Politecnico di Torino

Politecnico di Torino

Master's Degree in COMPUTER ENGINEERING

A.a. 2024/2025

Graduation Session October 2025

GenAI-NewsScraper

Automated News Scraping, Summarization, Enrichment, and Multimodal Content Generation

Supervisors:

Riccardo Coppola

Edoardo Morucci

Christian Cabrera

Candidate:

Luca Bergamini

Abstract

The rapid growth of digital media has created a need for automated systems capable of efficiently retrieving, processing, and delivering news content. This thesis presents the design and implementation of a generative AI (**GenAI**) system for automated news scraping, content enrichment and summarization, and multimodal output generation, aimed to support a scalable media workflows and interactive user experiences.

The main objective is to develop an agent-based architecture that autonomously collects news from different sources, enriches it with related material, summarizes key information, and delivers results via text and audio formats. The system relies on a Model Context Protocol (**MCP**) server for orchestration, with modular tools for vector-based data storage, LLM-driven web search, and Text-to-Speech (**TTS**) synthesis. Structured web scraping, multi-document summarization, and vector embeddings (using PostgreSQL with `pgvector`) enable efficient data processing, while TTS supports automated podcast generation and interactive newsletters.

The prototype demonstrates the system's ability to handle end-to-end news workflows with minimal human intervention. Cost and usage analysis indicates that, for a single user on the system, the daily operation—including scraping, summarization, embeddings, and podcast generation—amounts to approximately \$0.85, or \$25.5 per month. For 100 users, the daily per-user cost drops to about \$0.22 (\$6.6 per month), and for 1000 users, it further reduces to \$0.157 (\$4.71 per month), due to the **shared baseline scraping and content generation cost** of $\$\frac{0.70}{\sqrt{n}}$ per day. Typical usage (for 1000 users) with one access per day incurs \$0.157–\$0.25 daily per user, while high-activity scenarios with multiple accesses and queries increase costs to \$0.30–\$0.50 per day (\$9–\$15 per month). This analysis highlights the linear scaling of per-user operational costs versus the shared scraping baseline, providing insights for accurate budgeting and resource planning.

In conclusion, this thesis contributes a practical framework for integrating LLMs, modular tools, and MCP-based orchestration into automated news pipelines. The system demonstrates scalability, multimodal capabilities, and cost efficiency, illustrating the potential of agent-based architectures for smart, interactive media platforms.

Acknowledgements

First of all, I would like to thank my academic supervisor, Professor Riccardo Coppola, for his availability, valuable advice, and constant support throughout the process of this work. His experience and feedbacks have been fondamental in shaping this project.

I would also like to express my appreciation to Data Reply, for providing me the opportunity to develop this project within a stimulating and professional environment. I would also like to express my gratitude to my company supervisors, Edoardo Morucci and Christian Cabrera, for their constant support and advice.

I would like to express my gratitude to Martina, for her continuous support, to believe in me, and for being part of my every day life.

Finally, a special thought goes to my whole family and friends, for being by my side all these years, and to those who will continue to be.

Table of Contents

List of Figures

Chapter 1

Introduction

1.1 Background and Motivation

Today, professionals and companies are overwhelmed by the sheer volume of online content, especially in fast-paced sectors like finance and technology. Hundreds of articles, reports, and updates are published daily across multiple platforms. Manually monitoring, reading, and analyzing all this information is time-consuming and often impractical. As a result, there is a growing need for automated systems that can collect and summarize the most relevant content, helping users stay informed without investing excessive time or effort.

At the same time, progress in Natural Language Processing (NLP) and in large language models (LLMs) has made it possible to automate complex linguistic tasks, such as summarization, sentiment analysis, and question answering. Combined with modern scraping techniques and APIs, these models can have powerful information synthesis and delivery tools.

The purpose of the thesis is to design and implement a system that collects news articles from reliable sources through web scraping automatically, processes the content using a series of language model-based tools, and produces concise summaries. Additionally, the system converts these summaries into audio using Text-to-Speech (TTS) technology, creating automated newsletters and podcasts.

To orchestrate these processes in a dynamic way, the system relies on an agent-based architecture capable of invoking different tools depending on the context and the user's query. The tools, exposed through an MCP (Model Context Protocol) server, enable a scalable and extensible solution, following the user's needs and adaptable to multiple domains.

The motivation of this work is to reduce the cognitive and operational overhead associated with continuous information tracking, enhance user engagement through multimodal content delivery, and explore the capabilities of language models working

with real-world information. By combining automation, scraping, summarization, and audio generation, the proposed system offers a scalable and accessible solution for modern content consumption.

1.2 Goals

The implemented solution consists of four key components: an automated web scraping pipeline, an MCP server that provides all the needed tools, the integration with Large Language Models (LLMs), and a modular agent-based architecture. Each of these plays a central role in addressing the core challenges related to information overload, content accessibility, and process automation.

The main objectives of this thesis are to design and implement a system that automatically collects news articles from multiple online sources, generates concise and precise summaries of collected articles using LLMs, transforms the scraped articles into audio content using Text-to-Speech technology, facilitates dynamic interaction through an agent that can select and execute tools based on user input, and finally offers a scalable and extensible architecture adaptable to various domains.

1.3 Thesis structure

The thesis is organized into five main chapters, each addressing a key aspect of the work.

- **Chapter 1 – Introduction**: This chapter presents the motivation for the work, the challenges, and the objectives of the proposed system. It introduces the overall goals of developing an automated, agent-based framework for news scraping, summarization, enrichment, and multimodal delivery.

- **Chapter 2 – State of the art**: This chapter provides the theoretical and technological background of the thesis. It discusses the evolution of Large Language Models (LLMs) and their role in Natural Language Processing, the birth of tool-oriented agent-based architectures, and the introduction of the Model Context Protocol (MCP). It also discusses web scraping techniques, summarization approaches, multimodal generation, and the use of vector databases for semantic retrieval.

- **Chapter 3 – Methodology**: Describes the design and implementation of the system. The chapter details the agent architecture, workflow from user input to final output, and the selected instruments such as MCP, Firecrawl

for scraping, PostgreSQL with `pgvector` for storage, OpenAI models for summarization and TTS, and the frontend infrastructure.

- **Chapter 4 – Prototype and Use Case Demonstration**: This chapter shows the functioning of the implemented prototype through concrete use cases. It presents the website interface, newsletter subscription and delivery pipeline, podcast generation, and interactive chat system. The chapter also includes analyses of usage and cost, evaluating daily and multi-user scenarios.
- **Chapter 5 – Conclusion and Future Work**: Summarizes the contributions of the thesis, highlighting the effectiveness of MCP-based orchestration for building modular and scalable news delivery systems. It also outlines possible extensions, including multilingual support, advanced personalization, cloud-native deployment, and integration with third-party MCP servers.

Chapter 2

State of the art

The purpose of this chapter is to provide an overview of the main concepts, models, and frameworks that form the foundation of this thesis. Starting from the evolution of language models and their role in Natural Language Processing (NLP), we discuss how these systems have progressively advanced from sequential architectures to transformer-based approaches, setting the stage for modern Large Language Models (LLMs). The chapter then explores agent-based systems and their tool-oriented architectures, with a focus on the Model Context Protocol (MCP), followed by techniques for web scraping, data enrichment, summarization, and multimodal content generation. Together, these topics present the current landscape of research and development, highlighting practical implementations that inspire the design of our system.

2.1 Large Language Models (LLMs) and Natural Language Processing (NLP)

Large Language Models represent the keystone of recent progress in Artificial Intelligence, especially within the domain of Natural Language Processing (NLP). Their ability to capture long-range dependencies, generate coherent text, and adapt across multiple tasks has redefined the boundaries of what automated systems can achieve. In this section, we introduce the historical progression of NLP architectures, beginning with sequential models such as Recurrent Neural Networks (RNNs) and proceeding toward the transformer architecture. This highlights how foundational innovations in model design have enabled the emergence of state-of-the-art LLMs, which now power agent-based frameworks and multimodal applications.

2.1.1 From Sequential Models to the Transformer Architecture

Before the emergence of transformer-based models, *Recurrent Neural Networks (RNNs)* represented the foundational approach for modeling sequences in Natural Language Processing (NLP). These networks are designed to process input sequences one element at a time, maintaining a hidden state that propagates information from one timestep to the next. The goal is to capture dependencies between components in a sequence, such as words in a sentence or frames in a speech signal.

Recurrent neural networks (RNNs), particularly *Long Short-Term Memory (LSTM)* networks [1] and *Gated Recurrent Units (GRU)* [2], have long been considered the state of the art in sequence modeling. These gated variants were introduced to address certain weaknesses of vanilla RNNs, particularly the difficulty of learning long-term dependencies.

As shown in their structure, an RNN receives at each timestep t the current input x_t and the hidden state from the previous timestep h_{t-1}, and outputs both a new hidden state h_t and, optionally, a prediction y_t. The same model is applied recursively across the entire sequence.

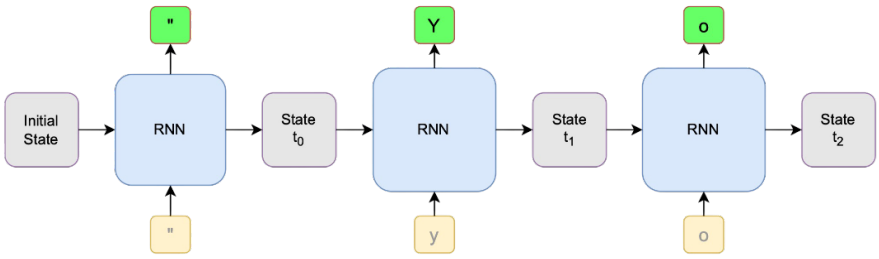

Figure 2.1: Structure of a basic RNN. Each timestep processes an input token and propagates a hidden state to the next.

The above Figure 2.1 illustrates an example of the sequential dependency: suppose we are processing the sequence `"you..."` and require the model to correctly capitalize. The model begins with `"`, and its hidden state learns that this likely starts a dialogue, so the next character should be capitalized. It then sees `y`, outputs `Y`, and updates its hidden state to remember this decision. The next character depends on the current one and the hidden state, continuing one step at a time.

This *inherent sequential nature* of RNNs limits parallelization during training, since the output at timestep t cannot be computed until the computation for $t-1$ is complete. This bottleneck leads to inefficiencies when training on modern hardware like GPUs, which excel at parallel computation. Furthermore, *long sequences* introduce memory constraints that restrict batch sizes and slow down training further.

During training, all timesteps in the unrolled network are computed forward, and a loss function (typically cross-entropy) is calculated based on the difference between predicted and target outputs. Gradients are then propagated backward across the entire sequence via *Backpropagation Through Time (BPTT)*. Since the weights of the network are shared across all timesteps, the gradient at each step is computed recursively. This introduces several stability issues:

- **Vanishing and exploding gradients:** Gradients are computed as products of many terms, which can shrink to near-zero or grow uncontrollably, impeding learning.
- **Long-term dependency problems:** Earlier states are gradually forgotten as new inputs arrive.
- **Computational inefficiency:** Training must proceed one step at a time due to hidden state dependencies.

To address these issues, *gated RNNs* like LSTM and GRU were introduced. They incorporate internal mechanisms (input, output, and forget gates) that help preserve information over longer sequences and regulate the flow of gradients. Nevertheless, they still do not completely solve the limitations around sequential computation and memory retention over long contexts.

These challenges paved the way for the development of a new architecture that could overcome the fundamental bottlenecks of RNNs. The breakthrough came with the introduction of the *Transformer* by Vaswani et al. [3], in their paper titled *"Attention is All You Need"*. This architecture replaced recurrence with a fully parallelizable *self-attention* mechanism, enabling tokens in a sequence to attend to each other directly and simultaneously, as shown in Figure 2.2.

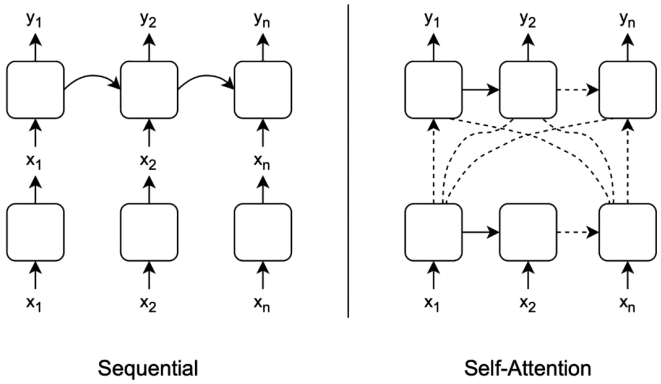

Figure 2.2: Comparison of RNN (sequential) and Transformer (Self-Attention) architectures. RNNs process input sequentially; Transformers process the entire sequence simultaneously through self-attention.

The Transformer architecture computes dependencies using a scaled dot-product attention mechanism, which calculates relevance scores between all token pairs. These attention scores are then used to compute contextualized representations of each token. Unlike RNNs, this method does not rely on hidden states or sequential propagation, making the model highly efficient and scalable on modern hardware.

Furthermore, because Transformers do not encode order by design, they include *positional encodings* to capture the position of tokens in a sequence. This allows the model to reason about word order even though it processes tokens in parallel.

Thanks to these innovations, the Transformer has become the foundation for all modern Large Language Models (LLMs), including BERT, GPT-2, GPT-3, and T5. It enables models to learn from massive corpora and generalize effectively across a wide range of NLP tasks, all while training efficiently and capturing long-range dependencies.

2.1.2 Architecture of the Transformer Model

The Transformer architecture, introduced by Vaswani et al. [3], brought a fundamental shift in how sequence data is processed. Unlike recurrent models, which rely on step-by-step computations and hidden states, the Transformer uses a purely attention-based mechanism that allows for parallel computation and direct access to all tokens in a sequence.

The model follows a classic *encoder–decoder* structure. The **encoder** receives an input sequence and transforms it into a sequence of contextualized vector representations. The **decoder** then uses these encoded representations, along with previously generated outputs, to produce the final output sequence in an autoregressive fashion.

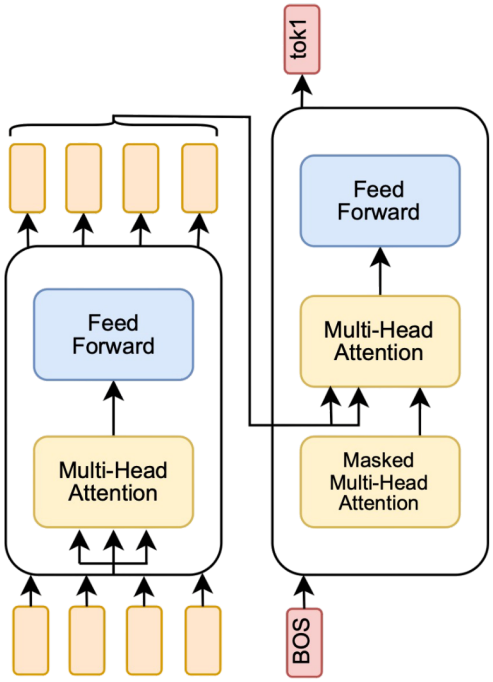

Figure 2.3: Simplified version of the Transformer architecture as proposed by Vaswani et al. [3], featuring stacked encoder and decoder blocks composed of multi-head attention and feed-forward sublayers.

As illustrated in Figure 2.3, each encoder layer consists of two primary components:

1. **Multi-Head Self-Attention**: This mechanism enables each token to attend to all other tokens in the sequence, capturing a wide range of dependencies.
2. **Position-Wise Feed-Forward Network**: A fully connected neural network applied identically and independently to each position, adding non-linearity and further transformation capacity.

