GenAI and financial data alert systems

GenAI Insight Overlays

Candidate Number: KYHL7[1]

MSc. Computer Science

Internal Supervisor: Prof. Lewis D. Griffin

Submission date: 8 September 2025

[1]**Disclaimer:** This report is submitted as part requirement for the MSc. Computer Science at UCL. It is substantially the result of my own work except where explicitly indicated in the text. The report may be freely copied and distributed provided the source is explicitly acknowledged.

Abstract

TODO: Summarise your report concisely.

Contents

Chapter 1

Introduction

According to a study by Forrester Consulting, knowledge workers spend 30% of their time — that is, 2.4 hours a day — searching for information within their own company[4]. This is a significant amount of time that could be better spent making decisions and taking action. As such, the aim of this project is to explore the capabilities of Large Language Models ("LLMs") to access a company's data, extract operational figures and provide initial insights to managers, directly to their inbox, saving time from the operational task of data gathering, and allowing them to focus on the more impactful parts of their roles — making data-informed decisions.

To perform this exploration, the goal is to creating a proof of concept system that acts as a junior analyst (or "AI Analyst") who autonomously generates reports about a specific Key Performance Index (KPI), at set time intervals, based on the company's data. In addition to the AI Agent, the system will have a simple user interface, where the user can set the KPI(s) to report on, and the interval at which their analyst should run.

For this, the project builds on the recent evolution of AI agents and agentic systems, as paradigms that allow the execution of complex, multi-step workflows autonomously. Nevertheless, this specific use case poses a few challenges to the current stat of the art in GenAI agents.

For these reports, the AI Analyst must go beyond simply retrieving high-level KPIs. Instead, it must also provide granular details that uncover deeper operational insights. For example, rather than merely reporting that sales are down by 10%, the Analyst should also identify which products or regions are driving this decline.

This type of work is inherently open-ended. There is no single "correct" answer, or even path to an answer, nor a clear point to stop, and the AI Analyst must adapt its approach depending on its own intermediate findings. Crucially, for the reports to be truly useful, the agent must not just report the data, but drawing causal inferences that can point the manager in the right direction to solve any issues found, something that remains a known challenge for LLMs [21].

Moreover, this kind of internal analysis is not likely to be included in LLM's training sets. Companies rarely publish their internal data or reports, and the requirements of operational reports are more granular than financial reports, which might be available for public companies. This makes the task a strong candidate for fine-tuning, but due to the lack of suitable training data this is outside of the project scope.

With all of this in mind, the project was executed in three consecutive stages: First, a Research stage, focused on understanding the state of the art in AI agent orchestration and prompting, as well as the software tools available; the results of this stage can be found in Chapter 2. Second, a Requirements and Analysis stage, where together with the client, Alvarez & Marsal, we defined the scope of the proof of concept; the results of this stage can be found in Chapter 3. This was also the bases for the evaluation framework for the project, which is described in Chapter 5. Third, a Design and Implementation stage, which was performed iteratively; the details of this stage can be found in Chapter 4. Finally, Chapter 6 contains the Reflections from the process of creating the proof of concept system, as well as the potential future work to create a production-ready system.

Chapter 2

Research

Research for this project was be split in two main lines. First, understanding the fundamentals of building applications with Large Language Models. For this, the project was largely influenced by Chip Huyen's book, AI Engineering, which "provides a framework for adapting foundation models, which include both large language models (LLMs) and large multimodal models (LMMs), to specific applications"[8]. In particular, a few learnings from the book were crucial in the development stage, which give structure to Section 2.1:

1. Its highlight of the importance, and difficulty, of evaluating AI systems
2. Its guideline of Prompt Engineering best practices
3. Its guideline of Retrieval Augmented Generation
4. Its overview of AI Agents

That said, this chapter does not intend to replicate the book's content, instead it will describe the most important learnings grasped from it, and add the learnings from other relevant information sources.

Afterwards, Section 2.2 will describe the research performed to define the technology stack used to develop the proof of concept, which per the client's request was built in a python environment.

2.1 Large Language Models and Agents Learnings

2.1.1 AI System Evaluation

In AI Engineering, Chip Huyen highlights two main reasons why evaluating the outputs of LLMs is harder than evaluating traditional machine learning models.

First, as tasks become more sophisticated, the more time it takes to evaluate them. As she puts it, "[y]ou can no longer evaluate a response based on how it sounds. You'll also need to fact-check, reason, and even incorporate domain expertise". Secondly, the open-ended and probabilistic nature of LLMs "undermines the traditional approach of evaluating a model against ground truths". There are too many correct responses, so it is impossible to have a comprehensive list of correct outputs as ground truth.

These problems are exacerbated when it comes to AI Agents, as they tackle more complex tasks. This is an on-going research field but, according to [10], the focus of evaluation so far has been “predominantly focused on accuracy metrics that measure task completion success”. This focus has been criticised for many reasons, particularly for limiting the evaluation to a task being completed or not, and not included other metrics such as the efficiency of the completion.

This type of measure is called “exact evaluation”, because it has no ambiguity — a task is complete or it is not — and can be viewed more generally to include any type of rules-based evaluation where, as the name indicates, a set or rules are defined and the output is measured against it adherence to these rules.

The alternatives to this type of evaluation are what has been called “objective evaluations”, where an external validator that reviews the agent’s output and “judges” it, assigning it a score based on some type of defined criteria — for example completeness or clarity. So far, studies such as [22] have found that this judgment can be performed by either LLMs (LLM as a judge) or Humans (Human as a judge), with high correlations in the scores given. Nevertheless, [7] has also found that human’s personal preferences can affect the score given to an output.

Moreover, other authors such as [25] criticize any approach for evaluating Agents that relies solely on the final output of their process. They believe that agents shouldn’t only be evaluated on their final output but on the intermediate steps taken, and thus propose agent as a judge -a methodology that uses agents to enable intermediate feedback- as a better alternative.

Evidently, despite the lack of a clear “best” strategy, evaluation remains a key ingredient in creating AI systems to ensure their reliability. The AI Engineering book goes as far as proposing a development methodology called Evaluation Driven Design (EDD). Similarly to Test Driven Design, in EDD the evaluation criteria for the application are defined before building the application, allowing safe iterations. Moreover, her recommendations align with [25] in the need to not only evaluate the final output, but also every component of the system. As will be discussed in detail in Chapter 5, this project followed the book’s recommendations.

But how do you define said evaluation criteria? Huyen proposes thinking in three categories which, despite being targeted for evaluating LLMs and not agents, can be adapted as follows:

- Domain-Specific Capacity: Evaluate the ability of the agent to perform tasks specific to the problem domain, such as coding or finance.
- Generation Capacity: Evaluate the quality of the text being generated, including the factual consistency of the outputs (that is, that the facts in the response are correct).
- Instruction-Following Capacity: Evaluate how well the specific instructions of a prompt are followed.

Choosing the Right Model

The difficulty in evaluation of LLMs and Agents highlights one specific buy key issue — how do you choose the right model, or at the very least model family, to power an agent?

The literature points towards using specialized model for tasks that require domain specific capabilities, such as data analysis, and more general models for task that are common in the general training data, such as coding, tool calling or general summarization (e.g., writing the final report). Studies such as [9] and [19] have shown that smaller models specifically trained for financial tasks outperform larger models in financial benchmarks.

Nevertheless, [14] created a financial benchmark that includes finance specific tasks such as forecasting market trends, predict asset movements, or analysing causal relationships in financial events, which closely resemble the types of task required in this project. Their evaluations of general models such as OpenAI's GTP-4o or o4-mini produced strong results (above 80%) for these tasks, and as-such, these are used for development in this project. This is in line with the client's preferences of using widely available models.

2.1.2 Prompt and Context Engineering

According to Chip Huyen, "[a] prompt is an instruction given to a model to perform a task". One of the main reasons to build an evaluation framework is to be able to perform Prompt Engineering, or the process of tweaking prompts and context to get the desired outcome from an LLM. This step is so important, that model providers often provide detailed guides on how to engineer prompts for their specific models, such as [16] from OpenAI, which guided the promp engineering process for this project. Most of guides, nonetheless, include two techniques that have become cornerstones for this process.

First is "few-shot" prompting, introduced by [1]. This paper, by Brown et al., demonstrated that language models can learn a desired behaviour from examples in within the prompt; in more colloquial terms, there is a benefit to providing examples in prompts.

Second, there is "Chain-of-Thought" prompting, introduced by [23]. In this type of prompt, the model is asked to perform a step by step process before providing an output — either steps pre-defined in the prompt or defined by the model during execution. Wei et al. where able to achieve state of the art results using this technique.

Note that these techniques are not mutually exclusive, and prompts for complex tasks might include both of them, or none, depending on the specific model used (e.g., reasoning models might not need explicit chain of thought prompting).

Context Engineering

Borrowing AI Engineering's definition, the context is "the information provided to the model so that it can perform a given task". While the instruction (Prompt) remains static for any query, the information (Context) varies.

One key pattern for context engineering is Retrieval Augmented Generation (RAG), which according to AI Engineering can be considered as "a technique to construct context specific to each query". The pattern of retrieving and then generating was first introduced by Chen et al. in [2], where their system would retrieve Wikipedia pages relevant to a question and add their content to a model's context, and then fully coined by [13].

While these two papers define the technique with the use of text, this pattern can be applied to any type of information. Most relevant to this project, tabular data can also be retrieved and provided as part of the context.

Finally, adding information is not the only way to manage a task's context. In perhaps the most important finding for this project, [20] demonstrates that as the size of the context grows closer to the model's limit, retrieval and reasoning become more biased towards data that is near the end of the context, and the model more easily fails to retrieve information provided in previous parts of the context.

2.1.3 Agent Architectures and Agentic Orchestration

According to [17], "AI Agents can be defined as autonomous software entities engineered for goal-directed task execution within bounded digital environments", usually with an LLM as their "core reasoning component". They distinguish AI Agents from Agentic AI, an "emerging class of systems [that] extends the capabilities of traditional AI Agents by enabling multiple intelligent entities to collaboratively pursue goals through structured communication, shared memory, and dynamic role assignment". This subsection summarizes research findings for state of the art architectures of both.

Agent Architecture

The basic structure of an agent is described in AI Engineering as an LLM, which can perform certain actions (through tools) in a defined environment. The LLM acts as the brain of this agent, processing the information it receives — e.g., the task given by the user or results from its own tools — and generating plans to perform the task.

The plans designed by the LLM can be performed in many ways, for example, tools can be called in parallel or sequentially, or the LLM and tools call be called in a Loop until the LLM itself decides the task has been completed. This designed is generally called the Agent's Architecture.

One of the key problems for any Agent Architecture is how to build in ways to adjust the plans laid out by the LLM, allowing them to correct potential errors. On this, one of the most common patterns is ReAct (Reasoning and Acting), proposed by [24]. In this pattern, two steps are taken in a loop until the task is completed: a Reasoning step, which encompass both planning and reflecting on tool outputs, and an Action step, where the next step in the plan is executed.

This architecture was later expanded upon in [18], in their proposed Reflexion architecture. In Reflexion, the agent has separate steps for evaluating the output of its tools and self-relect on what went wrong, on each loop, after evaluation and reflection, the agent proposes a new plan, and then executes the next step on that plan.

Agentic Orchestration Architecture

As described by [17], Agentic AI allows multiple AI Agents to collaborate for more complex goals. Agentic Orchestration thus refers to the way that the collaboration between agents is structured.

There are numerous ways to structure this collaboration. Some of the best known are, as described as part of LangGraph's documentation in [12], Networks (or Swarms) — where all agents communicate in a single group — and Supervisor — where an LLM acts as the coordinating node. But there are numerous ways of coordinating the work across agents, according to a specific task needs.

For instance, in [6], a team of Microsoft researchers described Magentic-One, a generalist system that demonstrated state of the art performance in multiple agentic benchmarks without changing how the agents collaborate. The architecture is described by them as "a multi-agent architecture where a lead agent, the Orchestrator, plans, tracks progress, and re-plans to recover from errors. [...] Moreover, Magentic-One's modular design allows agents to be added or removed from the team without additional prompt tuning or training, easing development and making it extensible to future scenarios".

One of the most interesting parts about Magentic-One is that it applies the learnings from [18] to multi-agent orchestration, that is, it has explicit steps to evaluate outputs, analyze what went wrong and re-plan. The architecture was open sourced by Microsoft and, for these reasons, was used extensively in this project for its generalist capabilities.

2.2 Software Tools

While the project involved numerous decisions about tools, such as libraries to use for certain tasks, or the type of database to use to store user preferences, this section will only detail the key decisions that had an system-architecture impact: the frameworks used for user interface and agent orchestration, and the tool to schedule agent runs. They will be presented in this chapter in order of importance.

An important note is that the client had a preference for using Azure/Microsoft products for any infrastructure requirements, due to its enterprise readiness. Azure was thus used as inference provider (i.e., LLMs calls are performed through Azure AI Foundry). The client could not provide access to their own instances though, so the Azure student subscription was used independently of some of its limitations in model access and rate limiting.

2.2.1 Agent Orchestration Frameworks

Selecting the appropriate orchestration framework was the most crucial decision for the project, and the key guiding point for this was the size of a framework's community, which is often indicated by the number of GitHub stars. Community size can significantly influence the availability of resources such as documentation, tutorials, and community support. A larger community typically leads to more comprehensive examples and a broader base of shared knowledge, facilitating smoother development and troubleshooting processes.

