THOMAS MORE

# Leo – Ordolio chatbot assistant

## Realization document

Mark Vaikul

**Bachelor of Applied Computer Science in Artificial Intelligence**

2024 – 2025

# Table of Contents


1. INTRODUCTION 3

2. ANALYSIS 3

2.1. Overview of Tools Explored 4

2.2. Language Model Evaluation 5

2.3. Flow Structuring with LangChain and CrewAI 5

2.4. Vector Database 6

2.5. Deployment Strategy 6

2.6. Stack 7

3. REALIZATION 9

3.1. MVP using GPT-TRAINER 9

3.2. MVP using CREWAI 10

3.2.1. Overview of the MVP full code 10

3.2.2. Explanation of “flow” 11

3.2.3. Explanation of “crew” 11

3.2.4. Filling in the Qdrant database 12

*3.2.4.1. Data Collection from External API 11*

*3.2.4.2. Embedding and Uploading to Qdrant 12*

3.2.5. Main logic of code 12

3.2.6. API 13

3.2.7. Dockerization 13

3.3. Local implementation in the Oracle platform 15

3.4. Host on AWS 23

4. CONCLUSION 25

REFERENCE LIST 26

ATTACHMENTS 22

4.1. Main_flow.py 27

4.2. crew.py 34

4.3. config.yaml 36

4.4. tool.py 37

4.5. endpoints.py 38

TIPS FOR WRITING IN YOUR GRADUATION PORTFOLIO 35

# 1. Introduction

This realization document describes the development and implementation of an AI-powered assistant, named **LEO**, created for the **Ordolio** platform during the internship period. The primary objective of the project was to design an intelligent, integrated chatbot capable of supporting clubs and associations in managing events, promoting activities, handling sponsorships, and offering personalized suggestions.
The assistant was initially envisioned as two distinct components:

- **Event Success Agent** – Focused on boosting event participation through promotion, communication support, and outreach.
- **Fun with Your Club Agent** – Intended to provide ideas for group activities and recommend relevant offers from partners.

Over time, and in response to evolving requirements, the assistant was extended and refined. Its scope expanded beyond static templates and business suggestions, evolving into a dynamic, multi-functional system with the following capabilities:

- Business analysis of planned or past events
- Actionable suggestions for improvement
- Assistance with navigating the Ordolio platform
- Multi-language support for a broader user base
- Access to and contextual analysis of the user's event data

To achieve these goals, a combination of advanced tools, frameworks, and techniques was applied throughout the project. This document presents a comprehensive overview of the process and decisions involved, divided into the following chapters:

- **Analysis** – A review of the evaluated tools, language models, frameworks, and infrastructure, with justification for each choice.
- **Realization** – A technical walkthrough of the implementation process, including:
    - *AI Assistant Development* – Initial prototyping, model selection, and agent design using CrewAI and LangChain.
    - *API and Flow Integration* – Asynchronous communication logic, API design, and flow orchestration.
    - *Database and Context Handling* – Implementation of Qdrant for semantic search and Redis for memory storage.
    - *Frontend Integration* – Embedding the assistant in the Ordolio UI and resolving integration challenges.
    - *Optimization and Testing* – Performance tuning, issue resolution, and final deployment.
- **Conclusion** – Reflections on the outcomes, challenges encountered, lessons learned, and recommendations for future enhancements.

The following chapters explain how each of these components was implemented and combined to produce a scalable, intelligent, and production-ready assistant solution.

# 2. Analysis

Before starting the development of the AI assistant for the Ordolio platform, a thorough comparison of different tools, platforms, and technologies was carried out. The aim was to find the best options for building a solution that would be scalable, reliable, and easy to maintain. Key points of focus included choosing the right language model, designing the conversational flow, and setting up an effective development and deployment environment.

This analysis helped to evaluate the available choices fairly and supported the final decisions with solid reasoning. It also created a clear starting point for the actual implementation.

## 2.1. Overview of Tools Explored

To make sure the AI assistant would be both effective and scalable, different tools and technologies were explored and compared. The selection focused on model performance, ease of integration, speed, and available community support. Below are the main components that were considered:

- **Language Models (LLMs):**

  Gemini 1.5 Flash, GPT-4.0 Mini, Mistral 7B, and Falcon 7B were compared. The evaluation focused on each model's reasoning capabilities, language understanding, response time, and overall resource usage. Cost-efficiency was also considered, especially for deployment in production environments.

- **Flow Management:**

  LangChain and CrewAI were reviewed as possible solutions for managing structured dialogue and multi-agent logic. The comparison looked at modularity, memory handling, ease of task delegation, and suitability for more advanced conversational workflows.

- **Vector Databases:**

  Qdrant and Pinecone were evaluated for semantic search and long-term memory storage. Factors like query performance, integration with embedding models, deployment flexibility, and community support were considered in the comparison.

- **API Framework:**

  FastAPI was compared with other backend frameworks based on its asynchronous support, development speed, and auto-generated documentation features. Its ability to streamline testing and integration was also taken into account.

- **Infrastructure:**

  Different containerization solutions were considered, with emphasis on deployment consistency, ease of local and cloud integration, and compatibility with CI/CD workflows.

- **Frontend Integration:**

  Integration approaches were explored for incorporating the assistant into the Django-based Ordolio platform. This involved assessing how well each approach handled session management, API endpoint alignment, and UI compatibility.

## 2.2. Language Model Evaluation

The development process began with the use of **GPT-Trainer**, which provided an opportunity for rapid in-code testing of early prototypes. This tool allowed for efficient experimentation with different prompts, tasks, and response formats in a controlled setting, helping to shape the assistant's initial capabilities.

After initial testing, several language models were evaluated to determine the most suitable one for deployment in the Ordolio platform. The comparison focused on four criteria: **performance**, **cost**, **context handling**, and **ease of use**. Each model was scored from 1 to 10 per category, and the total scores were used to support a data-driven selection process.

| Criteria | Weight | Gemini 1.5 Flash | | GPT-4.0 Mini | | Mistral 7B | | Falcon 7B | |
|---|---|---|---|---|---|---|---|---|---|
| | | Score | Total | Score | Total | Score | Total | Score | Total |
| Performance | 4 | 8 | 32 | 9 | 36 | 6 | 24 | 6 | 24 |
| Cost | 3 | 9 | 27 | 7 | 21 | 10 | 30 | 9 | 27 |
| Context Handling | 5 | 7 | 35 | 9 | 45 | 6 | 30 | 5 | 25 |
| Ease of Use | 2 | 8 | 16 | 9 | 18 | 6 | 12 | 6 | 12 |
| **Total Score** | | | **110** | | **120** | | **96** | | **88** |

**CONCLUSION:**
Based on the overall ranking, **GPT-4.0 Mini** was selected as the most effective and balanced option. It demonstrated strong accuracy, reliable context management, and easy integration into the existing development environment. While other models like **Gemini 1.5 Flash** and **Mistral 7B** offered strengths in specific areas, such as cost efficiency or multimodal support, GPT-4.0 Mini achieved the best overall performance and was chosen as the foundation for the assistant.

## 2.3. Flow Structuring with LangChain and CrewAI

To manage structured workflows and support complex interactions within the AI assistant, several tools were evaluated for their ability to orchestrate logic, maintain modularity, and enable multi-agent functionality.

LangChain was analyzed for its capacity to chain prompts, integrate memory, and connect tools within a single-agent framework. It offered flexibility for building linear workflows and provided a straightforward approach to incorporating tool usage and memory management.

CrewAI was reviewed for its focus on modular multi-agent workflows. It allows agents to operate with distinct roles and objectives, making the system more organized and goal-oriented. Its architecture supports parallel task delegation and coordinated response handling, which are important in systems requiring distributed logic and collaboration among components.

The comparison considered the following key factors:

- **Role-Based Agent Design:**

  CrewAI enables developers to assign specific roles and goals to individual agents, helping structure responsibilities in a clear, manageable way.

- **Flexible Orchestration:**

It supports dynamic coordination between agents, enabling more adaptive workflows and more efficient response handling.

- **Improved Code Structure:**

  A modular design contributes to cleaner code, reduced interdependency between components, and easier maintenance and debugging over time.

**CONCLUSION:**
Based on functionality, scalability, and developer experience, CrewAI proved to be the most effective solution for managing multi-agent workflows. While LangChain offered strong features for prompt chaining and memory integration, CrewAI's support for clear role separation, modular logic, and flexible orchestration led to significant improvements in system maintainability, clarity, and adaptability. As a result, it was adopted as the primary framework for workflow and agent management.

## 2.4. Vector Database

To enable efficient semantic search and memory management within the assistant, a suitable vector database was required. Two options were evaluated: **Qdrant** and **Pinecone**.

The assessment focused on five key criteria: performance, integration flexibility, scalability, documentation/support, and cost. Each criterion was assigned a weight based on its relevance to the project's needs. The following table presents the weighted ranking:

| **Criteria** | **Weight** | **Qdrant** | | **Pinecone** | |
|---|---|---|---|---|---|
| | | **Score** | Total | **Score** | Total |
| Performance | 4 | 8 | 32 | 9 | 36 |
| Integration Flexibility | 3 | 9 | 27 | 7 | 21 |
| Scalability | 5 | 7 | 35 | 9 | 45 |
| Documentation & Support | 2 | 8 | 16 | 9 | 18 |
| Cost | 5 | 9 | 45 | 6 | 30 |
| **Total Score** | | | **155** | | **150** |

**CONCLUSION:**
Both Qdrant and Pinecone demonstrated strong capabilities for vector-based semantic search. Pinecone scored slightly higher in scalability and documentation, making it suitable for large-scale enterprise applications. However, **Qdrant** achieved a higher overall score due to its superior integration flexibility, strong performance, and significantly lower cost. As an open-source solution with fast indexing and robust community support, Qdrant was selected as the preferred vector database for this project.

## 2.5. Deployment Strategy

To ensure consistent deployment and maintainability across various environments, all system components were containerized using **Docker**. In combination with **Docker Compose**, this setup enabled the orchestration of multiple services, such as the language model, vector database (Qdrant), backend API (FastAPI), and frontend (Django)—within a unified and configurable environment.

By using Docker and Docker Compose together, each component could be defined in a central configuration file, allowing services to be started, stopped, and updated in a controlled and repeatable manner. This significantly improved development workflows and made it easier to test, debug, and scale the system.

The containerized setup provided several key benefits:

- **Environment Consistency:** All services ran identically on different machines and operating systems.
- **Dependency Isolation:** Each container included only the packages and tools it required, reducing conflicts.
- **Simplified Orchestration:** Docker Compose enabled all services to be launched and linked automatically with a single command.
- **Faster Deployment:** The approach supported quicker testing and deployment cycles, especially when integrated into CI/CD pipelines.

For cloud hosting, both **Amazon Web Services (AWS)** and **Microsoft Azure** were considered. While both platforms offered robust support for containerized applications, **AWS** was selected due to the company's existing experience and infrastructure preferences.

**CONCLUSION:**
The combined use of **Docker** and **Docker Compose** ensured a modular, reproducible, and scalable deployment process. It allowed each system component to be independently developed, tested, and maintained, while still functioning seamlessly as part of an integrated solution. AWS was chosen as the hosting provider due to its compatibility with the existing organizational environment, streamlining the deployment pipeline, and reducing operational overhead.

