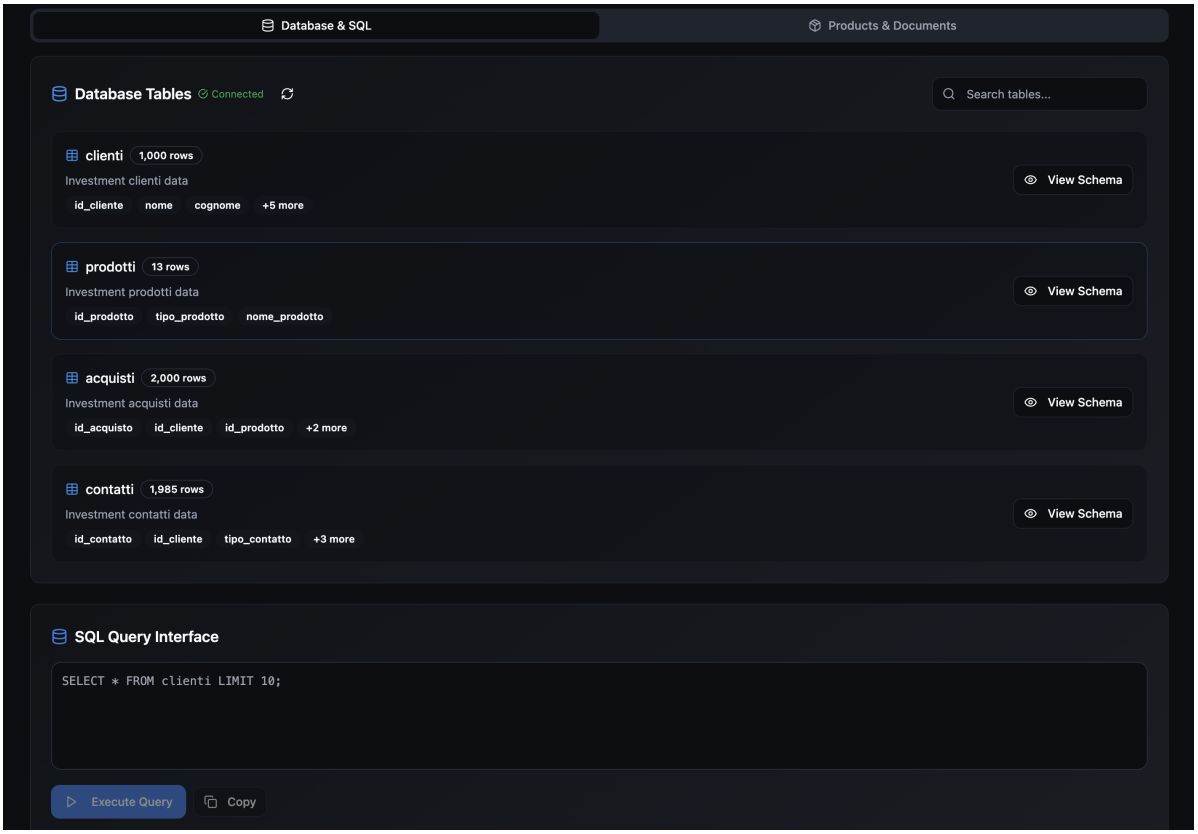


Figure 6.4: Preview of structured datasets and schema

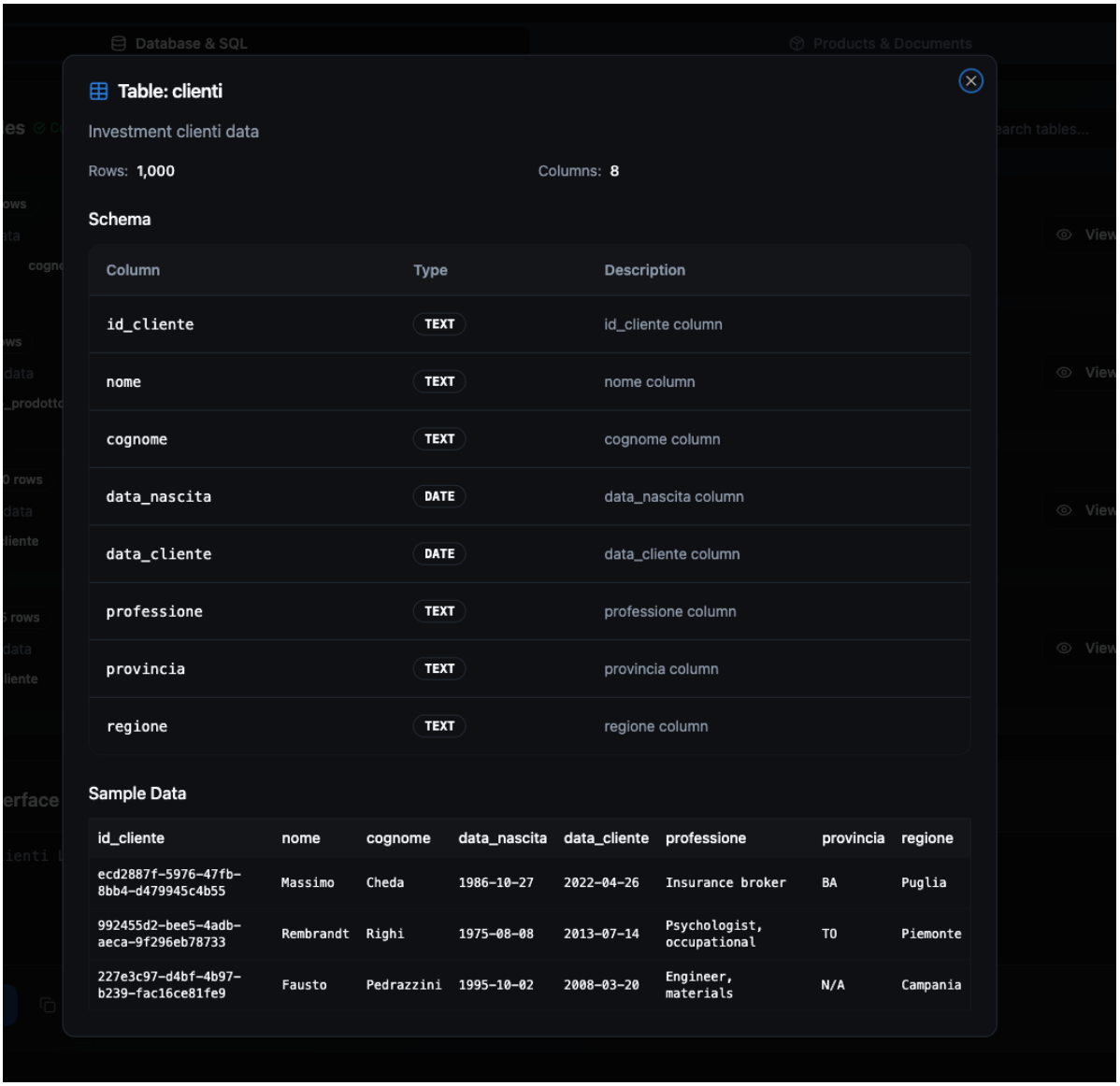
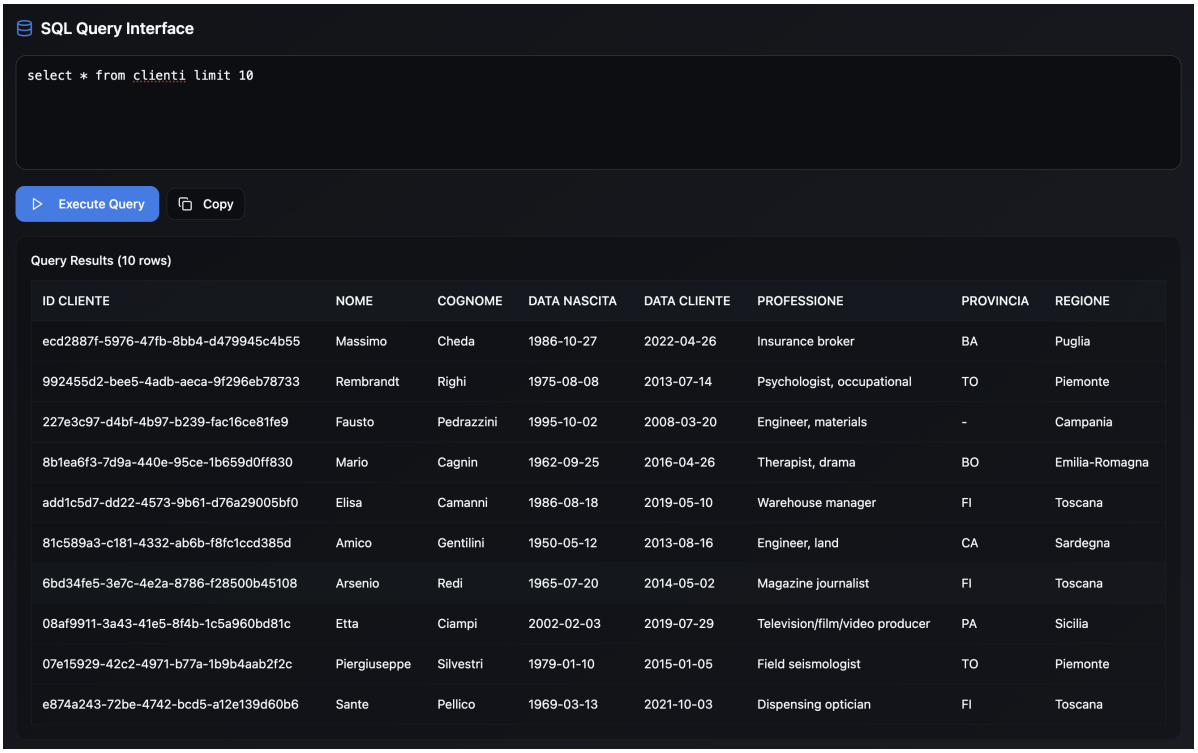


Figure 6.5: It is also possible to inspect the single tables clicking on *View Schema*. In this way we can see the schema and some sample data

Additionally, it is possible to directly query structured data directly using SQL:



ID CLIENTE	NOME	COGNOME	DATA NASCITA	DATA CLIENTE	PROFESSIONE	PROVINCIA	REGIONE
ecd2887f-5976-47fb-8bb4-d479945c4b55	Massimo	Cheda	1986-10-27	2022-04-26	Insurance broker	BA	Puglia
992455d2-bee5-4adb-aeca-9f296eb78733	Rembrandt	Righi	1975-08-08	2013-07-14	Psychologist, occupational	TO	Piemonte
227e3c97-d4bf-4b97-b239-fac16ce81fe9	Fausto	Pedrazzini	1995-10-02	2008-03-20	Engineer, materials	-	Campania
8b1ea6f3-7d9a-440e-95ce-1b659d0ff830	Mario	Cagnin	1962-09-25	2016-04-26	Therapist, drama	BO	Emilia-Romagna
add1c5d7-dd22-4573-9b61-d76a29005bf0	Elisa	Camanni	1986-08-18	2019-05-10	Warehouse manager	FI	Toscana
81c589a3-c181-4332-ab6b-f8fc1ccd385d	Amico	Gentilini	1950-05-12	2013-08-16	Engineer, land	CA	Sardegna
6bd34fe5-3e7c-4e2a-8786-f28500b45108	Arsenio	Redi	1965-07-20	2014-05-02	Magazine journalist	FI	Toscana
08af9911-3a43-41e5-8f4b-1c5a960bd81c	Etta	Ciampi	2002-02-03	2019-07-29	Television/film/video producer	PA	Sicilia
07e15929-42c2-4971-b77a-1b9b4aab2f2c	Piergiuseppe	Silvestri	1979-01-10	2015-01-05	Field seismologist	TO	Piemonte
e874a243-72be-4742-bcd5-a12e139d60b6	Sante	Pellico	1969-03-13	2021-10-03	Dispensing optician	FI	Toscana

Figure 6.6: SQL query interface

In the *Products and Documents* section, we can explore all available PDFs containing information about investments, insurance products, and additional uploaded documents.