As such, Table 2.1 compares several prominent Python frameworks for agentic orchestration, highlighting their community size and the implications for developers seeking robust support and resources.

Framework	**GitHub Stars**	**Community Notes**
LangChain / LangGraph	113.6k	Extensive tutorials, active forums, and a wide array of integrations.
AutoGen	43.1k	Used to have support from Microsoft, with growing number of examples, but currently in the process of being merged to Semantic Kernel.
OpenAI SDK	8.6k	Official OpenAI support, with increasing community contributions.
CrewAI	35.8k	Active development with a focus on multi-agent collaboration.
Semantic Kernel	25.8k	Backed by Microsoft, offering enterprise-grade features, but documentation focused on C#.

Table 2.1: Comparison of Python Agentic Orchestration Frameworks

Additionally, the Table 2.2 provides some findings on the same frameworks' functionality, ideal use cases, and the pros and cons associated with each.

As will be explained in more detail later, the development of the project consisted of two stages. The first stage tried to take advantage of Microsoft's Magentic-One orchestration architecture, and that, in combination with the use of Azure as inference provider, made it an easy choice to use Semantic Kernel. On the contrary, the second stage's approach is to build a detailed workflow with careful management of the data used in context, thus making LangChain/LangGraph a better fit, not just for its functionality but for the amount of documentation and examples available to support building this more detailed workflow. LangChain has its own observability platform (LangSmith), which was also used as it integrates easily with the framework.

Framework	Functionality	Use Cases	Pros and Cons
LangChain / LangGraph	Modular pipeline for LLMs, embeddings, tools	Complex workflows, data pipelines	Pros: Extensive integrations, strong community support; Cons: Steep learning curve, large framework size
AutoGen	Multi-agent orchestration with conversational agents	Multi-agent systems, collaborative tasks	Pros: Easy to implement multi-agent system, original implementation of Magentic-One; Cons: In process of being merged to Semantic Kernel
OpenAI Agents SDK	Lightweight framework with agents, handoffs, and guardrails	Rapid prototyping, simple agent workflows	Pros: Minimal abstractions, easy to learn; Cons: Heavily geared towards using the full OpenAI ecosystem
CrewAI	Role-based collaboration with task management	Team-based workflows, sequential tasks	Pros: Intuitive design, good for structured workflows; Cons: Less flexibility for complex scenarios, smaller community
Semantic Kernel	Enterprise-grade framework	Enterprise applications, integration with Microsoft tools	Pros: Strong enterprise support, robust features, includes a default implementation of Magentic-One; Cons: Limited Python support, smaller community

Table 2.2: Comparison of Python Agentic Orchestration Frameworks

2.2.2 Scheduling Tool

The project requires the AI agent to run at set intervals, as defined by the user preferences. Two primary scheduling alternatives were considered: Celery and cron (through the python crontab library).

Celery is a distributed task queue that supports asynchronous and real-time task execution, with advanced features such as retries, task prioritization, and integration with message brokers like Redis or RabbitMQ. While powerful, Celery introduces additional system complexity and overhead, requiring configuration of worker processes and a message broker.

In contrast, cron is a native, time-based scheduler available on Unix-like systems, designed specifically for executing tasks at fixed intervals. Cron is straightforward to configure, persistent across system reboots, and does not require any additional services.

Given the project's scope as a proof of concept, and the requirement of periodic execution at set intervals, cron was selected as the more practical solution.

2.2.3 User Interface Framework

Finally, for setting up the configuration interface, several Python frontend options were considered. The main alternatives evaluated were Streamlit, Gradio, and FastAPI with Jinja2 templates. While Streamlit and Gradio offer very quick setup and ease of use for prototypes, FastAPI with Jinja2 was chosen for this project due to its high customizability and the ability to easily scale the proof of concept into a full production application if needed.

Feature	**Streamlit**	**Gradio**	**FastAPI + Jinja2**
Primary Use Case	Interactive dashboards	ML model interfaces	API-driven web apps
Ease of Setup	Very easy; minimal code	Very easy; minimal code	Moderate; requires setup
Customization	Limited UI customization	Limited UI customization	High; full control
Performance	Moderate	Moderate	High; asynchronous support
Extensibility	Limited	Limited	High; supports plugins
Recommended Deployment	Streamlit Cloud	Gradio Hub	Flexible (e.g., Docker, Cloud)
Best For	Rapid prototyping	Quick ML demos	Scalable, production-ready apps

Table 2.3: Comparison of Python Frontend Frameworks for AI Agent Configuration

Chapter 3

Requirements and Analysis

As mentioned before, the client for this project is Alvarez & Marsal (A&M), a global professional services firm specializing in turnaround management, corporate restructuring, and performance improvement.

A&M clearly identified a problem to solve: Executives spend valuable hours navigating dashboards just to extract the specific information relevant to their needs. As such, they wanted to explore the capabilities of LLMs to help reduce this time by building an Agentic AI system that could access a company's data, extract operational figures and generate report about a specific KPI, delivered directly to a user's email. For this, A&M shared bullet points with their ideal set of functionalities for the system.

Nevertheless, as a professional services firm, Alvarez & Marsal was limited in the resources they could share for the development of the system. Particularly, due to client confidentiality clauses, they were only allowed to provide anonymized data, which severely limited the functionalities that could be built — as an example, while they wanted the system to retrieve news that could explain a reduction in sales, this only makes sense if the system knows which products the company is selling.

Due to this limitation, it was agreed to build a proof of concept system that could later be extended by their internal team. This chapter thus details the process for arriving to the requirements for this proof of concept system. First, the creation of user personas, and their scenarios for using the system, to help narrow the key need for users. Second, the definition of a set of requirements based on said personas.

Readers will note that this process did not include the definition of Use Cases. This is intentional, as the system has a single use case — a user setting the report they want to receive, and when they want to receive it. Other than that, the system is autonomous.

3.1 User Personas and Scenarios

A&M identified 2 types of users they wanted to target, based on their experience advising companies: a Sales Manager and a CFO-level user. The following are Fictional Personas, as well as a set of user scenarios where the system could improve their day to day work. The process of creating these drew heavily from [3] and [5], and was based both on A&M and the author's experience in industry.

John, the Sales Manager

Hard Facts	John lives in London, is 30–35 years old, and works as the Sales Manager for Germany for a car parts manufacturer.
Computer and Internet Use	He mainly uses a computer for work, including emails, reviewing dashboards, and browsing the internet for news and social media. He also owns an iPad and a MacBook. He is comfortable with computers but isn't particularly tech-savvy.
A Typical Monday	John starts his work week by reviewing unread emails to set priorities. He can spend up to two hours reviewing the company's dashboard to track sales in Germany. He then schedules follow-up meetings and spends the afternoon in meetings with his team and clients.
Current Scenarios (As is)	• **All good**: John navigates the dashboard to see that sales are on track and he can move on with his day. • **Issue found**: John notices a sales decline, investigates by manually filtering through different views (clients, number of active clients, churned clients), and then works with his team to create an action plan. • **Continue what's working**: John sees a better-than-expected sales increase, investigates by looking at top clients and specific products, and then finds a news article about a change in safety regulations that might be driving the trend. He then advises his team to review the regulation and try to up-sell more customers.
Future Scenarios (To be)	• **All good**: John receives an email with a short report showing sales are on target, skims it, and moves on with his day. • **Issue found**: John receives an email with a report attached that notes a sales decline and lists the clients who have churned, allowing him to forward the report and schedule a meeting with his team. • **Continue what's working**: John receives an email with a report pointing out a higher-than-expected increase in sales for a specific product and links to a news article about a change in safety regulations, enabling him to forward the report and advise his team.

Clara, the CFO

Hard Facts	Clara lives in Cambridge with her family, is 45–50 years old, and works as the CFO of John's company.
Computer and Internet Use	She mainly uses a computer for work, including emails, reviewing dashboards, and browsing the internet. She is definitely not a tech-savvy person.
A Typical Monday	Clara starts her week by reviewing emails and spends up to an hour reviewing the company's dashboard to identify high-level trends. She has lunch with the CEO to discuss strategic planning and spends the afternoon in meetings with her team.
Current Scenarios (As is)	• **All good**: Clara navigates through different dashboard screens to check various KPIs and finds they are all on track. • **Issue found**: Clara notices a decline in Gross Margin, investigates by looking at sales and costs, and finds an unexpected cost increase in France, which she then notes and instructs her team to investigate. • **Continue what's working**: Clara sees a positive increase in sales growth, investigates by looking at business lines, and finds an uptick in the Brakes division. She then looks for news, finds a regulation change in Germany, and schedules a meeting with the corresponding sales team to strategize.
Future Scenarios (To be)	• **All good**: Clara receives an email with a short report on all the KPIs she monitors, skims it, and moves on. • **Issue found**: Clara receives an email with a report mentioning a KPI (Gross Margin) decline and a detailed explanation of the cause (increased costs in the France unit), allowing her to instruct her team to investigate immediately. • **Continue what's working**: Clara receives an email with a report that points out an increase in sales for the Brakes product group and links to a news article about the related safety regulation change in Germany. This allows her to schedule a meeting with the corresponding sales team to discuss the impact and prepare for similar changes in other countries.

3.1.1 Analysis

As put by [3], Fictional Personas are deeply flawed. Nevertheless, they can be used as initial sketches of user needs. Based on these two particular user personas, a few things are important to drive adoption of any solution:

1. **User interface needs to be easy to use:**
    - Target personas are not particularly tech savvy.
    - They are used to the intuitive interface of dashboards.

- They don't want to spend 5 hours setting something up that only saves them a few hours per month—they want to spend 5 minutes.
- The key output they need is knowing where to look in detail for their next steps, so a high-level report can be good enough.

2. **Our users target a different number of KPIs, and this affects the level of detail they want to see:**

   - A Sales Manager tracks a single KPI (Sales), but they want to go into a loot of operational detail. A CFO, on the other side, follows multiple high-level KPIs, but only wants to have a general understanding of the situation.
   - Moreover, our target personas are used to being able to see exactly the level of operational detail they want from their dashboards. The system must be aware of that level of detail.

3. **Our users want a quick way of knowing if they need to review the report in detail:**

   - This can be addressed in two ways. First, the report must contain an Executive Summary. Second, the email for the report should include the key findings.

3.2 MoSCoW Requirements

Based on the user personas, and the limitations on data availability, the following requirements were defined for the system. Nevertheless, before reviewing them in detail there are a few important things to note.

First, as noted in the analysis of User Personas, the level of detail of the reports for the two types of personas are different — in practical terms, this means the prompts that are used for one type of user might not work for the other. As such, it was decided to split certain requirements related to the report content by type of users.

Second, due to the nature of this project as a proof of concept, and due to the limitations in data availability mentioned before, the system will only implement a subset of all potential requirements. However, knowing what requirements might be implemented in the future can inform design decisions in the present. As such, it was deemed acceptable to have a long list of Won't Have requirements — that is, a long list of requirements that are known and wanted for the system, but that will be outside of the scope of this project.

Finally, due to the limitations on data availability, it was decided that the proof of concept would only cover Sales Manager users. While A&M could anonymize sales data with relative speed [1], the time needed to anonymize enough data for other KPIs was deemed too long for the project.

With this in mind, the following tables reflect the requirements for the project's system.

[1] This speed came with a cost though, as you will notice if you review the sample report, there are some drastic fluctuations in sales that can only be explained by issues with this process.

3.2.1 Functional Requirements

Sales Manager

ID	**Requirement**	**Priority**
FR-1	The system shall allow a Sales Manager-type user to set up their Sales scope (e.g., a specific region or product)	Must Have
FR-2	The system shall be able to generate a Sales performance report based on the user's preferences	Must Have
FR-3	The system shall be able to identify "special cases" in Sales trends and perform a deeper analysis	Must Have
FR-4	The system shall be able to provide simple Sales forecasts based on current trends	Should Have
FR-5	The system shall be able to recommend potential actions to remedy "special cases" in Sales trends	Should Have
FR-6	The system shall be able to provide simple Sales forecasts based on the remedy actions recommended by itself	Could Have
FR-7	The system shall be able to provide advanced Sales forecasts using regression techniques or any other statistical method	Won't Have

CFO

ID	**Requirement**	**Priority**
FR-8	The system shall allow a CFO-type user to set up a list of KPIs they follow	Won't Have
FR-9	The system shall be able to generate a report for the user's KPI list	Won't Have
FR-10	The system shall be able to provide simple forecasts for each KPI based on current trends	Won't Have
FR-11	The system shall be able to identify "special cases" in the trend of any KPI and perform a deeper analysis	Won't Have
FR-12	The system shall be able to recommend potential actions to remedy "special cases" in each KPI's trend	Won't Have
FR-13	The system shall be able to provide simple forecasts for each KPI based on the remedy actions recommended by itself	Won't Have
FR-14	The system shall be able to provide advanced forecasts for each KPI using regression techniques or any other statistical method	Won't Have

User Type Independent Requirements

ID	Requirement	Priority
FR-15	The system shall email the resulting report to the email addresses configured by the user	Must Have
FR-16	The system shall customize the email with the key findings of the report	Should Have
FR-17	The system shall include both financial and operational information in a report	Must Have
FR-18	The system shall be able to load information from a CSV file	Must Have
FR-19	The system shall be able to load information from other files provided by the user	Won't Have
FR-20	The system shall be able to load information from a set of reliable online information sources	Won't Have

3.2.2 Non-Functional Requirements

ID	Requirement	Priority
NFR-1	The system shall be extensible to other sources of information	Must Have
NFR-2	The system shall run autonomously at fixed intervals	Must Have
NFR-3	The system shall allow the user to update the fixed interval for runs	Must Have
NFR-4	The system shall provide a user interface to update its configuration	Should Have
NFR-5	The system shall provide a user interface that is clear and easy to use	Should Have
NFR-6	The system shall allow multiple users to set up their reports	Could Have
NFR-7	The system shall allow users to authenticate before updating their report preferences	Won't Have

Chapter 4

Design and Implementation

As an exploratory proof of concept, the project's system exactly as built will likely never be used in production, and as such, at the time of writing it is not deployed. Nevertheless, its implementation followed a two guiding principles that should make it easy for the client to either build upon it or take useful parts: Modularity and Extensibility.