## 2.6. Stack

Following a phase of iterative testing and validation, a final architecture was established to support the production deployment of the AI assistant. Each component was selected based on performance, integration compatibility, and long-term maintainability. The finalized technology stack is outlined below:

- **Model:** *GPT-4.0 Mini* via the OpenAI API, selected for its balanced performance, context handling, and integration ease.
- **Conversation Flow:** *CrewAI* combined with *LangChain* for modular, role-based multi-agent workflows and structured prompt management.
- **API Layer:** *FastAPI*, used for its asynchronous capabilities, fast response times, and auto-generated documentation features.

- **Databases:** *Qdrant* for vector-based semantic search and *Redis* for lightweight context memory and state management.
- **Frontend:** Integration with the existing *Django*-based user interface of the Ordolio platform.
- **Infrastructure:** Fully containerized environment using *Docker* and *Docker Compose* for consistent local and cloud deployment.
- **Hosting:** *Amazon Web Services (AWS EC2)*, selected for its reliability, scalability, and alignment with the company's existing infrastructure.

## 2.7. Fallback Strategy

Given the modular nature of the system and its reliance on multiple services—including external APIs, containerized components, and cloud infrastructure—a fallback strategy was developed to ensure resilience and minimize disruption in case of partial or total failure.

### *API & Model Failures*

The assistant's most critical external dependency is the OpenAI API (GPT-4.0 Mini). To handle cases where the API becomes temporarily unavailable (due to timeouts, rate limits, or service interruptions), the system includes internal error-handling logic. If a failure is detected, the backend returns a polite, human-readable message such as: *"The assistant is currently unavailable. Please try again shortly."* This ensures a smooth user experience without exposing backend issues. Retries and exponential backoff mechanisms can also be implemented to auto-recover if the outage is brief.

### *Container and Service Failures*

Redis and Qdrant, being core to maintaining context and executing semantic search, are configured with Docker's restart: always policy to automatically recover in case of container crashes. Additionally, health checks can be defined in docker-compose.yml to ensure dependent services do not start until their requirements are healthy (e.g., Qdrant starting before the API that uses it). This prevents cascading errors during deployment or container restarts.

### *Cloud Hosting Failures (AWS)*

In the event of AWS instance failure (e.g., EC2 downtime, accidental termination), the system can be redeployed locally on any development machine using Docker Compose. All critical configuration files, environment variables, and volumes are managed within version-controlled repositories, enabling reproducibility across environments. In the future, introducing infrastructure-as-code (e.g., Terraform) could further automate this recovery.

### *Frontend or Integration Failures*

If the frontend (Django) fails to load the assistant or loses the connection to the API, the user interface will display fallback text or hide the chat window temporarily, avoiding confusion. Session memory managed by Redis ensures conversations are preserved when services recover, allowing the user to resume without data loss.

# 3. Realization

## 3.1. MVP using GPT-TRAINER

Following the analysis and tool selection phase, the development of the AI assistant, named **Leo**, was initiated. Leo was integrated into the Ordolio platform as an intelligent chatbot designed to provide more than basic support. The assistant combines event-specific insights, general support features, and business logic into a unified conversational interface, enhancing user experience and platform usability.

The first Minimum Viable Product (MVP) version of Leo was developed using **GPT-Trainer**. This tool enabled rapid prototyping and allowed for early testing of the assistant's core functionalities. Key focus areas included:

- Basic prompt-response interactions
- Integration with the existing Ordolio frontend
- Testing of initial conversation flow structures

While GPT-Trainer proved valuable for establishing the foundational logic and validating integration points, it also exposed limitations in response quality and adaptability. These findings led to a transition toward more advanced language models and more robust conversation flow frameworks as the project progressed.

At the moment of developing the product, most of the time went into learning the documentation of the framework and tests on a fully coded product; this game took a lot of time to correct errors, which helped a lot later on in development.

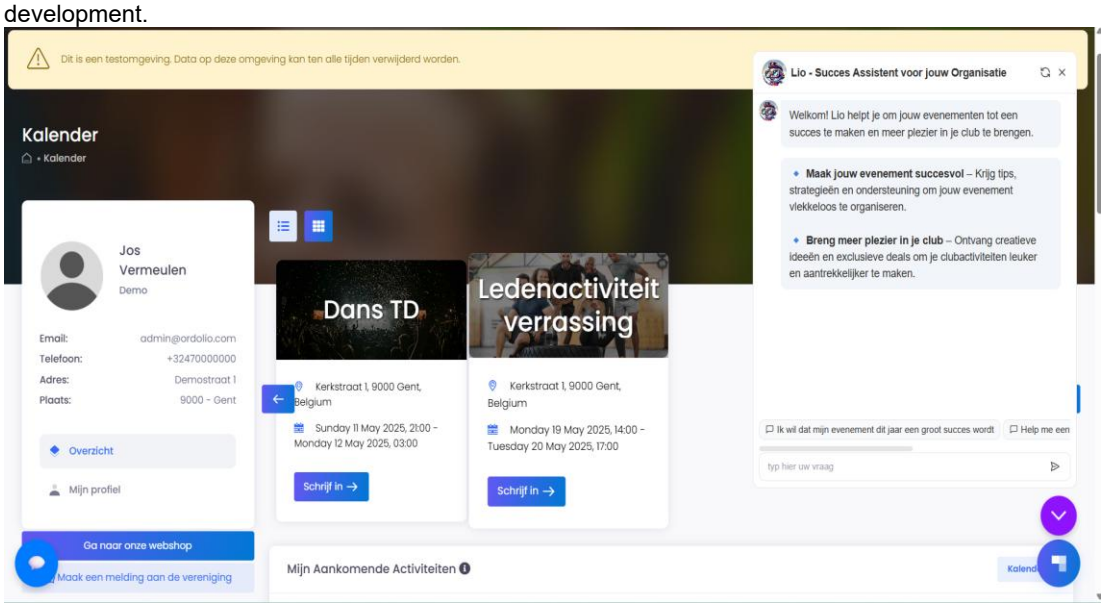



*Figure 1: Leo, the chatbot assistant, before the implementation of the full code version*

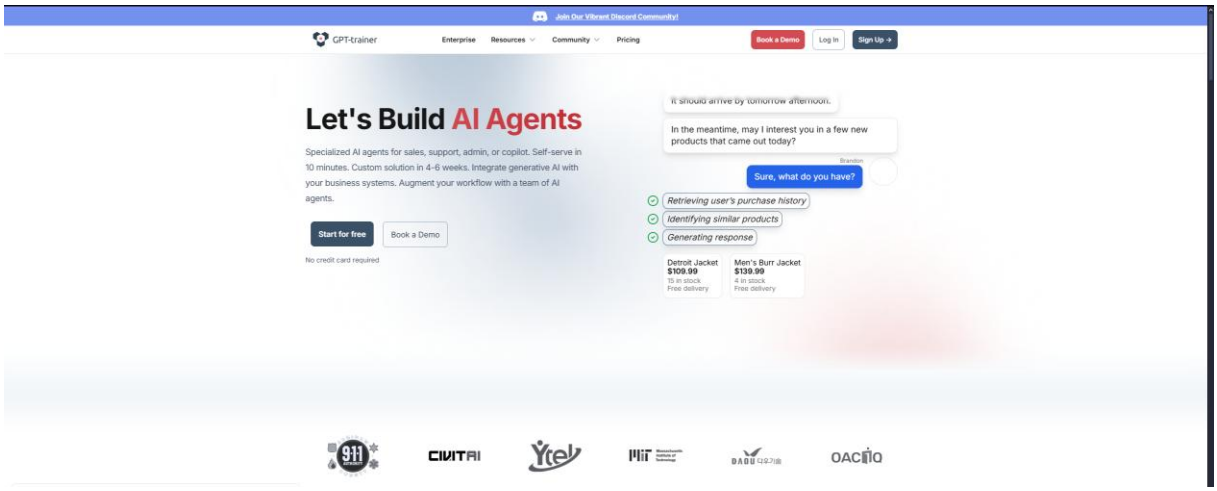



*Figure 2: GPT-trainer low-code framework, used for the first MVP*

## 3.2. MVP using CREWAI

### 3.2.1. Overview of the MVP full code

The assistant, named **Leo**, was designed to support a wide range of tasks relevant to clubs and associations using the Ordolio platform. These tasks combine event analysis, user support, and advanced language capabilities into a single, conversational interface. The key functions developed include:

- **Event Checking:** Leo can retrieve and list upcoming or past events that are relevant to the user or their associated club.
- **Business Analysis:** The assistant is capable of performing basic analytical operations on event-related data to generate insights and provide performance feedback.
- **Event Comparison:** Leo can identify similar events and compare them based on metrics such as attendance, engagement, and key features.
- **Support Assistance:** Users can rely on Leo for help navigating the Ordolio platform, accessing features, and resolving common usage questions.
- **Casual Chatting:** To create a more natural and engaging experience, Leo supports informal conversation in addition to task-specific queries.
- **Complex Queries:** The assistant is able to handle compound instructions, such as: *"Find an event like X, compare it with Y, and tell me which had the better turnout."*
- **Context Memory:** Using Redis, Leo can maintain context across multi-turn conversations, allowing for more coherent and intuitive dialogue over time.
- **Generation of suggested prompts:** generation of prompts that could help the user continue talking to the bot.

All features were developed individually and integrated into a cohesive framework. As no dedicated frontend, backend, or AI specialist was available in the intern team during the project, the full implementation, including agent logic, API development, and user interface integration, was completed independently.

The initial logic structure was built using a centralized **manager-agent** model within **CrewAI**, where multiple conditional tasks were delegated to subordinate agents. However, this approach led to performance bottlenecks and high complexity. Further exploration of CrewAI introduced the concept of **flows**, which improved performance and maintainability by separating task groups into specialized crews and defining structured execution paths.

### 3.2.2. Explanation of “flow”

The logic that drives the assistant is built around the **CrewAI flow architecture**, which allows for structured and stateful orchestration of tasks across multiple agents. The flow design consists of three core components:

1. **State Object** – Maintains variables and shared data between flow steps.
2. **Flow Function** – Acts as a reactive listener that triggers upon the completion of prior steps or specific events.
3. **Execution Function** – The entry point for initiating the flow, often implemented asynchronously (e.g., kickoff() or async_kickoff()).
4. **Decorators** – used to define the flow's execution order.

Mainly, code explanation will consist of explaining Flow Functions, as they are building blocks for flow, usually they split into 2 types:

- Crew/llm execution functions
- Code executing functions

A code sample demonstrating this architecture is provided in the attachment titled main_flow.py

### 3.2.3. Explanation of “crew”

After stabilizing the flow orchestration, the next phase of development focused on building and configuring **crews** using the CrewAI framework. In this context, a *crew* is a modular group of AI agents designed to work collaboratively toward completing a specific task or workflow. Each agent within the crew is assigned a clearly defined role, a set of tools, and an operational goal. Crew-based architectures are particularly useful for managing complex, multi-step interactions that benefit from delegation and specialization.

**CrewAI** enables developers to design agent ecosystems in which each member contributes based on its role—for example, one agent may be responsible for retrieving data, another for interpreting results, and a third for generating the final output. This separation of concerns improves modularity, clarity, and maintainability in the overall system.

Each crew is implemented using two main components:

- **Main Crew File:** Contains the logic for task delegation, the declaration of agents, tool assignments, and the orchestration structure.
- **Configuration File:** Provides detailed metadata, including agent descriptions, behavioral parameters, task definitions, and other dynamic settings.

An example of a crew can be found in the reference list **here.**
And the config example can be examined in the attachments section with the name config.yaml.