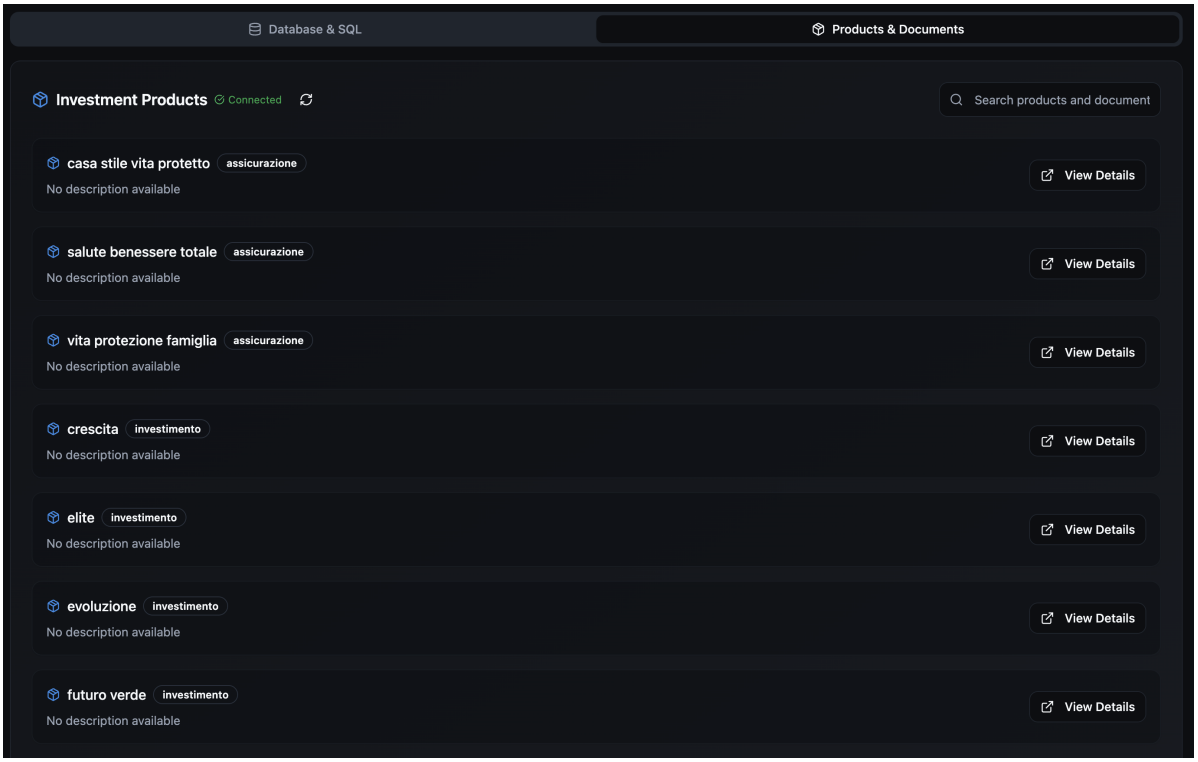


Figure 6.7: Products and documents section with available PDFs

6.4 Question Discrimination

As mentioned before, the main *magic* happens in the chat page, where we can ask for information about both structured and unstructured data. A key requirement for building such a system is the ability to discriminate among different types of user queries. Without an accurate classification mechanism, the assistant risks either overusing expensive retrieval components or under-serving the user by failing to access the relevant data source.

At the same time, performing data research for every question—even for those that do not need it, such as a normal conversational interaction—would be inefficient. For example, if the user simply prompts *"How are you?"*, it would be useless to trigger additional retrieval steps. Such behavior would increase API costs and response time.

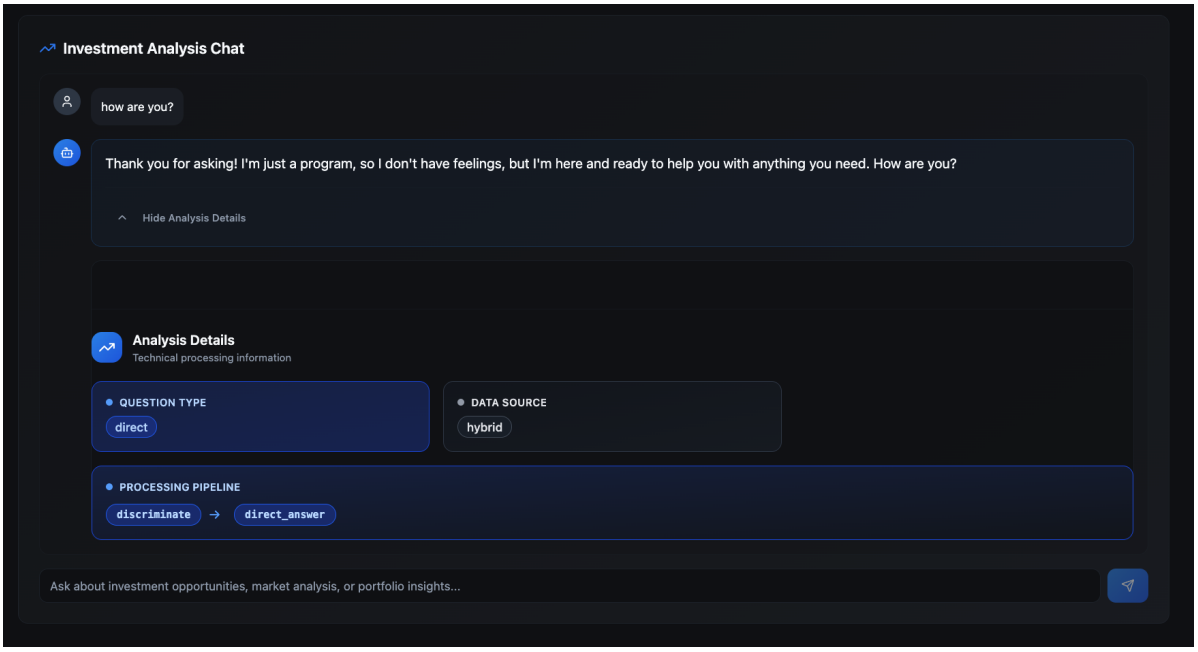


Figure 6.8: Example of how a direct question is handled

To ensure prompt responses and avoid unnecessary computational overhead, the system must route each question to one of three strategies:

1. **Direct answers:** General knowledge questions that can be answered without external data, using only the information embedded in the model during training.
2. **RAG (Retrieval-Augmented Generation):** Product-related questions requiring retrieval from unstructured documents such as brochures and policy sheets.
3. **Database queries (Text2SQL):** Client-specific or transactional questions requiring SQL queries over structured databases, with a conservative fallback to RAG when information is incomplete.

This discrimination problem was approached in three different ways: a K-Nearest Neighbors (K-NN) classifier, a fine-tuned BERT model, and a few-shot learning approach using Azure-hosted LLMs.

6.4.1 K-NN Based Approach

The first approach explored was based on the K-Nearest Neighbors (K-NN) algorithm. To enable classification, a dataset of representative questions spanning all categories of interest was required. The quality of this dataset is critical: well-curated data can

significantly improve classification accuracy, whereas sparse or noisy data degrades performance. While constructing such a dataset manually is time-consuming, in a production setting the system could iteratively collect user queries, gradually enriching its dataset. Misclassified queries could then be corrected manually or automatically, leveraging user feedback to refine the classifier over time.