These principles were followed not only in the high-level design, explained in Section 4.1, but in the detailed implementation, detailed in Section 4.2.

4.1 System Design

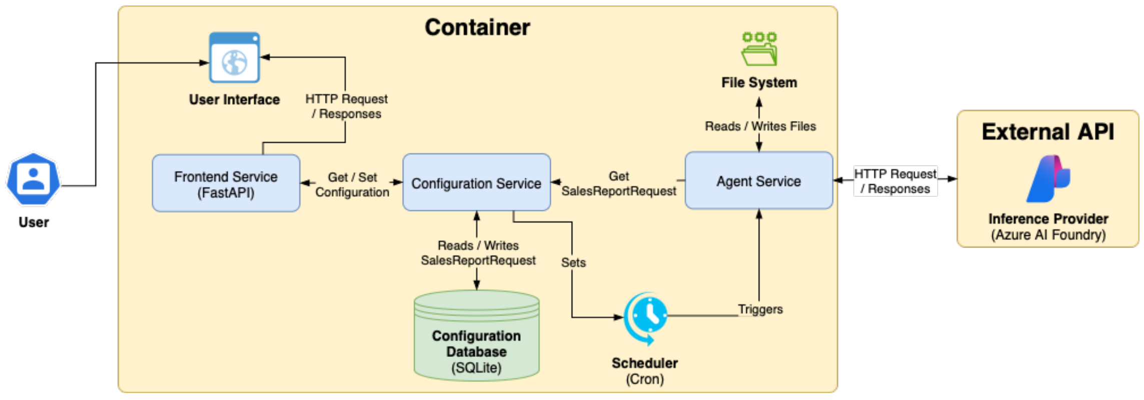


Figure 4.1: System Design Diagram

Following the principle of modularity, as can be seen in Figure 4.1 the system is composed of three main parts.

First, at the core of the system is a Configuration Service. This service is responsible for managing the user's configuration for the AI Analyst — how often it runs, which sets the Cron scheduler, and the topics of the reports it generates, called SalesReportRequest within the system, which are stored in a local SQLite Database. This Configuration Service is used by both the Frontend and Agent services.

Second, the user interface, which is served to the user through a FastAPI application. This user interface has a single functionality, that is, allowing the user to set up the configuration for their AI Analyst.

Third, the Agent service, where the AI Analyst is defined. This service uses the Configuration

Service to retrieve the SalesReportRequests it must execute, and while executing, makes extensive use of the File System to store and retrieve data. Additionally, as inference is provided by Azure, the Agent Service communicates with Azure AI Foundry through HTTP.

Finally, the system has been designed to run in a single container, implemented in Docker. The intention of the containerization is that the AI Analyst needs access to a company's private data, and as such should be easily deployed inside each company's private cloud, not as a stand-alone product.

4.2 Implementation

The process of implementing the system was divided in two parts, both of which will be explored in detail in this section.

First and most importantly, the implementation of the AI Analyst as an Agentic system. As an exploratory proof of concept, this was implemented through an iterative process, defined by a series of experiments with both agent architecture and prompt contents which form the backbone of the reflections in Chapter 6.

Separately, the implementation of the Configuration Service, as well as the user interface to update the configuration, which was a one-off implementation.

4.2.1 AI Analyst Agent

Before detailing the process of building the AI Analyst, it is important to note some practical challenges that influenced some key decisions. These limitation mostly came from the client's restrictions to provide certain resources for developing the system, as mentioned in Chapter 3.

First, the system needed to be developed using Azure's student subscription, which provides a total of USD100 in credits to be used across their platform, independently of the resources needed. Moreover, this subscription limited the LLMs that could be used; some of the highest-performing models, such as OpenAI's o3, were not available for testing.

Second, while the client reviewed if they could share data for testing, it was initially agreed to use [15], a public dataset that simulates entries a company's database, as the production system would be expected to access a company's actual database. Due to Microsoft's restrictions this dataset had to be hosted on Azure, and ran a the relatively high cost of USD1/day while it was being used.

The client was later able to provide anonymized sales data in the form of a CSV file. Thus, the database was turned off and the evaluations and integration tests that used it were removed from the repository, but the agent created for extracting data from it was kept in the repository for the client's future reference — thus, you will see that code related to the database agent has 0% coverage.

Finally, due to the high cost of the database it was decided to use a lower cost LLM to develop the system, as to reduce the risk of reaching the account's spending cap. As such, the system was built using OpenAI's gpt-4o-mini model, and only when the full structure was created was the model changed to OpenAI's o4-mini, a reasoning model, as it was more than 10 times more expensive per token.

Single Task Architecture

Considering the advances in agentic architectures mentioned in Section 2.1, the first group of experiments aimed to use [6]'s Magentic-One orchestration architecture, without modifications, for the entire task. This was done using Microsoft's Semantic Kernel framework, which exposes its own implementation of the Magentic-One architecture but, for reference, Figure 4.2 depicts its general structure. The key objective of this approach was extensibility. With this orchestration, adding new sources of information to the analysis should be as easy as adding a new member to the team of subagents, with everything else remaining as it was.

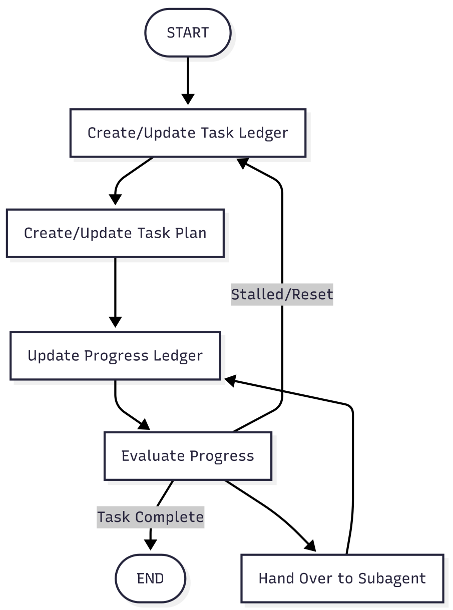


Figure 4.2: Magentic-One Orchestration Architecture

For this group of experiments, three initial subagents were defined:

1. A **Database Agent**, which had access to a tool to explore the tables in a database and execute arbitrary SQL queries. This agent used SemanticKernel's default ChatCompletion-Agent.

2. A **Coder (or Quantitative) Agent**, which had access to a code interpreter tool and thus could run arbitrary code. This agent used Azure's pre-defined agents, as this is the default way to access a code interpreter tool in the framework.

3. An **Editor Agent**, who's system prompt detailed the structure and editing guidelines of the report. This agent also used SemanticKernel's default ChatCompletionAgent, without tools.

The first challenge with this architecture, which became apparent as the first few attempts to run an experiment where unsuccessful, was the interaction with the Database Agent. It quickly became apparent that the Database Agent was essentially a text-to-sql agent, and thus needed request to be defined as queries for information. Magentic-One, however, defines the requests to the next subagent as tasks — to give an idea, the request might be something like "Investigate the main contributors to sales", but the Database Agent worked best with something like "Retrieve sales by product family". Moreover, the default implementation of Magentic-One passes the entire conversation history as context to every subagent, but the Database Agent worked best with a minimal context.

The solution to this was to create another agent, an **Internal Data Agent**, which could access the Database Agent as a tool. In this architecture, the Internal Data Agent could receive a task, including the entire conversation, break the task down into a set of natural language queries, and request them one by one to the Database Agent. The advantage of using the agent-as-tool architecture was that the Internal Data Agent need not know how the queries were executed, increasing the modularity of the design.

Another interesting question was where to use the Editor Agent, that is, whether to include it in the team of subagents for the Magentic-One task, or as a separate step that would take the results from the research task and generate the formatted report from it (similar to a traditional RAG architecture). After experimenting with both, including the Editor Agent within the Magentic-One task resulted in reports that did not follow the editing guidelines. Again, it seemed like the culprit was the increased context included in each request, in line with [20]'s findings.

This then set the architecture for the Single Task approach: first, a Magentic-One task for retrieving all the information required, using a team of the Internal Data Agent, who in turn used the Database Agent as tool, and the Quantitative Agent; second, the results from this task were formatted by the Editor Agent.

But setting the architecture is just one step in the process of creating an agent. As described in Section 2.1 the agent's output can also be improved with Prompt Engineering. While the experiments with prompts were limited in this stage, as it was quickly clear that the system would need a different architecture, one interesting question that would translate to the following stage arose: How reusable should prompts be? Some of the first prompt attempts were very general, without specific examples, with the idea that the client could change the description of the KPI and easily extend the functionality of the AI Analyst from Sales to any KPI — this is called a "zero-shot prompt" in LLM parlance. These prompts were, generally, not successful, which is in line with commentary on both the literature and prompting guides by LLM providers; the results were easily improved just by adding specific examples about detailed sales data that could be retrieved. It is possible to envision a prompt module that is designed in a way that makes it easy to inject different examples or terms depending on the requested KPI; but as the proof of concept only needed to extract Sales information, this was not implemented.

Workflow Driven Architecture

After a few experiments with the prompts for the Single Task Architecture, it was clear that some of the key nuances of the task were not being properly conveyed in the output. As such, the next stage of the project was to design a workflow that the agent could follow which reflected the "standard operating procedure (SOP), with step-by-step instructions for how a human would perform the task or process", as recommended by LangChain in [11]. Figure 4.3 presents the workflow designed as the SOP.

As the new workflow would not use Magentic-One as the basis of its implementation, an important decision had to be made: continue building using Microsoft's Semantic Kernel framework or move to a different framework. Moving to a different framework would mean losing work and manually implementing Magentic-One like architectures for some of the steps in the workflow, but working with Semantic Kernel had already shown some issues — it was not particularly intuitive for fine-grained control of agent's context, and it lacked examples for use cases that did not fall

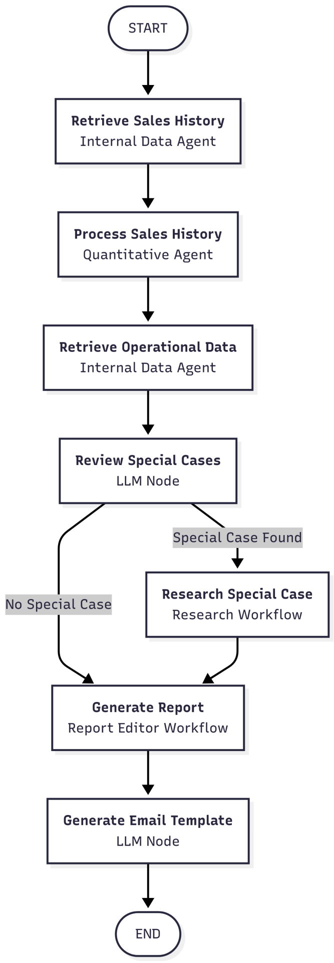


Figure 4.3: AI Analyst workflow, including agent or workflow that implements each step

within its pre-defined structures. LangGraph, in contrast, is specifically designed for this kind of agentic workflow, and due to its popularity there is extensive community content. As such, it was decided to use LangGraph from this point forward.

It was also close to this point that the client was able to provide anonymized sales data for testing. As mentioned before, this was provided as a CSV file and, as such, the Database Agent was not necessary any more. Its function could be replaced by an agent that could write and execute code in the environment that contained the CSV file.[1] As the files were stored in the filesystem of the AI Agent's container, all agents that need code are allowed to execute it directly in the container's environment. This also made it easier to share data between steps; any intermediate outputs could be stored as CSV files within a temporary folder, and other agents would have access to them if needed.

Furthermore, considering the change in both framework and data access, the new workflow was implemented iteratively, starting with a version that only included retrieving the sale's history, processing it and creating a report, and adding additional steps one at a time. Each time a step was added or altered significantly, a full experiment and evaluation was performed, so that the impact of each change on the overall output was clear. This also confirmed the extensibility of the design, as additional steps could be added without breaking the existing functionality.

[1] Though as mentioned before the code for the Database Agent was kept in the repository for the client's reference.

Following this iterative process, the first versions of the workflow used LangGraph's pre-set implementation of the ReAct architecture for any agent that required tool usage, such as the **Internal Data Agent**, which now used a tool to run arbitrary code to retrieve data from the provided CSV file, and the **Quantitative Agent**. Additionally, while the Magentic-One architecture did not generate the expected results for the entire task, it was clear that it had value due to its extensibility and deep research capabilities, so it was used for the **Research Workflow** in the Research Special Case step — that is, the step to explore the available data and find potential reasons for special cases, such as sales reductions — when it was added to the workflow.

Moreover, in early iterations the instruction for analysing data given to the Quantitative Agent included creating visualizations, but this resulted in visualizations that were poorly formatted despite iterations on the prompts. As the analysis task would only become more complex due to the in-depth research for special cases, the plot generation was moved to the end of the workflow. It then made sense to create a separate workflow for the generation of the report, the **Report Editor Workflow**, which consists of a Supervisor, which ensured task completion, a Report Writer, a Data Loader agent, which is actually code to load a CSV's file contents to the workflow's context from a plain text instruction, and a Data Visualization agent, which uses LangGraph's ReAct implementation.