### 3.2.4. Filling in the Qdrant database

To support advanced semantic search capabilities within the assistant, a custom vector database was implemented using **Qdrant**. Since no ready-made dataset was available for the specific needs of the Ordolio assistant, a full pipeline was developed to **collect**, **process**, **embed**, and **store** relevant event data.

The process was completed in two main stages: data acquisition and vector embedding.

#### *3.2.4.1. Data Collection from External API*

The first stage involved fetching structured event data from the **Uitdatabank.be** an API. A custom script was developed to request events that take place in the year 2025, filter out incomplete entries, and extract only relevant metadata such as event name, description, location, timing, organizer information, and pricing details.

Each event was parsed, enriched with missing fields (such as generating user-friendly URLs), and stored in a local JSON file for downstream processing. This file served as the raw input for the embedding phase.

**You can find an example of this data acquisition script** here**.**

#### *3.2.4.2. Embedding and Uploading to Qdrant*

Once the raw event data was collected, a second script was used to **vectorize** the data using OpenAI's text-embedding-3-small model. Each event was transformed into a descriptive text block (containing name, description, and location), then embedded into a fixed-size vector.

These vectors were wrapped in Qdrant-compatible data structures, tagged with metadata, and uploaded to a **dedicated collection** (events_2025) in Qdrant. If the collection did not yet exist, it was created dynamically with appropriate parameters (vector size and distance metric).

**The embedding and upload process is illustrated in the following script -** upload.py

### 3.2.5. Main logic of code

During the development of the assistant, several technical and practical challenges had to be addressed independently. These included the lack of consistent code examples for specific language models, variations in implementation approaches, limited framework documentation, and the substantial complexity of integrating with a large-scale platform such as Ordolio. Framework limitations, such as restricted support for asynchronous operations within flows, also required workaround solutions. Despite these obstacles, all critical issues were ultimately resolved through extensive testing, documentation analysis, and the development of custom components.

The **initial logic and flow implementation** were developed using **Gemini 1.5 Flash**. This model was selected at the early stages of the project due to its low cost and budget alignment with the company's resource constraints. It enabled rapid iteration and allowed foundational components to be tested before transitioning to **GPT-4.0 Mini** for the production version.
Once the main flow and crew architecture were complete, attention shifted to implementing practical features and real-world functionality. The hybrid strategy—balancing speed and quality between LLM-based responses and agent-based crews—enabled the assistant to handle a broad set of user requests effectively.
The final flow logic was created this way:

1. Take user input and save it in the state
2. Analyze the intent of the user to know to which function the user should be redirected (multiple functions possible
3. Intent was analyzed and distributed between functions: Answer general questions, Answer support question that uses the knowledge base ( big text files with all of the answers to general questions of Ordolio), fetch events from Qdrant database ( find relevant events), fetch personal events with the usage of a custom tool
4. Then it passed to Business analysis, if intent contains it as well, it analyzes outputs of all of the agents to take into account
5. Next, it is passed to summarization to get a unified response
6. Based on the response, context, and prompt of the user, suggested prompts are generated ( 3 of them )

Key features supported by the final system include:

- **Detailed Business Analysis:** Complex prompts related to event performance are processed by specialized crews with access to data and analytical tools.

- **Platform Support Queries:** Users can receive assistance navigating and using the Ordolio platform.

- **Personalized Event Retrieval:** Agents can fetch a user's events or identify similar ones based on descriptive criteria.

- **Event Comparison:** The assistant can compare multiple events across performance metrics, audience demographics, or other factors.

- **Generation of suggested prompts:** prompts for the user to help with conversation

### 3.2.6. API

To expose the assistant's functionalities to the frontend and manage interactions between the user and the internal services, a dedicated **API layer** was developed using **FastAPI**. This service acts as the secure and structured entry point to the system, handling client requests, managing session data, and forwarding messages to the internal CrewAI logic engine.

The code for the API is organized into a single FastAPI application with clearly defined responsibilities and route endpoints.

The API layer includes the following core elements:

- **CORS Middleware:**

  Configured to allow cross-origin communication between the API and the frontend (e.g., the Django UI served on localhost:8000 during development).

- **Redis Integration:**

  Used to persist conversation sessions. The API connects to Redis via environment variables, ensuring that context is maintained securely and efficiently across multiple messages in a session.

- **CrewAI Integration:**

  The core assistant logic resides within the internal crew_ai service. The API communicates with this service through HTTP POST requests, sending user messages and context, and receiving generated responses.

- **Pydantic Models for Validation:**

  All incoming requests are validated using Pydantic schemas. This ensures the presence and integrity of required fields such as conversation_id, message, and access_token.

You can find the code for the API here

### 3.2.7. Dockerization

The containerized deployment architecture was designed with a clear separation of concerns and layered service exposure in mind. The goal was to ensure maintainability, security, and scalability across development and production environments.
The architecture is composed of three primary services:

- **crew_ai (Internal)**

  This service handles the core logic of the assistant, including conversation flows, agent coordination, and task execution using CrewAI. It is kept entirely **internal** to protect the flow logic and underlying models

from direct external access. This separation ensures that AI processes remain encapsulated and cannot be directly invoked from the public network.

- **api (Public Interface)**

  The API service serves as the **only externally accessible component**. It acts as a secure intermediary between the frontend (e.g., Ordolio's Django interface) and the internal services. This service receives user requests, handles authentication and routing, and forwards queries to crew_ai or Redis as needed. By isolating the public interface, the system significantly reduces its attack surface and simplifies access control.

- **redis (Internal Context Memory)**

  Redis is used for fast, in-memory storage of temporary session data and conversation context. Like crew_ai, it is only accessible from within the internal network, which ensures privacy and minimizes potential data leaks.

This architectural separation is enforced through the use of **Docker Compose networks**:

- **internal_net** is used to isolate services that should not be exposed publicly (e.g., crew_ai, redis).
- **public_net** connects only the API service, allowing it to receive traffic from the outside (e.g., localhost or cloud server).
- **shared_net** is defined externally for scenarios where coordination across project containers is needed, but its usage remains restricted.

By design, this layered architecture enforces **security-by-default**, where all sensitive logic and data processing occur in private services, and only the essential API gateway is exposed. It also enhances maintainability by decoupling concerns and allowing each service to scale independently if needed.

## 3.3. Local implemenataion in Ordolio platform

After finishing all the required logic for the AI part, the frontend started to take place. Due to company policy, I can't show the full code of the Ordolio platform. Shortly, work on the Ordolio platform consisted of:

- Creating a separate view page that acted like a reusable component
- Integrating this view in the Ordolio base template
- Writing all backend logic and frontend logic, including design
- Attaching all necessary information to the window, creating endpoints
- Responsive design
- Attach the AI conversation to the session link so restart of the page does not restart the conversation itself.

After it, the project could work locally and display all of the required needs from the internship supervisor.

*Tabel 1 Example of usage: Chatbot: Greeting chatbot and seeing auto-generated prompts based on context*

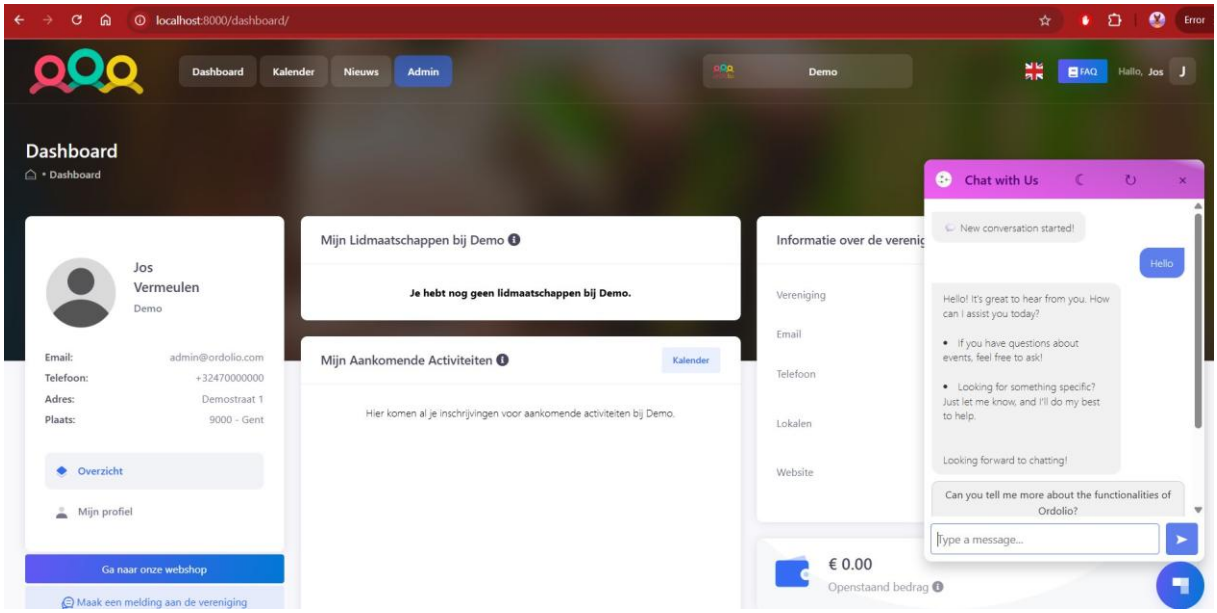
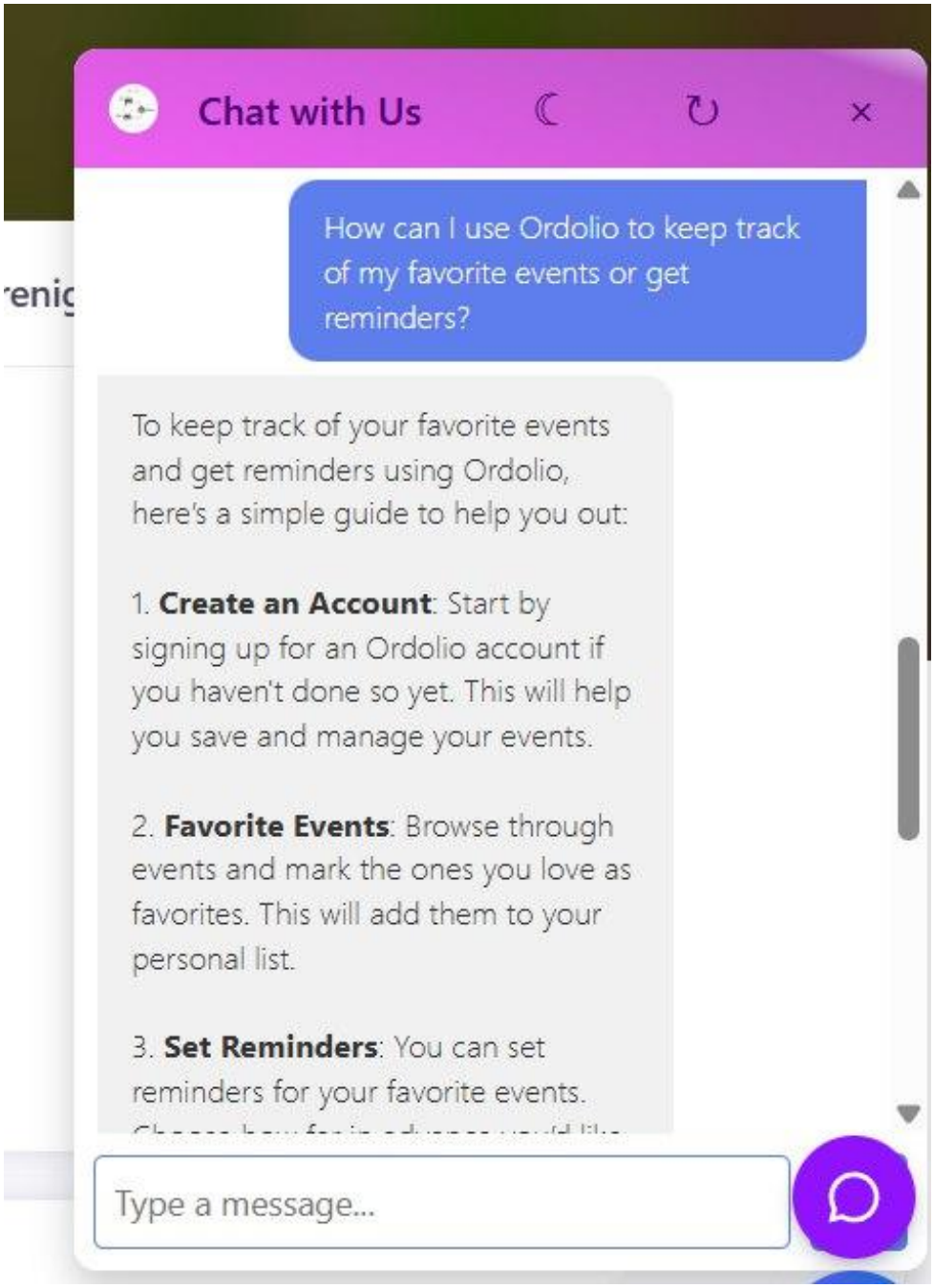


