“NoiPA Decoded: Agentic Intelligence with CrewAI”

Reply

- Di Maio Maria Vittoria (792171) maria.dimaio@studenti.luiss.it
- Montesi Maria Elisa (793891) maria.montesi@studenti.luiss.it
- Traversa Alessia (800131) alessia.traversa@luiss.it

1. Introduction - Multi-agent framework for dataset handling

Our project aimed at creating a multi-agent framework to handle four datasets for the NoiPA portal. The guidelines specifically requested three agents: one to analyze data, one to create visualizations, and one to talk to users. Following an initial Exploratory Data Analysis (EDA) phase on the provided datasets, several key insights emerged that guided our development. Notably, we found that SPID (Sistema Pubblico di Identità Digitale) was the most predominantly used method for accessing the NoiPA portal across the user base. Concurrently, analysis of payment preferences indicated that direct crediting to a bank account ('carta/conto corrente') was the most common method for salary disbursement. Recognizing the specific context of the NoiPA portal, which serves Italian public administration employees, a crucial first design decision was made: we would have created agents that would have worked well with both English and Italian queries.

To orchestrate the workflow between these specialized agents, we implemented a sequential crew architecture using the CrewAI library (Section 2). This crew was designed to process user queries in a step-by-step manner. The first agent in the sequence is the DataAnalystAgent, responsible for receiving the user's query, performing in-depth analysis of the relevant NoiPA datasets using its programmatic data access tools, and preparing both a textual summary of its findings and machine-readable data. Following the Analyst, the DataVisualizerAgent takes the processed data and, if a visualization is requested or deemed appropriate, generates and executes the Python code necessary to create a visual representation (like a bar chart or line graph), along with metadata about the plot. Finally, the ReporterAgent acts as the concluding agent; it receives the textual analysis from the DataAnalystAgent and the visualization blueprint (in JSON format) from the DataVisualizerAgent. The ReporterAgent then utilizes a specialized tool to render both the textual insights and the generated visualization directly within a Streamlit application, providing a final, user-facing consolidated report.

2. Methods

2.1 Framework: CrewAI

To create our multi agent architecture for the project we decided to take an experimental route using CrewAI. CrewAI is a new library created specifically for the

construction of agents. We decided it was a good fit for us due to its modularity and its intuitiveness. Furthermore, this framework heavily relies on the power of prompting and minimizes the actual coding that would go into such a project, making it accessible and implementable even without extensive machine learning and coding knowledge.

2.2 Pre-processing and Architecture

Before designing the architecture, we conducted an exploratory data analysis to gain a thorough understanding of the provided datasets. This step served two main purposes: to organize the data into clear and accessible formats for our agents, and to equip ourselves with deeper insights into the data, enabling more effective and targeted prompting during the construction phase. An example of data preprocessing we did was standardizing the column names across the four datasets, assigning the same names to columns that contained identical information. Furthermore, because the datasets included range-based information with significant missing data—particularly where values were grouped rather than tied to individual employees—we created a new column to explicitly represent these ranges. We assigned a value of zero to indicate the lower bound of each range and left the upper bound blank to reflect maximum values. More detailed information can be found in our directory, specifically in the path “/data/cleaning code and EDA”. Although some preprocessing was performed to address these issues, the agents overall received largely unstructured and mostly raw data.

At the outset, we believed a hierarchical approach would yield the best results for the architecture of our multi-agent system. The core idea was to have one agent act as a manager, overseeing the workflow and coordinating the tasks of the other two agents. After some experimenting, although the idea logically made sense, we observed that in reality this formation was not the most adequate one for the task at hand. Thus, we decided to switch to a sequential architecture, in that it would have established a cleaner and more efficient workflow amongst the agents, resulting in an overall better performance.

2.3 Description of Agents

The final sequential architecture begins with the DataAnalystAgent, which is equipped with a customizable data analysis tool provided by Crew AI. This tool grants access to dataset directories and essential Python libraries, enabling the agent to perform dynamic, context-aware data exploration and processing. The primary responsibility of the DataAnalystAgent is to interpret user queries and extract meaningful insights from the

diverse NoiPA datasets, which include information on portal access, income brackets, commuting patterns, and salary payments.

Upon receiving a user query, the agent identifies the relevant data sources and applies necessary preprocessing operations—such as filtering, aggregation, and computation—using Python, primarily through the Pandas library. The agent's task, defined as create_analyst_task, involves generating a comprehensive textual summary of its findings that is accessible and informative to the end user. In addition to this narrative output, the agent also produces a structured, machine-readable version of the processed data (typically in CSV string format), which is forwarded to the subsequent agent in the system. Crucially, the DataAnalystAgent is designed to handle incomplete or ambiguous data with robustness. When exact answers are not possible, it provides the closest relevant analysis and explicitly communicates any limitations or uncertainties present in the datasets.