Each decoder layer includes the same components as the encoder but adds a third:

3. **Masked Self-Attention**: Prevents the decoder from attending to future positions, preserving the autoregressive property.
4. **Encoder–Decoder Attention**: Enables each decoder token to attend to all encoder outputs, facilitating alignment between input and output.

All sublayers are surrounded by **residual connections** followed by **layer normalization**, ensuring better gradient flow and stable training. This results in a

consistent architecture where each layer can refine representations while preserving prior context.

Self-Attention Mechanism. The core idea of attention is to compute a weighted sum of value vectors based on a similarity between queries and keys. Given input vectors $X \in \mathbb{R}^{n \times d}$, the model learns linear projections to form:

$$Q = XW^Q, \quad K = XW^K, \quad V = XW^V$$

Then computes:

$$\text{Attention}(Q, K, V) = \text{softmax}\left(\frac{QK^T}{\sqrt{d_k}}\right)V$$

To allow the model to jointly attend to information from different representation subspaces, the Transformer uses **multi-head attention**, as reported in Figure 2.4, computing multiple attention distributions in parallel and concatenating their results.

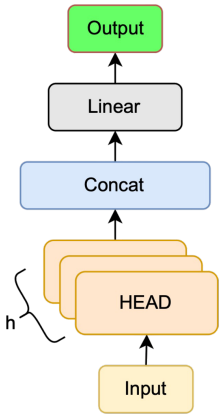

Figure 2.4: Multi-head attention mechanism: parallel attention heads allow the model to focus on different semantic relationships simultaneously.

Feed-Forward Network. Each attention output is passed through a position-wise feed-forward network:

$$\text{FFN}(x) = \text{ReLU}(xW_1 + b_1)W_2 + b_2$$

This is applied to each token independently, enabling non-linear transformations while preserving position specificity.

Positional Encoding. Since the Transformer does not use recurrence or convolution, it needs a way to incorporate information about token order. This is achieved through **positional encoding** vectors added to input embeddings. Vaswani et al. proposed a deterministic encoding based on sinusoidal functions:

$$\mathrm{PE}_{(pos,2i)} = \sin\left(\frac{pos}{10000^{\frac{2i}{d}}}\right), \quad \mathrm{PE}_{(pos,2i+1)} = \cos\left(\frac{pos}{10000^{\frac{2i}{d}}}\right)$$

These encodings allow the model to learn relative and absolute positions of tokens, facilitating the learning of order-sensitive tasks such as translation or summarization.

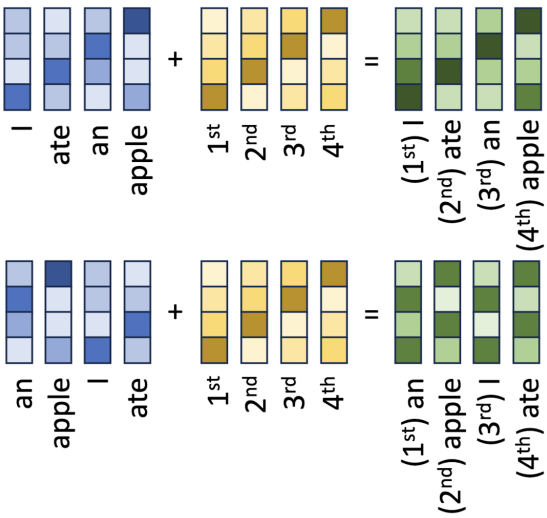

Figure 2.5: Example of sinusoidal positional encodings for different embedding dimensions. The pattern of each dimension varies across positions, helping the model infer token order.

From the Original Transformer to the Current State of the Art. Since the introduction of the original Transformer in 2017, numerous architectural variants and scaling strategies have been developed, each addressing specific limitations such as computational efficiency, context length, or parameter scaling. Today, state-of-the-art (SOTA) large language models (LLMs) are still based on the Transformer architecture, but incorporate significant modifications:

- **GPT-family (GPT-2, GPT-3, GPT-4, GPT-4o, GPT-5)**: autoregressive decoders optimized for large-scale pretraining, characterized by billions of parameters and optimized scaling laws. These models dominate in open-ended text generation and conversational AI.

- **BERT and Derivatives (RoBERTa, DistilBERT, ALBERT)**: encoder-only architectures designed for bidirectional context modeling, widely used in classification and retrieval tasks. They trade off generative ability for efficiency in discriminative tasks.
- **T5 and Encoder–Decoder Models (BART, FLAN-T5)**: encoder–decoder Transformers fine-tuned for multi-task learning and instruction following, enabling high performance across summarization, translation, and reasoning benchmarks.
- **Efficient Transformers (Longformer, BigBird, Performer)**: introduce sparse or low-rank attention mechanisms to reduce the quadratic complexity of self-attention ($O(n^2)$) to linear or near-linear, making it possible to process sequences with thousands of tokens.
- **Mixture-of-Experts Models (Switch Transformer, GLaM, Mixtral)**: employ sparse activation of expert subnetworks, increasing model capacity (trillions of parameters) while keeping inference cost manageable.
- **Multimodal Transformers (CLIP, Flamingo, GPT-4V)**: extend the Transformer to vision–language and speech–language settings, integrating cross-modal attention for joint reasoning across text, images, and audio.

In terms of complexity, the baseline Transformer scales quadratically with sequence length due to the self-attention mechanism. Modern approaches mitigate this by introducing **sparse attention**, **low-rank approximations**, or **state space models** (e.g., S4, Mamba), which provide linear-time alternatives while preserving representational power. As of today, the Transformer and its derivatives remain the backbone of all leading LLMs, with progress driven by scaling model size, optimizing efficiency, and extending context length.

Model Type	**Representative Models**	**Complexity / Features**
Encoder-only	BERT, RoBERTa, DeBERTa	Strong for classification, NLU; quadratic attention
Decoder-only	GPT-3, GPT-4, GPT-4o, LLaMA, Mixtral	Generative models; large-scale pretraining; autoregressive decoding
Encoder–Decoder	T5, BART, mT5, UL2	Effective for translation, summarization; bidirectional + autoregressive
Efficient Transformers	Longformer, BigBird, Performer, Linformer	Sparse / low-rank / kernelized attention; complexity $\mathcal{O}(n \log n)$ or $\mathcal{O}(n)$
Mixture-of-Experts	Switch Transformer, GLaM, Mixtral	Trillions of parameters; only a subset of experts activated per forward pass
Multimodal Transformers	CLIP, Flamingo, GPT-4V, Gemini	Joint text–image (and audio/video) understanding; cross-attention fusion

Table 2.1: Overview of Transformer-based model variants and their main features.

2.2 Agents and Tool-Oriented Architectures

Recent advances in artificial intelligence have led to increasingly autonomous systems capable of interacting with their environment in complex and dynamic ways. In this context, the concept of an *agent* plays a central role: it represents an entity able to perceive, reason, and act in order to achieve specific goals. At the same time, the integration with external tools (*tool-oriented architectures*) has expanded agents' capabilities, allowing them to combine internal reasoning with the use of external resources such as APIs, databases, or search engines. This section explores the fundamental characteristics of agents, their historical evolution, and their integration with tool-oriented architectures, with a particular focus on recent implementations leveraging LLMs.

2.2.1 Definition and Evolution of LLM-Based Agents

The concept of an **agent** has been foundational in artificial intelligence. Traditionally, agents are defined as autonomous systems that interact with an **environment** by receiving inputs, processing them, and executing outputs to achieve specific goals. These interactions occur through **sensors** (which receive *percepts*, such as text or data) and **actuators** (which produce *actions*, such as web searches or tool usage). Classical agents were constructed with modular components for perception, reasoning, planning, and execution. While structured and interpretable, these systems generally lacked adaptability and generalization capabilities beyond predefined tasks.

The emergence of **Large Language Models (LLMs)** has significantly transformed agent architectures. Pretrained on vast and diverse corpora of text, LLMs like GPT-3 and its successors embed broad linguistic knowledge, commonsense reasoning, and factual understanding. This obviates the need for task-specific rules or programming. As a result, a new class of **LLM-based agents** has emerged, where the LLM functions as an **"augmented" reasoning core**—capable of interpreting percepts, planning, and dynamically executing actions in open-ended, evolving environments.

In this paradigm, the LLM-based agent is not a static predictor but an **interactive system**. As illustrated in Figure 2.6, the agent continuously engages its environment, receiving percepts through **sensors**, performing planning *through reasoning*, and influencing the external world through **actuators**. The agent may utilize external **tools** (e.g., search engines, APIs) and internal **memory** to store and retrieve context or historical information. According to the taxonomy proposed by Xu et al. [4], such agents operate in iterative cycles composed of five key processes:

- **Perception**: Interpreting inputs received via sensors, including natural language instructions, search results, or API responses.

- **Planning**: Decomposing tasks into actionable steps and identifying intermediate goals.

- **Memory**: Storing and recalling relevant information from prior observations, plans, or interactions.

- **Reflection**: Evaluating past reasoning and actions to identify errors and improve future strategies.

- **Action**: Executing outputs via actuators—such as generating text, invoking tools, or performing actions that affect the environment.

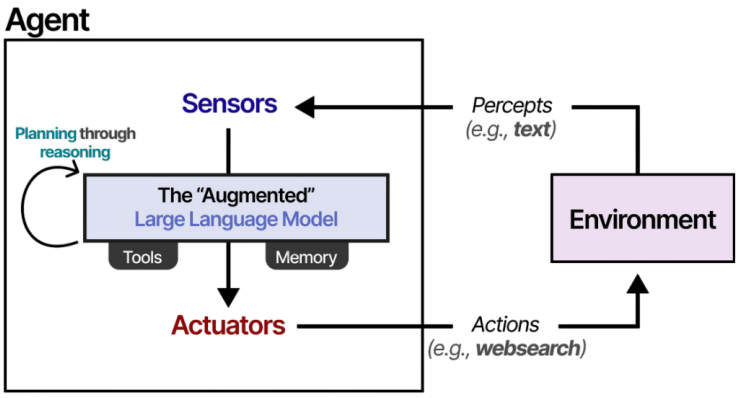

Figure 2.6: Typical decision loop of an LLM-based agent: perception, planning, memory, reflection, and action (adapted from Xu et al. [5]).

This architecture fundamentally departs from traditional single-pass LLM usage. Rather than generating static outputs from a fixed input, **LLM-based agents function as dynamic, goal-driven entities** capable of reasoning, remembering, adapting, and acting through continuous interaction with their environment.

Having introduced the five foundational processes -Perception, Planning, Memory, Reflection, and Action—that compose the interactive loop of LLM-based agents, we now provide a deeper examination. This section outlines their roles, implementation challenges, and examples based on recent developments in tool-augmented reasoning, as surveyed by Zhou et al. [5].

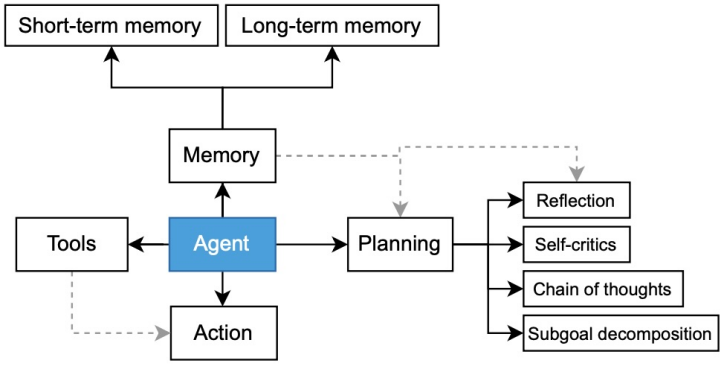

Figure 2.7: Illustration of an LLM-based agent architecture showing the interaction between core components, memory systems, and planning submodules.

The architecture in the above Figure 2.7 illustrates how an LLM-based agent is structured around modular and interactive components. Let's now break down each key module based on the design elements described in Zhou et al. [5] and the conceptual layout shown in the figure.

The Agent

At the center of the architecture lies the **Agent**, the core decision-making unit responsible for interpreting tasks, orchestrating components, and selecting suitable reasoning or tool-use strategies. The agent evaluates task complexity and determines whether to engage tools, access memory, decompose subgoals, or reflect on past decisions. It serves as the dynamic controller of all downstream processes.

Tools

Tools extend the agent's native capabilities by providing interfaces to specialized operations. Examples include:

- *Calculator* – for arithmetic and numerical reasoning,
- *Code Interpreter* – for executing and debugging code,
- *Calendar* – for time-related planning and scheduling,
- *Search* – for retrieving real-time or external knowledge.

These tools are invoked when the agent recognizes that a subtask exceeds its intrinsic reasoning ability and requires external functionality. Their integration transforms LLMs into versatile, interactive problem-solvers.

Memory

The agent uses both **short-term** and **long-term memory**:

- *Short-term memory* retains transient information like recent interactions or intermediate tool outputs.
- *Long-term memory* stores persistent information, such as user preferences or previously learned knowledge.

This memory separation allows the agent to maintain context within a session while also personalizing responses across sessions.

Planning

Planning enables the agent to organize complex tasks into structured procedures. The agent can choose among different strategies:

- Implicit planning through chain-of-thought reasoning,
- Explicit planning by generating structured action plans,
- Mixed approaches involving dynamic replanning.

Planning also involves selecting tool sequences and managing dependencies between subtasks. It supports coherent behavior across multiple steps.

In contemporary LLM-based agent systems, planning is not a singular process but a composition of complementary reasoning and control mechanisms. As shown in Figure 2.7, several techniques are frequently employed as subcomponents of planning:

Reflection *Reflection* introduces a meta-cognitive dimension to planning. After executing a reasoning or action step, the agent evaluates whether its decision was effective. If not, it may revise earlier steps, try alternative tools, or replan from a different perspective. Reflection increases the agent's resilience and learning capacity in uncertain or iterative settings.

Self-Critics *Self-Critics* builds on reflection by allowing the agent to proactively critique its own outputs before finalizing them. This internal feedback loop functions as a quality-control mechanism, helping identify flawed logic, suboptimal plans, or misused tools. It is especially useful in domains requiring high accuracy and minimal human intervention.

Chain-of-Thought *Chain-of-Thought* (CoT) is a reasoning strategy where the agent articulates intermediate steps in natural language before arriving at a conclusion. This promotes transparency, improves logical consistency, and reduces hallucination. CoT is particularly effective for arithmetic, multi-hop question answering, and tasks requiring logical deduction.

Subgoal Decomposition *Subgoal decomposition* complements CoT by providing structural scaffolding. Instead of solving an entire task in one go, the agent first splits it into smaller subgoals or milestones. Each subgoal can then be tackled with its own plan, tool usage, and reasoning process. This modularity enhances robustness and allows dynamic adaptation in case of failures.

2.2.2 Advanced Architectures for LLM-Based Agents

As LLM-based agents evolve beyond single-shot prompting paradigms, recent architectural surveys, such as Zhou et al. [5], highlight the emergence of sophisticated modules that underpin agentic behavior. These include memory systems, external knowledge integration, planning strategies, and interaction schemes—each designed to enhance decision-making, adaptability, and multi-agent coordination. This section draws upon the structural and conceptual breakdown presented in the architectural summary (Figure 2.8) and extended in the technical framework outlined in [5], providing a comprehensive view of how modern LLM agents operate.