At this early stage of development, however, sufficient real-world data was not available. To overcome this limitation, synthetic training data was generated using an LLM prompted to produce diverse question formulations conditioned on previously generated examples. This technique reduces redundancy and encourages greater diversity, resulting in a dataset with more linearly independent examples—an important property for improving classifier performance.

Once the dataset was prepared, questions were embedded into vector space and classified using K-NN. Given an input question $\mathbf{x} \in \mathbb{R}^d$, the algorithm identifies the k nearest neighbors from the labeled dataset $\mathcal{D} = \{(\mathbf{x}_i, y_i)\}_{i=1}^n$ according to cosine or Euclidean distance. The predicted class $\hat{y}$ is then determined by majority vote:

$$\hat{y} = \arg\max_{c \in \mathcal{C}} \sum_{i \in \mathcal{N}_k(\mathbf{x})} \mathbb{I}(y_i = c),$$

where $\mathcal{N}_k(\mathbf{x})$ is the set of k nearest neighbors and $\mathbb{I}$ is the indicator function.

While conceptually simple and interpretable, the method's effectiveness proved highly dependent on the representativeness of the training data. In experiments, the synthetic dataset lacked sufficient diversity to cover real-world variability, leading to suboptimal results. Consequently, this approach was not retained for the final system.

6.4.2 BERT Fine-Tuning

A second approach involved fine-tuning a transformer-based classifier. The BERT family of models provides robust language representations and is well-suited to sentence-level classification tasks. In this case, the model was adapted to predict whether a question required a direct answer, a database query, or a RAG retrieval.

Several variants were considered, including the standard `bert-base-uncased`, lighter architectures such as DistilBERT and MiniLM, and domain-adapted encoders like FinBERT. The latter approach involved additional pretraining on finance-related documents to improve domain specificity. For deployment in a latency-sensitive environment, a lightweight model was deemed most appropriate.

A classification head was added, and the model was trained using cross-entropy loss:

$$\mathcal{L}(\theta) = -\sum_{(x,y)\in\mathcal{D}} \sum_{c\in\mathcal{C}} \mathbb{I}[y = c] \log p_\theta(c \mid x).$$

Labeled data included generic finance questions for the direct category, product-related questions for RAG, and client-specific questions for the database category. To address class imbalance, synthetic examples and paraphrases were added.

Optimization employed AdamW with learning rates in the range 1e-5 to 5e-5, batch sizes of 32–64, and early stopping. Evaluation metrics included macro-F1, per-class precision and recall, and expected utility scores reflecting the cost of misclassification.

Although fine-tuned BERT models offer advantages such as low latency and control over deployment, they require substantial labeled data, retraining for evolving product catalogs, and ongoing MLOps. Given these constraints, a more pragmatic solution was pursued.

6.4.3 Few-Shot LLM Routing (Azure)

The final and most effective approach used few-shot learning with an Azure-hosted LLM. Instead of training a classifier, the system relies on a carefully designed prompt containing exemplars of direct, database, and RAG questions. The LLM is asked to output a structured JSON object specifying the class of the query.

This approach offers several advantages. It is fast to implement, requiring no labeled dataset or retraining. It is flexible, allowing prompt modifications to immediately update behavior as products or policies change. It also proved more accurate in edge cases, where users asked ambiguous or mixed-intent questions.

Operational settings included low temperature (0–0.2) and short token limits to reduce variance and cost. Additional safeguards such as threshold-based fallbacks and optional reasoning fields allow for transparency and continual improvement.

In practice, this approach outperformed the alternatives and was chosen as the final solution for query discrimination.

6.5 Text-to-SQL Pipeline with Entity Resolution and RAG Fallback

6.5.1 Motivation

One of the central challenges in investment management lies in handling questions that require precise information from structured databases. These questions often involve client demographics, transaction histories, portfolio allocations, or aggregated performance metrics. While such information is stored in relational databases, end users do not formulate their queries in SQL. Instead, they phrase them in natural language, often including vague references, abbreviations, or misspellings of entities such as product names or client surnames.

A simple text-to-SQL translation approach is insufficient in this setting. First, the domain suffers from *high cardinality*: thousands of possible client identifiers or product names exist, and a direct mapping from text to SQL cannot enumerate all possibilities. Second, natural language inputs are prone to spelling errors or paraphrases, which do not match the exact values in the database. Finally, even a correct SQL query may not yield a complete answer if the relevant information resides in unstructured documents rather than in tables.

The pipeline described here addresses these issues through a sequence of stages: entity identification, entity correction via vector search, SQL query generation, query execution, answer synthesis, and a completeness check that triggers fallback to retrieval-augmented generation (RAG) when necessary.

6.5.2 Entity Identification

The first step is to extract entities from the user's natural language question. Entities are specific values that appear as row entries in the database, such as product titles, surnames, geographic regions, or years. To perform this step, an LLM is prompted with explicit instructions to output only entities that are likely to exist as cell values, rather than generic concepts or column names.

For example, in the question:

> *"How many clients purchased the Giovani Digital investment plan?"*

the model should output the entity list: `[Giovani Digital]`, since this string is expected to match an entry in the `products` table. This filtering avoids including irrelevant terms

such as "clients" or "investment," which are column names rather than row values.

6.5.3 Entity Correction and Candidate Generation

Once the entities are extracted, the pipeline must address the dual challenges of high cardinality and incorrect spelling. For instance, if the user types "Giovvani Digittal," a direct match to the database would fail. To resolve this, an embedding-based vector index of canonical entities is maintained. Each entity is stored as a dense vector embedding, enabling semantic similarity search.

For every extracted entity, the system performs a k-nearest neighbor search against the vector index. The top-k candidates (typically $k = 5$) are returned as possible corrections. Continuing the example, the search for "Giovvani Digittal" may return candidates such as:

- Giovani Digital
- Giovani Flex
- Digital Growth Plan

An LLM is then prompted to rewrite the user's question, replacing the noisy entity mention with the best-matching canonical form. This rewriting step ensures that the SQL query operates on valid and consistent database values. Importantly, the rewriting is constrained to substitution only: the model is not allowed to add new content or alter the semantic intent of the question.

6.5.4 SQL Query Generation

With the rewritten question, the system proceeds to generate SQL. The LLM is provided with:

1. The target database dialect (e.g., PostgreSQL, SQLite).
2. A detailed schema description of the available tables and columns.
3. Constraints that forbid the model from hallucinating non-existent tables or selecting all columns indiscriminately.