With this, the system's development was complete, and it was time to use the more expensive reasoning model, o4-mini, for evaluation and prompt engineering. As expected, reasoning models are better at complex tasks, resulting in more detailed analysis even before optimizing the prompts, but they come with a drawback: configurations that limit the range of potential results from the LLM, such as Temperature, Top P or Top K are not available for reasoning models. This resulted in an increased likelihood of agents getting stuck in infinite loops, trying to perform more in depth analysis, find more data than available or attempting to run code with errors without being able to fix it between loops.

Some of these issues were improved upon with prompt engineering, such as providing clearer instructions from when to stop, but the stochastic component of LLMs meant there was no way of ensuring this won't happen on any given experiment. As such, changes to the agent's architecture were needed.

First, for the Research Workflow, which used the Magentic-One architecture, the main issue was that the model could not consistently decide when the task was considered complete. While a human would execute a judgment call considering that they have "enough" findings for the report, an LLM needs a clear stopping rule. Thus, it was decided to add a forceful exit condition, where the workflow would break the execution loop after a number of plan updates (note that this is different to progress ledger updates, which happen every time a sub-agent returns a response, plan updates only happen when the execution is deemed stalled by the orchestrator). This approach is in line with Semantic Kernel's implementation.

Additionally, building from the learnings of the first implementation stage, a Summarize Findings step was added to the Magentic-One workflow. Without this step, the best way to share the findings from the task would be to share the entire conversation, adding a large amount of context to the report generation step. Instead, by summarizing the findings, only the actually relevant parts of this conversation, such as findings and files generated, are shared with the main agent, maintaining the total size of the context contained. The final architecture for the Research Workflow can be seen in Figure 4.4.

Second, both the Internal Data Agent and the Quantitative Agent generate code that needs to be executed in the local environment, using a Python Read-Evaluate-Print-Loop (REPL) Tool.

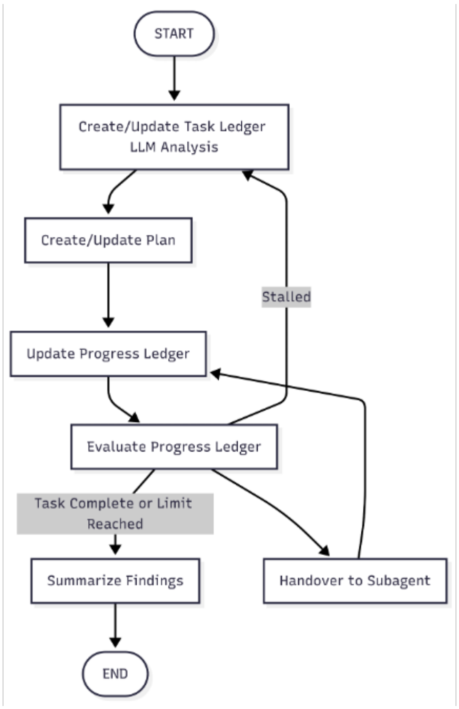


Figure 4.4: Research Workflow representation.

For these agents, which first used a basic ReAct architecture, the issues were twofold. On one side, the LLM had a hard time recovering when the code generated had issues, such as not receiving any outputs due to a lack of print statement or error messages. On the other side, it would sometimes return an intermediate thinking step without code generated, which in a default ReAct agent is taken as a final response.

To improve the agent's self recovery capabilities, the workflow includes a step where the code outputs are reviewed for errors. If errors are present, a separate LLM is requested to provide a code review, which is added to the agent's context before looping. Additionally, when an error is detected several times in a row (5 by default) or the agent has looped more than a certain threshold of times (25 by default), the prompt for the next loop is changed for one that requires the LLM to review its attempts, analyse what went wrong and provide the analysis as final response. This approach was particularly for use within the Magentic-One orchestrator, as it can provide information for the orchestration for either retrying the request or replanning.

To avoid early returns due to thinking steps, the workflow includes a step where that reviews the response from the agent. If the response includes a tool call for code execution, it moves to the next step. If the response is only text, it is reviewed by an LLM to decide whether it is meant as a final response, in which case it is returned, or as an intermediate thinking step, in which case the loop starts again.

These additional steps have a key trade off, they make the agent more expensive and slower to run, but reduce errors. Nevertheless, as the system is designed to run as a background task, adding latency does not affect user experience. The extra cost could be reduced by using a non-reasoning model for these additional reviews, as neither is a complex task, so gpt-4o-mini was used.

The final architecture, which can be seen in Figure 4.5, was implemented as a reusable agent that can be initiated with a different system prompt depending on if it is needed as an Internal Data or a Quantitative Agent, reducing potential duplications.

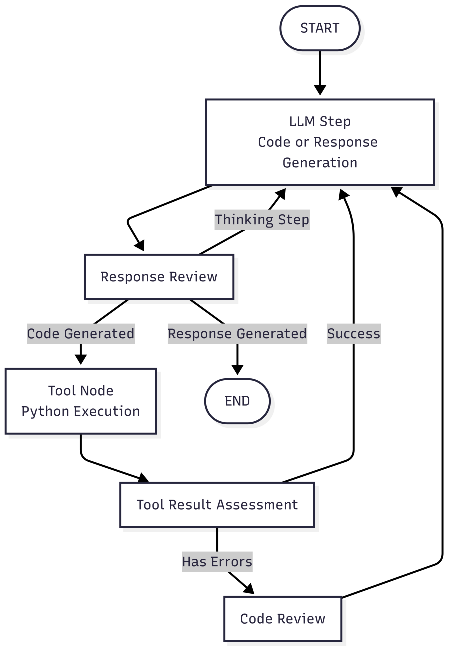


Figure 4.5: Coding Agent workflow representation.

4.2.2 Configuration Service and Interface

The Configuration Service is, in reality, a module within the system. This module is responsible for three things:

First, is is responsible for loading the system's settings, such as API keys, from the environment, ensuring they are available for the rest of the system. As FastAPI uses Pydantic for type validation, it was an easy decision to use Pydantic's `pydantic_settings` library to validate the presence and types of environment variable — using this package ensure the system will fail on startup if the environment is set incorrectly. Similarly, this module is used to set up any system-wide constant, such as the location of data and temporary files.

Second, it is responsible for handling setting and updating the Cron job that runs the AI Analyst according to the user's preferences. The system uses the `python-crontab` library for this purpose.

Finally, the service is responsible for storing, retrieving and updating user preferences regarding the report to be generated. The preferences are represented in code as a Pydantic model (essentially a class that represents data) called SalesReportRequest, which included attributed for the characteristics of the request — such as the "grouping", that is, if the report should be about total sales or sales for a specific country, product, etc. — and for the list of emails that should receive the report when generated.

The key decision to make, though, was how to persist these classes. While it was clear that setting up an external database was unnecessary overhead for storing local configuration, two alternatives were contemplated: serializing the data and storing as files or storing in a sqlite database within the same container.

Serialization would have been the easiest option to set up, and should have performed well for the needs of the proof of concept, which only handles a few reports. Nevertheless, it was decided to use a local sqlite database as it would be more robust if the system is ever implemented within a large company, as A&M's clients usually are. A large company may have hundreds of sales managers setting up their reports, while databases can handle this without any issue, there is a high risk of a file becoming corrupt through two concurrent attempts at updating.

With this in mind, a simple database was defined with two tables: `sales_report_requests` and `recipient_emails`, sharing a one to many relation (one report request can have many recipients). Additionally, following the principle of extensibility, it was decided to use an ORM (`sqlalchemy`) for handling this database, as it would make the system easier to connect to a different database or add additional configuration if needed — for example, some type of user authentication.

User Interface

To update the Configuration, a frontend was implemented as a FastAPI web application using Jinja2 templates. This implementation follows a modular router-based architecture that separates concerns — templates can easily be changed to suite the style of a company that will use the AI Analyst internally. The initial templates created consider the analysis in Chapter 3 and as such try to be simple and intuitive, using Bootstrap 5 to speed implementation.

The FastAPI application consists of three main router modules as detailed in Table 4.1:

Router	**Routes**	**Functionality**

Index Router	`GET /`
	`POST /run_now`
		Main dashboard displaying existing sales report requests and cron schedule configuration.

Sales Report Router	`GET /sales_report/create`
	`POST /sales_report/create`
	`GET /sales_report/edit/{id}`
	`POST /sales_report/edit/{id}`
	`POST /sales_report/delete/{id}`
		Complete CRUD operations for sales report requests. Handles form validation, data persistence, and user feedback with proper error handling and success messages.

Cronjob Router	`POST /crontab/update`	Manages the cron schedule configuration for automated report generation.

Table 4.1: Frontend Router Structure and Functionality

Chapter 5

Evaluation and Testing

As mentioned before, this project followed [8]'s recommendations regarding LLM evaluations. This included following Evaluation Driven Design, or the practice of writing the evaluations before working on the agent. As such, Section 5.1 will discuss the evaluation framework designed for the AI Analyst's final output across experiments, which was defined before the start of implementation. Moreover, also following the recommendations of [8], each of the agent's components was evaluated separately. The approach taken to the evaluation of the components is described in Section 5.2. Finally, Section 5.3 discussed the approach taken to tests for the non-agentic parts of the code, such as the configuration module.

5.1 Agent Output Evaluation

Evaluating the outputs of the AI Analyst was an essential part of the project. Knowing the complexities of Prompt and Context Engineering, it was essential for the development to have a defined evaluation for the AI Analyst's outputs to ensure that iterations could be performed safely, and improvements, or regressions, could be objectively defined.

Nevertheless, in line with some of the research described in Section 2.1, evaluating the AI Analyst posed a series of challenges.

First, even for a single type of Sales Report Request, it would be impossible to create a comprehensive set of all possible reports that could be generated, so the traditional Machine Learning strategy of evaluating against ground truths was not viable.

Second, evaluating task completion was not relevant. With the current state of LLMs, obtaining a report is relatively straightforward; the greater challenge lies in ensuring that the report is actually what the users would expect, that is, it has the structure expected, the data extracted is correct and the insights are relevant.

Finally, because the scope of the project is only a proof of concept, the evaluations defined are unlikely to be used in production, so it was important to limit the effort dedicated to creating an evaluation framework.

Considering these challenges, it was clear that the best path forward was an "subjective evaluation". That is, having an external validator that would take the report generated, review it and grade it. This grading could consider the particularities of the task, and could act as north star for development: higher scores could mean forward progress.

Moreover, the final constraint about limiting effort made it easy to decide to use a Human Judge as evaluator. While AI Judges have advantages, particularly the potential of running them autonomously and thus increase the number of potential iterations, setting up an AI Evaluator added unnecessary complexity at this stage, namely the need to set up an additional evaluation — you have to "evaluate the evaluator" to make sure that its grading is accurate.

Knowing that outputs would be evaluated by a human, the key challenge became ensuring that the score accurately reflected the needs of the task, specially considering the findings of [7] regarding potential biases in human evaluation. The solution was to use a tried-and-tested framework from the education industry — a grading rubric[1]. In essence, this grading rubric could contain the most relevant aspects of the report and assign a grade to each, thus producing an overall grade for the result.

Rubric Definition

As mentioned before, when evaluating an Agent [8] recommends thinking of three groups: Domain-Specific Capacity, Generation Capacity and Instruction-Following Capacity.

In the context of a report, it is possible to think about both Instruction-Following as the capacity of the agent to follow the structure of the report, while Generation as the capacity to retrieve and present the correct data. Both of these can be considered as part of the "Form" of the report — does the report include the right sections, is the data correct, is the data presented graphically.

Defining a grading rubric for these thus required defining the structure of the output report. While this was not considered part of the requirements elicitation process, as the structure of the report is a variable within the system and not a requirement, it was agreed with Alvarez & Marsal during the same process, arriving at:

1. Executive Summary: A quick glance of the key findings.
2. Overview: An overview of the value of the KPI for the period.
3. Trends and Context: An overview of the trend for the KPI and high-level operational metrics.
4. In depth analysis: A detailed analysis of operational metrics that drive the trend.
5. Forward Outlook and Recommendations: A projection of the KPI and potential ways to correct negative trends.

On the contrary, Domain-Specific Capacity on this context can be considered the capacity of the model to perform financial analysis. This is things like accurately representing an evolution instead of just mentioning values or only presenting relevant data. These are more reflective of the Content of the report, not its form.

Finally, in addition to defining the types of questions, it was important to define the level of specificity. If questions became too specific, for example expecting a per-product analysis, the evaluation only becomes usable to one type of Sales Report Request, but very open ended questions lead to subjective grading.

This was solved by defining questions that are wide in scope, but allowing only a Yes or No answer, in the style of "Does the analysis in the report accurately represent the evolution of the KPI?". While this question requires reviewing the entire report and subjectively defining what accurate is, there is less risk that the same evaluator can have different responses to similar outputs, unlike if required to provide a specific grade.

[1]Thanks to Professor Griffin for his comments that pointed in this direction.

Moreover, this type of questions had two added benefits. First, it made it easier, and thus quicker, to evaluate a specific report. Second, it simplified the score definition — the total evaluation score could be calculated as the percentage of positive responses over the total questions.