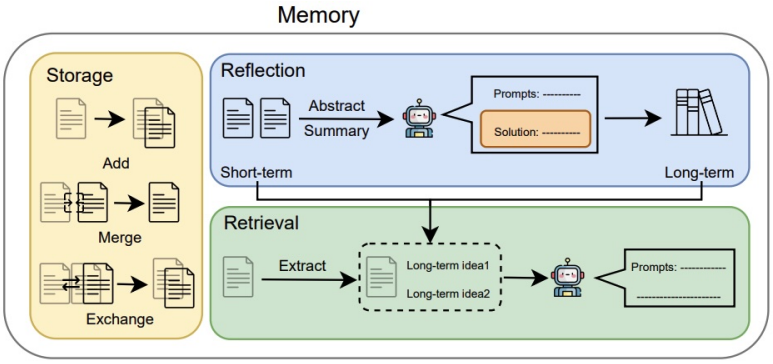

Figure 2.8: Overview of the memory architecture in LLM-based agents. The memory system consists of storage, retrieval, and reflection.

The Memory Model

Memory in LLM-based agents is no longer limited to the input context window. Instead, it is structured across multiple layers:

Memory Retrieval allows agents to extract contextually relevant data from either short-term or long-term storage. In short-term cases, this often involves appending recent tool outputs or user queries. Long-term memory, by contrast, requires filtering and relevance-based retrieval, often implemented through embedding-based semantic search in vector databases. These mechanisms mirror techniques explored in retrieval-augmented generation (RAG) models [6] and agent platforms such as LangChain.

Memory Reflection refers to the process by which agents consolidate experiences from past interactions to refine future performance. This can involve summarizing recent sessions, generalizing patterns of error or success, and updating internal representations. In multi-agent contexts, Zhou et al. [5] describe *centralized memory reflection*, where a master agent integrates knowledge from subordinate agents. Memory reflection is typically achieved through mechanisms such as episodic memory storage, which records past interactions; pattern extraction or clustering, which identifies recurring trends or mistakes; and representation updates, where learned knowledge modifies the agent's internal models or decision policies. These processes enable agents to adapt over time, improving both efficiency and accuracy in subsequent tasks.

Memory Storage and Modification involves determining how to encode and persist information. Depending on the task, this may involve natural language, structured data, or multimodal content (e.g., image features or audio embeddings). Modification strategies include appending new data, merging with similar entries, or replacing outdated or incorrect knowledge. Such dynamics align with continual learning frameworks and emphasize the shift toward lifelong learning in agent systems.

Utilization of Knowledge

LLM-based agents enhance their performance by combining internal, multimodal, and external knowledge sources:

- **Internal Textual Knowledge:**
  - Learned during pretraining on large-scale corpora.
  - Supports language understanding, generation, translation, etc.
  - *Limitation:* Static and potentially outdated [5].
  - *Best used for:* Tasks relying on general language understanding, commonsense reasoning, or knowledge that does not require real-time updates.

- **Multimodal Knowledge:**
  - **Visual:** Represented through embeddings (e.g., visual transformers) and integrated via self-attention.
  - **Audio:** Processed through spectrograms or speech encoders embedded in shared vector spaces.
  - Enables tasks like VQA, image captioning, and speech interaction [7].
  - *Best used for:* Tasks requiring perception and interpretation of non-textual information, such as images, videos, or speech.
- **External Knowledge Integration:**
  - **Web Scraping & API Calls:** Fetch real-time data for dynamic tasks (e.g., news, stock prices) [8].
  - **Database/Knowledge Base Queries:** Access structured repositories (e.g., PubMed, ChatDB).
  - **Retrieval-Augmented Generation (RAG):** Combines retrieval with generation to improve factuality [6].
  - *Best used for:* Tasks requiring up-to-date information, domain-specific data, or high factual accuracy beyond the agent's pretrained knowledge.

Reasoning and Planning Strategies

LLM agents can plan and reason using different approaches:

- **One-Step Planning:**
  - Single-pass decomposition of tasks into subgoals.
  - Fast but less adaptive to changes or feedback.
- **Multi-Step Planning:**
  - Iterative refinement over multiple reasoning cycles.
  - Supports feedback, reflection, and dynamic task adjustment [9].

Having this two strategies, we can note that **One-step** planning is best suited for straightforward or time-critical tasks where speed is more important than adaptability. In contrast, **multi-step** planning is preferred for complex, long-horizon tasks that require iterative reasoning, the ability to handle uncertainty, and the incorporation of feedback.

Agent Interaction Schemes

Multi-agent systems (MAS) follow different coordination models:

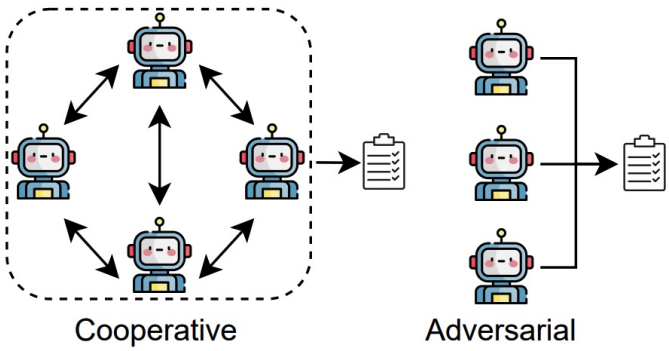

Figure 2.9: Illustration of interaction schemes in multi-agent systems, cooperative and adversarial.

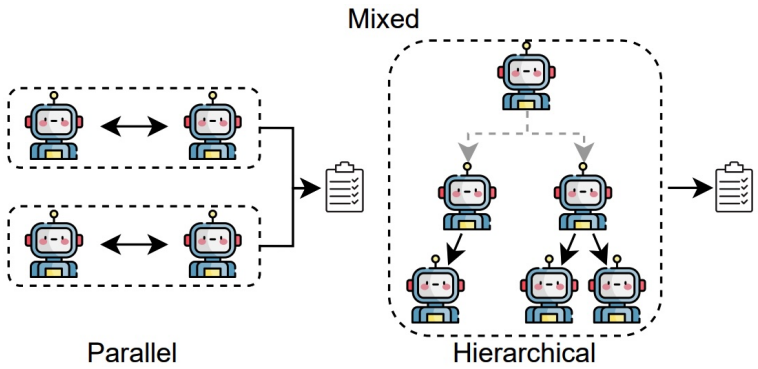

Figure 2.10: Illustration of parallel and hierarchical interaction schemes in multi-agent mixed systems.

- **Cooperative:**
  - As shown in the (Figure 2.9), agents share goals, decompose tasks, and collaborate through communication.
- **Adversarial:**
  - As shown in the (Figure 2.9), agents pursue competing objectives with strategic behavior (e.g., negotiation, game theory).
- **Mixed:**

– As shown in the (Figure 2.10), combines cooperative and competitive elements; can be parallel or hierarchical. Commonly used in complex systems with partial alignment between agents.

Considering these interaction schemes, **cooperative** models are best suited for scenarios where agents share common goals and benefit from collaboration, while **adversarial** models are appropriate when agents have competing objectives and strategic behavior is required. **Mixed** interaction schemes are particularly useful in complex systems where agents may partially align, combining both cooperative and competitive elements.

Prompting Techniques Prompting serves as a primary method for guiding the behavior of LLM-based agents and structuring their reasoning processes. By providing carefully designed instructions or examples, agents can be directed to produce desired outputs without altering their internal parameters. The main prompting strategies include:

- **Zero-shot prompting:** Used when no examples are available and the agent must generalize from the instruction alone; ideal for novel or exploratory tasks.
- **Few-shot prompting:** Provides a small set of exemplars to demonstrate the expected behavior, improving performance on tasks with moderate complexity or ambiguous instructions.
- **Chain-of-thought prompting:** Encourages the model to generate intermediate reasoning steps, effective for multi-step or logical tasks requiring transparency and interpretability.
- **Instruction-tuned prompting:** Leverages pre-trained models fine-tuned on large instruction datasets, yielding higher reliability on standard benchmarks and repetitive tasks.

Prompting is typically performed through natural language instructions, optionally augmented with structured formats, templates, or retrieval-augmented context to enhance factual accuracy and domain specificity. Effective prompting allows agents to dynamically adapt their reasoning, planning, and decision-making behaviors without explicit retraining, complementing the modular architectures and memory systems described in this chapter.

2.2.3 Agent Frameworks: LangChain and LangGraph

The rapid adoption of LLM-based agents has been accompanied by the emergence of specialized frameworks that simplify their construction, orchestration, and

deployment. Two of the most widely used open-source frameworks are **LangChain** and **LangGraph**, each offering complementary approaches to agent design and execution.

LangChain

LangChain provides a high-level framework for building applications powered by LLMs. Its architecture centers on the composition of *chains*, which are modular sequences of prompts, models, and tools. It also integrates mechanisms for **retrieval-augmented generation (RAG)**, memory management, and external API interaction. The main strengths of LangChain lie in its flexibility and ecosystem support, making it suitable for a wide variety of use cases, from simple chatbots to enterprise-grade knowledge assistants. However, its general-purpose nature can lead to higher complexity and overhead in agent-specific workflows [10].

LangGraph

LangGraph extends the agent paradigm by modeling agentic workflows as **graphs**, where nodes represent steps such as reasoning, tool invocation, or memory retrieval, and edges define the flow of execution. This design allows agents to operate as **state machines** with deterministic control over transitions, while still leveraging the generative capabilities of LLMs. Compared to LangChain, LangGraph emphasizes **determinism, reproducibility, and control**, making it particularly suited for research and production systems where predictable behavior is essential. As highlighted in recent surveys [5], graph-based orchestration is emerging as a key direction for agent reliability and safety.

Comparison and Role in Agent Architectures

While LangChain focuses on composability and ecosystem integration, LangGraph provides a more structured and transparent execution model. In practice, they can be seen as complementary: LangChain for rapid prototyping and integration of diverse components, LangGraph for precise control over agent reasoning and reproducibility. Together, these frameworks exemplify the trend of moving from ad-hoc LLM wrappers toward systematic, modular, and controllable agent infrastructures.

In our methodology (see Section **??**), we adopt **LangGraph** as the foundation for the scraping agent, leveraging its graph-based design to manage tool orchestration and ensure coherent, context-aware responses.

2.3 Model Context Protocol (MCP) for Agent-Based Systems

As LLM-based agents become more capable and integrated into complex environments, the need for robust context management and standardized interactions grows. Agents frequently rely on multiple tools, APIs, and knowledge sources to perform tasks that require both reasoning and real-time information retrieval. The Model Context Protocol (MCP) provides a framework to address these challenges, offering a consistent interface for managing contextual information, coordinating tool usage all in one place, and ensuring coherent multi-step execution. This section introduces the motivation behind MCP, its role in agent architectures, and the mechanisms through which it enables scalable and reliable agent-based systems.

2.3.1 Motivation and Role in Agent Architectures

Modern LLM-based agents must interact with a variety of tools, APIs, and external data sources while maintaining coherent, persistent context. The **Model Context Protocol (MCP)** addresses this need by defining a structured interface that mediates between models and their operational environments. MCP was first introduced by **Anthropic**, the organization behind the Claude AI models, in late 2023. Its primary purpose is to provide a standardized method for AI models to communicate with external tools such as calendars, task managers, CRMs, Notion, or Slack in a structured and consistent manner.

Before MCP, AI models were often connected to external services through ad hoc, tool-specific APIs, leading to fragile integrations, inconsistent data handling, and difficulties in scaling agent architectures. MCP was designed to solve these challenges by providing a unified communication protocol that abstracts the underlying tool interfaces. This allows developers to integrate a wide variety of resources while maintaining persistent model context, improving both interoperability and maintainability.

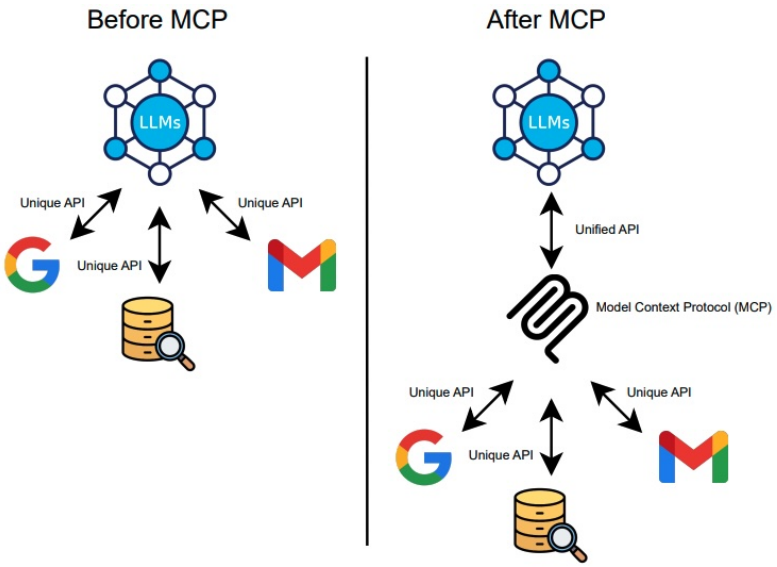

Figure 2.11: Comparison of tool integration before and after the introduction of the Model Context Protocol (MCP).

Figure 2.11 illustrates the fundamental motivation behind MCP. In the traditional setup (left), large language models interact with external services through multiple tool-specific APIs, each requiring custom integration logic. This results in increased development complexity, inconsistent data handling, and poor scalability. By contrast, the MCP-based architecture (right) introduces a unified communication layer between LLMs and external resources. Through the MCP server, agents can access a wide variety of tools and services without being tightly coupled to their individual interfaces. This abstraction simplifies system design, improves maintainability, and enables more flexible and interoperable agent-based applications.

2.3.2 Protocol Structure and Components

MCP follows a client-server model, where **clients** act as intermediaries embedded in model-hosting applications (e.g., IDEs, desktop agents, applications) and **servers** expose tool functionalities or contextual resources. The interaction between model, client, and server occurs through structured messages and lifecycle phases that ensure stability and traceability.

Key elements of the protocol include:

- **Hosts:** Applications that the user interacts with. They contain the model and the MCP client, responsible for initiating and routing requests.

- **MCP Clients:** Embedded components that format, serialize, and dispatch model-generated requests to the appropriate server endpoints.
- **MCP Servers:** External programs or services that expose tools, memory access, or resources through the MCP interface.

This structure allows for clean separation of concerns: the model focuses on reasoning, the client on mediation, and the server on executing actions or returning results.

2.3.3 MCP Server Lifecycle

As Figure 2.12 illustrates, the MCP is organized around a three-phase server lifecycle, each with distinct technical implications:

1. **Creation:** An MCP server is instantiated with a schema that defines available tools and data modalities. This allows the agent to discover and index callable functions.
2. **Operation:** The server processes structured requests sent via the MCP client, executes the corresponding action, and returns a standardized response (often JSON).
3. **Update:** The toolset or state of the server may evolve, and this phase governs the communication and synchronization of such changes.

This lifecycle ensures that models interact with a consistent view of tool capabilities and responses, improving robustness and facilitating auditability.