Following the DataAnalystAgent in the sequential pipeline is the DataVisualizerAgent, which assumes the role of a Visualization Code Architect. Its principal function lies in transforming the structured data produced by the analyst into a coherent and informative visual representation. Unlike the analyst agent, the DataVisualizerAgent does not possess the capability to execute code. Instead, its purpose is centered on generating executable Python code—primarily utilizing libraries such as Matplotlib and Seaborn—which can be run by subsequent system components to produce visual output. This functionality is operationalized through the create_visualization_code_task, which instructs the agent to construct a JSON-formatted visualization blueprint that encapsulates all components necessary for the generation of a plot. Within this blueprint, the agent includes a Python code snippet that, when executed, renders the intended visual representation of the data. Alongside this, the agent incorporates the specific subset of structured data that the visualization is based upon, ensuring clarity and precision in what is to be represented. Additionally, the blueprint contains a complete set of configuration parameters—such as plot titles, axis labels, legends, and other stylistic metadata—that guide the rendering process and contribute to the interpretability and aesthetic quality of the final visualization. In addition to producing this structured output, the visual content—once executed by a downstream component—is saved to a designated directory (plots/) to ensure persistence and traceability. The use of a standardized JSON format not only promotes consistency across the pipeline but also facilitates smooth integration with the final reporting module, which consumes this blueprint for rendering.

An important aspect of the DataVisualizerAgent's design is its capacity to handle imperfect correspondence between user queries and the available data. In such cases, the agent is

instructed to generate the most relevant and contextually appropriate visualization possible, while transparently documenting any necessary adjustments or interpretative choices within the description field of the JSON object. This ensures that the visual output remains both meaningful and faithful to the underlying data constraints.

Concluding the sequential workflow is the ReporterAgent, which operates in the capacity of the Chief Communications Officer and Streamlit Report Publisher. Its primary role is to synthesize the outputs generated by both the DataAnalystAgent and the DataVisualizerAgent, integrating them into a cohesive and interactive report presented through a Streamlit application. Acting as the final point of interaction in the multi-agent system, the ReporterAgent is tasked with translating analytical results and visual components into a user-facing format that is both accessible and informative.

To fulfill this role, the ReporterAgent receives two key inputs: a textual summary containing the analytical findings from the DataAnalystAgent, and a structured JSON blueprint from the DataVisualizerAgent, which includes both the Python code necessary to generate the visualizations and the data that those visualizations are based on. The final step in the ReporterAgent's task is to issue a status message summarizing the outcome of the reporting process, confirming the successful generation of the Streamlit report or, if necessary, detailing any errors encountered during execution. In this way, the ReporterAgent serves as both the culmination of the system's analytical pipeline and the bridge between technical output and user experience.

2.4 Memory

In an effort to enhance contextual understanding and potentially improve performance across more complex or multi-turn interactions, we also explored the implementation of memory capabilities within our CrewAI framework. CrewAI provides mechanisms for equipping agents with memory, allowing them to retain information from previous steps or even past interactions. We experimented with basic sequential memory where the context from one task is passed to the next, which was fundamental to our sequential crew's operation. However, when investigating more advanced memory types, such as those designed for long-term persistence or semantic recall using embeddings (e.g., MemoryRAG or integrations with vector stores like ChromaDB), we encountered a practical constraint. These sophisticated memory solutions typically rely on generating embeddings from text, a process that often requires API calls to paid embedding model services (such as OpenAI's Ada or Cohere). Given our project's reliance on the free-tier gemini-1.5-flash model and the associated API access, which primarily focuses on text generation rather than

providing free embedding endpoints, we did not fully implement these advanced, persistent memory features.

As a result, we wrote code to implement short-term memory leveraging OpenAI's embedding models, but we commented it out. It can only work with a paid API key.

3. Experimental Design

(The complete evaluation score' measurements are in the "Agents Evaluation" spreadsheet in the directory)

In order to evaluate our architecture, we designed a query score (0 to 5). This score was based on analysis (0 to 3) and visualization (0 to 2). In particular, the analysis score was divided in: 0 for not working, 1 for showing reasoning but no satisfactory answer, 2 for showing a correct simple answer, 3 for an in-depth correct answer. The visualizer instead was assigned 1 for correctly generating a plot, 2 for extremely accurate plots.

3.1 English and Italian queries

We measured this score for both English and Italian queries. As already mentioned, we decided to create agents that could work with Italian as well as English, since the real use-case application of our fleet would have been for the NoiPA portal (a portal that manages Italian public administration's employees). We used the English queries' performances as baseline reference, and then evaluated the Italian queries' metrics based on those results.

In general, the average score for Italian queries was around 2.96 (as is shown on the "Agents Evaluation" spreadsheet), whereas the English queries averaged 2.85. Overall, we deemed this a significant indication that our agents were achieving moderate results in both languages, as we expected.

3.2 Timing of response

We also measured responses' timing to be able to better assess the system's performance. On average, the response time was 38 seconds, which could be considered moderately good since our agents were tasked with processing not only the queries, but also extremely specific and accurate prompts.

Moreover, it is to be noted that our agents were powered by the gemini-1.5-flash model, which is freely available with a google developers' API key. Consequently, we are assuming response timing and accuracy could improve if our system was linked to a more powerful LLM.

