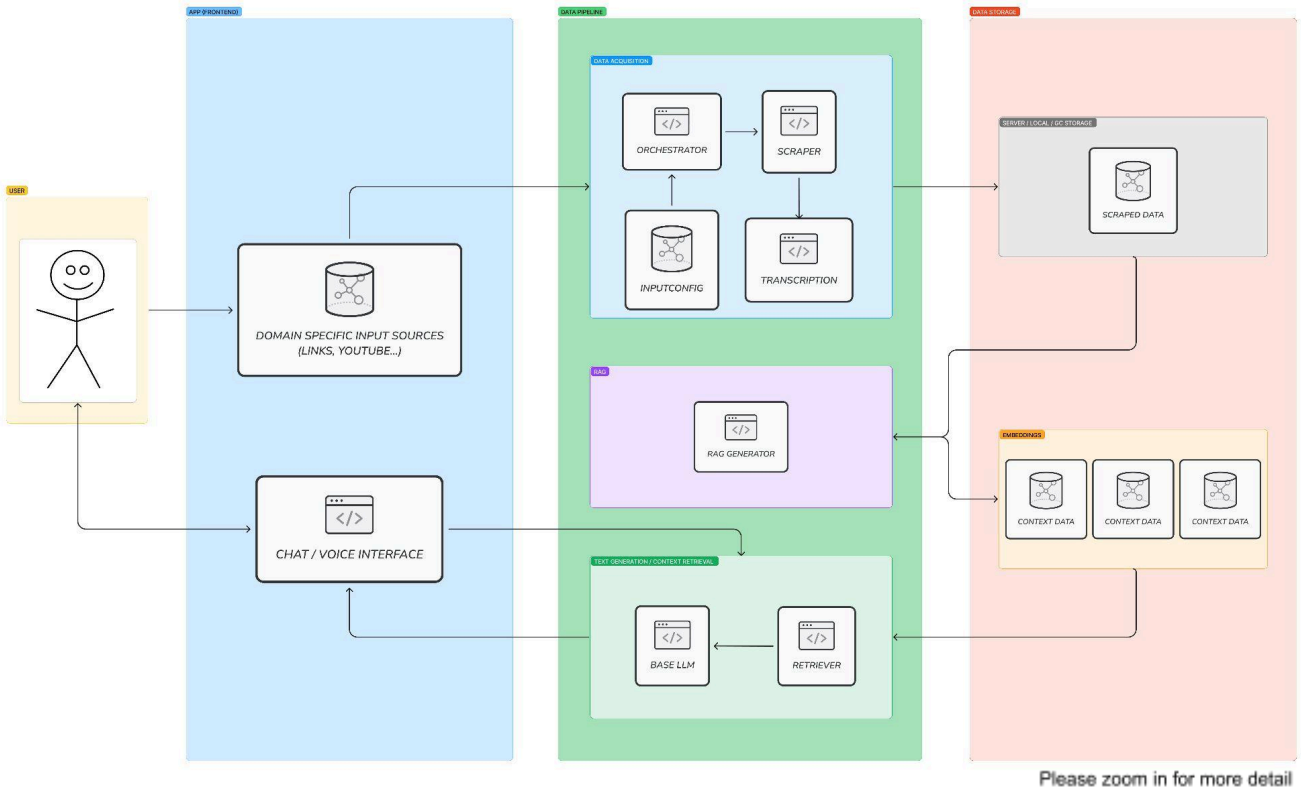
Ailixir - customised AI agent generation framework

Introduction

The Ailixir project proposes a framework that leverages Large Language Models for developing customised AI agents specialised in different contexts (medical, business, biology, etc.). The agents can dynamically adapt to user input, progressively refine interactions over time and are capable of conversing via text and voice with a user.
The user is an entrepreneur-developer who wants to rapidly create agent prototypes which are based on handpicked content sources and evaluate the quality of the results they produce. The custom agent that is produced as the end product for each knowledge domain helps the entrepreneur-developer to decide whether the specific knowledge domain is worth investing more effort into building a better agent.
By harnessing diverse training data from sources such as academic literature repositories, platforms like YouTube, podcast episodes and selected website data, the agents gain nuanced understanding of specific contexts and can be adjusted to accommodate user preferences. The framework prioritises transparency by providing links to the sources used to generate the responses to afford the user the ability to further verify their accuracy and reliability.