The SQL generation prompt enforces best practices. Queries are restricted to at most a specified number of rows (e.g., 100), ensuring that execution is efficient. The model is

also instructed to select only the columns relevant to the user's question, reducing both latency and risk of irrelevant output.

For the example question, the generated SQL might be:

```
SELECT COUNT(*)
FROM transactions t
JOIN products p ON t.product_id = p.id
WHERE p.name = 'Giovani Digital';
```

6.5.5 Query Execution

Once generated, the SQL is executed against the database. The system makes use of a dedicated tool wrapper that ensures syntactic validity and catches execution errors. If the query runs successfully, the raw tabular results are returned to the pipeline.

Execution errors can still occur, especially if the LLM incorrectly referenced a column or misapplied a join. These cases are logged, and the system gracefully falls back to reporting the failure to the user with an apologetic message, rather than exposing raw error messages.

6.5.6 Answer Synthesis

The tabular results are then translated into natural language. For this, the LLM is prompted with the original user question, the SQL query, and the returned result set. It generates a concise, human-readable answer. For instance:

> *"According to the database, 152 clients have purchased the Giovani Digital investment plan."*

This synthesis step makes the assistant more user-friendly, removing the need for end users to interpret raw tables.

6.5.7 Completeness Check

Even when the database query executes successfully, it may not fully answer the user's question. For example:

> *"What are the risks associated with the Giovani Digital plan?"*

In this case, the database might return only information about the number of clients or sales volume, which does not address the request. To guard against such scenarios, a completeness check is applied.

The LLM evaluates whether the generated database answer is "complete" or "incomplete" with respect to the user's intent. If the answer is incomplete, the system triggers a fallback to RAG.

6.5.8 RAG Fallback and Hybrid Responses

When completeness is judged insufficient, the system reformulates the query for retrieval, searches the product document index, and composes a response based on relevant chunks. For example, in the previous question about risks, the RAG component would locate passages in the product brochure describing exposure to market fluctuations or specific regulatory warnings.

In certain cases, both database results and document context are relevant. For example:

> *"How many clients purchased the Giovani Digital plan, and what fees apply to it?"*

Here, the number of clients comes from the database, while the fee structure is stored in PDFs. The system combines both sources into a hybrid response:

> *"152 clients have purchased the Giovani Digital plan. According to the product brochure, the plan applies an annual management fee of 1.2%."*

6.5.9 Summary

The Text-to-SQL pipeline thus operates as a robust multi-stage system. By combining entity identification, embedding-based correction, SQL generation, query execution, natural-language synthesis, and completeness checks, it overcomes the challenges of high cardinality, spelling errors, and partial information. The integration of RAG as a fallback ensures that users receive complete and interpretable answers, even when data is scattered across structured and unstructured sources. This design is especially well-suited to investment management, where both numerical facts and descriptive product information are essential for informed decision-making.

6.5.10 Example Usage

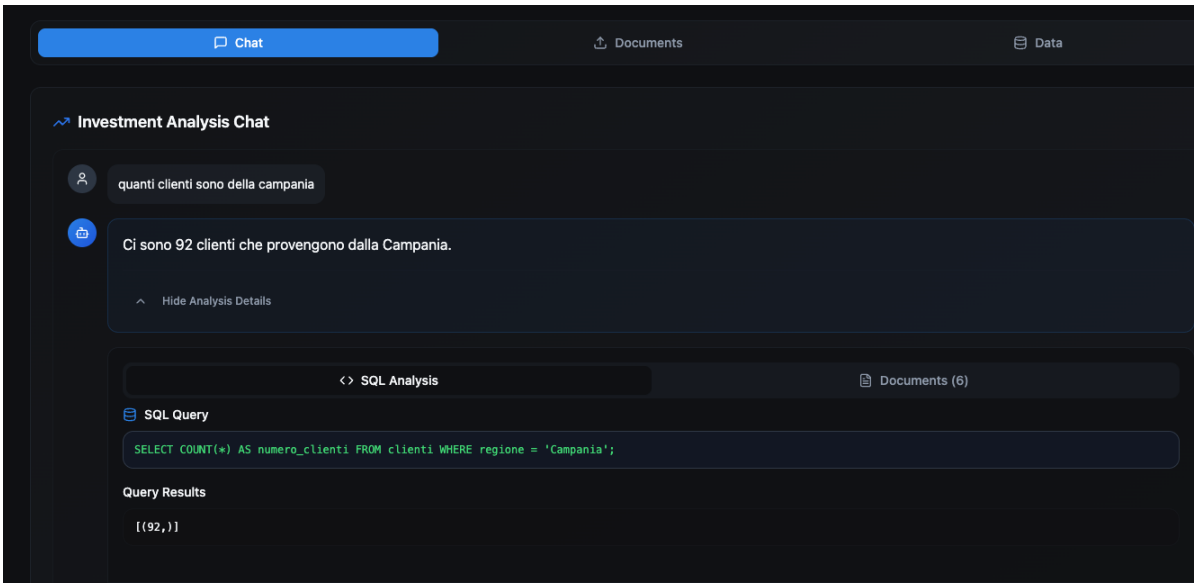


Figure 6.9: Example of answer to a question requiring Text2SQL

Figure 6.10: Processing Pipeline

In this example, it is possible to see how the agent correctly classified the question as one requiring a database query. Even though the input name was `campania`, the agent was able to identify that the intended entity was *Campania* and correctly queried the database, obtaining the result of `92`.

As mentioned earlier, the approach is conservative: often, in addition to the SQL query, a RAG (retrieval-augmented generation) pass over documents (e.g., PDF files) is performed. This ensures that any additional information that may be contained in the unstructured sources can also be retrieved and presented to the user.

6.6 Retrieval-Augmented Generation (RAG) Route

This section details the RAG pathway used when a query requires product- or policy-level information that resides in unstructured documents (e.g., brochures, term sheets, regulatory notes). The route combines classical retrieval with LLM-based grading and generation, and it can synthesize answers jointly with database outputs when needed. The implementation follows a modular pattern built on top of LangGraph's `StateGraph`, with explicit nodes for retrieval, answer drafting, document grading, query rewriting, and hybrid answer construction.

6.6.1 Objectives and High-Level Flow

The RAG route is invoked in two cases: (i) the question discriminator directly classifies the input as *RAG*, or (ii) a *database* question is deemed incomplete by the completeness checker. The goal is to (1) retrieve a focused set of document passages, (2) retain only passages that are relevant to the question, (3) generate a concise answer grounded in those passages, and (4) optionally merge document-based context with database facts into a single, coherent response.