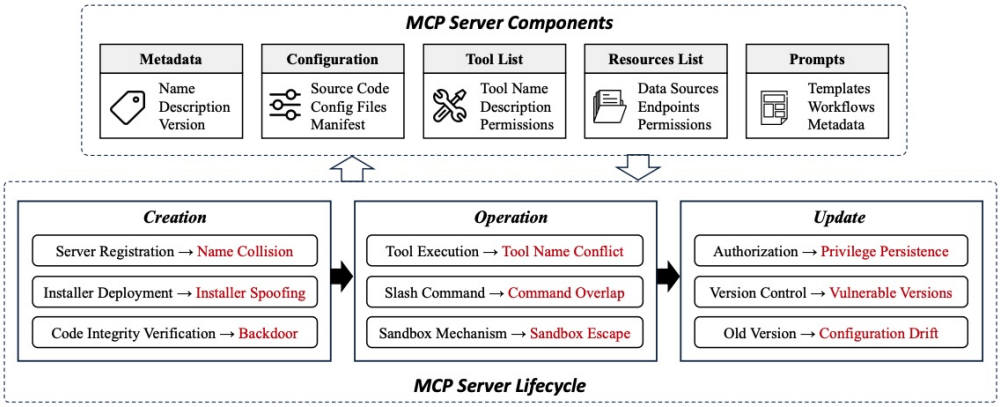

Figure 2.12: Illustration of MCP server components and lifecycle.

2.3.4 Security and Context Isolation

Hou et al. [11] emphasize the importance of security and context management in MCP. Since the protocol enables access to potentially sensitive tools (e.g., file systems, code interpreters), strong guarantees are required regarding access control, input sanitization, and data isolation.

Each MCP session operates within a defined scope, which can be aligned with user identity, task context, or memory segments. This scoped interaction model supports granular permissioning and reduces the risk of accidental cross-contamination of state or intent.

2.3.5 Implications for Agent-Based Systems

From an agent architecture perspective, MCP offers several advantages:

- **Modularity:** Tools and capabilities can be added or removed without retraining the model or rewriting prompts.
- **Traceability:** Every interaction is structured and serializable, enabling better debugging, logging, and reflection.
- **Multi-agent support:** MCP's structured communication enables multiple agents to share tools and operate over a shared memory or state space.
- **Composability:** Complex workflows involving multiple tools or agents can be composed from discrete, well-defined actions governed by the protocol.

These properties make MCP especially well-suited for advanced LLM applications such as IDE assistants, research agents, and collaborative agent ecosystems.

2.4 Web Scraping and Data Extraction

Access to real-time, structured, or semi-structured web content is crucial for LLM-based agents tasked with open-domain reasoning, data-driven decision-making, or dynamic workflows. Web scraping and data extraction have thus become foundational components in enabling agents to interact with the web as an external source of truth or memory.

This section outlines the current state of the art in data extraction, spanning from low-level browser automation to high-level agentic reasoning frameworks that parse and interpret web content.

2.4.1 Visual and DOM-Based Extraction

State-of-the-art extraction systems combine visual rendering with DOM structural analysis to generalize across diverse web layouts. The **Document Object Model (DOM)** is a tree-structured representation of an HTML or XML document, where each element (such as headings, paragraphs, images, or links) is represented as a node in a hierarchical structure. This model provides both the syntactic organization of the page and the relationships between elements, making it the standard way for programs and scripts to dynamically access and manipulate web content.

Traditional extraction approaches often relied on brittle selectors (e.g., XPath or CSS paths) tied to specific DOM positions. However, small layout changes could easily break such rules. To address this, modern systems analyze the DOM in combination with visual cues such as spatial layout, rendering order, or saliency. By segmenting the page into meaningful regions and clustering elements with similar visual or structural features, these systems can robustly identify and extract relevant content across heterogeneous websites.

This combination of visual and DOM-based techniques is particularly effective for extracting:

- Product listings or result pages,
- Articles, posts, and reviews,
- Navigation menus and metadata.

Recent research has also extended transformer architectures to operate directly on DOM structures. Approaches like **WebFormer** and **TreeBERT** model the DOM tree as a sequence enriched with layout and semantic signals, enabling more accurate recognition of key content blocks across dynamic or complex web pages.

2.4.2 Headless Browsers and Page Interaction

Modern websites increasingly render content dynamically through JavaScript, making traditional HTTP-based scraping insufficient. To handle these complexities, contemporary extraction pipelines rely on **headless browser engines** such as `Playwright`, `Puppeteer`, and `Selenium`.

These tools simulate full browser environments, enabling automated agents to:

- Wait for asynchronous content to fully render in the DOM,
- Interact with elements such as dropdowns, modals, or login forms,
- Traverse complex navigation flows and multi-step user interactions,
- Extract information directly from rendered views (e.g., tables, popups).

Headless browsers expose granular control over the browser context, allowing the automation of nearly any interaction a human user could perform. They serve as the foundational execution layer for higher-level scraping agents.

While headless browsers handle the *rendering and interaction*, they lack the capacity for semantic reasoning or strategic content selection. This gap is filled by the next generation of systems: LLM-augmented scraping agents.

2.4.3 LLM-Augmented Scraping Agents

Building on top of browser automation, recent research has introduced **language model-based agents** capable of reasoning about web pages. These agents do not merely execute scripted actions—they interpret content, make decisions, and extract structured data with minimal supervision.

In tool-augmented paradigms like ReAct [8], LLM agents can:

- Read and understand raw HTML or DOM fragments,
- Decide which elements to click or follow based on task objectives,
- Parse unstructured content into structured outputs (e.g., JSON or key-value pairs).

A state-of-the-art example is **FireCrawl**, an end-to-end framework that integrates:

- A headless browser for robust page rendering,
- DOM segmentation and heuristic clustering for content organization,
- An LLM-based extraction module that semantically labels and structures content.

Unlike rule-based scrapers, LLM-augmented agents adapt to varied layouts and noisy content, making them ideal for dynamic and heterogeneous web environments. They are particularly well-suited for use in research agents, retrieval-augmented generation systems, and general-purpose web-based assistants.

2.4.4 Structured vs. Unstructured Content

Agents often need to extract both:

- **Structured data** — e.g., tables, lists, metadata (JSON-LD, RDFa),
- **Unstructured text** — e.g., articles, comments, or freeform inputs.

While structured data can often be directly parsed, unstructured content benefits from summarization, named entity recognition, and LLM-based transformation into knowledge graphs or intermediate representations.

2.5 Content Enrichment and Summarization

Web scraping will almost always return content in a noisy, verbose, or inconsistent format, because websites are primarily designed for human consumption rather than machine processing. A typical web page intermixes the main text with advertisements, navigation menus, user comments, pop-ups, and stylistic elements that carry little or no semantic value. Moreover, the same type of content (e.g., news articles, product pages) can be presented in highly heterogeneous layouts across different domains, with variations in HTML structure, embedded scripts, and dynamically generated elements. As a result, the raw output of a scraper often contains redundant or irrelevant information, formatting artifacts, or incomplete fragments of the target content.

Automated agents therefore need to perform **content enrichment** to transform these rough inputs into well-structured data suitable for downstream tasks. Content enrichment involves a combination of processes such as summarization, fact extraction, entity tagging, and adaptation for decision-making or dialogue systems.

Summarization plays a foundational role, as it allows agents to distill essential insights from verbose inputs, reducing cognitive workload for both the system and the end user. The evolving landscape of research in extractive and abstractive summarization, as well as more specialized applications such as real-time, user-adaptive, and multi-document summarization (both synchronous and asynchronous), demonstrates the central importance of this capability in modern agent architectures.

This section surveys the main approaches, contrasting extractive and abstractive summarization methods, exploring techniques for multi-document and long-context scenarios, and highlighting the role of LLM-based agents in advancing the field.

2.5.1 Extractive vs. Abstractive Summarization

Traditional summarization methods fall into two broad categories:

- **Extractive summarization** selects and concatenates salient spans directly from the source. Classical models include TextRank, LexRank, and more recently BERTSum [12]. These methods are robust but often lack fluency or semantic coherence.
- **Abstractive summarization** generates new text that paraphrases or compresses the source content. Transformer-based models such as BART [13],

PEGASUS [14], and T5 [15] have achieved strong performance in this setting. These are especially effective for news, reviews, and dialogue summarization.

Abstractive models require more data and supervision but yield higher-quality outputs in terms of fluency and informativeness.

2.5.2 Multi-Document and Long-Context Summarization

Agents often encounter scenarios where information is spread across multiple pages or sources. Recent work addresses this through:

- **Multi-document summarization** models like Hierarchical Transformers and GSum [16], which condition on multiple passages and generate unified outputs.
- **Long-context models** such as Longformer [17], BigBird [18], and Gemini [19], which can attend to entire documents or multi-page corpora efficiently.

These systems enable context-aware reasoning and summarization over large or distributed inputs—crucial in academic search, legal documents, or technical pipelines.

2.5.3 LLM-Based Summarization Agents

Large language models (LLMs) like GPT-4, Claude now enable agents to perform summarization as a multi-step reasoning task. These agents can:

- Adapt summaries based on user intent or style,
- Generate query-focused or comparative summaries,
- Reflect and revise outputs based on quality signals [9].

Few-shot prompting and tool-augmented reasoning (e.g., ReAct [8]) allow flexible adaptation to unseen domains without retraining. Such agents are increasingly used in search assistants, research tools, and report generation pipelines.

2.6 Multimodal Output and Podcast Generation

Modern agents are expected to communicate with users across different modalities—text, image, audio, and video. This has driven the development of **multimodal output** capabilities that allow agents not only to understand but also to produce expressive, human-aligned content in various formats.

One of the most practical and fast-evolving applications is **podcast generation**, where language models automatically transform written content into rich, engaging audio experiences. This involves both linguistic planning (e.g., summarization, script generation) and high-fidelity voice synthesis. OpenAI and other leaders in the field have released models that tightly integrate these capabilities.

The following subsections explore the foundations and applications of these technologies, focusing on multimodal LLMs and real-time voice cloning for podcast narration, advances in text-to-speech and multilingual support, and a discussion of the opportunities and current limitations of multimodal output systems.

2.6.1 Multimodal LLMs and Real-Time Generation

With the release of GPT-4o [20], OpenAI introduced a unified model capable of processing and generating across **text, audio, and vision**—in real time. Unlike previous architectures with separately trained modules, GPT-4o handles all modalities natively, enabling seamless tasks like:

- Spoken dialogue with emotional nuance,
- On-the-fly image explanation and voice narration,
- Real-time feedback for tutoring or accessibility.

Similar trends are visible in Gemini [19] and Meta's ImageBind [21], with increasing attention on cross-modal alignment and reasoning. These models are setting a new standard for how agents perceive and respond within multimodal environments.

2.6.2 Voice Cloning and Podcast Narration

Text-to-speech (TTS) systems have seen massive improvement in the quality and controllability of generated audio. OpenAI's **Voice Engine** [22] can produce highly expressive speech that maintains clarity, natural rhythm, and speaker consistency—even with limited training samples.

State-of-the-art capabilities in this area include:

- **Zero- or few-shot voice cloning**, which replicates speaker identity from short audio clips.
- **Emotion and tone control**, enabling varied narrations for news, fiction, or education.
- **Integration with summarizers**, allowing content transformation pipelines for podcast production.

This allows agents to automate end-to-end workflows: ingesting large content volumes (e.g., news articles), summarizing key insights, and narrating them with natural prosody.

2.6.3 Text-to-Speech and Multilingual Support

Text-to-speech (TTS) capabilities are increasingly central to LLM-based agents, enabling them to communicate in natural spoken language and operate in audio-first contexts such as podcast narration, virtual assistants, and accessibility tools. The current state of the art in TTS emphasizes **expressiveness**, **language coverage**, and **integration with conversational agents**.

OpenAI has developed a proprietary TTS system known as the **Voice Engine** [22], which underpins voice interactions in ChatGPT and the GPT-4o framework [20]. Unlike traditional TTS pipelines, the Voice Engine produces speech that is not only intelligible and fluent, but also expressive, natural-sounding, and context-aware.

The system supports speech synthesis in over 35 languages, including:

- **European languages:** English, Spanish, French, German, Italian, Dutch, Portuguese, Swedish, Polish, Russian;
- **Asian languages:** Chinese (Mandarin), Japanese, Korean, Hindi, Thai, Vietnamese, Indonesian;
- **Others:** Turkish, Arabic, Hebrew, and more.

Support for these languages varies in quality and expressiveness. English remains the most expressive, with nuanced intonation, named voice personas, and emotional modulation. Other languages are typically rendered with clear pronunciation and fluent cadence, but may lack advanced prosodic features or stylistic variation.

The integration of TTS into fully multimodal agents—such as those built with GPT-4o—allows for seamless real-time dialogue, storytelling, and audio-first applications. These agents can listen, process, and respond with speech in a conversational loop, opening up new use cases in education, accessibility, entertainment, and beyond.

2.6.4 Opportunities and Limitations

Multimodal generation introduces significant opportunities for LLM-based agents, including:

- **Immersive experiences:** Real-time outputs improve context understanding and human-like communication.

- **Personalization and accessibility:** Voice cloning, expressive TTS, and multilingual support enable scalable content creation, such as podcasts and narrated summaries.
- **User engagement and flexibility:** Agents can adapt across tasks and deliver interactive tutorials or content pipelines.

However, multimodal agents also face notable limitations:

- **Latency and coordination:** Real-time generation across modalities requires low-latency decoding, modality synchronization, and efficient resources.
- **Dialogue memory and modality awareness:** Agents must reason over content, prosody, visual cues, and intonation.
- **Agent identity and continuity:** Maintaining a consistent persona across voices, contexts, and modalities is critical for user trust.

These opportunities and limitations are particularly relevant for podcast generation, where agents must seamlessly combine summarization, voice synthesis, emotional expression, and multilingual support while maintaining a coherent and engaging narrative for listeners.

2.7 Vector Databases

As LLM-based agents increasingly rely on semantic representations of text, traditional relational databases are often insufficient for efficient retrieval of contextually relevant information. Vector databases are designed to store high-dimensional embeddings, enabling similarity-based queries that go beyond exact keyword matching. This capability is particularly important for applications such as news summarization, semantic search, recommendation systems, and any scenario where LLMs must reason over related content rather than literal matches.

This section examine the utility of vector databases in modern AI systems, discuss embeddings and their role in vector representations, and highlight practical implementations such as PostgreSQL with the `pgvector` extension.

2.7.1 Utility of Vector Databases

Vector databases allow agents to perform similarity searches in embedding space, which captures semantic meaning rather than surface-level text features. This enables:

- **Semantic retrieval:** Quickly finding articles, documents, or other content that is conceptually related to a query, even if no exact keywords match.
- **Context-aware summarization:** Aggregating information from multiple related sources to create coherent summaries or multi-document synthesis.
- **Scalability:** Efficiently handling large collections of embeddings while maintaining fast query performance.
- **Integration with LLMs:** Supporting retrieval-augmented generation (RAG) pipelines where retrieved vectors can be fed into LLMs for improved factuality and context.

2.7.2 Embeddings and Vector Representations

LLM-based agents rely on transforming textual or multimodal content into numerical representations known as **embeddings**, as shown in Figure 2.13. An embedding is a high-dimensional vector that captures the semantic meaning of the input, such that similar concepts are represented by vectors that are close in space. This transformation allows agents to reason about content based on meaning rather than exact wording.