Thus, the final rubric, which can be found in Appendix C, consists of a list of 23 Yes or No questions — 14 questions focused on the Form of the report, as this covers two types of capacity analysis, and 9 questions focused on its Content.

5.2 Components Evaluation

In addition to evaluating the final output, evaluating an Agent's components can help spot issues earlier in the development or identify the causes of unexpected behaviour. Nevertheless, it has most of the same challenges described before.

There is one exception to this though, just evaluating the intermediate steps completion does add valuable information, specially for development, as it allows to confidently build the system modularly without the need to run full experiments on every change — a relevant issue when considering that full experiments could take up to one hour when the system was complete.

Moreover, in many cases there is one aspect to the intermediate step that can act as a proxy for its utility to the system, such as making sure that a file is generated or that the output has a certain shape.

For these reasons, it was decided to use "exact evaluation" for the Agent's components. Whats better, using strict rules to evaluate the results, such as confirming completion or that a file was generated, makes the evaluations similar to a regular automated test, and can be expressed as such in `pytest`, meaning they can run automatically. In total, 37 evaluations were written with this structure. You can find an example in Appendix E.

One of the biggest challenges of this approach is that while the evaluations are **expressed** as tests, they are not actual tests and should be interpreted differently. In some cases, the evaluation rule might check that a specific word or numeric value is included in the test, this is specially useful when evaluating summarization steps. That expected text might not always be present due to the probabilistic nature of LLMs. In such cases, a failing "test" does not mean the system is not working; instead, it means that a more detailed experiment needs to be run for this specific component.

A solution to this issue is to use a subjective evaluation, but as mentioned before this has additional challenges which, for the scope of this project as proof of concept, are considered an unnecessary effort.

Another unexpected challenge of using this strategy was handling the file system. As defined before, one of the most important aspects of an agent is its capacity to perform actions on its environment. In this project, that mostly meant reading and writing files to store intermediate data, mostly in a temporary folder that is created for each request, all handled in a central module.

Ideally, the testing environment is isolated from the rest of the application, which meant that it needed its own temporary folder, something that sounds easy but becomes difficult with the way `pytest` manages patching imported modules. Because modules in python are resolved the first time they are imported on a process, patching a central module requires special care — things like never importing at the module level, and transitively patching when a module imports a module that imports the patched module. While this was solved, if development of the system will continue it is probably worth the time to streamline this.

Finally, the biggest challenge of this approach is how long it takes to run the test suite. Evaluations expressed as tests are essentially integration tests; they perform real calls to the model provider, facing latency, and evaluated subagents perform real, if reduced, tasks, which take time. This discourages running the entire evaluation suit on code changes, thus sometimes voiding the benefit of identifying unexpected regressions from a change. A production version of the system might benefit from splitting pure tests and evaluations as two suites that do not have to run together.

5.3 Other Tests

While the LLM components of the system need to be evaluated, an Agent is also composed of parts that are indeed subject to traditional tests; for example, the tools it uses or the integration with model providers. Furthermore, the entire system goes beyond the actual AI Analyst Agent. These more traditional software components also need testing.

Nevertheless, as mentioned a few times already, as a proof of concept, iteration speed outweighed testing completeness, as this system might face significant changes before becoming production-ready. As such, the approach taken to testing can be summarized as:

1. Focus on unit testing, with some narrow integration tests.
2. Avoid writing tests for functionalities that only wrap external libraries, such as email sending or pdf generation. It is assumed that the libraries themselves are well tested.
3. For the most important functionalities, such as the configuration modules, 100% coverage was ensured.
4. For less critical functionalities, only testing the "happy path" was considered acceptable.
5. For the frontend, as it is expected to completely change when integrated to a company's systems, manual acceptance testing was considered acceptable.

With all of these in mind, 23 test between unit and integration were written for the configuration module, which was considered the most critical, 17 tests were written for functionalities related to the agent execution, such as tools and utilities, and 5 unit test for the frontend. The result of those test, as well as the coverage summary, can be found in Appendix F.

Chapter 6

Reflections

As a refresher, the aim of this project was to explore the capabilities of LLMs to access a company's data, extract operational figures and provide initial insights to managers, which was to be done by creating a proof of concept system that acts as a junior analyst who autonomously generates reports about a specific KPI, with a simple user interface where the user can set the specific needs for their report and the interval at which the analyst should run.

Success, in this particular project, had two complementary angles. The more tangible angle was to successfully create the proof of concept system, with the set of requirements discussed in Chapter 3. The other, more open-ended angle, was to gather insights from building the proof of concept about using LLMs for this type of task, which can be useful for the client's future projects.

The project can be considered a success on both fronts. Section 6.1 will reflect on the requirements of the proof of concept system, how many were actually included in the system and why others were not. Yet, as mentioned before, with AI Agents it is important to differentiate between a the functionality existing and it being useful, which is where Evaluations come into play. Section 6.2 will thus reflect on the evolution of the key Evaluation metrics, and through this will explore the question of using LLMs for this type of analysis, and the key learnings from the project. Finally, Section 6.3 will reflect on how the proof of concept system can be improved upon to arrive at a production-level application.

6.1 Proof of Concept Requirements Achievement

In general terms, the proof of concept system includes all key requirements. It achieved 100% of both Must Have and Should Have requirements, and 1 of the 2 Could Have requirements. Additional, the system can create reports for Total Sales, which is a KPI within the scope of the CFO, showing that the system as designed can be used as a base for further development of functionality for this type of user. However, implementing this type of user was outside the scope of this project.

The only Could Have requirement that was excluded from the proof of concept was FR-6. This was decided as the agent does not reliably recommended remedy actions that are measurable, but could be addressed considering the insights described in Section 6.2 if needed as part of the production system.

6.1.1 Functional Requirements

Sales Manager User

ID	Requirement	Priority	Included
FR-1	The system shall allow a Sales Manager-type user to set up their Sales scope	Must Have	Yes
FR-2	The system shall be able to generate a Sales performance report based on the user's preferences	Must Have	Yes
FR-3	The system shall be able to identify "special cases" in Sales trends and perform a deeper analysis	Must Have	Yes
FR-4	The system shall be able to provide simple Sales forecasts based on current trends	Should Have	Yes
FR-5	The system shall be able to recommend potential actions to remedy "special cases" in Sales trends	Should Have	Yes
FR-6	The system shall be able to provide simple Sales forecasts based on the remedy actions recommended by itself	Could Have	No
FR-7	The system shall be able to provide advanced Sales forecasts using regression techniques or any other statistical method	Won't Have	No

Table 6.1: Sales Manager Type User Functional Requirements Achievement

CFO User

ID	Requirement	Priority	Included
FR-8	The system shall allow a CFO-type user to set up a list of KPIs they follow	Won't Have	No
FR-9	The system shall be able to generate a report for the user's KPI list	Won't Have	No
FR-10	The system shall be able to provide simple forecasts for each KPI based on current trends	Won't Have	No
FR-11	The system shall be able to identify "special cases" in the trend of any KPI and perform a deeper analysis	Won't Have	No
FR-12	The system shall be able to recommend potential actions to remedy "special cases" in each KPI's trend	Won't Have	No
FR-13	The system shall be able to provide simple forecasts for each KPI based on the remedy actions recommended by itself	Won't Have	No
FR-14	The system shall be able to provide advanced forecasts for each KPI using regression techniques or any other statistical method	Won't Have	No

Table 6.2: CFO Type User Functional Requirements Achievement

User Type Independent Requirements

ID	Requirement	Priority	Included
FR-15	The system shall email the resulting report to the email addresses configured by the user	Must Have	Yes
FR-16	The system shall customize the email with the key findings of the report	Should Have	Yes
FR-17	The system shall include both financial and operational information in a report	Must Have	Yes
FR-18	The system shall be able to load information from a CSV file	Must Have	Yes
FR-19	The system shall be able to load information from other files provided by the user	Won't Have	No
FR-20	The system shall be able to load information from a set of reliable online information sources	Won't Have	No

Table 6.3: User Type Independent Functional Requirements Achievement

6.1.2 Non-Functional Requirements

ID	Requirement	Priority	Included
NFR-1	The system shall be extensible to other sources of information	Must Have	Yes
NFR-2	The system shall run autonomously at fixed intervals	Must Have	Yes
NFR-3	The system shall allow the user to update the fixed interval for runs	Must Have	Yes
NFR-4	The system shall provide a user interface to update its configuration	Should Have	Yes
NFR-5	The system shall provide a user interface that is clear and easy to use	Should Have	Yes
NFR-6	The system shall allow multiple users to set up their reports	Could Have	Yes
NFR-7	The system shall allow users to authenticate before updating their report preferences	Won't Have	No

Table 6.4: Non-Functional Requirements Achievement

6.2 Evaluation Results and Key Insights

Figure 6.1 displays the evolution of the total evaluation score for the system generated report [1]. At first glance, there was a consistently improvement over time, nevertheless, this does not show the entire story. To complement, Figure 6.2 shows the evolution of the score when considering only items that evaluate the form of the report and Figure 6.3 shows the score considering only items that evaluate the content of the report. This section will break down the meaning and insights that can be extracted from these three charts.

[1]Remember the score is just the percentage of items checklist that are correct in the output.

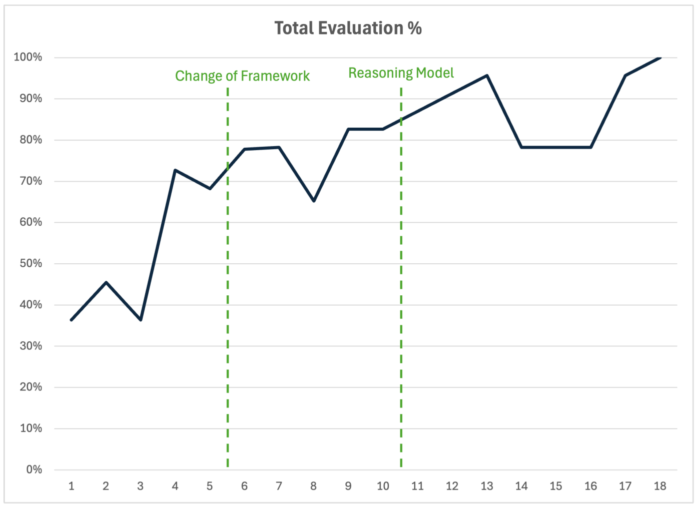


Figure 6.1: Evolution of evaluation results

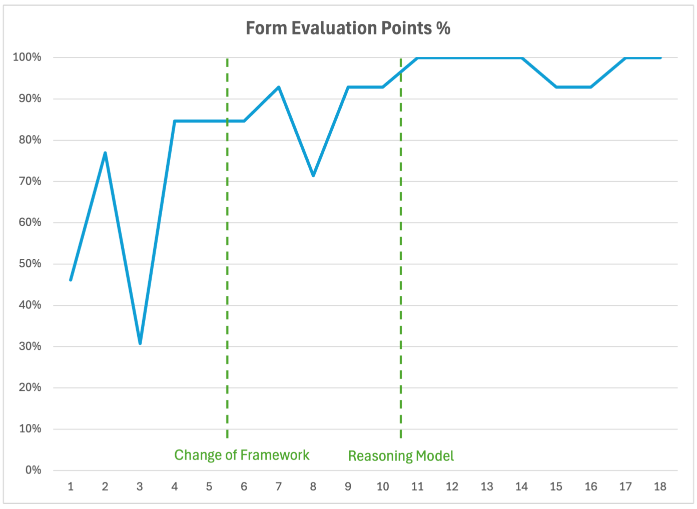


Figure 6.2: Evolution of evaluation results considering only form items

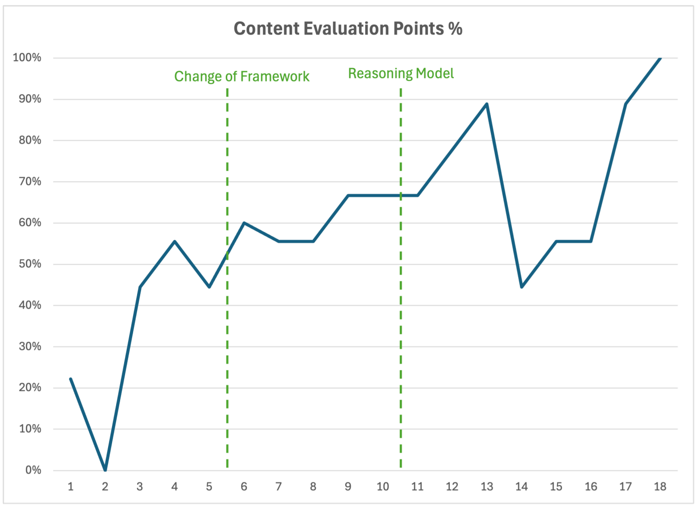


Figure 6.3: Evolution of evaluation results considering only content items

As mentioned before, the first iteration considered a single orchestration entity, with a single task to create the entire report. As can be seen in Figure 6.2, this iteration could get a reasonably high score on the form evaluation, but never perfect. It consistently failed to properly present the "current" period of data — in every case, the report analysed the entire sales history, which is not useful for a report that will be issued periodically. This is a key nuance of the task.

Moreover, as evidenced in Figure 6.3, this agentic architecture failed to score more than 60% on the content score, despite changes to the task description or architecture. In fact, as can be reviewed in Figure D.1, every iteration had issues with different parts of the content.