Operationally, the compiled graph executes the following chain:

$$\texttt{retrieve} \rightarrow \texttt{generate\_rag} \rightarrow \texttt{grade} \rightarrow \texttt{rewrite} \rightarrow \texttt{hybrid} \rightarrow \texttt{END}.$$

Each node reads and writes to a shared `FullBotState`, ensuring that intermediate artifacts (retrieved titles/contents, graded passages, and final answer) are available to downstream steps.

6.6.2 Retrieval and State Population

The node `perform_rag_retrieval` calls a domain-configured retriever (e.g., a Chroma-backed vector index) with the current `state["question"]`. It returns a ranked list of `Document` objects; for reproducibility and later auditing, the implementation separates *titles* (from `metadata`) and *page contents*:

- `state["retrieved_docs_title"]` ← list of titles ("Unknown Title" if missing),
- `state["retrieved_docs_content"]` ← list of corresponding text chunks.

Using titles alongside content is deliberate: titles typically encode product names or series identifiers and are useful during hybrid synthesis to name recommended products explicitly.

6.6.3 First-Pass Answer Drafting

The node `generate_rag_answer` performs a first-pass answer using all retrieved chunks as context. The `RAG_GENERATION_PROMPT` instructs the LLM to produce a brief, grounded response from the provided passages and the active question. This step gives the user a quick draft while longer-running grading operations proceed, and it also establishes a baseline answer for comparison with the refined, graded set of chunks.

6.6.4 Document Grading (Parallelized)

Not all retrieved chunks are equally relevant. The node `grade_documents` filters the set using a binary relevance grader defined by the `GradeDocuments` schema:

$$\texttt{binary\_score} \in \{\text{“yes”}, \text{“no”}\}.$$

For each retrieved item, the system constructs a grading prompt (`RAG_GRADING_PROMPT`) that presents the document and the question to the LLM. Grading is parallelized with a `ThreadPoolExecutor` (up to 10 workers or the number of documents, whichever is smaller) to reduce latency on multi-document inputs. Only documents with a ‘‘`yes`’’ score are retained and stored in:

$$\texttt{state["relevant\_docs"]} \leftarrow \{\text{“\#Title”} \,\|\, \text{content}\}.$$

This binary gate substantially improves answer precision, as it removes semantically adjacent but ultimately off-topic text (e.g., similar funds with different fee structures).

6.6.5 Query Rewriting for Retrieval Robustness

RAG often benefits from iterative refinement. The node `rewrite_question` rewrites the user query with `RAG_REWRITE_PROMPT`, producing a semantically clearer formulation targeted at the vector index. This step is particularly helpful when the original question is short, colloquial, or contains domain-specific shorthand. The rewritten text replaces `state["question"]`, and a counter `grade_negative_counter` is incremented for telemetry and loop control. In practice, an upper bound on rewrite attempts (e.g., 1–2 passes) should be enforced to avoid indefinite cycling.

6.6.6 Hybrid Answer Synthesis

The final node, `generate_hybrid_answer`, merges structured and unstructured evidence when both are available. It constructs a *combined context* by concatenating:

1. a database stub (if present), e.g., ‘‘`Database Query Result: ...`’’;
2. a formatted list of RAG passages where each block is prefixed with ‘‘`Product Name:`’’ followed by the title and the content.

This combined context is given to `HYBRID_ANSWER_PROMPT`, which instructs the LLM to produce a concise investment-focused response (4–5 lines), emphasizing product identity, fees, eligibility constraints, and other actionable attributes. The result is saved as:

$$\texttt{state["answer"]} \leftarrow \text{final hybrid text}, \quad \texttt{state["source"]} \leftarrow \text{“hybrid”}.$$

If neither DB nor RAG context is available, the node returns a polite failure message and records the source as ‘‘`hybrid`’’ for observability.

6.6.7 Graph Orchestration and Design Rationale

The RAG graph is defined with `StateGraph(FullBotState)` and compiled into an executable plan. The current edge ordering is:

$$\texttt{retrieve} \rightarrow \texttt{generate\_rag} \rightarrow \texttt{grade} \rightarrow \texttt{rewrite} \rightarrow \texttt{hybrid}.$$

This ordering prioritizes responsiveness (a draft is produced early) and then improves precision via grading and rewriting before constructing the final hybrid answer. Two design notes are worth highlighting:

1. **Parallel Grading for Throughput.** Grading is independent across chunks; parallelism yields significant wall-clock savings when k is moderately large (e.g., 6–10).
2. **Rewrite as Retrieval Optimizer.** Rewriting refines future retrievals. In a production loop, a second retrieval after rewriting (i.e., `retrieve` → `grade` → `rewrite` → `retrieve` → `grade`) can further improve precision. The present implementation increments a counter to support such extensions.

6.6.8 Prompting and Domain Sensitivity

All prompts are domain-calibrated for wealth management. For instance, the rewrite prompt is language-aware (Italian in the deployment example) and emphasizes investment-plan terminology. The generation prompt encourages concise, decision-support answers rather than generic summaries. The grading prompt is deliberately binary to simplify thresholding and to avoid nuanced—but operationally ambiguous—scores.

6.6.9 Failure Modes and Mitigations

Typical failure modes include: (i) empty retrieval sets; (ii) grader over-pruning (no documents survive); (iii) overly terse answers; and (iv) mismatch between database and document terminology. Mitigations are:

- *Empty retrieval:* trigger a single rewrite and re-run retrieval; if still empty, return a clear "not found" message.
- *Over-pruning:* reduce aggressiveness by allowing a small number of borderline documents (e.g., keep at least top-1).
- *Answer quality:* enforce a minimal structure in `HYBRID_ANSWER_PROMPT` (e.g., "product name, key attribute, numeric figure if available").
- *Terminology mismatch:* rely on titles in the combined context to anchor product identities; consider adding alias fields to the index.

6.6.10 Practical Considerations

To keep latency low while maintaining quality:

- Set the retriever's k modestly (e.g., $k = 6$) and chunk sizes appropriate to the document style (e.g., 600–1,000 characters with 50–150 overlap for brochures).
- Use low-temperature inference for grading and generation to reduce variance.
- Cache embeddings and retriever configurations (via `get_cached_config`) and periodically refresh the index to reflect new or updated product documents.

6.6.11 Outcome

The RAG route produces answers that are both grounded and concise, with explicit product naming and references to the retrieved content. When paired with database outputs in the hybrid step, the assistant can simultaneously report quantitative metrics (e.g., client counts) and qualitative product details (e.g., fees, eligibility, and risk warnings), thereby supporting rapid and explainable decision-making in the investment management workflow.