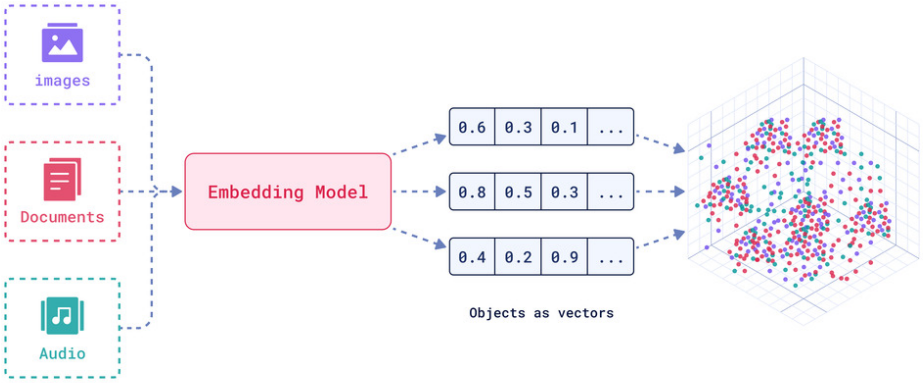

Figure 2.13: Illustration of how embedding works with embedding models.

Vector Space Representation

Embeddings are organized within a **vector space**, where each dimension encodes latent semantic features learned by the model. In this space:

- Each document, sentence, or token corresponds to a point (vector) in high-dimensional space.

- Semantic similarity between items is reflected in geometric closeness: vectors representing similar concepts have smaller distances between them.

- Vector spaces allow LLM-based agents to perform operations like clustering, nearest-neighbor search, and semantic reasoning.

The choice of vector dimensionality depends on the embedding model and the complexity of the semantic information it needs to capture. Typical LLM embeddings have hundreds or thousands of dimensions.

Similarity Search in Vector Databases

Similarity search is the process of retrieving vectors that are most semantically similar to a given query vector. In practice, this is performed using **distance metrics** such as:

- **Cosine similarity:** Measures the cosine of the angle between two vectors, capturing orientation similarity regardless of magnitude.

- **Euclidean distance:** Measures straight-line distance in the vector space, suitable when magnitude encodes meaningful information.

- **Manhattan distance:** Sum of absolute differences across dimensions, occasionally used for sparse embeddings.

Vector databases like `pgvector` allow efficient indexing and retrieval of nearest neighbors for a given query vector, often using optimized structures such as **approximate nearest neighbor (ANN) algorithms**. This enables agents to:

- Retrieve semantically relevant articles, even when keywords differ from the query.

- Aggregate information from multiple sources for summarization or reasoning.

- Provide context-aware answers in real-time applications.

By combining embeddings, vector space representation, and similarity search, LLM-based agents gain the ability to operate over content semantically rather than literally, greatly enhancing the accuracy and utility of information retrieval.

2.7.3 PostgreSQL + pgvector

As mentioned earlier, one popular approach combines the reliability of relational databases with vector search capabilities. `PostgreSQL`, a mature and widely-used relational database, can be extended with the `pgvector` extension to store embeddings as vectors. This setup offers several advantages:

- **Familiar interface:** Developers can continue using SQL while leveraging vector similarity search.
- **Hybrid queries:** Combine traditional filters (e.g., by date, category, or author) with semantic similarity search in a single query.
- **Open-source and flexible:** free, well-supported, and integrates easily with Python-based agent pipelines.
- **Efficient storage and retrieval:** High-dimensional embeddings are indexed to allow fast nearest-neighbor searches, critical for real-time applications.

By integrating vector databases into LLM-based agent architectures, systems can perform more accurate, context-aware retrieval, which enhances downstream tasks such as summarization, recommendation, and multimodal content generation.

Chapter 3

Methodology

This chapter presents the design and implementation of the automated **NewsScraper** system developed in this project. It provides a detailed overview of the system architecture, describing how modular agents interact with tools and external services to collect, process, and generate content. We discuss the design principles guiding the system, including modularity, multimodal output, personalization, and agent autonomy. The chapter then examines the agent architecture, data flow, and the specific roles of each component, including the MCP server, scraping service, and email distribution module. Finally, we describe the selected instruments and technologies, such as LLMs, text-to-speech models, vector databases, and the Streamlit interface, highlighting how they are integrated to enable robust, scalable, and user-centered functionality. This chapter lays the foundation for the system's prototype demonstration and use-case evaluation presented in the following chapter.

3.1 System Overview and Design Principles

The goal of this project is to develop an automated system capable of scraping online news, summarizing relevant content, and generating a podcast based on it. The resulting pipeline enables efficient, accessible, and personalized information consumption, particularly useful for users with time constraints or those who prefer auditory content.

The system is structured as a modular and agent-based architecture, which promotes reusability, scalability, and robustness. Agents operate as autonomous entities that interact with modular tools—such as scrapers, summarizers, and speech generators, thanks to a unified communication interface provided by the **Model Context Protocol (MCP)**. This abstraction allows developers to focus on high-level workflows, finding already done MCP servers ready for use.

From a design perspective, the system has the following principles:

- **Modularity:** Each component (scraping, summarization, TTS) operates as an independent tool, allowing for easy replacement or improvement without disrupting the overall pipeline.

- **Multimodal Output:** The final output is not just textual but vocal, leveraging advanced text-to-speech models to create natural-sounding podcast-like results.

- **Personalization:** The system adapts to user preferences by filtering and selecting news based on pre-defined topic categories (e.g., Economy, Politics, AI, Cybersecurity).

- **Agent Autonomy:** Agents operate in a goal-directed manner, orchestrating tool execution through the MCP without requiring hardcoded task flows.

- **Cloud-First and API-Centric:** The system integrates cloud-based LLMs and TTS models (via OpenAI APIs), ensuring state-of-the-art capabilities without the need for self-hosted models.

This chapter provides a detailed description of the system's architecture, data flow, and implementation decisions, laying the foundation for the prototype and use-case demonstration in the next chapter.

3.2 Agent Architecture and Data Flow

As illustrated in Figure 3.1, the system is designed around a modular agent-based architecture powered by the **Model Context Protocol (MCP)**. The MCP agent acts as the central coordinator, mediating between the user and the set of tools provided by the MCP server. This setup promotes **separation of concerns**, **scalability**, and **extensibility**, as new components can be integrated without major structural changes.

The architecture at a high level integrates several services: an MCP server that facilitates summarization, storage, and text-to-speech functionalities; a specialized scraping service responsible for acquiring news content; and an email service designated for newsletter distribution. These services collaboratively function to deliver personalized news summaries and podcasts.

Figure 3.1 illustrates the overall design of the system, showing how the different modules interact with each other and with external APIs.

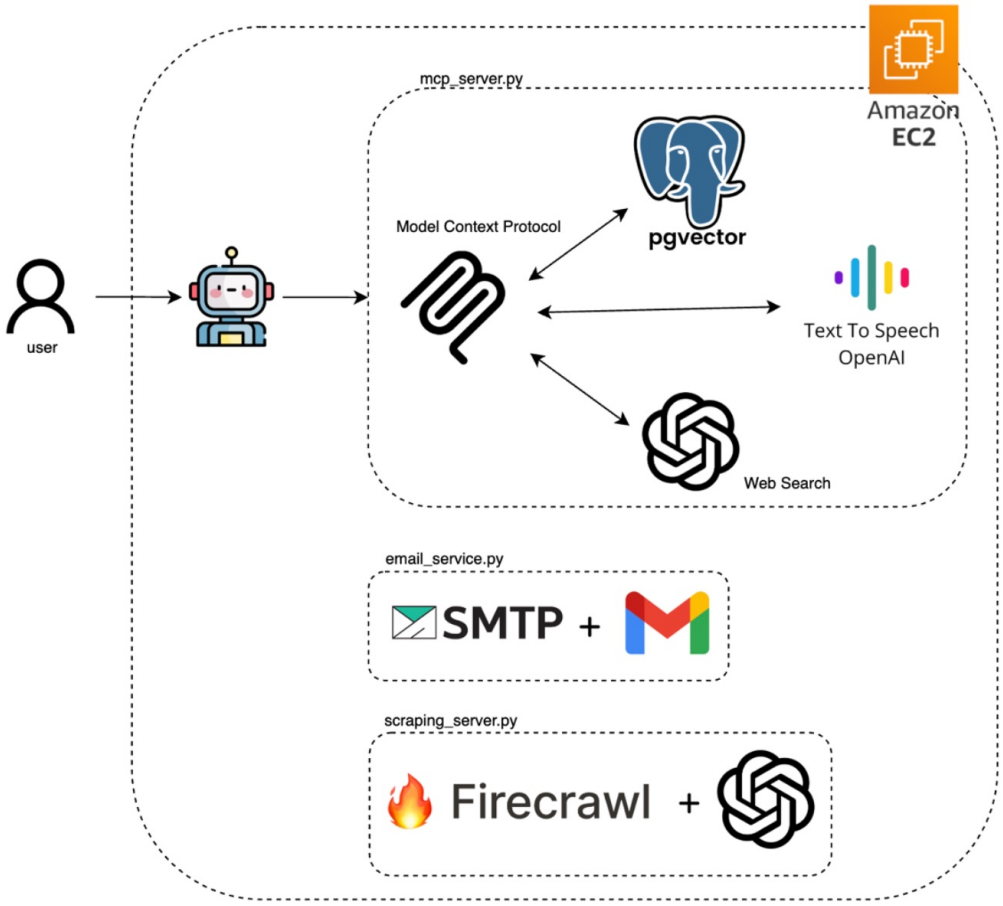

Figure 3.1: High-level architecture of the agent-based system.

3.2.1 Component Breakdown

The architecture relies on a set of modular Python scripts, each responsible for a specific functionality. Figure 3.1 already introduced the overall integration; here, we detail the role of each module.

- `mcp_server.py`: This script is the backbone of the system. It runs the **MCP server**, exposing the tools required for the agent, including:
  - **Summarization tool**, which communicates with the OpenAI API.
  - **Vector storage tool**, powered by **PostgreSQL + pgvector**, for storing articles, embeddings, and metadata.
  - **Text-to-Speech tool**, connected to the OpenAI Voice Engine for podcast generation.

  - **Web search tool**, for retrieving updated content if required.

- `scraping_server.py`: This script is responsible for **news extraction**. It uses **Firecrawl**, a web crawling framework, to retrieve clean article content from predefined trusted sources. The content is then optionally processed with the OpenAI API for cleaning and summarization before being stored in the database.

- `email_service.py`: This script manages the **weekly newsletter service**. It collects relevant articles and podcasts generated during the week, assembles them into a digest, and sends them to subscribed users. It integrates both:

  - The **SMTP protocol**, and
  - The **Gmail API**,

  to deliver messages reliably and securely.

Each component could be hosted on an **Amazon EC2 instance**, ensuring scalability and remote accessibility.

The technologies and frameworks selected for these components were chosen to balance performance, scalability, and ease of integration. **OpenAI's** models were selected for summarization and text generation due to their state-of-the-art capabilities, reliability, and strong support for multimodal content. The **OpenAI Voice Engine** provides high-quality text-to-speech, which is essential for producing engaging podcast output, even if it has some difficulties with the Italian language. **PostgreSQL + pgvector** was chosen to enable semantic similarity search and efficient storage of vectorized article embeddings, allowing fast and context-aware retrieval, with its ease and simplicity too. **Firecrawl** was selected as the web scraping framework for its robustness, flexibility, and ability to extract clean article content from multiple sources, having the possibiliy to extract content with ad-hoc schemas done for the project. For email distribution, the **SMTP protocol** and **Gmail API** were used to ensure secure, reliable, and scalable newsletter delivery. Finally, hosting components on **Amazon EC2 instances** provides the system with flexible cloud-based infrastructure, ensuring availability, remote access, and the ability to scale resources according to workload demands.

3.2.2 Workflow: From User Input to Final Output

Once the components are integrated, the data flows through the system in a sequential yet modular pipeline. The workflow can be summarized as follows:

1. **User Interaction:** The user selects categories of interest (e.g., *Economy*, *Politics*, *AI*). This input is sent to the `mcp_server.py`.

2. **News Retrieval:** The `scraping_server.py` is invoked, which runs Firecrawl to scrape new articles. Content is checked against the database to avoid duplicates.
3. **Summarization and Storage:** Articles are passed to the OpenAI summarization model. Both the raw and summarized content are stored in **pgvector**, enabling similarity search and reusability.
4. **Podcast Generation:** Summarized articles are converted into speech using the OpenAI Voice Engine. Audio files are stored and indexed for later access.
5. **Newsletter Assembly:** The `email_service.py` gathers weekly summaries and podcasts, builds the digest, and sends it to users via Gmail SMTP.
6. **Output Delivery:** Users can then access the news through two modalities:
   - Web platform, where users can browse articles, read summaries, listen to audio, and ask questions directly about the news content.
   - Email newsletters with both summaries and embedded audio links.

This modular workflow ensures robustness and scalability, while also facilitating easy extension of the system with additional tools (e.g., multilingual support or advanced analytics).

3.3 Selected Instruments

This section presents the technologies and frameworks that underpin the system's implementation. While Section 2.3 discussed these tools in a theoretical context and from a state-of-the-art perspective, here we describe their concrete use within the architecture. Each component was selected to balance **scalability**, **interoperability**, and **extensibility**.

3.3.1 Model Context Protocol (MCP)

The Model Context Protocol (MCP) offers a standardized architecture for presenting modular tools that an agent can dynamically invoke, as covered in Section 2.3. Here, we describe the specific implementation of MCP in our system, whereas the previous part concentrated on its philosophical underpinnings and use in agent systems.

The MCP server is implemented using the `FastMCP` library. Each tool encapsulates a specific functionality, and the MCP agent orchestrates their invocation depending on user preferences or queries.

The main tools exposed by the MCP server are:

- **Database Retrieval (`get_articles_from_db`)**: Allows the agent to query the **pgvector** database using **semantic similarity search**. This provides fast access to previously scraped and summarized articles without redundant computation, giving the news that is closest to the given query.
- **Preference Expansion (`expand_article`)**: Retrieves news from the system's database, filtered by the user's selected categories or subcategories. This tool not only returns structured article data (title, link, publication date, summary). After the article expantion, the **podcast generation module** is invoked to produce audio content, using the expanded articles, from the user preferences.
- **Web Search (`web_search`)**: When local data or stored articles are insufficient, the agent can query the web in real time. This is implemented using the `gpt-4o-mini-search-preview` model, which integrates search capabilities with natural language summarization. The tool takes a query as input, triggers a web search, and returns a concise summary of the most relevant information retrieved.

Finally, the server runs with a lightweight `SSE transport`, ensuring efficient communication between the MCP agent and the tools.

This implementation highlights how MCP supports modularity and extensibility: additional functionalities (e.g., translation, sentiment analysis, or advanced filtering) could be integrated by simply adding new tools, without requiring changes to the agent logic.

3.3.2 Scraping Process

The scraping pipeline is responsible for acquiring raw news data from online sources, transforming it into structured representations, and preparing it for subsequent enrichment and storage. This process is implemented using the `Firecrawl` API, which provides robust extraction capabilities while respecting the dynamic and heterogeneous structures of modern news websites.

The pipeline is organized into distinct stages:

- **Metadata Extraction.** Given a user-selected source (e.g., a news website corresponding to a category or subcategory), the system extracts the list of available articles. For each item, structured metadata is retrieved, including *title*, *description*, *link*, and *source*. This information is represented in a standardized schema, ensuring interoperability across heterogeneous publishers.
- **Article Content Retrieval.** Once candidate links are collected, the system scrapes the full text of each article, including the *main content*, *publication*

date, and, if available, the *author*. Non-relevant sections (advertisements, navigation menus, footers, etc.) are discarded. To increase robustness, the scraping functions implement a retry mechanism, mitigating the impact of temporary failures or network instability.