After an in-depth review of the intermediate steps taken by the system, the reason for this issue became clear: while the agent loops, the context keeps growing and growing, even if the steps taken in that loop do not actually move the task forward. In this particular architecture, this growth in context also affected the context of each sub-agent, making their responses worse as more loops took place. This led to two insights, which drove the approach to the second iteration:

- **Key Insight 1**: while LLMs can indeed perform large tasks with some success, by breaking the task into subtasks we can manage the context at each point, improving their effectiveness and reliability.
- **Key Insight 2**: deep research tasks increase the context quickly, but not all intermediate findings actually add information, so they benefit for summarization steps within the workflow[2]. This is in line with the findings of [20].

The first point can be seen clearly in Figure 6.2. After the change to a step-by-step workflow, fluctuations in form score reduced vastly, while the content score slowly increased as the entire workflow was rebuilt. This is also highlighted by the fact that LLM steps were not reliably recommended remedy actions that were measurable; by making the recommendations part of the single deep research task, and not a separate step, the nuance is lost within the large context the task itself creates, resulting in very general recommendations.

Later on, after finishing rebuilding the workflow, the system's default model was changed to use one of OpenAI's reasoning models, leading to **Key Insight 3**: changing to a reasoning model has a smaller impact than the change to a step-by-step workflow, as the reasoning step can be approximated by the architecture of each agent within the workflow. That said, the quality of certain parts of the output, such as the visualizations generated, did improve markedly by using a more powerful model.

Nevertheless, as mentioned in Chapter 4 with the change to reasoning model runs became more volatile, as reasoning models do not accept configuration variables such as Temperature, Top K or Top P. This was particularly evidenced in sub-agents that work iteratively before responding, which seemed to fall into infinite loops more easily. This led to **Key Insight 4**: autonomous agents are not guaranteed to self-recover, and the system must be planned around that. Common architectures use steps to reflect and re-plan, but this assumes that the agent will at some point finalize the task. For cases such as deep research, where there might not be clear measure for the task to be "done", it is important to create, and properly handle, forceful ending points.

Finally, as mentioned in Chapter 5 the evaluation score does not measure how useful the report is for decisions making. Some reports, for the same request and with access to the same data, might show detailed operational information by City and Product Family, while others might extract insights by Customer. All of these would get the same evaluation score, but might be more

[2]As opposed to most common architectures which only summarise at the end of the task

or less useful. This highlights **Key Insight 5**: at the current state of LLMs, they can be used to reduce the time spent in data extraction, but companies will still need a human in the loop to review the output, confirm whether additional information is needed, and decide how to act based on the extracted insights.

6.3 Future Work

The current system is a proof of concept. As such, it can — and should — be improved upon to create production-ready system. The most important changes to get there would be:

1. Improvements to the evaluation framework:
    - Fully automate the evaluation of outputs, to allow for quicker iterations.
    - Evaluate intermediate steps within a full execution, in addition to individually.
    - Add objective evaluation to intermediate steps when appropriate.
2. Improvements to the product readiness of the user interface for configuration:
    - If the system will be deployed in a publicly accessible location, add authentication.
    - Add acceptance tests that can be run on the CI/CD pipeline.
    - Potentially, allow different reports to run on different cron configurations.
3. Improvements to the agent architecture and outputs:
    - Make the corrective actions proposed by the LLM when there is a “special case” measurable, by creating a separate step in the workflow dedicated to providing and measuring the recommendations.
    - Improve retry logic so that the entire workflow is not repeated when one of its steps fails.

Bibliography

[1] Tom B. Brown et al. *Language Models are Few-Shot Learners.* 2020. arXiv: 2005.14165 [cs.CL]. URL: https://arxiv.org/abs/2005.14165.

[2] Danqi Chen et al. *Reading Wikipedia to Answer Open-Domain Questions.* 2017. arXiv: 1704.00051 [cs.CL]. URL: https://arxiv.org/abs/1704.00051.

[3] Rikke Friis Dam and Teo Yu Siang. *Personas – A Simple Introduction.* Accessed: 2025-08-26. 2025. URL: https://www.interaction-design.org/literature/article/personas-why-and-how-you-should-use-them?srsltid=AfmBOooAYhlEhOmlGj-G2Ah1HQjFMXz2c1lRuLjGAk-7T5qYeFTLfwXQ#10_steps_to_creating_your_engaging_personas_and_scenarios-6.

[4] Forrester Consulting. *The Crisis of Fractured Organizations: How Teams Can Address Organizational Misalignment & Achieve More In The Modern Work Environment.* Thought Leadership Paper. Commissioned by Airtable. Forrester Research, Inc., Dec. 2022. URL: https://www.airtable.com/lp/resources/reports/crisis-of-the-fractured-organization.

[5] Interaction Design Foundation. *What are User Scenarios?* Accessed: 2025-08-26. 2025. URL: https://www.interaction-design.org/literature/topics/user-scenarios.

[6] Adam Fourney et al. *Magentic-One: A Generalist Multi-Agent System for Solving Complex Tasks.* 2024. arXiv: 2411.04468 [cs.AI]. URL: https://arxiv.org/abs/2411.04468.

[7] Yebowen Hu et al. *DecipherPref: Analyzing Influential Factors in Human Preference Judgments via GPT-4.* 2023. arXiv: 2305.14702 [cs.CL]. URL: https://arxiv.org/abs/2305.14702.

[8] Chip Huyen. *AI Engineering.* USA: O'Reilly Media, 2025. ISBN: 978-1801819312.

[9] Zixuan Ke et al. *Demystifying Domain-adaptive Post-training for Financial LLMs.* 2025. arXiv: 2501.04961 [cs.CL]. URL: https://arxiv.org/abs/2501.04961.

[10] Naveen Krishnan. *AI Agents: Evolution, Architecture, and Real-World Applications.* 2025. arXiv: 2503.12687 [cs.AI]. URL: https://arxiv.org/abs/2503.12687.

[11] LangChain. *How to Build an Agent.* Last Accessed: 2025-08-26. July 2025. URL: https://blog.langchain.com/how-to-build-an-agent/.

[12] LangChain Documentation. *Multi-agent systems.* Last Accessed: 2025-08-26. 2025. URL: https://langchain-ai.github.io/langgraph/concepts/multi_agent/.

[13] Patrick Lewis et al. *Retrieval-Augmented Generation for Knowledge-Intensive NLP Tasks.* 2021. arXiv: 2005.11401 [cs.CL]. URL: https://arxiv.org/abs/2005.11401.

[14] Guilong Lu et al. *BizFinBench: A Business-Driven Real-World Financial Benchmark for Evaluating LLMs.* 2025. arXiv: 2505.19457 [cs.AI]. URL: https://arxiv.org/abs/2505.19457.

[15] Microsoft Corporation. *WideWorldImporters Sample Database*. Sample database for Microsoft SQL Server and Azure SQL, retrieved from Microsoft documentation. 2025. URL: https://learn.microsoft.com/en-us/sql/samples/wide-world-importers-what-is?view=sql-server-ver17.

[16] OpenAI Help Center. *Best practices for prompt engineering with the OpenAI API*. Last Accessed: 2025-08-26. Aug. 2025. URL: https://help.openai.com/en/articles/6654000-best-practices-for-prompt-engineering-with-the-openai-api.

[17] Ranjan Sapkota, Konstantinos I. Roumeliotis, and Manoj Karkee. *AI Agents vs. Agentic AI: A Conceptual Taxonomy, Applications and Challenges*. 2025. DOI: https://doi.org/10.1016/j.inffus.2025.103599. arXiv: 2505.10468 [cs.AI]. URL: https://arxiv.org/abs/2505.10468.

[18] Noah Shinn et al. *Reflexion: Language Agents with Verbal Reinforcement Learning*. 2023. arXiv: 2303.11366 [cs.AI]. URL: https://arxiv.org/abs/2303.11366.

[19] Kota Tanabe et al. "Enhancing Financial Domain Adaptation of Language Models via Model Augmentation". In: *2024 IEEE International Conference on Big Data (BigData)*. IEEE, Dec. 2024, pp. 6661–6669. DOI: 10.1109/bigdata62323.2024.10825292. URL: http://dx.doi.org/10.1109/BigData62323.2024.10825292.

[20] Blerta Veseli et al. *Positional Biases Shift as Inputs Approach Context Window Limits*. 2025. arXiv: 2508.07479 [cs.CL]. URL: https://arxiv.org/abs/2508.07479.

[21] Lei Wang and Yiqing Shen. "Evaluating Causal Reasoning Capabilities of Large Language Models: A Systematic Analysis Across Three Scenarios". In: *Electronics* 13.23 (2024). ISSN: 2079-9292. DOI: 10.3390/electronics13234584. URL: https://www.mdpi.com/2079-9292/13/23/4584.

[22] Ruiqi Wang et al. "Can LLMs Replace Human Evaluators? An Empirical Study of LLM-as-a-Judge in Software Engineering". In: *Proceedings of the ACM on Software Engineering* 2.ISSTA (June 2025), pp. 1955–1977. ISSN: 2994-970X. DOI: 10.1145/3728963. URL: http://dx.doi.org/10.1145/3728963.

[23] Jason Wei et al. *Chain-of-Thought Prompting Elicits Reasoning in Large Language Models*. 2023. arXiv: 2201.11903 [cs.CL]. URL: https://arxiv.org/abs/2201.11903.

[24] Shunyu Yao et al. *ReAct: Synergizing Reasoning and Acting in Language Models*. 2023. arXiv: 2210.03629 [cs.CL]. URL: https://arxiv.org/abs/2210.03629.

[25] Mingchen Zhuge et al. *Agent-as-a-Judge: Evaluate Agents with Agents*. 2024. arXiv: 2410.10934 [cs.AI]. URL: https://arxiv.org/abs/2410.10934.

Appendix A

System Manual

This section provides the technical details necessary for developers to understand, deploy, and maintain the AI Analyst system. The system is implemented as a containerized Python application with clear separation of concerns across its core components.

A.1 System Requirements

The AI Analyst system requires the following technical environment:

- Python 3.13 or higher
- uv package manager for dependency management
- Docker and Docker Compose for containerized deployment
- Azure account for AI model access (Azure OpenAI or Azure Foundry)
- SMTP server for automated email delivery

A.2 Code Repository Structure

The system follows a modular architecture with clear separation between agent logic, configuration management, and user interface components:

```
src/
-agents/           # AI agent implementations and workflows
-configuration/    # System settings and agent configuration
-frontend/         # Web interface and API routes

agent_main.py      # Primary agent execution entry point
frontend_main.py   # FastAPI web application
data/              # Financial datasets
outputs/           # Generated reports and artifacts
documentation/     # Technical and user documentation
tests/             # Test suite and evaluation framework
```

The source code is available at the project repository, with comprehensive documentation in the `documentation/` folder covering setup, architecture, and deployment procedures.

A.3 Installation and Configuration

System deployment follows a straightforward process:

Configuration is managed through environment variables defined in a `.env` file. Essential settings include AI model authentication keys, database connection strings, and email service configuration; see `.env.example`. The system uses Pydantic Settings for type validation, ensuring startup failures if the environment is incorrectly configured.

Deependencies are defined in `pyproject.toml`, and can be installed with a single command: `uv sync`.

Finally, the system includes a Docker configuration and thus can run as `docker-compose up`.

A.4 Data and Testing

Test execution uses: `uv run pytest`. Note that the sample data is not included in the repository as it is too large, and thus the test suite will fail. Moreover, consider that the testing framework combines unit tests and agent evaluation, so it needs Azure API Keys to be set in the `.env` file before running.

Appendix B

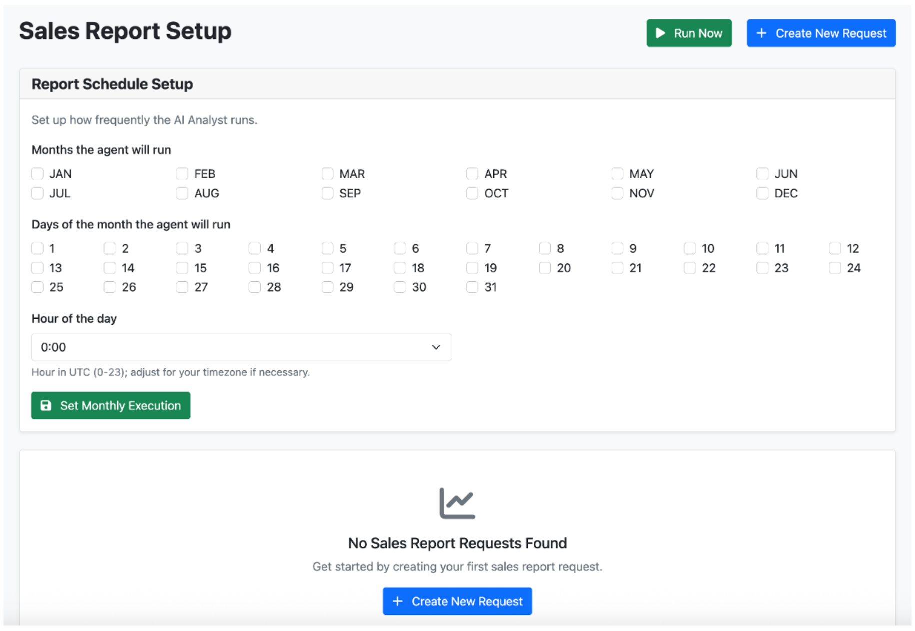
User Manual

This manual provides a guide on how to use the Sales Report Setup user interface to configure your AI Analyst. It is organized by functionality.

B.1 Getting Started

Before setting up your AI Analyst, it is adviced that you familiarize yourself with the user interface, which will look like Figure B.1 if you have not set up your Analyst yet, and like Figure B.2 if you have.