*Table 2 Example of Chatbot usage: Asked Ordolio a specified question, which is taken from the Knowledge file.*



*Tabel 2 Example of usage: Simple query (3-4 seconds to respond)*

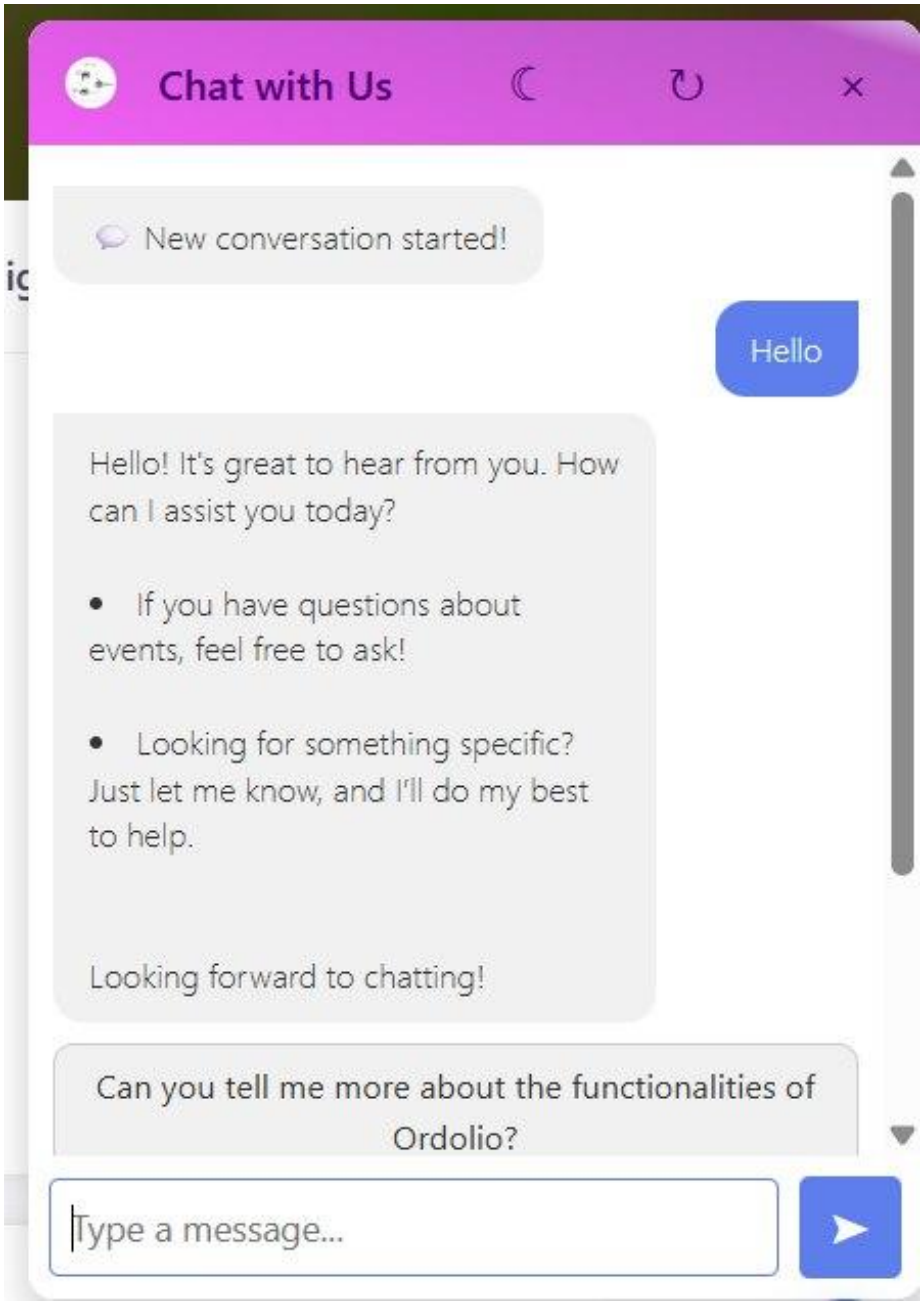


*Table 3 Example of Chatbot usage: more detailed photo of the suggested prompts*

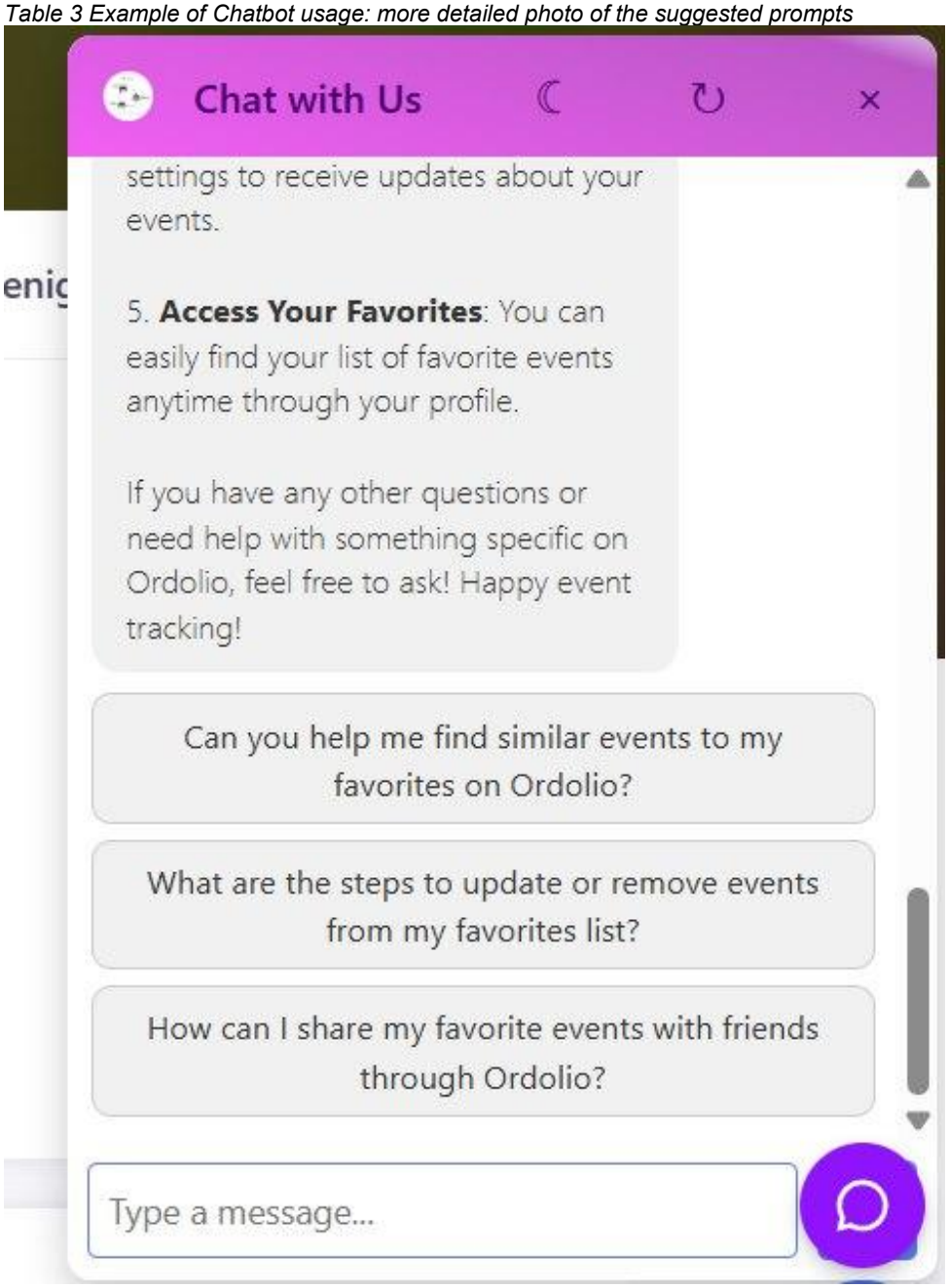


*Figure 5 Example Chatbot usage: Request for personal information*

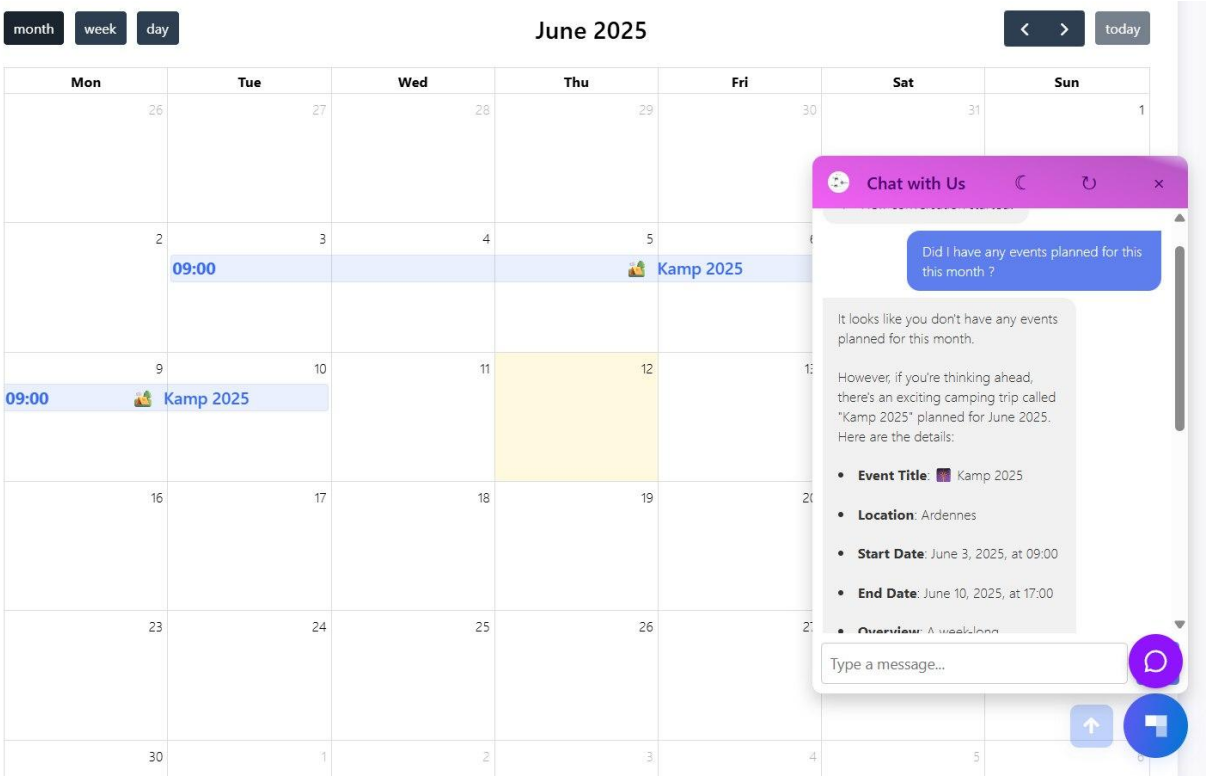


*Figure 6 Example Chatbot usage: Analysis of personal event from a business perspective*