- **Duplicate Detection.** To avoid storing duplicates, a newly scraped article is compared with the database. The system applies a two-level filtering: (i) comparison of title, and (ii) comparison of semantic content, where a language model determines textual similarity for the candidate article and those already saved. This ensures that the database remains clean and non-redundant even when many sources are reporting the same story.

- **Expansion and Summarization.** Articles are expanded upon in more detail text data acquired from searching on the web. This allows the system to expand the original content by adding explanations of key terms, background context, and potential impacts (e.g., market consequences of an economic event). Summarization modules condense the elaborated text into short summaries, which are later reused for newsletter and podcast generation.

- **Embedding and Storage.** Finally, each processed article is embedded into a high-dimensional vector representation and stored in the PostgreSQL database with `pgvector`. The embeddings combine the article's title, content, category, and subcategory, allowing efficient semantic retrieval during user queries. We employ the `text-embedding-3-small` model from OpenAI, offering a short but highly comprehensive representation of text data. This choice ensures a balanced trade-off between computational efficiency and retrieval accuracy, making it suitable for real-time interaction with the news assistant.

This modular design allows the scraping process to remain extensible. New data sources can be integrated by simply defining their metadata schema, while additional enrichment steps (e.g., translation or sentiment analysis) can be incorporated without altering the core pipeline. By leveraging Firecrawl's structured extraction, semantic embeddings, and GPT-based contextualization, the system ensures both reliability and adaptability in processing heterogeneous news streams.

3.3.3 Database and Vector Storage with `pgvector`

A central component of the system is the database, which stores both raw and processed content. Each news article is saved together with its metadata (title, source, publication date, link, and summary), ensuring persistence and efficient access to past results while avoiding redundant scraping.

To enable semantic retrieval, the system integrates the `pgvector` extension for PostgreSQL, which allows storing high-dimensional embeddings and performing

similarity search directly within the database. As discussed in Section 2.7, embeddings map text into a vector space where semantically related items are positioned closer together. This makes it possible to retrieve conceptually relevant articles even when no exact keywords overlap with the query.

The adoption of `pgvector` was motivated by three main considerations:

- **Hybrid queries:** Combine semantic similarity search with classical SQL filters (e.g., by date, category, or source).
- **Scalability and efficiency:** Store and retrieve large volumes of embeddings with low latency, supported by optimized nearest-neighbor indexing.
- **Seamless integration:** Extend PostgreSQL's reliability and familiarity with vector search without introducing a separate database system.

With this setup, user interactions—whether through direct questions, semantic lookups, or preference-based searches—return results that are both accurate and context-aware. By bridging traditional relational queries with vector similarity search, the system achieves a flexible and robust database layer, aligning with the principles outlined in Section 2.7.

Database Schema and Columns

The database schema is centered around the `articles` table, which stores both metadata and content. The main columns are:

- `title, author, link, source`: Identifying information and provenance of the article.
- `publication_date`: Timestamp associated with the news item.
- `description, summary`: Concise versions of the content for quick access and filtering.
- `category, subcategory`: Used to match articles to user preferences.
- `text, expanded_article`: Raw article text and a processed and enriched version.
- `embedding`: High-dimensional vector representation of the article text, stored via `pgvector`.

This schema balances relational querying (on metadata) with semantic search (on embeddings).

Embedding Integration

As presented in Section 3.3.2, the embedding pipeline transforms each article into a semantic vector using the `text-embedding-3-small` model from OpenAI.

- The embedding input concatenates the title, article text, category, and subcategory.
- The computed vector is stored in the `embedding` column using `pgvector`.
- This enables direct similarity queries inside PostgreSQL by leveraging vector operators (e.g., cosine distance).

The use of a compact embedding model ensures computational efficiency while maintaining semantic richness. The design can be scaled to larger models (e.g., `text-embedding-3-large`) when higher precision is needed.

Saving Articles to the Database

The function `save_articles_to_db` handles the ingestion of new articles:

- Computes the semantic embedding and inserts all article fields into the database.
- Uses safe database transactions with error handling and rollback in case of failure.

This ensures articles are persistently stored along with their semantic vectors.

Retrieving All Articles

The function `get_all_db_articles` retrieves all entries from the `articles` table:

- Maps each row back into an `Article` object.
- Preserves both text and metadata for full downstream processing.
- Returns an empty list if no entries are available.

This provides a direct bridge between the database and the higher-level application logic.

Semantic Search with Similarity Queries

The function `search_similar_articles` performs semantic retrieval. It firstly takes the user query and embeds it into the same vector space as the articles, then executes a SQL query, ranking articles by cosine similarity with the query vector, and finally applies a threshold to retain only the most relevant articles.

As discussed before, to measure how close two embeddings are in the vector space, the system relies on **cosine similarity**. This metric computes the cosine of the angle between two vectors, providing a score in the range [-1, 1]. A value close to 1 indicates that the vectors point in nearly the same direction (high semantic similarity), while a value close to 0 suggests little or no relation. Unlike Euclidean distance, cosine similarity focuses on the orientation of the vectors rather than their magnitude, making it particularly suited for text embeddings where semantic meaning is captured by direction in the vector space.

Preference-Based Retrieval

The function `get_articles_preferences` filters articles by user-specified categories or subcategories:

- If all categories are selected, it retrieves the three most recent articles per category.
- Otherwise, it filters by the provided categories or subcategories.
- Only considers articles published in the last seven days, ensuring recency.

This mechanism supports personalized news delivery with minimal latency.

Updating Existing Articles

The function `update_articles` allows partial updates:

- Modifies only the `summary` and `expanded_article` fields of existing rows.
- Uses batch updates for efficiency.
- Ensures atomicity with transaction management.

This function is crucial for workflows where article summaries and article text are refined or enriched after the initial ingestion.

3.3.4 LLMs and Summarization Models

The system makes use of two main models from OpenAI's family:

- `gpt-4o-mini`: employed for text summarization, multi-article synthesis, and article expansion.
- `gpt-4o-mini-search-preview`: a specialized variant capable of performing **web-augmented queries**, used to retrieve contextual information beyond the scraped articles.

Together, these models allow the system to provide a differentiated approach which is adapted to the different use cases as follows. The choice of these specific variants was motivated by two main considerations: they ensure **minimum operational cost**, making the pipeline scalable and sustainable, while at the same time providing sufficient **precision and reliability** in tasks such as summarization, synthesis, and web-augmented retrieval. This balance makes them particularly well suited for applications where frequent queries and high responsiveness are required.

Single-Article Summarization. Each article that is scraped is processed with `gpt-4o-mini` to create a summary. The created outputs are clear, informative, and easy to understand, keeping the key facts from the original text. The summaries are short, usually three to four paragraphs long, and serve as the foundation for both text presentation and podcast narration.

Multi-Article Synthesis. When several related articles are retrieved, like multiple sources reporting on the same event, the system produces a joint summary. This avoids repetition, emphasizes shared points among sources, and ensures that numerical details like percentages or figures are maintained.

Web-Context Expansion. In cases where articles lack context, an external model is used to perform web queries.
Specifically, the `gpt-4o-mini-search-preview` model is used. Retrieved information is structured into thematic sections such as *Introduction*, *Context*, *Impact*, and *Synthesis*. This structured output is then integrated into the original text by another `gpt-4o-mini` call, creating an enriched version that includes background explanations and a clearer understanding of the event.

3.3.5 Text-to-Speech with OpenAI Voice Engine

A central feature of the system is the ability to transform textual news summaries into high-quality spoken output, enabling podcast-style delivery. This is achieved

through the integration of OpenAI's `gpt-4o-mini` and `gpt-4o-mini-tts`. First, the system generates a podcast-ready script by aggregating the selected and expanded articles. The summarization model is instructed to rewrite the content in a spoken-friendly style, ensuring smooth transitions between articles, concise yet informative coverage, and a total length adapted to the number of items (ranging from one to five minutes). The result is a coherent script that reads naturally, making the narration engaging and easy to follow.

Once the script is created, the text is converted into audio using OpenAI's `gpt-4o-mini-tts` model. The chosen configuration employs the voice *Sage*, which delivers a clear and professional narration in Italian. Specific instructions guide the voice to maintain an authoritative tone, with careful handling of percentages, dates, and numerical values, as well as intonation to improve comprehension. The output is exported as an `.mp3` file and stored with a structured naming. These audio files are then distributed through weekly email newsletters with embedded audio links or accessed directly on the website.

3.3.6 Frontend and Integration Infrastructure

The backend of the system plays a central role in connecting the user-facing interface with the underlying agentic logic and the MCP-based tool ecosystem. It ensures that user interactions, such as selecting preferences, subscribing to newsletters, or querying the system for specific financial information, are seamlessly translated into tool invocations and data processing workflows.

This component integrates three main layers:

1. The **Streamlit interface**, which provides an interactive and accessible web application where users can customize preferences, trigger scraping tasks, listen to generated podcasts, and subscribe to the newsletter.

2. The **scraping Agent**, which acts as the reasoning engine, deciding which tool to call based on user queries and conversation history, ensuring coherence and contextuality.

3. The **integration with MCP tools**, which manages the execution of operations such as database similarity search, article expansion, and web search, returning structured results to the user.

The user-facing side remains simple and intuitive, while the internal orchestration performs complex reasoning and content generation. Both the frontend and backend can be executed locally during development or deployed on an Amazon EC2 instance for scalability, remote accessibility, and production readiness.

Streamlit Frontend and User Interface

Streamlit is an open-source Python framework designed for the rapid development of interactive data applications and dashboards. It enables developers to build web-based interfaces directly in Python without requiring extensive knowledge of frontend technologies such as HTML, CSS, or JavaScript. Its declarative, high-level API is particularly suited for research prototypes and production-ready applications where fast iteration and clear visualization are priorities.

The adoption of Streamlit in this system is motivated by its ability to provide a seamless bridge between backend logic and user-facing components. Compared to traditional web frameworks, Streamlit reduces development complexity and accelerates deployment, making it a practical choice for agent-based applications where iterative design and frequent updates are expected.

The main advantages of Streamlit include:

- **Ease of use:** Applications can be built entirely in Python, without the need for frontend code.
- **Rapid prototyping:** Ideal for quickly iterating on user interfaces and validating system features.
- **Native integration with Python libraries:** Easily integrates with `pandas`, `matplotlib`, machine learning models, and APIs.
- **Interactive widgets:** Provides out-of-the-box components such as sliders, buttons, checkboxes, and forms for building dynamic interfaces.

On the other hand, Streamlit also presents some limitations:

- **Customization constraints:** Less flexibility for advanced UI/UX design compared to fully-fledged frontend frameworks.
- **Scalability:** While suitable for medium-scale applications, very high-traffic scenarios may require additional optimization or integration with external services.
- **State management:** Though improved in recent releases, handling complex application states can be less intuitive than in dedicated web frameworks.

Within this system, Streamlit acts as the main entry point for user interaction, allowing users to select their news preferences across categories and subcategories. The interface dynamically manages the state of user selections, validates that at least one preference is active, and generates the corresponding query prompt. A dedicated sidebar also enables users to subscribe to the weekly newsletter by

providing their personal information, such as name, surname, email, and chosen preferences, which are then stored in the database.

Streamlit additionally integrates the podcast player, retrieving available audio files from the backend via a **REST API** and displaying them in descending chronological order. This design ensures that users can either listen to the latest generated podcast directly from the interface. The system maintains a conversational interface through a chat window, where scraped results and summaries are returned, and where users can further interact by asking additional questions. Overall, this module represents the bridge between the user and the MCP agent, enabling an accessible and customizable experience.

Scraping Agent and Tool Orchestration

At the core of the backend logic lies the **scraping agent**, which manages the interaction between user queries and the available MCP tools. The agent is implemented using `langgraph` and the `MultiServerMCPClient` adapter, enabling it to connect to the MCP server and dynamically determine which tool to invoke based on the query content and chat history. A set of rules guides this decision-making process: if the query relates to specific financial events or entities, the agent uses the `get_articles_from_db` tool to retrieve the related articles from the internal database; if it corresponds to broader categories or subcategories selected by the user, the `expand_article` tool is invoked to expand the articles in the database with the selected category or subcategory; and if the query is of a general or definitional nature, the `web_search` tool is selected to search about the query on the web.

As discussed in Section 2.2.3, both *LangChain* and *LangGraph* are widely adopted frameworks for building LLM-based agents. While LangChain offers a broad ecosystem of integrations and rapid prototyping capabilities, we opted for **LangGraph** due to its emphasis on reproducibility, and graph-based control of agent workflows. These properties are particularly advantageous in our setting, where tool orchestration must follow explicit rules and ensure consistent behavior across user sessions.

The scraping agent also integrates contextual reasoning by analyzing the history of the conversation, ensuring continuity when handling short or ambiguous user inputs. When the `expand_article` tool is invoked, the agent returns a structured output containing titles, publication dates, links, and summaries of each retrieved article, maintaining full coherence with the database content. This orchestration mechanism ensures that the system provides the most relevant and contextually appropriate response while avoiding redundant tool calls. In this way, the scraping agent constitutes the reasoning layer of the backend, coordinating user interactions, database access, and external information retrieval in a unified workflow.

Chapter 4

Prototype and Use Case Demonstration

To illustrate the functionality of the prototype, we describe a representative use case where a new user interacts with the system from the first access to the reception of the weekly newsletter. This narrative highlights how the different components of the architecture integrate seamlessly to provide a coherent user experience.

4.1 Website Homepage

When the user first accesses the website, they are welcomed by the interface of the news assistant. The homepage serves as the central hub of the application, presenting a clear and minimal design to guide user interaction. From this entry point, users can explore the available **categories** and **subcategories**, which represent the thematic organization of the scraped content. Each category corresponds to a macro-topic like *Mercati*, *Economia*, or *Tecnologia*, while subcategories allow for more personalization.

The homepage also provides a direct entry to the **newsletter subscription panel**, where users can personalize the system to their own interests. In addition, a section is dedicated to the archive of **generated podcasts**, allowing users to browse previously created audio summaries. This combination ensures that the homepage functions as both a dashboard and a starting point for deeper interaction with the assistant.

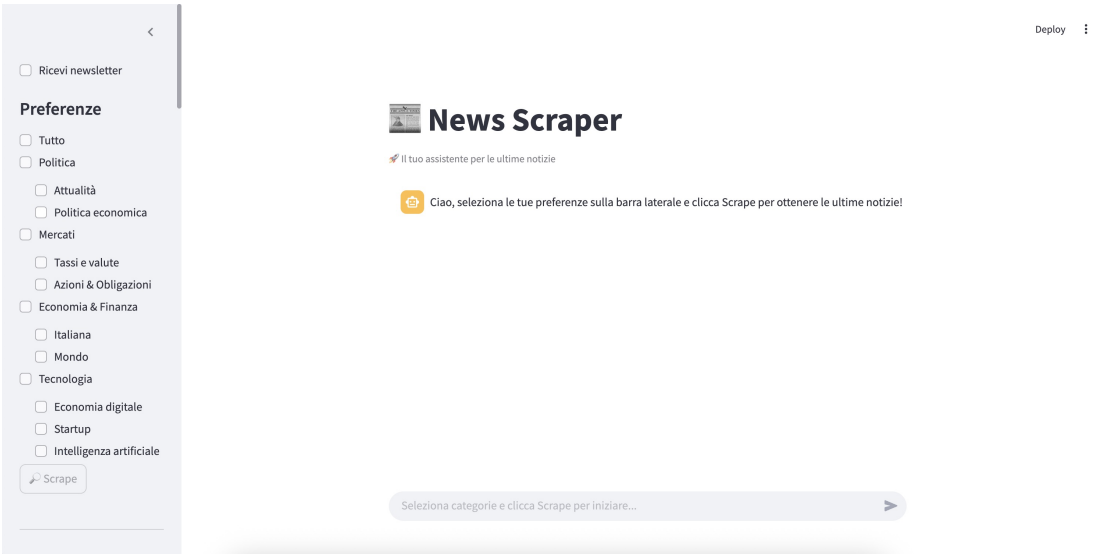

Figure 4.1: Initial interface of the website upon first access.