6.6.12 Example

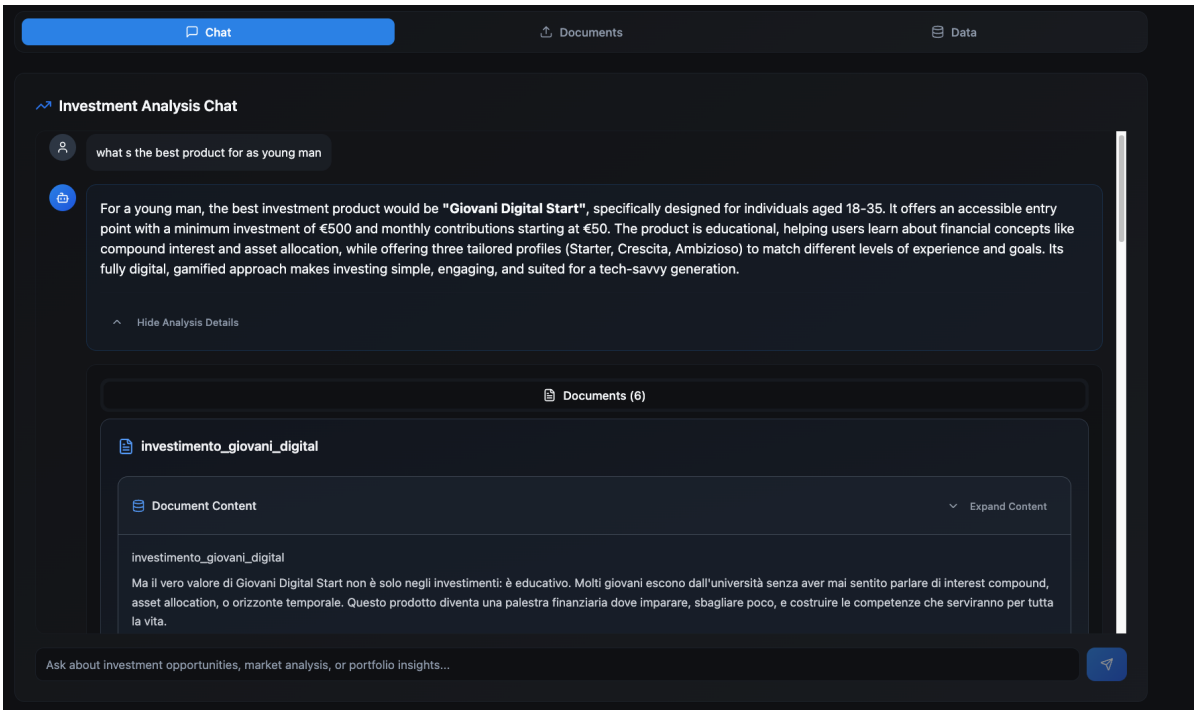


Figure 6.11: Example of answer to a question requiring RAG

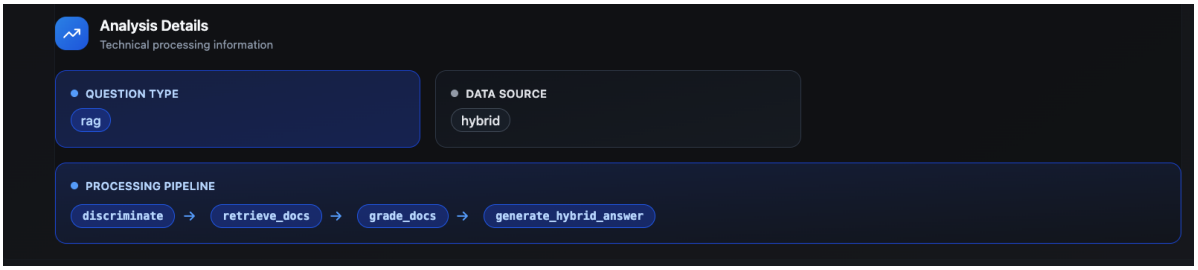


Figure 6.12: Processing Pipeline

Bibliography

[1] Encyclopaedia Britannica.
Chomsky's grammar, n.d.
Accessed 2025-09-07.

[2] Claude E. Shannon.
Prediction and entropy of printed english.
Bell System Technical Journal, 30(1):50–64, 1951.

[3] Noam Chomsky.
Three models for the description of language.
IRE Transactions on Information Theory, 2(3):113–124, 1956.

[4] History of Information.
The georgetown-ibm experiment demonstrates machine translation, 2014.
On the 1954 Georgetown-IBM MT demonstration; accessed 2025-09-07.

[5] David Mumford and Agnès Desolneux.
Pattern Theory: The Stochastic Analysis of Real-World Signals.
A K Peters/CRC Press, Natick, MA, 2010.
Recounts Markov's 1913 analysis of *Eugene Onegin*.

[6] David E. Rumelhart and James L. McClelland.
On learning the past tense of english verbs.
In James L. McClelland and David E. Rumelhart, editors, *Parallel Distributed Processing: Explorations in the Microstructure of Cognition, Volume 2*, pages 216–271. MIT Press, 1986.

[7] James L. McClelland and David E. Rumelhart.
A distributed model of the english past tense.
Cognitive Science, 12(2):255–306, 1986.

[8] Jeffrey L. Elman.
Finding structure in time.
Cognitive Science, 14(2):179–211, 1990.

[9] Yoshua Bengio, Réjean Ducharme, Pascal Vincent, and Christian Janvin.
A neural probabilistic language model.
Journal of Machine Learning Research, 3:1137–1155, 2003.

[10] Mitchell P. Marcus, Beatrice Santorini, and Mary Ann Marcinkiewicz.
Building a large annotated corpus of english: The penn treebank.
Computational Linguistics, 19(2):313–330, 1993.

[11] Yoshua Bengio, Patrice Simard, and Paolo Frasconi.
Learning long-term dependencies with gradient descent is difficult.
IEEE Transactions on Neural Networks, 5(2):157–166, 1994.

[12] Tomáš Mikolov, Martin Karafiát, Lukáš Burget, Jan Černocký, and Sanjeev Khudanpur.
Recurrent neural network based language model.
In *INTERSPEECH*, pages 1045–1048, 2010.

[13] Tomáš Mikolov, Stefan Kombrink, Lukáš Burget, Jan Černocký, and Sanjeev Khudanpur.
Extensions of recurrent neural network language model.
In *ICASSP*, 2011.
RNNLM extensions and toolkit.

[14] Mike Schuster and Kuldip K. Paliwal.
Bidirectional recurrent neural networks.
IEEE Transactions on Signal Processing, 45(11):2673–2681, 1997.