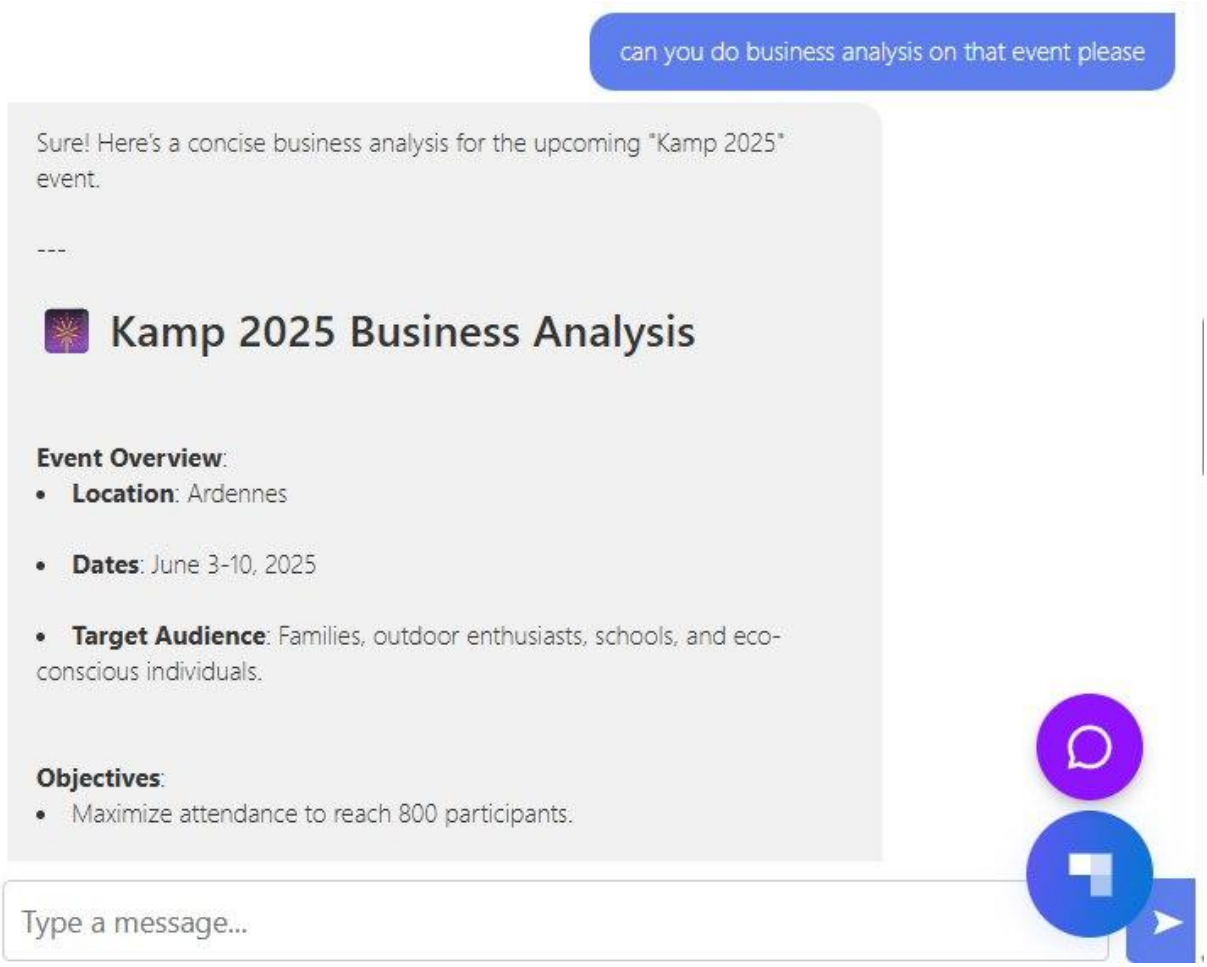


*Figure 7 Example Chatbot usage: Analysis of personal event from a business perspective*

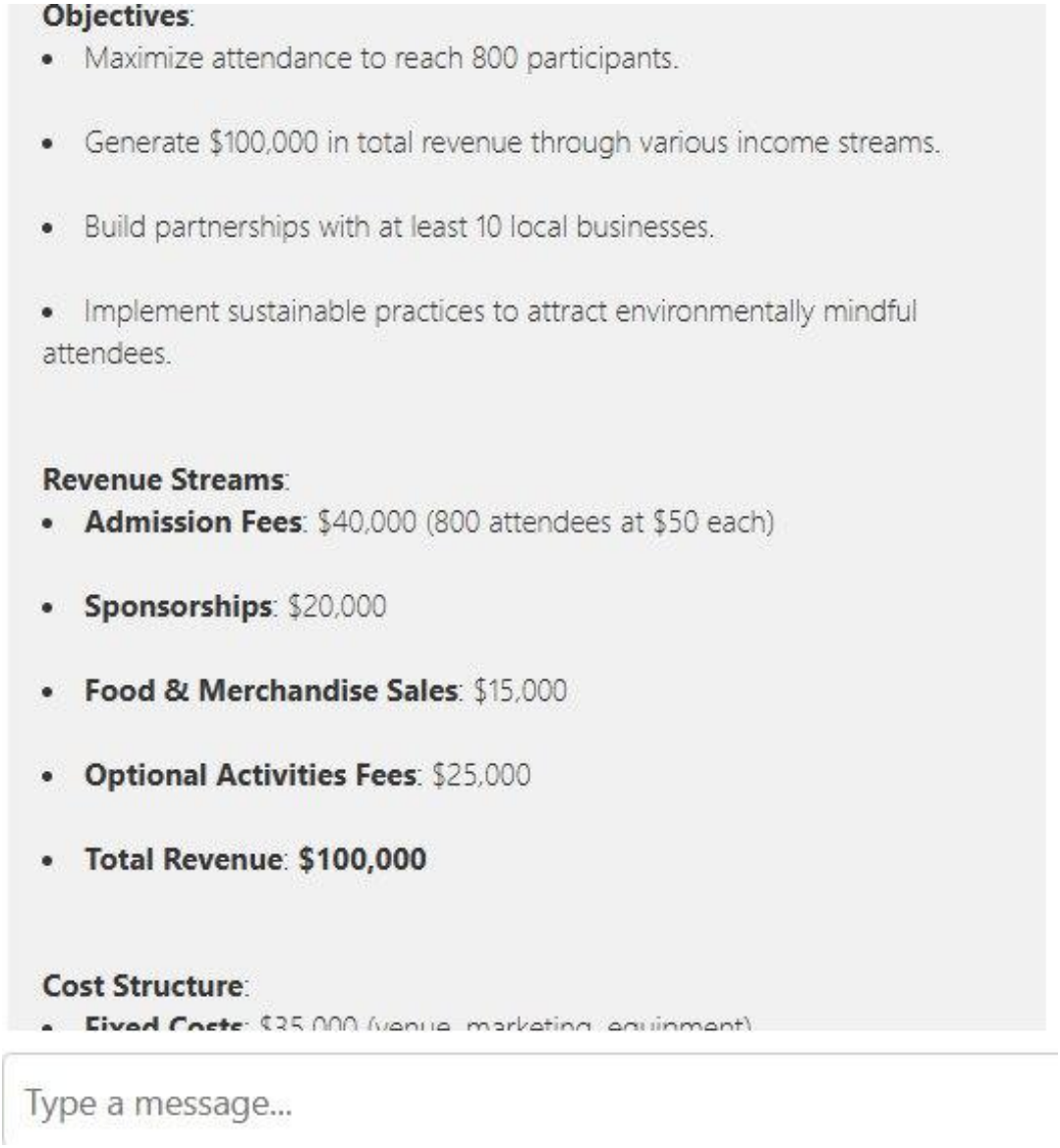


*Figure 8 Example Chatbot usage: Analysis of personal event from a business perspective*

**Cost Structure**:

- **Fixed Costs**: $35,000 (venue, marketing, equipment)
- **Variable Costs**: $30,000 (staff and materials)
- **Total Costs**: **$65,000**
- **Estimated Profit**: **$35,000**

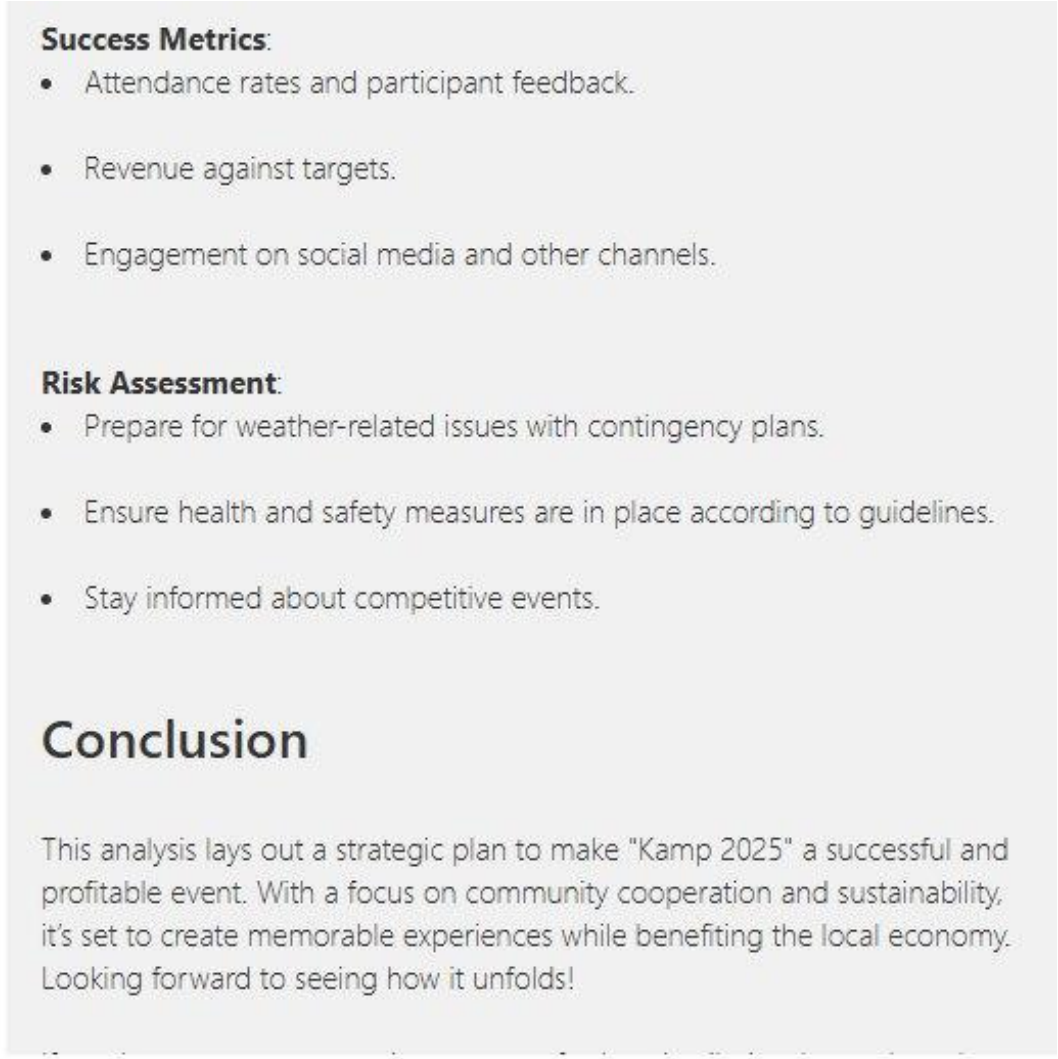
**Marketing Strategy**:

- Utilize social media platforms for targeted promotions.
- Build an email list for direct communication and updates.
- Partner with local schools and businesses for outreach.
- Engage local influencers for wider visibility.

**Success Metrics**:

- Attendance rates and participant feedback.
- Revenue against targets.

*Figure 9 Example Chatbot usage: Analysis of personal event from a business perspective*




## 3.4. Host on AWS

For hosting the assistant system in a stable and scalable production environment, **Amazon Web Services (AWS)** was selected. This choice was based on the platform's reliability, comprehensive support for containerized deployments, and prior familiarity within the company infrastructure.

An **EC2 (Elastic Compute Cloud)** instance was provisioned to serve as the deployment target. This virtual server provided a flexible Linux-based environment where the containerized system could be launched and maintained efficiently.

The deployment process involved the following key steps:

- **Source Code Retrieval:**

  The entire project was cloned directly from a private GitHub repository onto the EC2 instance using Git. This ensured that the latest stable version of all components—API, CrewAI logic, Docker configuration, and environment files—was available on the host system.

- **Docker Environment Setup:**

  Docker and Docker Compose were installed on the EC2 instance. Using the docker-compose.yml

configuration, all services (redis, crew_ai, and api) were launched with a single command. Environment variables (including API keys and configuration settings) were injected via a secure .env file.

- **Port Configuration and Exposure:**

  Only the **API service** was exposed to the public internet via port **9090**, mapped to internal FastAPI port **8000**. This was configured in both the docker-compose.yml file and the EC2 security group rules. All other services, including redis and crew_ai, remained **completely internal**, connected via private Docker networks to reduce the attack surface and improve system security.

- **Persistent Storage and Monitoring:**

  The Redis container was configured with a persistent volume to store conversation memory across restarts. Additionally, standard logging and container restart policies (restart: always) were used to maintain service reliability.

This setup allowed the assistant system to operate continuously, securely, and with minimal overhead. Future updates to the project can be deployed by simply pulling the latest version from GitHub and restarting the containers.

## 3.5. Testing & Evaluation

### 3.5.1. Testing

A series of technical tests were conducted to ensure the reliability, security, and performance of the AI assistant under different operational scenarios.

- **Container Connection Failure:** The system was tested for scenarios where connection to critical containers (such as Redis or Qdrant) was lost. The assistant handled these cases successfully using Docker's restart policies, and communication was restored automatically once services became available again.
- **Redis Expiry & Session Integrity:** To simulate a real-world session timeout, data in Redis was manually expired during an active session. Despite the missing context, the assistant preserved the session ID and gracefully resumed the conversation, confirming robust session linkage and state recovery logic.
- **Security – Input Injection:** The assistant was tested against input injection attempts, including code snippets and malformed inputs. These were safely ignored, with no execution or crash occurring. This demonstrates that user input is correctly sanitized and parsed without introducing risk.
- **Asynchronous Workflow Validation:** The multi-agent flow architecture was verified for asynchronous consistency. All async tasks (e.g., simultaneous agent calls and flow transitions) performed reliably and in the expected order, confirming correct flow orchestration and state propagation.
- **Response Time & Load Handling:** The assistant was benchmarked for response time across varying query complexity:
  - *Simple prompts*: average response time ~4 seconds
  - *Intermediate prompts*: ~10–15 seconds
  - *Complex prompts with full CrewAI utilization*: up to 45 seconds

  These results represent a significant improvement over early development phases (weeks 1–6), where latency and instability were frequent issues.

### 3.5.2. Evaluation

Feedback from the company supervisor was highly positive. The solution met all functional expectations and was described as a “very good result.” The assistant's integration into the Ordolio platform, both backend and frontend, was described as complete and effective. The hosting setup on AWS and the fallback mechanisms were particularly well received, as they support future scalability.

From a personal perspective, the final system shows measurable progress in performance and robustness. It handles connection loss gracefully, maintains session continuity, and returns fallback messages when encountering critical failures. The assistant is now capable of handling complex queries across multiple agents

while remaining responsive and stable. The improvements made in code structure, Dockerization, and testing have contributed to a mature, production-ready implementation. But there is much more to optimize and handle to make this system perfect, and I see it.

# 4. Conclusion

This internship project was both a challenge and a rewarding experience. My main goal was to design and implement a smart AI assistant—LEO—for the Ordolio platform. What started as a simple idea for a template-suggestion bot quickly grew into something much more complex and capable. In the end, I built a modular assistant that could do business analysis, compare events, support users, and even work in multiple languages with contextual memory.

Throughout the process, I had to explore different tools, models, and frameworks. I experimented with various large language models and finally chose GPT-4.0 Mini because it gave the best mix of speed, quality, and memory handling. One of the biggest game-changers for me was discovering CrewAI. It helped me organize my agents and workflows in a clean, logical way and made it much easier to manage complicated conversations using flows.

From a technical point of view, I'm proud of what I achieved:

- The assistant is live and working on the Ordolio platform.
- It can understand what users want, fetch and process information, and respond intelligently.
- It runs on a backend that I fully dockerized and connected through FastAPI to the frontend interface.

Of course, things didn't always go smoothly. I often ran into missing or confusing documentation, had to figure out tricky bugs alone, as well as initial complexity of task was much lighter, so I had to create a lot of features you see and hear about this chatbot nowdays and even wrote custom tools when the ones provided didn't fit my use case. There were moments when I felt stuck or overwhelmed, especially since I was handling most of the backend and AI work by myself, but those challenges forced me to become more independent and confident in my problem-solving.

Looking back at my original project goals, I can proudly say I met them—and maybe even went a bit beyond. The system I created is scalable, flexible, and already helping users in real-world situations. That feels pretty amazing.

That said, there's always room to grow. If I were to continue this project, I'd:

- Improve the performance for real-time responsiveness.
- Add fallback options for handling model uncertainty.
- Integrate better logging and monitoring to catch issues more easily.

This internship taught me a lot—not just technically, but also about managing my time, staying motivated when things get hard, and pushing forward even when answers aren't easy to find. I'm really glad I got to be part of something useful and meaningful. If I could do it all over again, I would—only faster, cleaner, and with even more creativity.

This experience will stay with me as I take my next steps in tech.

# REFERENCE LIST


**Official Documentation**

- CrewAI Documentation. (n.d.). https://docs.crewai.com/
- LangChain Documentation. (n.d.). https://docs.langchain.com/
- OpenAI API Documentation. (n.d.). https://platform.openai.com/docs
- GPT-Trainer Documentation. (n.d.). https://guide.gpt-trainer.com/introduction
- FastAPI Documentation. (n.d.). https://fastapi.tiangolo.com/
- Qdrant Documentation. (n.d.). https://qdrant.tech/documentation/
- Redis Documentation. (n.d.). https://redis.io/docs/
- Docker Documentation. (n.d.). https://docs.docker.com/

**Community Forums & Technical Help**

- CrewAI Community Forum. (2025). https://community.crewai.com/
- Stack Overflow. (2025). https://stackoverflow.com/ – Used for solving issues in Python, FastAPI, Docker, and async workflows.
- GitHub Discussions – CrewAI. (2025). https://github.com/CrewAI/crewai/discussions

**AI Assistance Tools**

- ChatGPT (GPT-4) by OpenAI. https://chat.openai.com
    - Used extensively for: prompt design, debugging, flow structuring, technical writing