4.2 Newsletter Subscription & Delivery

The user can subscribe to the newsletter directly from the left sidebar of the interface. The subscription process is simple: by entering their **name**, **surname**, **email**, and selecting their **preferences**, users enable the system to build a personalized information flow tailored to their interests. These preferences are stored in the database and used to filter relevant content during the scraping and summarization processes.

Once subscribed, users receive weekly emails automatically generated by the backend. These newsletters are not simple collections of articles, but **curated digests**, enriched with AI-generated summaries and contextual insights. Furthermore, each newsletter includes the link to the latest generated **podcast**, ensuring multimodal access to information: users can either read the summaries or listen to them on the go. This design choice addresses different consumption habits, reinforcing engagement.

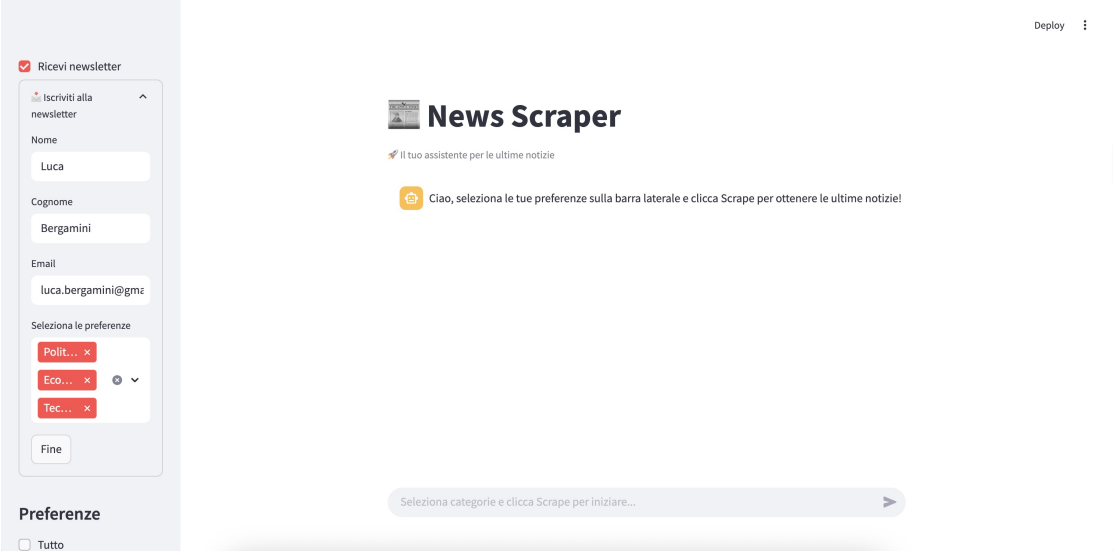

Figure 4.2: Subscription panel for the weekly newsletter.

Finally, at the end of the week, the system sends the newsletter via email. This message contains a summary of all relevant articles published during the week according to user preferences, together with the podcast that covers the same topics in spoken format. This provides users with a seamless **cross-channel experience**, allowing them to stay informed in the way that best suits their context.

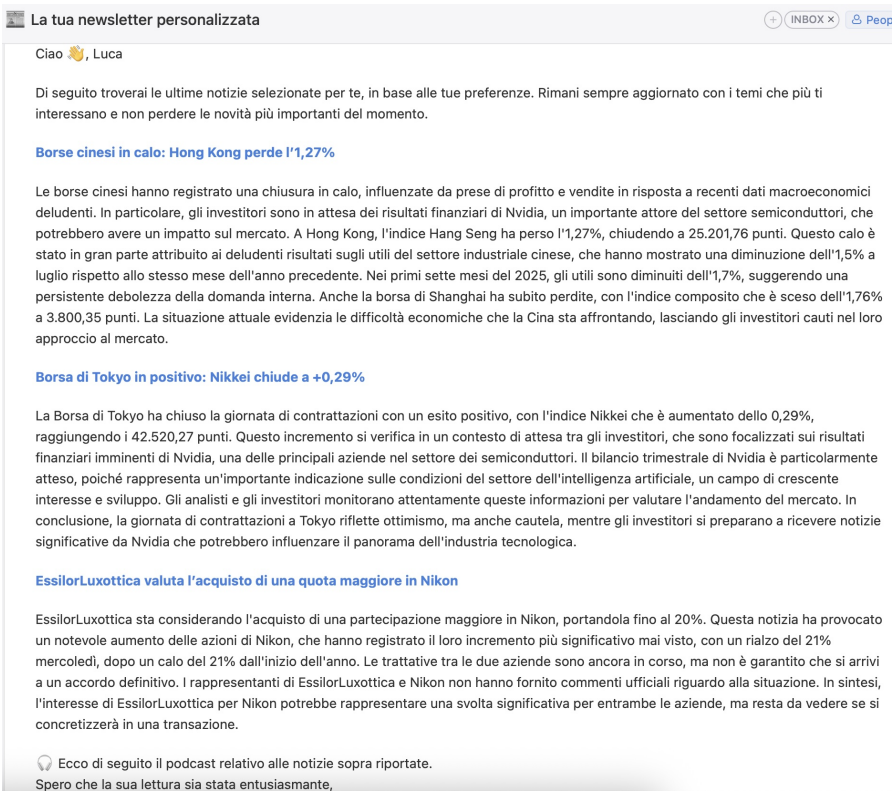

Figure 4.3: Weekly newsletter received by the user, including summaries and a podcast.

4.3 News Scraping & Podcast Generation

On the left sidebar, the user specifies their interests. For example, in Figure 4.4, the category *Mercati* is selected. This action triggers the **scraping pipeline**, which is responsible for collecting fresh articles from predefined news sources. Once retrieved, the content undergoes **summarization** and, if necessary, **expansion** via the MCP tools (see Section 3.3.1). The results are then displayed in the chat interface, structured with title, publication date, source, and a concise summary.

At the same time, the system launches the **podcast generation pipeline**, which transforms the textual summaries into audio narration. The generated podcast is automatically stored and made available in the sidebar, organized chronologically. This multimodal approach ensures that users are not limited to textual summaries but can consume content in auditory form, increasing accessibility and engagement.

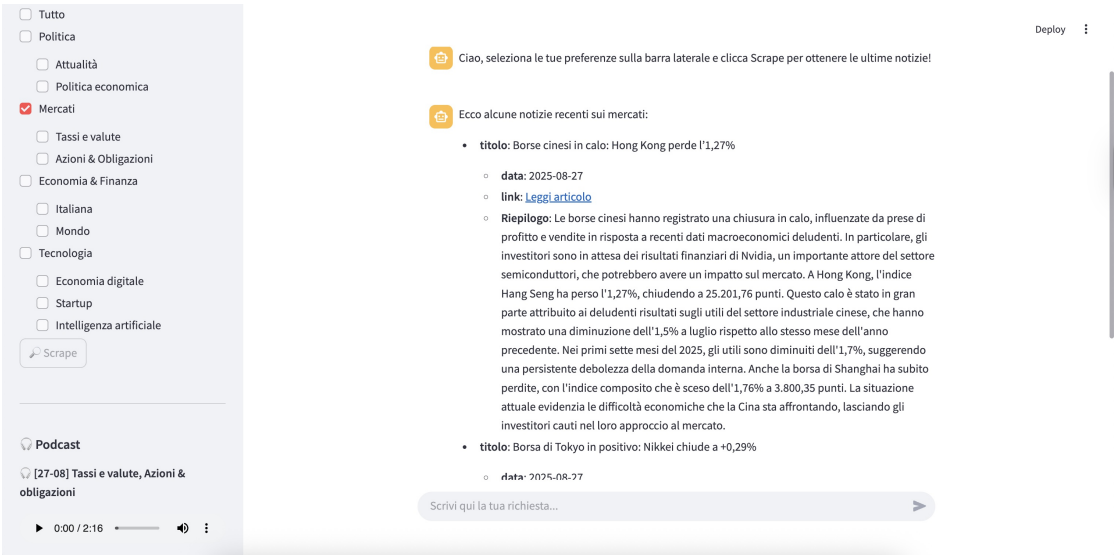

Figure 4.4: User selects preferences (*Mercati*) and the system retrieves matching articles.

4.4 Interactive Chat

In addition to static subscription and category-based retrieval, the user can also interact dynamically with the assistant through the **chat interface**. This conversational component enables open-ended queries, where the agent decides the best strategy to respond. For example, if the query refers to specific financial entities or events, the agent performs a similarity search in the internal database. Conversely, if the query is general or definitional, the system relies on the `web_search` tool to retrieve external information.

The chat interface thus acts as a flexible exploration tool, where results are returned in natural language and grounded on structured evidence. Importantly, the agent maintains **conversation history**, allowing it to disambiguate short or implicit user inputs (e.g., “sì”, “dimmi di più”). This feature ensures continuity and provides a more human-like interaction experience.

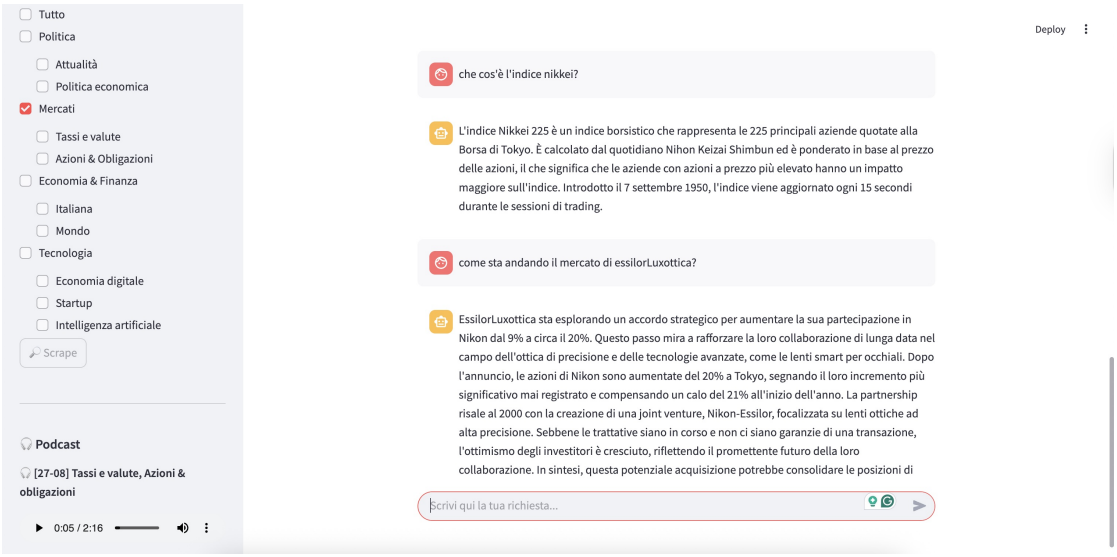

Figure 4.5: The chatbot answers user queries by using database similarity search or web search.

4.5 Usage and Cost Analysis

In this section, we analyze the usage and costs associated with the **OpenAI API** during the daily execution of the project. Two main experiments were tracked: the daily cost of scraping new articles, generating website output, and producing one podcast episode; and the costs over five-days. Figures 4.6 and 4.7 present the results.

4.5.1 Single-Day Usage

Figure 4.6 shows the distribution of costs across different models for a typical day. The main cost drivers are the high-usage calls to `web-search tool gpt-4o-high` and the audio output of `gpt-4o-mini-ts`, which dominate the expenses. Minor contributions come from embedding generation (`text-embedding-3-small`) and cached inputs, which help reduce overall cost. This reflects the computationally expensive nature of generating structured long-form outputs such as podcasts, while lighter tasks like embeddings and small queries contribute only marginally.

4.5.2 Five-Day Usage

Figure 4.7 aggregates costs over a five-day period. The stacked bars differentiate between input, output, cached input, and high-complexity calls. The data highlight

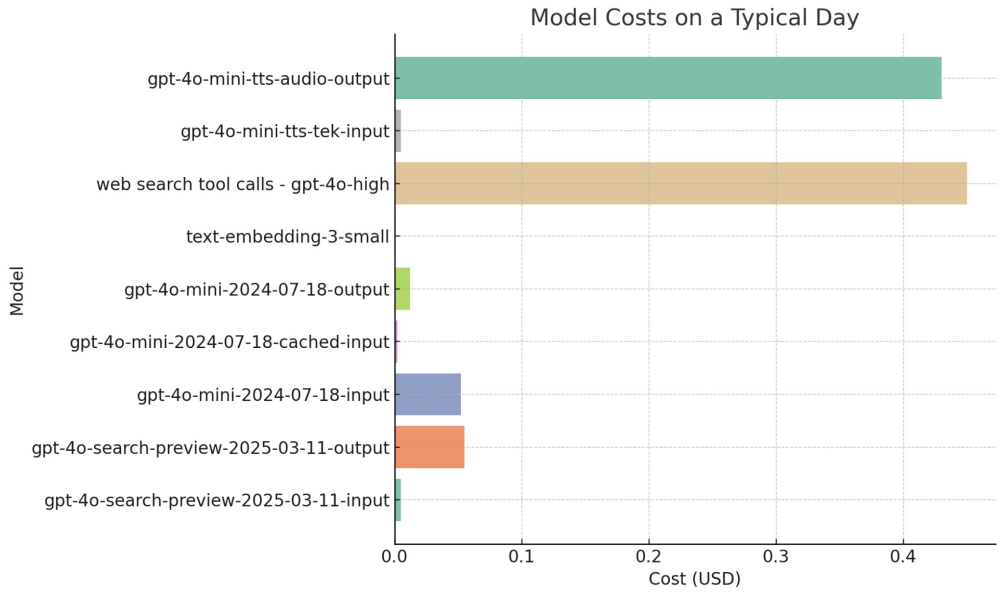

Figure 4.6: Model costs on a typical day.

two key observations:

1. Output tokens (pink) and high-complexity calls (orange) represent the majority of expenses across all days.

2. Cached input (light green) and input costs (purple) are relatively smaller, but consistently present.

The variation across days reflects the different workloads: some days are dominated by content generation (higher output costs, like on the day 5/9/2025 where only scraping phase was done), while others are lighter, relying mainly on cached results or simpler queries, with a podcast generation (like on the day 5/8/2025). This pattern illustrates the efficiency gains from reusing cached inputs while also showing the unavoidable high cost of large-scale text and audio generation.

4.5.3 Cost Estimations for Multiple Users

Based on the results shown in Figures 4.6 and 4.7, the total daily cost for running the system—including scraping, summarization, embeddings, podcast generation, and website updates—is approximately **$0.85** for a single user platform. However, the $\$\frac{0.70}{\sqrt{n}}$ daily scraping cost is a *shared baseline*, regardless of the number of users (where $\sqrt{n}$ represents how equal preferences are grouped). This cost should be divided among all users when estimating the per-user cost.