The upper section of the page allows you to run the AI Agent immediately with the "Run Now" button, and to add additional reports to be generated by the AI Analyst through the "Create New Request" button. See Section B.3 for more detail.



Sales Report Setup

Report Schedule Setup

Set up how frequently the AI Analyst runs.

Months the agent will run

☐ JAN	☐ FEB	☐ MAR	☐ APR	☐ MAY	☐ JUN
☐ JUL	☐ AUG	☐ SEP	☐ OCT	☐ NOV	☐ DEC

Days of the month the agent will run

☐ 1	☐ 2	☐ 3	☐ 4	☐ 5	☐ 6	☐ 7	☐ 8	☐ 9	☐ 10	☐ 11	☐ 12
☐ 13	☐ 14	☐ 15	☐ 16	☐ 17	☐ 18	☐ 19	☐ 20	☐ 21	☐ 22	☐ 23	☐ 24
☐ 25	☐ 26	☐ 27	☐ 28	☐ 29	☐ 30	☐ 31					

Hour of the day

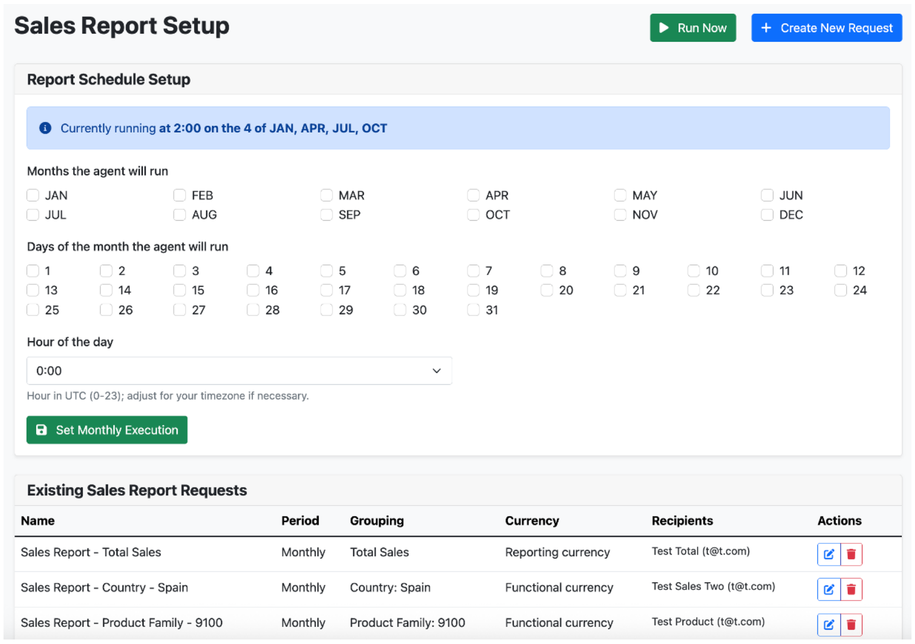
0:00

Hour in UTC (0-23); adjust for your timezone if necessary.

No Sales Report Requests Found

Get started by creating your first sales report request.

Figure B.1: User interface before any set up.



Sales Report Setup

Report Schedule Setup

Currently running **at 2:00 on the 4 of JAN, APR, JUL, OCT**

Months the agent will run

☐ JAN	☐ FEB	☐ MAR	☐ APR	☐ MAY	☐ JUN
☐ JUL	☐ AUG	☐ SEP	☐ OCT	☐ NOV	☐ DEC

Days of the month the agent will run

☐ 1	☐ 2	☐ 3	☐ 4	☐ 5	☐ 6	☐ 7	☐ 8	☐ 9	☐ 10	☐ 11	☐ 12
☐ 13	☐ 14	☐ 15	☐ 16	☐ 17	☐ 18	☐ 19	☐ 20	☐ 21	☐ 22	☐ 23	☐ 24
☐ 25	☐ 26	☐ 27	☐ 28	☐ 29	☐ 30	☐ 31					

Hour of the day

0:00

Hour in UTC (0-23); adjust for your timezone if necessary.

Existing Sales Report Requests

Name	Period	Grouping	Currency	Recipients	Actions
Sales Report - Total Sales	Monthly	Total Sales	Reporting currency	Test Total (t@t.com)	
Sales Report - Country - Spain	Monthly	Country: Spain	Functional currency	Test Sales Two (t@t.com)	
Sales Report - Product Family - 9100	Monthly	Product Family: 9100	Functional currency	Test Product (t@t.com)	

Figure B.2: User interface after set up.

B.2 Scheduling Report Generation

The top section of your screen is used for scheduling when the AI Agent will run, and thus generate your reports. You can select specific months and days. For example, you can select the months of January, April, July and October to receive reports quarterly.

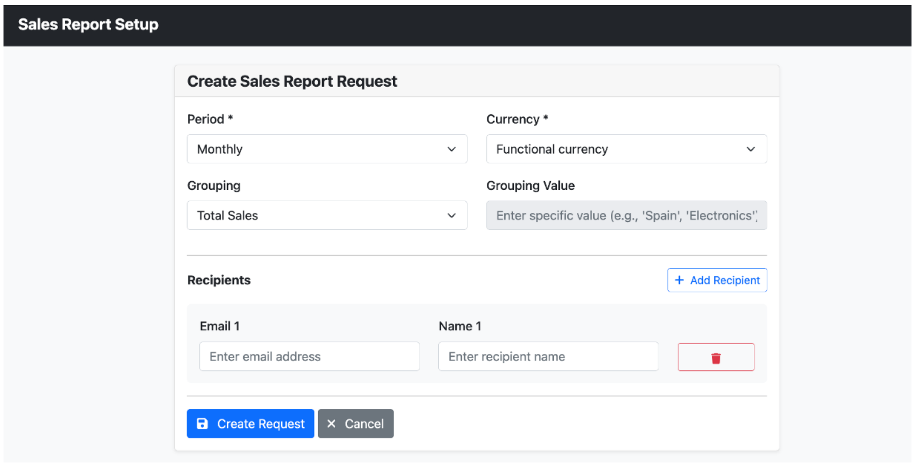
After you have set up a schedule for your report, the current settings will be displayed, as you can see in Figure B.2.

B.3 Adding or Updating Report Requests

When you click "Create New Request", you will be redirected to a screen that looks like Figure B.3. This is the form you will fill up to add a new report that you want to receive, or to update an existing one. The fields to set are:

- **Period**: Whether the report covers Monthly, Quarterly or Yearly data.
- **Currency**: Whether the report should consider the Functional or Reporting Currency. This is only relevant for regions that use a different currency than the company's reporting currency.
- **Grouping**: Whether the report should consider sales by a specific product or geography. If not, select Total Sales.
- **Grouping Value**: If your report should consider only sales for a specific grouping, this is where you say which — for example, the specific city or product.
- **Recipients**: The emails and names of the users that should receive the report.

When you finish filling the form, click "Create Request".


Sales Report Setup

Create Sales Report Request

Period *	Currency *
Monthly	Functional currency
Grouping	Grouping Value
Total Sales	Enter specific value (e.g., 'Spain', 'Electronics',

Recipients

Email 1	Name 1
Enter email address	Enter recipient name

Figure B.3: Sales report request setup form.

If you have already set up one or more reports to receive, you will be able to see them in the main screen, as Figure B.2 shows. The buttons under "Actions" allow you to manage these requests. The left, blue button allows you to edit the report, while the right, red button allows you to delete reports you don't need any more. Note that when you edit the report, you will be redirected to a screen that looks like Figure B.3, but pre-filled with its current settings.

Appendix C

Report Evaluation Questions

Form Items
Does the report include all the required sections?
Does the report actually address the required KPI?
Does the report include an analysis of the evolution of the KPI?
Does the report include detailed data, in addition to the high level KPI information?
Does the report include at least one graph?
Does the report include more than one graph?
Does the report include a projection of the KPI?
Is the data retrieved correct?
Does the report accurately present the last information period?
Is the content of the Executive Summary accurate to the section's description?
Is the content of the Overview accurate to the section's description?
Is the content of the Trends & Context accurate to the section's description?
Is the content of the In Depth Analysis accurate to the section's description?
Is the content of the Forward Outlook accurate to the section's description?

Table C.1: Checklist of Form Items

Content Items
Does the analysis in the report accurately represent the evolution of the KPI?
Does the report include an analysis of the detailed data?
Does the detailed analysis mention specific drivers of decline?
Are all graphs (visualisations) relevant to the report?
Are all graph types adequate for the data presented?
Are all projections adequate for the data presented?
Does the report accurately note any "special case" present?
Does the report provide next steps for any "special case" present?
Are all statements in the report sustained with data?

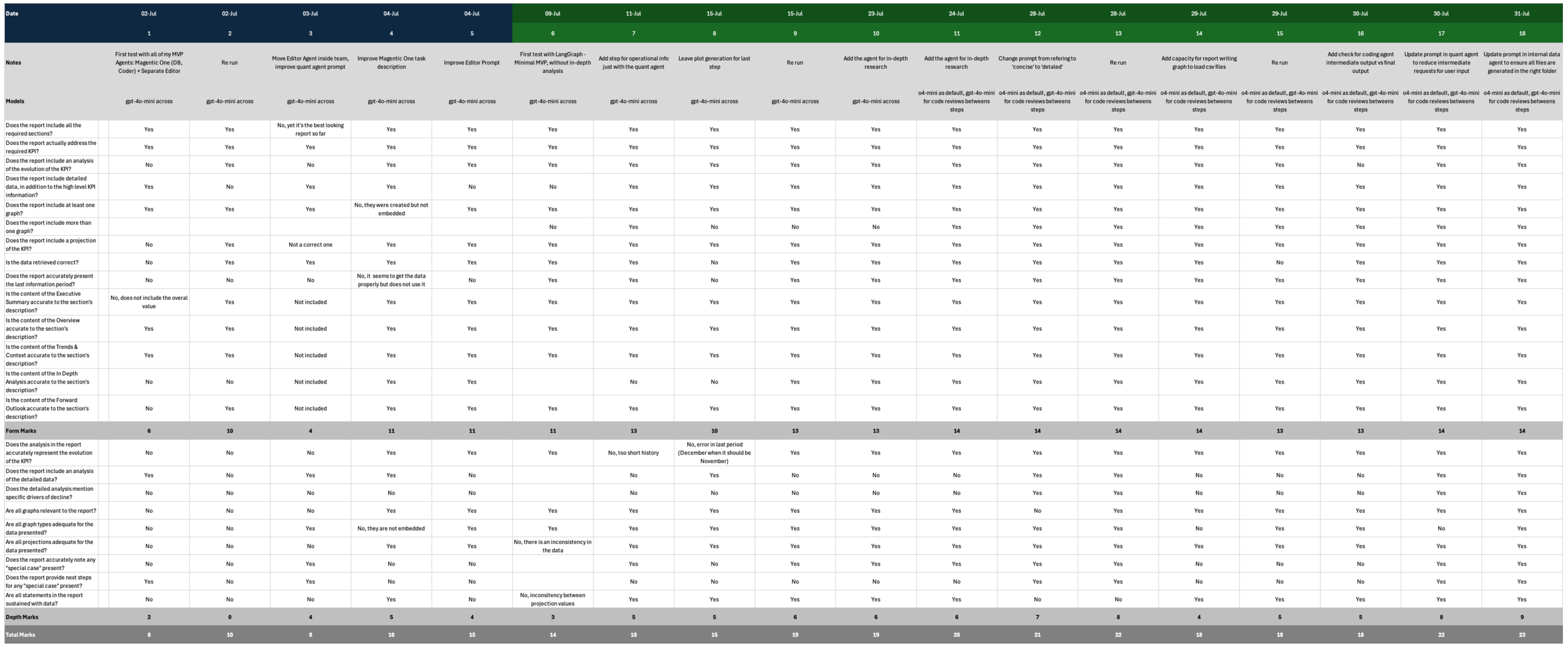
Table C.2: Checklist of Content Items

Appendix D

Experiments Performed and Evaluation Results

No.	Date	Notes	Default Model	Form	Content	Total
1	02-Jul	First test with all of my MVP Agents: Magentic-One (DB, Coder) + Separate Editor	gpt-4o-mini	50%	29%	36%
2	02-Jul	Re run of previous experiment	gpt-4o-mini	75%	29%	45%
3	03-Jul	Move Editor Agent inside team, improve quant agent prompt	gpt-4o-mini	50%	29%	36%
4	04-Jul	Improve Magentic-One task description	gpt-4o-mini	75%	71%	73%
5	04-Jul	Improve Editor Prompt	gpt-4o-mini	75%	64%	68%
6	09-Jul	First test with LangGraph — Minimal MVP, without in-depth analysis	gpt-4o-mini	78%	78%	78%
7	11-Jul	Add step for operational info just with the quant agent	gpt-4o-mini	100%	64%	78%
8	15-Jul	Leave plot generation for last step	gpt-4o-mini	67%	64%	65%
9	15-Jul	Re run of previous experiment	gpt-4o-mini	89%	79%	83%
10	23-Jul	Add the agent for in-depth research	gpt-4o-mini	89%	79%	83%
11	24-Jul	Add the agent for in-depth research	o4-mini	100%	79%	87%
12	28-Jul	Change prompt: ‘concise’ to ‘detailed’	o4-mini	100%	86%	91%
13	28-Jul	Re run of previous experiment	o4-mini	100%	93%	96%
14	29-Jul	Add capacity for report writing graph to load csv files	o4-mini	100%	64%	78%
15	29-Jul	Re run of previous experiment	o4-mini	89%	71%	78%
16	30-Jul	Add check for coding agent intermediate input vs final input	o4-mini	89%	71%	78%
17	30-Jul	Update prompt in quant agent to reduce intermediate requests for user input	o4-mini	100%	93%	96%
18	31-Jul	Update prompt in internal data agent to ensure all files are generated in the right folder	o4-mini	100%	100%	100%