# ATTACHEMENTS

## 4.1. Main_flow.py

```
import json
import re
from datetime import date
from random import randint
from typing import List

From pydantic import BaseModel
from crewai import LLM
from crewai.flow import Flow, listen, start, and_

from flow_Ordolio_ai.crews.database_crew.database_crew import DatabaseCrew
from flow_Ordolio_ai.crews.fix_personal_data_crew.fix_personal_data_crew
import FixPersonalDataCrew
from flow_Ordolio_ai.crews.business_crew.business_crew import BusinessCrew
from flow_Ordolio_ai.crews.support_crew.support_crew import SupportCrew


class ConversationState(BaseModel):
    intent: List[str] = []
    prompt: str = ""
    context: dict = {}
    debug_context: dict = {}
    access_token: str = ""
    current_org: str = ""


class PoemFlow(Flow[ConversationState]):
    def __init__(self):
        super().__init__()

    llm = LLM(model="gpt-4o-mini")

    @start()
    def take_user_input(self):
        print(self.state)
        return self. state

    @listen(take_user_input)
    def intent_classification(self, state):
        print("defying intent execution")

        messages = [
```

```
            {"role": "system", "content": "You are a helpful assistant
designed to output JSON."},
            {"role": "user", "content": f"""
        You are identifying the user's intent based on their current prompt
and the context of the previous conversation.

        - Context of conversation: {state.context}
        - User prompt: "{state.prompt}"

        ---

        Instructions:

        Determine the user’s intent by logically connecting their current
prompt with prior conversation. Only add `req-inf` or `req-act` if truly
needed, since these are time-consuming operations.

        Key Reminders:
        - There is always an intent for each request, there can be multiple
intents for each request, and there can be none for each request.
        - Users may refer to previously mentioned events or data, so always
check if the required information is already available in the context.
        - Avoid repeating data fetch (`req-inf`) or database actions (`req-
act`) unless:
          - The user explicitly requests more information.
          - A new event or topic is introduced.
          - If the request is about a previously discussed event, summarize or
respond using existing knowledge unless the user asks for more.

        Examples:
        - "Based on what you found, can you give me business advice?"
          → Intent: [rec-business]
        - "Can you give me a summary of this event?"
          → Intent: [req-inf] only if the event was not mentioned before IN
CONTEXT.
          → Intent: [] if the event was already covered BEFORE IN
CONVERSATION, and the user just wants a brief.

        Intent Labels:
        1. req-gen-qn: General questions/statements
        2. req-inf: Request for events NOT in user schedule
        3. req-act: Request to search in the user's schedule
        4. req-business: Business-related queries
        5. req-sup: Support-related queries

        ---
```

```
        Output format:
        JSON object with intent as a list of strings.
        Example: "intent": ["req-inf", "req-act"]
            """}
        ]

        response = self.llm.call(messages=messages)

        Try:
            json_str = re.search(r"\{.*\}", response, re.DOTALL).group(0)
            json_response = json.loads(json_str)
            print(f"JSON response: {json_response}")
            print(f"Raw response: {response}")
            state.intent = json_response["intent"]
            print(f"Intent: {state.intent}")
        Except Exception as e:
            raise ValueError(f"Failed to extract JSON from response:
{response}") from e

        return state

    @listen(intent_classification)
    def answer_general_question_or_statement(self, state):
        print("defying general question or statement check")

        If "req-gen-qn" is in state.intent:
            print("check passed")

            messages = [
                {"role": "system", "content": "You are a helpful assistant of
the company Ordolio"},
                {"role": "user", "content": f"""
                context of conversation that you need to take into account:
                {state.context}
                Answer on human prompt, but only on general questions, if it
considers finding an event or something in the database, ignore it
"{state.prompt}"
                """}
            ]

            llm_response = self.llm.call(messages=messages)
            self.state.debug_context["req-gen-qn"] = llm_response
            print(llm_response)
        else:
            Self.state.debug_context["req-gen-qn"] = "NONE"

        return state
```

```
    @listen(intent_classification)
    def support_action(self, state):
        print("suppoert action check")

        If "req-sup" is in state.intent:
            print("check passed")

            messages = [
                {"role": "system", "content": "You are a helpful assistant of
the company Ordolio"},
                {"role": "user", "content": f"""
                context of conversation that you need to take into account:
                {state.context}
                Answer on human prompt, but only on general questions, if it
considers finding an event or something in the database, ignore it
"{state.prompt}"
                """}
            ]

            llm_response = self.llm.call(messages=messages)
            self.state.debug_context["req-gen-qn"] = llm_response
            print(llm_response)
        else:
            Self.state.debug_context["req-gen-qn"] = "NONE"

        return state

    @listen(intent_classification)
    def find_relevant_event(self, state):
        print("finding relevant event check")

        If "req-inf" in state.intent:
            print("check passed")
            result = DatabaseCrew().crew().kickoff(inputs={"prompt":
state.prompt})
            self.state.debug_context["req-inf"] = result
            print(result)
        Else:
            Self.state.debug_context["req-inf"] = "NONE"

        return state

    @listen(intent_classification)
    def support_agent(self, state):
        print("support agent check")
```

```
        If "req-sup" is in state.intent:
            print("check passed")
            result = SupportCrew().crew().kickoff(inputs={"prompt":
state.prompt})
            self.state.debug_context["req-sup"] = result
            print(result)
        Else:
            Self.state.debug_context["req-sup"] = "NONE"

        return state

    @listen(intent_classification)
    def personal_event_search(self, state):
        If "req-act" is in state.intent:
            print("check passed")

            Try:
                Today = date.today()
                session_token = state.access_token

                print("🖋 Pr:", state.prompt)
                print(f"🖋 Session Token: {session_token}")
                print("🖋 Context Type:", type(state.context))
                print("🖋 Context Value:", state.context)
                print("added new")

                result = FixPersonalDataCrew().crew().kickoff(inputs={
                    "prompt": state.prompt,
                    "context": state.context,
                    "session_token": session_token,
                    "current_org": state.current_org,
                    "current_date": today.strftime("%Y-%m-%d")
                })

                self.state.debug_context["req-act"] = result
                print("☑ PersonalDataCrew result:", result)

            Except Exception as e:
                error_msg = f"❌ Error in sql_query_finding: {e}"
                print(error_msg)
                self.state.debug_context["req-act"] = error_msg
        Else:
            Self.state.debug_context["req-act"] = "NONE"

        return state

    @listen(and_(
```

```
        find_relevant_event,
        personal_event_search,
        answer_general_question_or_statement,
        support_agent
    ))
    def business_crew(self, state):
        print("stage 1")
        print(state.debug_context)

        If "req-business" is in state.intent:
            print("check passed for business crew")

            Try:
                print("initiating business crew")
                result = BusinessCrew().crew().kickoff(inputs={
                    "prompt": state.prompt,
                    "context": state.context
                })

                self.state.debug_context["req-business"] = result
                print("☑ BusinessCrew result:", result)

            Except Exception as e:
                error_msg = f"❌ Error in business_crew: {e}"
                print(error_msg)
                self.state.debug_context["req-business"] = error_msg
        Else:
            Self.state.debug_context["req-business"] = "NONE"

        return state

    @listen(business_crew)
    def summarize_answer(self, state):
        print("Conversation ended")
        print("stage 2")
        print(state.debug_context)

        messages = [
            {"role": "system", "content": "You are a helpful summary
assistant."},
            {"role": "user", "content": f"""
Prompt: "{self.state.prompt}"
Context of conversation: "{self.state.context}"

Answers to consider:
1. {self.state.debug_context["req-gen-qn"]}
2. {self.state.debug_context["req-inf"]}
```

```
3. {self.state.debug_context["req-act"]}
4. {self.state.debug_context["req-business"]}
5. {self.state.debug_context["req-sup"]}

Instructions:
- Combine all inputs into one cohesive human response.
- Don’t mention agents.
- Format for JS frontend (e.g., clear paragraphs, bullets).
- Be concise and friendly.
            """}
        ]

        print(f"Summary messages: {messages}")
        try:
            response = self.llm.call(messages=messages)
            self.state.debug_context["summary"] = response
        except Exception as e:
            response = f"❌ Summary failed: {str(e)}"
            self.state.debug_context["summary"] = response
            print(response)

        return response

    def kickoff_async(self, inputs=None):
        If inputs:
            for key, value in inputs.items():
                if hasattr(self.state, key):
                    setattr(self.state, key, value)

        return super().kickoff_async()


def kickoff():
    poem_flow = PoemFlow()
    poem_flow.kickoff()


def plot():
    poem_flow = PoemFlow()
    poem_flow.plot()


if __name__ == "__main__":
    kickoff()
```

## 4.2. crew.py

```
from crewai import Agent, Crew, Process, Task
from crewai.project import CrewBase, agent, crew, task

# If you want to run a snippet of code before or after the crew starts,
# you can use the @before_kickoff and @after_kickoff decorators
# https://docs.crewai.com/concepts/crews#example-crew-class-with-decorators

@CrewBase
class BusinessCrew():
    "BusinessCrew crew"

    # Learn more about YAML configuration files here:
    # Agents: https://docs.crewai.com/concepts/agents#yaml-configuration-
recommended
    # Tasks: https://docs.crewai.com/concepts/tasks#yaml-configuration-
recommended
    agents_config = 'config/agents.yaml'
    tasks_config = 'config/tasks.yaml'

    # If you would like to add tools to your agents, you can learn more about
it here:
    # https://docs.crewai.com/concepts/agents#agent-tools
    # tools_config = 'config/tools.yaml' # Uncomment this line to add tools to
your agents
    @agent
    def market_researcher(self) -> Agent:
        return Agent(
            config=self.agents_config['market_researcher'],
            verbose=True
        )

    @agent
    def financial_analyst(self) -> Agent:
        return Agent(
            config=self.agents_config['financial_analyst'],
            verbose=True
        )

    @agent
    def strategy_consultant(self) -> Agent:
        return Agent(
            config=self.agents_config['strategy_consultant'],
            verbose=True
        )
```

```python
    @agent
    def marketing_specialist(self) -> Agent:
        return Agent(
            config=self.agents_config['marketing_specialist'],
            verbose=True
        )

    @agent
    def operations_analyst(self) -> Agent:
        return Agent(
            config=self.agents_config['operations_analyst'],
            verbose=True
        )

    @task
    def market_research_task(self) -> Task:
        return Task(
            config=self.tasks_config['market_research_task'],
        )

    @task
    def financial_analysis_task(self) -> Task:
        return Task(
            config=self.tasks_config['financial_analysis_task'],
        )

    @task
    def strategy_planning_task(self) -> Task:
        return Task(
            config=self.tasks_config['strategy_planning_task'],
        )

    @task
    def marketing_plan_task(self) -> Task:
        return Task(
            config=self.tasks_config['marketing_plan_task'],
        )

    @task
    def operations_blueprint_task(self) -> Task:
        return Task(
            config=self.tasks_config['operations_blueprint_task'],
        )
    @crew
    def crew(self) -> Crew:
        "Creates the BusinessCrew crew"
```

```
        # To learn how to add knowledge sources to your crew, check out the
documentation:
        # https://docs.crewai.com/concepts/knowledge#what-is-knowledge

        return Crew(
            agents=self.agents, # Automatically created by the @agent
decorator
            tasks=self.tasks, # Automatically created by the @task decorator
            process = Process.sequential,
            verbose=True,
            # process=Process.hierarchical, # In case you wanna use that
instead https://docs.crewai.com/how-to/Hierarchical/
        )
```

## 4.3. config.yaml

```yaml
market_research_task:
  description: >
    {prompt}
    Use {context} to guide your market research.
  expected_output: >
    A list of 10 bullet points summarizing the most relevant market insights
based on the prompt and context.
  agent: market_researcher

financial_analysis_task:
  description: >
    {prompt}
    Use {context} to guide your financial analysis.
  expected_output: >
    A list of 10 bullet points detailing financial projections, assumptions,
and risks based on the prompt and context.
  agent: financial_analyst

strategy_planning_task:
  description: >
    {prompt}
    Use {context} to synthesize market and financial inputs into a strategic
roadmap.
  expected_output: >
    A markdown-formatted plan with objectives, prioritization framework,
recommended initiatives, and timelines derived from the prompt and context.
  agent: strategy_consultant

marketing_plan_task:
```

```
  description: >
    {prompt}
    Use {context} to develop a go-to-market marketing plan.
  expected_output: >
    A markdown-formatted outline of target personas, messaging, channel mix,
campaign milestones, and success metrics based on the prompt and context.
  agent: marketing_specialist

operations_blueprint_task:
  description: >
    {prompt}
    Use {context} to map out operational requirements and dependencies.
  expected_output: >
    A markdown-formatted blueprint detailing workflows, staffing needs,
technology stack, and timeline dependencies based on the prompt and context.
  agent: operations_analyst
```

## 4.4. tool.py

```
import requests
import json
from typing import Type
from crewai.tools import BaseTool
from pydantic import BaseModel, Field

class GetEventsToolInput(BaseModel):
    "Input schema for GetEventsTool."""
    session_token: str = Field(..., description="Token of session for the
currently logged-in user.")
    organization_id: int = Field(..., description="ID of the organization.")
    start: str = Field(..., description="Start date of the event period in
YYYY-MM-DD format.")
    end: str = Field(..., description="End date of the event period in YYYY-
MM-DD format.")

class GetEventsTool(BaseTool):
    name: str = "Personal Calendar Events Tool"
    description: str = "Fetches calendar events for a user in a specific
organization within a given time range."
    args_schema: Type[BaseModel] = GetEventsToolInput

    def _run(self, session_token: str, organization_id: int, start: str, end:
str) -> str:
        api_url = f"http://Ordolio-web-
1:8000/api/v2/{organization_id}/events/my-calendar/"
        headers = {
            "Authorization": f"Bearer {session_token}"
```

```
        }
        params = {
            "start": start,
            "end": end
        }

        try:
            response = requests.get(api_url, headers=headers, params=params)
            response.raise_for_status()
            events = response.json()
            print(f"events: {events}")
            return json.dumps(events, indent=2)
        except requests.exceptions.RequestException as req_err:
            return f"Request error occurred: {req_err}"
        except Exception as e:
            return f"An error occurred: {e}"
```