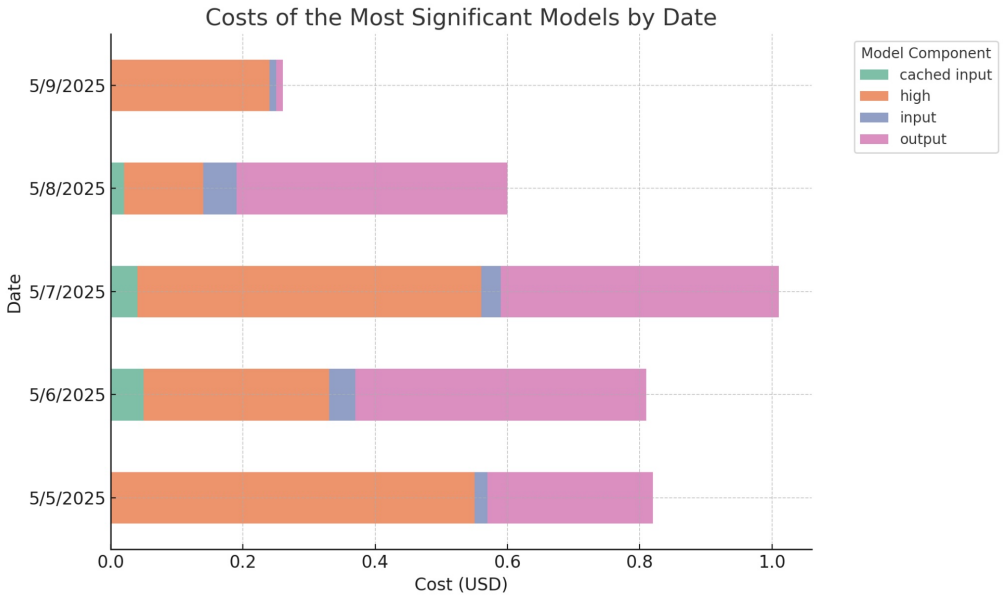

Figure 4.7: Costs over five days, broken down by input, output, cached input, and high-cost calls.

Let us denote:

- The per-user variable cost (email delivery, customization, etc.) as \$0.15.
- The shared daily scraping, and content generation cost as $\$\frac{0.70}{\sqrt{n}}$.
- The number of users as n.

Then, the daily cost per user can be expressed as:

$$\text{Cost per user} = 0.15 + \frac{0.70}{\sqrt{n}}.$$

Using this formula, we can estimate costs under different user scenarios:

- For a single user ($n = 1$), the daily cost is $0.15+0.70/1 =$ **\$0.85**, corresponding to about **\$25.5 per month**.
- For 100 users ($n = 100$), the daily cost per user is $0.15 + 0.70/10 \approx$ **\$0.22**, corresponding to approximately **\$6.6 per month**.
- For 1000 users ($n = 1000$), the daily cost per user is $0.15+0.70/100 \approx$ **\$0.157**, corresponding to about **\$4.71 per month**.

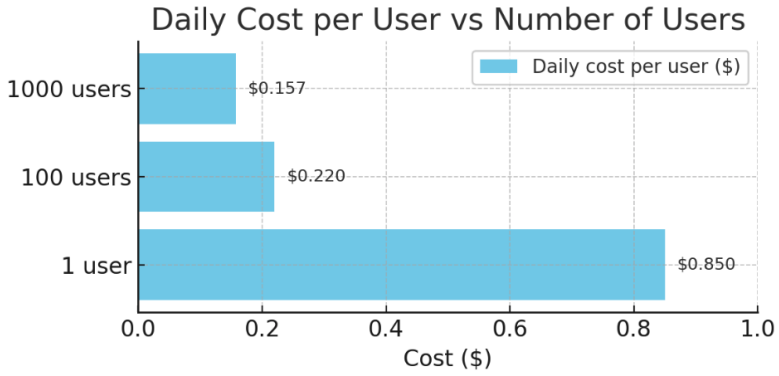

Figure 4.8: Costs per user for different number of users.

Those results are shown in Figure 4.8.

The total daily cost for all users can be calculated by multiplying the per-user cost by n:

- 100 users: $100 \times 0.22 \approx$ **\$22** per day, or approximately **\$660 per month**.
- 1000 users: $1000 \times 0.157 \approx$ **\$157** per day, or about **\$4,710 per month**.

Additional usage, such as multiple chatbot queries, article retrievals, summarization, or podcast generation, increases the per-user variable cost. For example:

- **Typical usage:** One access per day with one or two questions results in a daily cost per user of approximately **\$0.157–\$0.25**, corresponding to **\$4.71–\$7.5 per month** depending on the number of users sharing the scraping and generation cost.
- **High usage:** If a user accesses the system three times per day and asks three questions each time, the daily cost per user can increase to approximately **\$0.30–\$0.50**, corresponding to **\$9–\$15 per month**.

This approach clarifies that the baseline scraping cost is dependent of the user activity, while other operational costs scale linearly with user activity. Properly distinguishing shared and per-user costs is essential for accurate budgeting and operational planning.

Summary. Reviewing the results, the usage and cost analysis highlights that the main contributors to daily expenses are high-complexity operations such as news summarization, embeddings computation, and podcast generation, while cached inputs and shared scraping costs account for minor, but consistent, cost reductions. The mail delivery and customization cost of approximately **\$0.15** is personal for

each user, whereas other operations, like scraping and content generation, scale with individuals.

Over a five-day period, total costs fluctuate according to workload: days dominated by content generation and multiple queries incur higher expenses, while lighter days benefit from cached results and fewer active requests. The average daily cost per user is approximately **$0.157–$0.25**, corresponding to about **$4.71–$7.5 per month** for typical usage, assuming one access per day with one or two bot queries, with linear scaling for multiple users. In high-activity scenarios, such as three accesses per day with three queries each, daily expenses per user can rise to roughly **$0.30–$0.50**, corresponding to about **$9–$15 per month**.

This analysis underscores the importance of distinguishing between the shared baseline costs and per-user variable costs when planning operational budgets, as both the number of users and their usage patterns significantly influence overall system expenses.

Chapter 5

Conclusion and Future Work

5.1 Conclusion

This thesis presented the design and development of an agent-based system for news (financial in this case) aggregation, summarization, and personalized delivery. Leveraging the Model Context Protocol (MCP) as a backbone, the system integrated multiple tools for scraping, summarization with LLMs, storage with `pgvector` database, and multimodal output through text-to-speech and email newsletters.

The prototype demonstrated how a user can access the platform, select preferences, receive personalized summaries in both textual and audio form, and interact with the agent through natural language queries. This pipeline shows the feasibility of combining LLM-based summarization with personalized delivery channels, bridging the gap between unstructured news streams and user-centered consumption.

The main contribution of this work is the demonstration of how the Model Context Protocol (MCP) can act as a backbone for building an AI-driven news extraction and conversational system. The project shows how a custom MCP server, equipped with purpose-built tools for scraping, summarization, and database querying, can be orchestrated to meet the concrete goal of delivering personalized news experiences. At the same time, the design highlights the broader potential of MCP: since the protocol allows a client to connect to multiple independent servers, the architecture can easily integrate external tools developed by other systems, without additional engineering effort. This dual capability—combining tailored in-house tools with reusable third-party MCP servers—illustrates the strength of MCP as a modular and extensible framework.

The project also demonstrates how this approach supports adaptability, interoperability, and the creation of agents that can evolve with new functionalities over time.

5.2 Future Work

While the system is functional as a prototype, several areas remain open for further research and development:

- **Scalability and Deployment.** The current prototype is designed for a controlled environment. Future work should investigate deployment at scale, ensuring robustness under higher user loads and broader data sources, potentially through containerization and cloud-native architectures.
- **Multilingual and Cross-Domain Support.** Extending the platform beyond Italian would enable comparative analysis across international financial markets. Similarly, adapting the architecture to new domains (like healthcare, cybersecurity, ..) would demonstrate its generality.
- **Advanced Personalization.** User experience could be improved by incorporating adaptive recommendation mechanisms, where preferences are refined over time based on feedback, reading patterns, or engagement with newsletters and podcasts.
- **Real-Time Market Alerts.** An extension of the system could focus on immediacy, providing users with push notifications or urgent newsletters when significant financial events occur (e.g., market crashes, central bank announcements). This would shift the system from being purely aggregative to proactive, transforming it into an intelligent assistant for decision-making.
- **Security and Privacy.** Handling sensitive user information such as email addresses and preferences necessitates stricter privacy-preserving mechanisms and compliance with data protection regulations (e.g., GDPR).
- **Integration of Existing MCP Servers.** A natural extension of this work lies in exploiting the interoperability offered by the Model Context Protocol. By connecting to already available MCP servers developed by other systems, the prototype could quickly expand its functionalities without requiring additional ad-hoc implementations. For example, integrating calendar-based MCP servers would allow the assistant to automatically remind users of key events (such as central bank meetings, earnings reports, or regulatory deadlines). Similarly, messaging-related MCP servers (e.g., for Telegram) could enable delivery of summaries and podcasts directly within chat platforms. This approach would turn the news assistant into part of a broader ecosystem of interoperable agents, offering richer services and more personalized use cases.

Bibliography

[1] Sepp Hochreiter and Jürgen Schmidhuber. «Long short-term memory». In: *Neural computation* 9.8 (1997), pp. 1735–1780. URL: 10.1162/neco.1997.9.8.1735 (cit. on p. 5).

[2] Kyunghyun Cho, Bart Van Merriënboer, Caglar Gulcehre, Dzmitry Bahdanau, Fethi Bougares, Holger Schwenk, and Yoshua Bengio. «Learning phrase representations using RNN encoder-decoder for statistical machine translation». In: *arXiv preprint arXiv:1406.1078* (2014). arXiv: 1406.1078 [cs.CL]. URL: https://arxiv.org/abs/1406.1078 (cit. on p. 5).

[3] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Lukasz Kaiser, and Illia Polosukhin. «Attention is all you need». In: *Advances in Neural Information Processing Systems*. Vol. 30. 2017, pp. 5998–6008. URL: https://arxiv.org/abs/1706.03762 (cit. on pp. 6–8).

[4] Meng Xu et al. «LLM Agents: A Survey». In: *arXiv preprint arXiv:2309.07864* (2023). URL: https://arxiv.org/abs/2309.07864 (cit. on p. 12).

[5] Minghao Xu, Yuzhuo Huang, Xiaojian Zheng, and Minlie Huang. «Exploring Large Language Model-Based Intelligent Agents: Definitions, Methods, and Prospects». In: *arXiv preprint arXiv:2401.03428* (2024). URL: https://arxiv.org/abs/2401.03428 (cit. on pp. 13, 14, 16, 17, 21).

[6] Patrick Lewis et al. «Retrieval-augmented generation for knowledge-intensive NLP tasks». In: *Advances in neural information processing systems* 33 (2020), pp. 9459–9474. URL: https://proceedings.neurips.cc/paper/2020/file/6b493230205f780e1bc26945df7481e5-Paper.pdf (cit. on pp. 17, 18).

[7] Jean-Baptiste Alayrac, Jeff Donahue, Paul Luc, Antoine Miech, Serkan Cabi, Yee Whye Teh, et al. «Flamingo: a Visual Language Model for Few-Shot Learning». In: *arXiv preprint arXiv:2204.14198* (2022). URL: https://arxiv.org/abs/2204.14198 (cit. on p. 18).

[8] Shinn Yao, Juncheng Zhao, Dian Yu, Nan Wang, Zhou Yu, et al. «ReAct: Synergizing reasoning and acting in language models». In: *Advances in Neural Information Processing Systems* (2023). URL: https://arxiv.org/abs/2210.03629 (cit. on pp. 18, 27, 29).

[9] Noah Shinn, Lizi Liu, Will Tam, and Edward Lee. «Reflexion: Language agents with verbal reinforcement learning». In: *arXiv preprint arXiv:2303.11366* (2023). URL: https://arxiv.org/abs/2303.11366 (cit. on pp. 18, 29).

[10] LangChain contributors. *LangChain: Building applications with large language models*. https://github.com/langchain-ai/langchain. Accessed on 2025-09-03. 2023 (cit. on p. 21).

[11] Haixuan Hou et al. «Model Context Protocol (MCP): Landscape, Security Threats, and Future Research Directions». In: *arXiv preprint arXiv:2503.23278* (2025). URL: https://arxiv.org/abs/2503.23278 (cit. on p. 25).

[12] Yang Liu and Mirella Lapata. «Text Summarization with Pretrained Encoders». In: *arXiv preprint arXiv:1908.08345* (2019). URL: https://arxiv.org/abs/1908.08345 (cit. on p. 28).

[13] Mike Lewis, Yinhan Liu, Naman Goyal, Marjan Ghazvininejad, Abdelrahman Mohamed, Omer Levy, Veselin Stoyanov, and Luke Zettlemoyer. «BART: Denoising Sequence-to-Sequence Pre-training for Natural Language Generation, Translation, and Comprehension». In: *arXiv preprint arXiv:1910.13461* (2020). URL: https://arxiv.org/abs/1910.13461 (cit. on p. 28).

[14] Jingqing Zhang, Yao Zhao, Mohammad Saleh, and Peter J Liu. «PEGASUS: Pre-training with Extracted Gap-sentences for Abstractive Summarization». In: *arXiv preprint arXiv:1912.08777* (2020). URL: https://arxiv.org/abs/1912.08777 (cit. on p. 29).

[15] Colin Raffel, Noam Shazeer, Adam Roberts, Katherine Lee, Sharan Narang, Michael Matena, Yanqi Zhou, Wei Li, and Peter J Liu. «Exploring the Limits of Transfer Learning with a Unified Text-to-Text Transformer». In: *Journal of Machine Learning Research* 21.140 (2020), pp. 1–67. URL: https://arxiv.org/abs/1910.10683 (cit. on p. 29).

[16] Zi-Yi Dou, Demian Gholipour Ghalandari, Mina Neysiani, Pengfei Liu, Graham Neubig, and Alexander M Rush. «GSum: A General Framework for Guided Abstractive Summarization». In: *arXiv preprint arXiv:2104.02112* (2021). URL: https://arxiv.org/abs/2104.02112 (cit. on p. 29).

[17] Iz Beltagy, Matthew E Peters, and Arman Cohan. «Longformer: The Long-Document Transformer». In: *arXiv preprint arXiv:2004.05150* (2020). URL: https://arxiv.org/abs/2004.05150 (cit. on p. 29).

[18] Manzil Zaheer et al. «Big Bird: Transformers for Longer Sequences». In: *arXiv preprint arXiv:2007.14062* (2020). URL: https://arxiv.org/abs/2007.14062 (cit. on p. 29).

[19] Rohan Anil, Yen-Chun Chen, Aakanksha Chowdhery, et al. «Gemini: A Family of Highly Capable Multimodal Models». In: *arXiv preprint arXiv:2312.11805* (2023). URL: https://arxiv.org/abs/2312.11805 (cit. on pp. 29, 30).

[20] OpenAI. «GPT-4o: OpenAI's Omni Model». In: *OpenAI Blog* (2024). URL: https://openai.com/index/gpt-4o (cit. on pp. 30, 31).

[21] Rohit Girdhar, A Girish, A Payne, Carl Vondrick, Andrew Zisserman, and Joao Carreira. «ImageBind: One Embedding Space to Bind Them All». In: *arXiv preprint arXiv:2305.05665* (2023). URL: https://arxiv.org/abs/2305.05665 (cit. on p. 30).

[22] OpenAI. «Voice Engine: High-Quality Text-to-Speech from OpenAI». In: *OpenAI Blog* (2024). URL: https://openai.com/blog/voice-engine (cit. on pp. 30, 31).