Table D.1: Experiments performed with Evaluation scores



Date	02-Jul	02-Jul	03-Jul	04-Jul	04-Jul	09-Jul	11-Jul	15-Jul	15-Jul	23-Jul	24-Jul	28-Jul	28-Jul	29-Jul	29-Jul	30-Jul	30-Jul	31-Jul
	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18
Notes	First test with all of my MVP Agents: Magentic One (DB, Coder) + Separate Editor	Re run	Move Editor Agent inside team, improve quant agent prompt	Improve Magentic One task description	Improve Editor Prompt	First test with LangGraph - Minimal MVP, without in-depth analysis	Add step for operational info just with the quant agent	Leave plot generation for last step	Re run	Add the agent for in-depth research	Add the agent for in-depth research	Change prompt from refering to 'concise' to 'detaled'	Re run	Add capacity for report writing graph to load csv files	Re run	Add check for coding agent intermediate output vs final output	Update prompt in quant agent to reduce intermediate requests for user input	Update prompt in internal data agent to ensure all files are generated in the right folder
Models	gpt-4o-mini across	gpt-4o-mini across	gpt-4o-mini across	gpt-4o-mini across	gpt-4o-mini across	gpt-4o-mini across	gpt-4o-mini across	gpt-4o-mini across	gpt-4o-mini across	gpt-4o-mini across	o4-mini as default, gpt-4o-mini for code reviews betweens steps	o4-mini as default, gpt-4o-mini for code reviews betweens steps	o4-mini as default, gpt-4o-mini for code reviews betweens steps	o4-mini as default, gpt-4o-mini for code reviews betweens steps	o4-mini as default, gpt-4o-mini for code reviews betweens steps	o4-mini as default, gpt-4o-mini for code reviews betweens steps	o4-mini as default, gpt-4o-mini for code reviews betweens steps	o4-mini as default, gpt-4o-mini for code reviews betweens steps
Does the report include all the required sections?	Yes	Yes	No, yet it's the best looking report so far	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes
Does the report actually address the required KPI?	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes
Does the report include an analysis of the evolution of the KPI?	No	Yes	No	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	No	Yes	Yes
Does the report include detailed data, in addition to the high level KPI information?	Yes	No	Yes	Yes	No	No	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes
Does the report include at least one graph?	Yes	Yes	Yes	No, they were created but not embedded	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes
Does the report include more than one graph?						No	Yes	No	No	No	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes
Does the report include a projection of the KPI?	No	Yes	Not a correct one	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes
Is the data retrieved correct?	No	Yes	Yes	Yes	Yes	Yes	Yes	No	Yes	Yes	Yes	Yes	Yes	Yes	No	Yes	Yes	Yes
Does the report accurately present the last information period?	No	No	No	No, it seems to get the data properly but does not use it	No	Yes	Yes	No	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes
Is the content of the Executive Summary accurate to the section's description?	No, does not include the overal value	Yes	Not included	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes
Is the content of the Overview accurate to the section's description?	Yes	Yes	Not included	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes
Is the content of the Trends & Context accurate to the section's description?	Yes	Yes	Not included	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes
Is the content of the In Depth Analysis accurate to the section's description?	No	No	Not included	Yes	Yes		No	No	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes
Is the content of the Forward Outlook accurate to the section's description?	No	Yes	Not included	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes
Form Marks	**6**	**10**	**4**	**11**	**11**	**11**	**13**	**10**	**13**	**13**	**14**	**14**	**14**	**14**	**13**	**13**	**14**	**14**
Does the analysis in the report accurately represent the evolution of the KPI?	No	No	No	Yes	Yes	Yes	No, too short history	No, error in last period (December when it should be November)	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes
Does the report include an analysis of the detailed data?	Yes	No	Yes	Yes	No		No	Yes	No	No	No	Yes	Yes	No	No	No	Yes	Yes
Does the detailed analysis mention specific drivers of decline?	No	No	No	No	No		No	No	No	No	No	Yes	Yes	No	No	No	Yes	Yes
Are all graphs relevant to the report?	No	No	No	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	No	Yes	Yes	Yes	Yes	Yes	Yes
Are all graph types adequate for the data presented?	No	No	Yes	No, they are not embedded	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	No	Yes	Yes	No	Yes
Are all projections adequate for the data presented?	No	No	No	Yes	Yes	No, there is an inconsistency in the data	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes
Does the report accurately note any "special case" present?	No	No	Yes	No	No		Yes	No	Yes	Yes	Yes	Yes	Yes	No	No	No	Yes	Yes
Does the report provide next steps for any "special case" present?	Yes	No	Yes	No	No		No	No	No	No	No	Yes	Yes	No	No	No	Yes	Yes
Are all statements in the report sustained with data?	No	No	No	Yes	No	No, inconsitency between projection values	Yes	Yes	Yes	Yes	Yes	No	No	Yes	Yes	Yes	Yes	Yes
Depth Marks	**2**	**0**	**4**	**5**	**4**	**3**	**5**	**5**	**6**	**6**	**6**	**7**	**8**	**4**	**5**	**5**	**8**	**9**
Total Marks	**8**	**10**	**8**	**16**	**15**	**14**	**18**	**15**	**19**	**19**	**20**	**21**	**22**	**18**	**18**	**18**	**22**	**23**

Figure D.1: Detailed evaluation scores; empty cells are items that do not apply to that specific experiment.

Appendix E

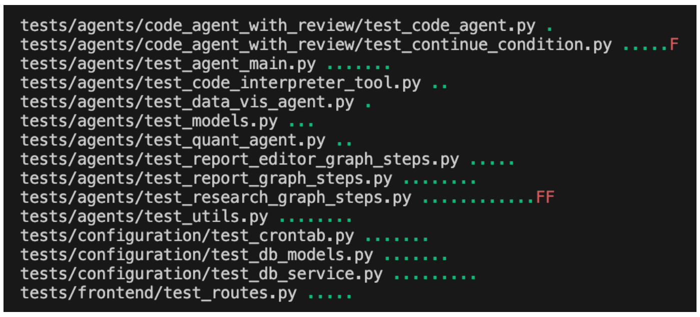
Sample Automated Evaluation

```
async def test_file_creation(quantitative_agent):
    """
    Test the agent creates at least one file in the temp directory.
    """

    query = f"""The company’s sales for the last three years are as follows:’

    {california_monthly_sales_in_db}

    Perform a detailed analysis of the sales data, including trends, patterns, and insights."""

    response = await quantitative_agent.ainvoke(messages=[HumanMessage(content=query)])
    response_content = extract_graph_response_content(response)

    from src.agents.utils.output_utils import get_all_files_mentioned_in_response

    files_mentioned = get_all_files_mentioned_in_response(response_content)
    try:
        # Assert that there is at least one file created in the temp directory
        assert len(files_mentioned) > 0, "No files were created in the temp directory."
        # Assert all files are csv files
        assert all(file.endswith(".csv") for file in files_mentioned), (
            "Not all created files are CSV files."
        )
        # Assert that the file actually exists
        for file_name in files_mentioned:
            file_path = test_temp_dir / file_name
            assert file_path.exists(), f"File {file_name} does not exist at {file_path}"
    finally:
        # Clean up the temp directory after the test
        for file in files_mentioned:
            file_path = test_temp_dir / file
            file_path.unlink(missing_ok=True)
```

Appendix F

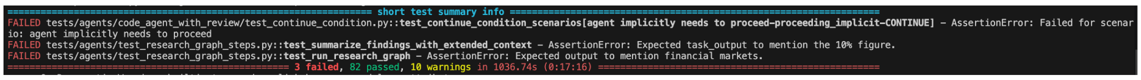
Automated Testing Suite Results

The following are the results of the last run of the testing suite. Please note that, as Section 5.2 notes, this includes both evaluations and actual test, and thus failing tests do not necessarily represent a system failure.



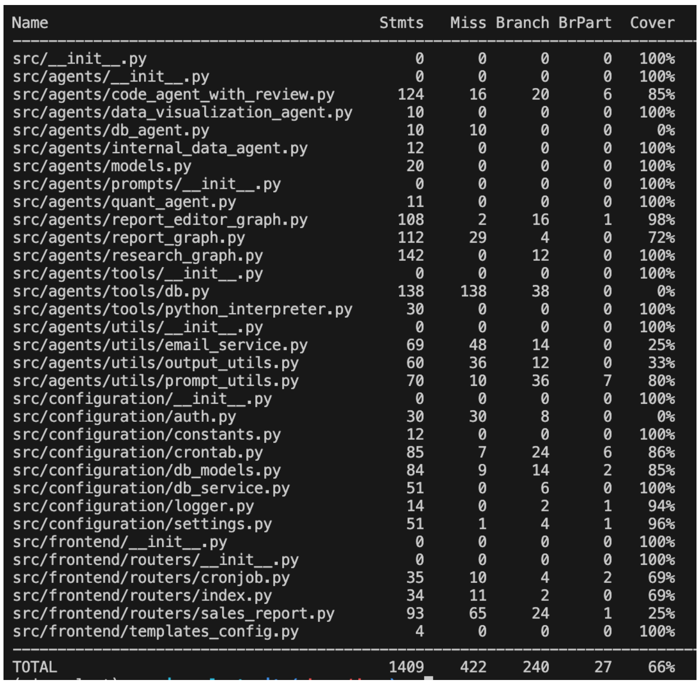
```
tests/agents/code_agent_with_review/test_code_agent.py .
tests/agents/code_agent_with_review/test_continue_condition.py .....F
tests/agents/test_agent_main.py .......
tests/agents/test_code_interpreter_tool.py ..
tests/agents/test_data_vis_agent.py .
tests/agents/test_models.py ...
tests/agents/test_quant_agent.py ..
tests/agents/test_report_editor_graph_steps.py .....
tests/agents/test_report_graph_steps.py ........
tests/agents/test_research_graph_steps.py ............FF
tests/agents/test_utils.py ........
tests/configuration/test_crontab.py .......
tests/configuration/test_db_models.py .......
tests/configuration/test_db_service.py .........
tests/frontend/test_routes.py .....
```

Figure F.1: Test results by test file



```
============================================================ short test summary info ============================================================
FAILED tests/agents/code_agent_with_review/test_continue_condition.py::test_continue_condition_scenarios[agent implicitly needs to proceed-proceeding_implicit-CONTINUE] - AssertionError: Failed for scenar
io: agent implicitly needs to proceed
FAILED tests/agents/test_research_graph_steps.py::test_summarize_findings_with_extended_context - AssertionError: Expected task_output to mention the 10% figure.
FAILED tests/agents/test_research_graph_steps.py::test_run_research_graph - AssertionError: Expected output to mention financial markets.
===================================================== 3 failed, 82 passed, 10 warnings in 1036.74s (0:17:16) =====================================================
```

Figure F.2: Test results summary and failures



```
Name                                      Stmts   Miss Branch BrPart  Cover
-----------------------------------------------------------------------------
src/__init__.py                               0      0      0      0   100%
src/agents/__init__.py                        0      0      0      0   100%
src/agents/code_agent_with_review.py        124     16     20      6    85%
src/agents/data_visualization_agent.py       10      0      0      0   100%
src/agents/db_agent.py                       10     10      0      0     0%
src/agents/internal_data_agent.py            12      0      0      0   100%
src/agents/models.py                         20      0      0      0   100%
src/agents/prompts/__init__.py                0      0      0      0   100%
src/agents/quant_agent.py                    11      0      0      0   100%
src/agents/report_editor_graph.py           108      2     16      1    98%
src/agents/report_graph.py                  112     29      4      0    72%
src/agents/research_graph.py                142      0     12      0   100%
src/agents/tools/__init__.py                  0      0      0      0   100%
src/agents/tools/db.py                      138    138     38      0     0%
src/agents/tools/python_interpreter.py       30      0      0      0   100%
src/agents/utils/__init__.py                  0      0      0      0   100%
src/agents/utils/email_service.py            69     48     14      0    25%
src/agents/utils/output_utils.py             60     36     12      0    33%
src/agents/utils/prompt_utils.py             70     10     36      7    80%
src/configuration/__init__.py                 0      0      0      0   100%
src/configuration/auth.py                    30     30      8      0     0%
src/configuration/constants.py               12      0      0      0   100%
src/configuration/crontab.py                 85      7     24      6    86%
src/configuration/db_models.py               84      9     14      2    85%
src/configuration/db_service.py              51      0      6      0   100%
src/configuration/logger.py                  14      0      2      1    94%
src/configuration/settings.py                51      1      4      1    96%
src/frontend/__init__.py                      0      0      0      0   100%
src/frontend/routers/__init__.py              0      0      0      0   100%
src/frontend/routers/cronjob.py              35     10      4      2    69%
src/frontend/routers/index.py                34     11      2      0    69%
src/frontend/routers/sales_report.py         93     65     24      1    25%
src/frontend/templates_config.py              4      0      0      0   100%
-----------------------------------------------------------------------------
TOTAL                                      1409    422    240     27    66%
```

Figure F.3: Test coverage output

Appendix G

Generated Report — Sample