The slightly higher average score for Italian queries, while a positive indicator of the system's adaptation to the target language, warrants further investigation. This could be attributed to the direct interpretation of domain-specific terms when processed in their native language. Future work could involve a more granular error analysis for both languages to pinpoint specific areas of strength or weakness, such as the complexity of query that each language handling excels at, or if certain types of analytical tasks (e.g., correlation vs. distribution) show language-dependent performance. Furthermore, the 38-second average response time, while acceptable for some use-cases, could be a bottleneck for real-time interactive applications. Optimizing prompt structures, exploring more efficient data parsing within tools, or investigating caching strategies for frequently accessed data transformations could be avenues to reduce this latency, even before considering an upgrade to a more computationally intensive LLM, which itself would likely impact both cost and speed.

4. Results

4.1 Architectural choices

Our exploration and implementation of a multi-agent system using the CrewAI framework for analyzing the NoiPA datasets yielded several noteworthy results, particularly concerning architectural choices and linguistic performance. The experimental results revealed that our adopted sequential architecture consistently outperformed an initially trialed hierarchical alternative. This suggests that for this specific task and dataset, maintaining a natural, linear flow of information from data analysis through visualization to reporting played a crucial role in the model's ability to accurately process and understand the input data. Unlike hierarchical models, which attempt to break down and interpret data at multiple levels of abstraction, the sequential approach appeared better suited to capturing the contextual dependencies inherent in the query processing workflow, reinforcing the importance of architectural choice in designing systems tailored to the structure and nature of the data.

4.2 Efficient prompting

A critical factor in the performance of our agents, regardless of architecture, was the efficacy of our prompting strategies. These heavily relied on establishing a clear chain-of-thought for each agent and ensuring their prompts facilitated an accurate interpretation of the provided NoiPA data. We discovered that detailed, step-by-step instructions within the agent goals and task descriptions were paramount for guiding the LLM's reasoning process. This was especially true for the DataAnalystAgent, which needed to navigate multiple datasets and perform specific calculations. The initial EDA was invaluable here, as it allowed us to craft prompts that nudged the agent towards correct data linkages and analytical methods.

A key decision in our data handling approach was to provide the agents, particularly the DataAnalystAgent, with access to the raw datasets with only minimally processed columns (such as standardized naming and range clarifications). We hypothesized that allowing the agent to perform its own dynamic data loading and manipulation via its Python execution tool would lead to more flexible and context-aware analysis. This approach proved successful; the DataAnalystAgent was generally able to identify relevant files and columns and perform the necessary operations based on the user's query, demonstrating the LLM's capability to reason over and interact with data programmatically.

4.3 Linguistic trend

Furthermore, an interesting linguistic trend emerged from our evaluations: queries formulated in Italian showed slightly better performance compared to their English counterparts. While the difference was not drastic (average scores of approximately 2.96 for Italian versus 2.85 for English), it was consistent enough to call our attention. This disparity may be explained by the directness with which Italian domain-specific terms (like those relevant to NoiPA) are interpreted. This factor could have contributed to a model more finely tuned to understanding and generating responses in Italian.

4.4 CrewAI evaluation

Finally, regarding the framework itself, we found CrewAI to be a fairly new yet remarkably solid platform for agentic AI development. Its modular design and intuitive approach to defining agents, tasks, and tools made it accessible and allowed for rapid prototyping. While it is still an evolving library, its current capabilities show significant promise for future applications. However, while CrewAI offers straightforward mechanisms

for implementing basic memory, we noted a practical limitation in our setup. The more advanced memory features, particularly those relying on embeddings for semantic similarity, typically require integration with embedding models that often necessitate paid API access. Our project, leveraging the free-tier gemini-1.5-flash model, meant that we relied more on explicit context passing between sequential tasks rather than a sophisticated, persistent memory store.

5. Conclusions

Our project successfully demonstrated the viability of constructing a multi-agent system using the CrewAI framework to analyze and visualize complex, real-world datasets like those from the NoiPA portal. The sequential architecture, coupled with carefully engineered prompts, enabled our agents—the DataAnalystAgent, DataVisualizerAgent, and ReporterAgent—to process user queries in both English and Italian, deliver analytical insights, and prepare visualizations for a Streamlit-based user interface. The primary takeaway from this endeavor is the promising potential of CrewAI as an accessible and modular framework for developing sophisticated agentic workflows; while its core functionalities are robust and intuitive, the practical implementation of advanced, persistent memory systems currently presents a hurdle when relying on free-tier LLMs and embedding models, suggesting an area for future improvement or alternative strategies within such constraints.

Despite these achievements, our exploration leaves some questions unanswered and avenues for future research open. While the 38-second average response time was deemed acceptable given the free-tier model and prompt complexity, optimizing this for near real-time interaction remains a significant challenge. It would be valuable to investigate the performance trade-offs of integrating more powerful (and potentially costly) LLMs and embedding models, not only for accuracy and speed but also for enabling more sophisticated memory solutions. Finally, exploring adaptive error handling within the agents, where they might attempt self-correction or request clarification upon encountering data inconsistencies or ambiguous instructions, could further enhance the system's robustness and user experience.