Code components (system architecture)



Frontend

The user provides a set of input sources in the form of a file and probably a set of user-specific parameters to configure the data pipeline. When the agent has been generated, the user converses with the agent through a chat or voice interface.

Backend

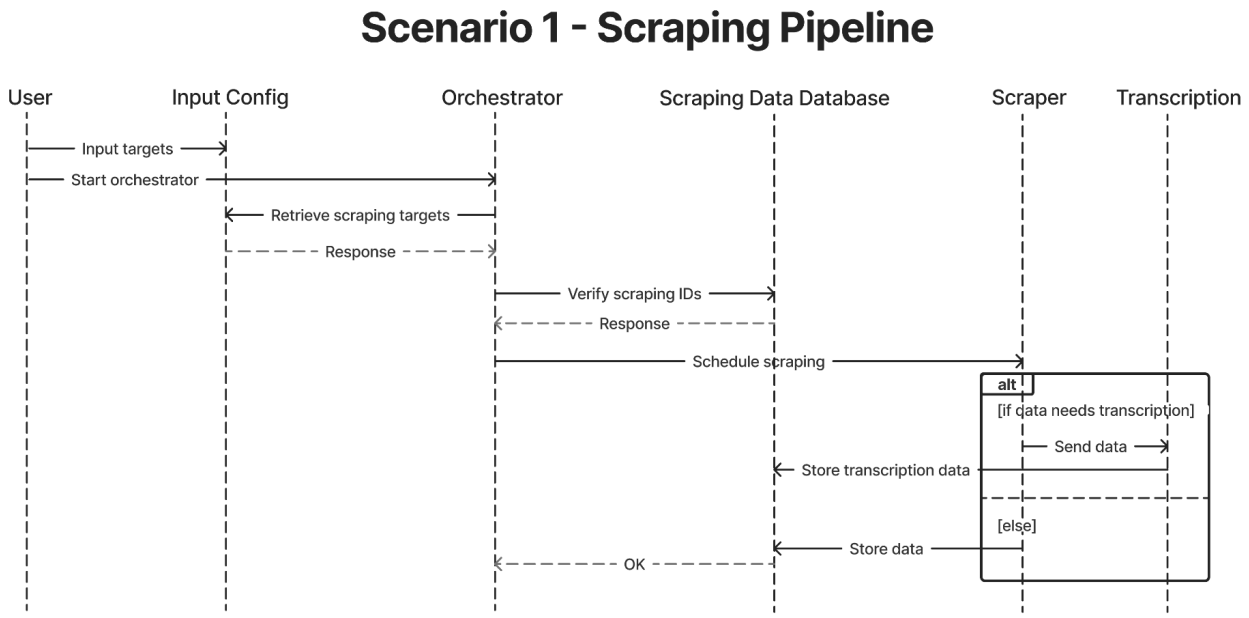
In the backend three main processes take place:

- The data acquisition pipeline which comprises the configuration functionality, the orchestrator functionality, scraping functionalities and transcription functionalities with the goal of obtaining useful topic related text and transcriptions of context specific sources. The scraped data is passed on and stored in a database in the “data storage part” of the architecture.
- The generator of the RAG framework takes information stored in the texts of the DB and creates embeddings based on them and then passes and stores them in the vector store.
- The text generation / Context retrieval where the user prompt is passed on to the RAG framework, where the context is determined and the retriever enriches the LLM’s answer with context by providing the LLM with domain specific knowledge from the embeddings database/vector store.

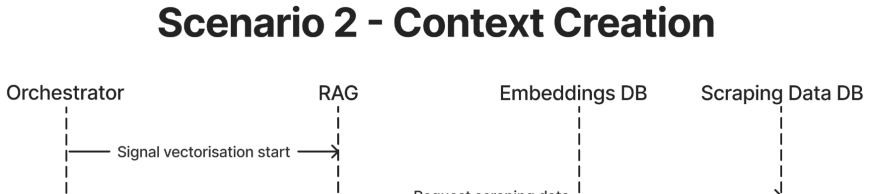
Data Storage

The data storage part of the architecture comprises a data bank or GC storage for storing text (in jsons) and a vector store storing the embeddings of the context data, which can later be retrieved and used by the RAG retriever to enrich the context.