[15] Sepp Hochreiter and Jürgen Schmidhuber.
Long short-term memory.
Neural Computation, 9(8):1735–1780, 1997.

[16] Felix A. Gers, Jürgen Schmidhuber, and Fred Cummins.
Learning to forget: Continual prediction with lstm.
In *Neural Computation*, 2000.
Forget gate addition.

[17] Wojciech Zaremba, Ilya Sutskever, and Oriol Vinyals.
Recurrent neural network regularization.
In *ICLR*, 2014.

[18] Kyunghyun Cho, Bart van Merriënboer, Çaglar Gülçehre, Dzmitry Bahdanau, Fethi Bougares, Holger Schwenk, and Yoshua Bengio.
Learning phrase representations using rnn encoder–decoder for statistical machine translation.
In *EMNLP*, 2014.

[19] Junyoung Chung, Çaglar Gülçehre, KyungHyun Cho, and Yoshua Bengio.
Empirical evaluation of gated recurrent neural networks on sequence modeling.
arXiv:1412.3555, 2014.
Additive updates in GRU vs LSTM.

[20] Dzmitry Bahdanau, Kyunghyun Cho, and Yoshua Bengio.
Neural machine translation by jointly learning to align and translate.
In *ICLR*, 2015.

[21] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Łukasz Kaiser, and Illia Polosukhin.
Attention is all you need.
In *Advances in Neural Information Processing Systems (NeurIPS)*, 2017.

[22] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova.
Bert: Pre-training of deep bidirectional transformers for language understanding.
In *NAACL*, 2019.

[23] Alec Radford, Karthik Narasimhan, Tim Salimans, and Ilya Sutskever.
Improving language understanding by generative pre-training.
Technical report, OpenAI, 2018.

[24] Alec Radford, Jeffrey Wu, Rewon Child, David Luan, Dario Amodei, and Ilya Sutskever.
Language models are unsupervised multitask learners.
Technical report, OpenAI, 2019.
GPT-2 zero-shot results.

[25] Tom B. Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared D. Kaplan, et al.

Language models are few-shot learners.
In *Advances in Neural Information Processing Systems (NeurIPS)*, 2020.

[26] OpenAI.
Gpt-4 technical report.
arXiv:2303.08774, 2023.
Multimodal; human-level performance on many benchmarks.

[27] Colin Raffel, Noam Shazeer, Adam Roberts, Katherine Lee, Sharan Narang, Michael Matena, et al.
Exploring the limits of transfer learning with a unified text-to-text transformer.
Journal of Machine Learning Research, 21(140):1–67, 2020.

[28] Rohan Paul.
Accelerating llm inference without attention approximation.
`https://www.rohan-paul.com/p/accelerating-llm-inference-without`, 2025.

[29] Sebastian Ruder, Matthew E. Peters, Swabha Swayamdipta, and Thomas Wolf.
Transfer learning in natural language processing.
arXiv:1910.07370, 2019.

[30] Kyle Wiggers.
Ai models from microsoft and google already surpass human performance on the superglue language benchmark.
VentureBeat, 2021.

[31] NVIDIA.
What is retrieval-augmented generation aka rag.
Blog, 2023.

[32] Cem Dilmegani.
Top vector database for rag: Qdrant vs weaviate vs pinecone.
`https://aimultiple.com/`, 2023.

[33] Manish Shivanandhan.
How ai agents remember things: Vector stores in llm memory.
`https://www.freecodecamp.org/news/how-ai-agents-remember-things-vector-stores-in-ll`
2023.

[34] PromptingGuide.ai.
React prompting — prompt engineering guide.
https://www.promptingguide.ai/techniques/react, 2023.

[35] Yao et al.
Tree of thoughts: Deliberate problem solving with large language models.
https://arxiv.org/abs/2305.10601, 2023.

[36] IBM Think.
What is tree of thoughts prompting?
https://www.ibm.com/think/topics/tree-of-thoughts, 2023.

[37] NocoBase.
Top 20 open source ai projects with the most github stars.
https://medium.com/@nocobase/top-20-open-source-ai-projects-with-the-most-github-st
2025.

[38] IBM Think.
What is langchain?
https://www.ibm.com/think/topics/langchain, 2023.

[39] r/singularity (Reddit).
Autogpt just reached 100k stars on github.
https://www.reddit.com/r/singularity/comments/12uixed/this_is_very_
impressive_autogpt_just_reached_100k/, 2023.

[40] OpenAgents.
Openagents: An open framework for connecting ai agents at scale.
https://openagents.org/, 2023.

[41] Designveloper.
Langchain: What it is, how it works, and how to build llm apps.
https://www.designveloper.com/blog/what-is-langchain/, 2025.

[42] Microsoft.
React — autogen 0.2 - microsoft open source.
https://microsoft.github.io/autogen/0.2/docs/topics/
prompting-and-reasoning/react/, 2024.

[43] Red Hat Blog.
Meet vllm: For faster, more efficient llm inference and serving.
https://www.redhat.com/en/blog/meet-vllm-faster-more-efficient-llm-inference-and-se
2024.

[44] ZenML.
Openrouter: Building a multi-model llm marketplace and routing platform.
https://www.zenml.io/llmops-database/building-a-multi-model-llm-marketplace-and-rou
2024.

[45] Zhengbao Ji et al.
Survey of hallucination in large language models.
arXiv preprint, 2023.

[46] BizTech Magazine.
Llm hallucinations: What are the implications for businesses?
https://biztechmagazine.com/article/2025/02/llm-hallucinations-implications-for-bus
2025.

[47] Salesforce.
Salesforce announces einstein gpt, the world's first generative ai for crm.
https://www.salesforce.com/news/press-releases/2023/03/07/
einstein-generative-ai/, 2023.

[48] SemiAnalysis.
The inference cost of search disruption – large language model cost analysis.
https://semianalysis.com/2023/02/09/the-inference-cost-of-search-disruption/,
2023.

[49] Andreessen Horowitz (a16z).
Welcome to llmflation – llm inference cost is going down fast.
https://a16z.com/llmflation-llm-inference-cost/, 2024.

[50] Amazon.
Amazon completes $4b anthropic investment to advance generative ai.
https://www.aboutamazon.com/news/company-news/
amazon-anthropic-ai-investment, 2024.

[51] OpenAI.

Enterprise privacy at openai.
https://openai.com/enterprise-privacy/, 2024.

[52] Florian Reifschneider.
Open-source llms for data privacy and compliance in corporate use cases.
https://rocketloop.de/en/blog/open-source-llms-for-data-privacy-and-compliance/, 2023.