## 4.5. endpoints.py

```
from fastapi import FastAPI, HTTPException
from pydantic import BaseModel
import json
from flow_Ordolio_ai.main import PoemFlow  # Import your PoemFlow class

app = FastAPI()

# Pydantic model for request validation
class CrewAIRequest(BaseModel):
    prompt: str
    context: str
    token: str
    current_org: str

@app.post("/generate")
async def generate_response(data: CrewAIRequest):
    """
    Endpoint to generate an AI response.
    Expects a JSON payload with "prompt" and "context".
    """
    try:
        print(f"🔍 Crew AI received prompt: {data.prompt} with context:
{data.context} and token {data.token} for organization {data.current_org}")
        # Create a fresh instance of PoemFlow for this request
        bot = PoemFlow()
        # Generate the AI response asynchronously
```

```
        bot_response = await bot.kickoff_async(inputs={"prompt": data.prompt,
"context": data.context, "access_token": data. token, "current_org":
data.current_org})
        print(f"🤖 Crew AI response: {bot_response}")
        return {"response": bot_response}
    except Exception as e:
        print("error in endpoints")
        raise HTTPException(status_code=500, detail=str(e))

if __name__ == "__main__":
    import uvicorn
    print("🚀 Starting Crew AI Service...")
    uvicorn.run("endpoints:app", host="0.0.0.0", port=5000, reload=True)
```

## 4.6. Upload.py

```
#!/usr/bin/env python3
"""
vectorize_events.py

A script to embed events from a JSON file using OpenAI embeddings and upload
them to a Qdrant collection.
"""
import os
import json
import logging
from tqdm import tqdm
from openai import OpenAI
from qdrant_client import QdrantClient
from qdrant_client.models import VectorParams, Distance, PointStruct
from dotenv import load_dotenv

# Load environment variables from .env file
load_dotenv()

# Determine script directory for relative paths
SCRIPT_DIR = os.path.dirname(os.path.abspath(__file__))

# Configure logging
def configure_logging():
    logging.basicConfig(
        level=logging.INFO,
        format="%(asctime)s [%(levelname)s] %(message)s",
        datefmt="%Y-%m-%d %H:%M:%S"
    )
```

```
# Load events from a JSON file
def load_events(path: str) -> list:
    with open(path, "r", encoding="utf-8") as f:
        return json.load(f)

# Build input text for embedding
def build_embedding_input(event: dict) -> str:
    def grab(field, fallback=""):
        return field if field else fallback

    # Name
    name_map = event.get("name") or {}
    name = grab(name_map.get("nl")) or next(iter(name_map.values()), "")

    # Description
    desc_map = event.get("description") or {}
    description = grab(desc_map.get("nl")) or next(iter(desc_map.values()),
"")

    # Location (only city name)
    location = event.get("location", {}).get("city", "")

    # Build the embedding input string
    parts = [
        f"Name: {name}",
        f"Description: {description}",
        f"Location: {location}"
    ]

    return "\n".join(parts)

# Main routine
def main():
    configure_logging()

    # Initialize OpenAI and Qdrant clients
    openai_client = OpenAI(api_key=os.getenv("OPENAI_API_KEY"))
    qdrant_url = os.getenv("QDRANT_URL")
    qdrant_api_key = os.getenv("QDRANT_API_KEY")
    collection_name = os.getenv("QDRANT_COLLECTION", "events_2025")
    default_file = os.path.join(SCRIPT_DIR, "events_2025.json")
    input_file = os.getenv("INPUT_FILE", default_file)
    vector_dim = int(os.getenv("VECTOR_DIM", 1536))

    # Check required settings
    missing = []
```

```
    if not os.getenv("OPENAI_API_KEY"):
        missing.append("OPENAI_API_KEY")
    if not qdrant_url:
        missing.append("QDRANT_URL")
    if missing:
        logging.error(f"Missing environment variable(s): {',
'.join(missing)}")
        return

    client = QdrantClient(url=qdrant_url, api_key=qdrant_api_key)

    # Verify events file exists
    if not os.path.exists(input_file):
        logging.error(f"Events file not found at {input_file}")
        return

    # Load events
    events = load_events(input_file)
    logging.info(f"Loaded {len(events)} events from {input_file}")

    # Generate embeddings and collect points
    points = []
    for ev in tqdm(events, desc="Embedding events"):
        text = build_embedding_input(ev)
        try:
            resp = openai_client.embeddings.create(input=text, model="text-
embedding-3-small")
            vector = resp.data[0].embedding
            points.append(PointStruct(id=ev.get("id"), vector=vector,
payload={"metadata": ev}))
        except Exception as e:
            logging.warning(f"Failed to embed event {ev.get('id')}: {e}")

    # Ensure or create collection
    existing = [c.name for c in client.get_collections().collections]
    if collection_name not in existing:
        logging.info(f"Creating collection '{collection_name}'...")
        client.recreate_collection(
            collection_name=collection_name,
            vectors_config=VectorParams(size=vector_dim,
distance=Distance.COSINE)
        )
    else:
        logging.info(f"Collection '{collection_name}' already exists")

    # Upload vectors
    if points:
```

```
        logging.info(f"Uploading {len(points)} points to
'{collection_name}'...")
        BATCH_SIZE = 50  # You can tune this

        for i in range(0, len(points), BATCH_SIZE):
            batch = points[i:i + BATCH_SIZE]
            logging.info(f"Uploading batch {i // BATCH_SIZE + 1} ({len(batch)}
points)...")
            try:
                client.upsert(collection_name=collection_name, points=batch)
            except Exception as e:
                logging.error(f"Failed to upload batch {i // BATCH_SIZE + 1}:
{e}")

        logging.info("Upload complete")
    else:
        logging.warning("No points to upload.")

if __name__ == "__main__":
    main()
```

## 4.7. Api.py

```
import os
import json
import uuid
import requests
import redis
from fastapi import FastAPI, HTTPException
from pydantic import BaseModel, field_validator

app = FastAPI()
from fastapi.middleware.cors import CORSMiddleware

app.add_middleware(
    CORSMiddleware,
    allow_origins=["http://localhost:8000"],  # Frontend domain
    allow_credentials=True,
    allow_methods=["*"],
    allow_headers=["*"],
)
# --- Redis Setup ---
# Use environment variables so the host/port can be configured via docker-
compose.
REDIS_HOST = os.getenv("REDIS_HOST", "redis")
REDIS_PORT = int(os.getenv("REDIS_PORT", "6379"))
```

```
redis_client = redis.StrictRedis(host=REDIS_HOST, port=REDIS_PORT, db=0,
decode_responses=True)

try:
    redis_client.ping()
    print("☑ Redis is running!")
except redis.ConnectionError:
    print("❌ Could not connect to Redis. Make sure it's running.")

# --- Crew AI Service Configuration ---
# The API calls the Crew AI service via its HTTP endpoint.
CREW_AI_HOST = os.getenv("CREW_AI_HOST", "crew_ai")
CREW_AI_PORT = os.getenv("CREW_AI_PORT", "5000")
CREW_AI_URL = f"http://{CREW_AI_HOST}:{CREW_AI_PORT}/generate"

# --- Helper Functions for Conversation Storage ---
def save_conversation(conversation_id: str, data: dict):
    redis_client.set(conversation_id, json.dumps(data), ex=1800)  # Expire
after 30 minutes

def get_conversation(conversation_id: str):
    data = redis_client.get(conversation_id)
    return json.loads(data) if data else None

# --- Pydantic Model for Request Validation ---
class EndConversationRequest(BaseModel):
    conversation_id: str
    @field_validator("conversation_id")
    def not_empty(cls, value):
        if not value.strip():
            raise ValueError("Conversation ID cannot be empty")
        return value
class MessageRequest(BaseModel):
    conversation_id: str
    message: str
    access_token: str
    current_organization_id: str

    @field_validator("conversation_id", "message", "access_token",
"current_organization_id")
    def not_empty(cls, value):
        if not value.strip():
            raise ValueError("Fields cannot be empty")
        return value

# --- API Endpoints ---
```

```python
@app.post("/create_conversation")
async def create_conversation():
    """
    Starts a new conversation and returns a unique conversation ID.
    """
    conversation_id = str(uuid.uuid4())
    conversation_data = {"messages": [], "active": True}
    save_conversation(conversation_id, conversation_data)
    return {"conversation_id": conversation_id, "message": "Conversation
created"}

@app.post("/generate_response")
async def generate_response(data: MessageRequest):
    """
    Appends the user message, builds context from the last 20 messages,
    then calls the Crew AI service to generate a response.
    """
    conversation_id, user_message, token, current_org = data.conversation_id,
data.message, data.access_token, data.current_organization_id

    conversation_data = get_conversation(conversation_id)
    if not conversation_data:
        raise HTTPException(status_code=404, detail="Conversation not found or
expired")
    if not conversation_data.get("active", False):
        raise HTTPException(status_code=400, detail="Conversation has ended")

    # Append the user message to the conversation history.
    conversation_data["messages"].append({"role": "user", "text":
user_message})
    conversation_history = conversation_data["messages"][-20:]
    context_messages = "\n".join(f"{m['role']}: {m['text']}" for m in
conversation_history)

    # Prepare payload for the Crew AI service.
    payload = {"prompt": user_message, "context": context_messages, "token":
token, "current_org": current_org}

    try:
        response = requests.post(CREW_AI_URL, json=payload)
        response.raise_for_status()
        response_data = response.json()
    except Exception as e:
        print("error in main.py")
        print(e)
        raise HTTPException(status_code=500, detail=f"Error calling Crew AI
service: {str(e)}")
```

```
    bot_response = response_data.get("response", "")
    conversation_data["messages"].append({"role": "bot", "text":
bot_response})
    save_conversation(conversation_id, conversation_data)
    print(f"🤖 Crew AI response: {bot_response}")

    suggested_prompts = response_data.get("suggested_prompts", [])

    if not isinstance(suggested_prompts, list):
        try:
            import json
            suggested_prompts = json.loads(suggested_prompts)
        except (ValueError, TypeError):
            suggested_prompts = [suggested_prompts]
    return {
        "conversation_id": conversation_id,
        "response": bot_response,
        "suggested_prompts": suggested_prompts
    }

@app.post("/end_conversation")
async def end_conversation(data: EndConversationRequest):
    """
    Marks a conversation as ended.
    """
    conversation_id = data.conversation_id
    conversation_data = get_conversation(conversation_id)
    if not conversation_data:
        raise HTTPException(status_code=404, detail="Conversation not found or
expired")

    conversation_data["active"] = False
    save_conversation(conversation_id, conversation_data)
    return {"message": "Conversation ended", "conversation_id":
conversation_id}

@app.get("/get_conversation/{conversation_id}")
async def get_conversation_api(conversation_id: str):
    """
    Retrieves conversation details by conversation ID.
    """
    conversation_data = get_conversation(conversation_id)
    if not conversation_data:
        raise HTTPException(status_code=404, detail="Conversation not found or
expired")
```

```
    return {"conversation_id": conversation_id, "conversation":
conversation_data}

# --- Run the API Service ---
if __name__ == "__main__":
    import uvicorn
    print("🚀 Starting API Service...")
    uvicorn.run("main:app", host="0.0.0.0", port=8000, reload=True)
```