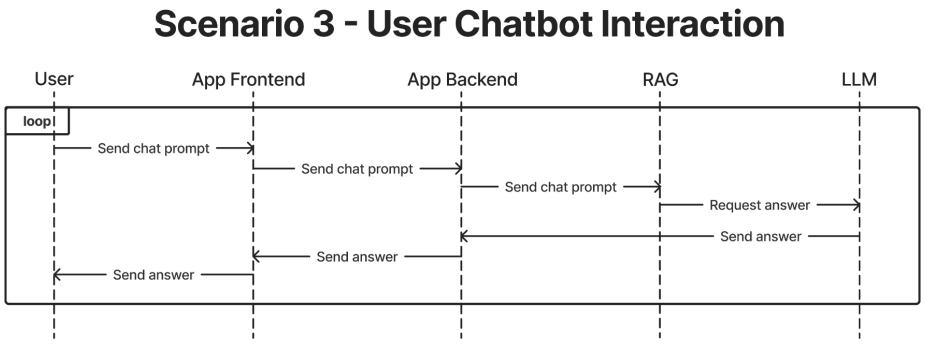
Runtime components



Scenario 1: The user specifies targets to scrape data from into a config file (e.g. youtube video urls). The orchestrator retrieves all the scraping targets, verifies that a scraping target has not already been scraped, and schedules scraping jobs. If needed, scraped data is transcribed to the correct format (e.g. convert audio files to text).



Scenario 2: Context data for the LLM is generated from the scraping data by the RAG.



Scenario 3: A user interacts with the chat interface. The user prompt is propagated to the RAG, where the domain context data is retrieved. The LLM generates an answer which is sent back to the user.

Components analysis and related tech stack

App Frontend (Chat-based / voice interface)

The frontend application will be developed using React Native. The application will be written in TypeScript. For authentication and user data management Firebase Auth, provided by Google Firebase, will be used.

App Backend

The backend will communicate with the frontend via REST APIs. Yarn will be used as the package manager to handle dependencies.

Data Pipeline

The pipeline revolves around workflow management, orchestrating the execution of data acquisition scraping tasks.

Data acquisition

- The architecture incorporates a configuration file containing detailed specifications regarding the targets earmarked for scraping. This enables the generation of new agents simply by changing the scraping sources stored in the configuration file.
- An “orchestrator” module is used to automate the definition, execution and monitoring of scraping processes; this is implemented with the assistance of Google Cloud Composer and Apache Airflow.
- A data scraper module is composed from multiple parts tailored to scrape data from various sources such as YouTube, websites, and academic repositories.
- After scraping the data are transcribed to machine readable format.

Retrieval Augmented Generation (RAG)

In the context of the present agent generation framework, responses that are relevant, up-to-date, accurate and whose origin can be traced is a central requirement. This requirement can be fulfilled by utilising a RAG technique. The contents of a given knowledge graph are turned into vectors (embeddings) and stored in a DB which can be enriched in regular intervals reducing or eliminating the need for expensive model retraining. A popular framework used to implement RAG is LangChain, since it enables the use of various LLMs while also allowing access to retrievers, vector stores and even more functionalities and thereby unites various parts of the RAG process in a single framework.

Context generation using a base LLM

The base LLM first interprets the user input, by parsing the text or voice input to understand the intent and extract relevant information. The base LLM understands the context of the interaction and then uses the additional domain specific data to refine its response. This is done using the Retrieval Augmented Generation (RAG). The LLM queries the embeddings stored in MongoDB and enriches its response. The base LLM works in synchronisation with the data acquisition pipeline and RAG system. As and when the embedded data source is enhanced the response of the LLM could also be improved. Also, the RAG would help the LLM to inform the user about the data sources used to produce the response, increasing the trustworthiness and verification of the information provided by LLM.

Data Storage

The storage aspect enables and facilitates the collection and processing of data from source to storage

- A document database is used for storing the text data and their metadata that are collected by the scraper modules.
- MongoDB serves as the primary database for storing the embeddings while cloud storage buckets are employed for scalable and accessible storage of different data types.

Brief Summary of Technology Stack

Categories	Technologies
Dev Environment	● Docker / Apptainer (for Containerization)
Frontend	● React Native (for cross-platform mobile app development) ● Typescript ● Google Firebase (for Authentication) ● Yarn (packet management)
Data Pipeline	● Python (executable) ● PDM (packet management) ● Google Cloud Composer (Airflow management) ● Apache Airflow (for Workflow Management) ● LangChain (RAG)
Data Storage	● GCP Storage / FAU HPC / MongoDB (for user and scraped data) ● MongoDB (for Embeddings)