UNIMORE
UNIVERSITÀ DEGLI STUDI DI
MODENA E REGGIO EMILIA

UNIVERSITÀ DEGLI STUDI DI MODENA

E REGGIO EMILIA

Dipartimento di Ingegneria "Enzo Ferrari"

Corso di Laurea Magistrale in Artificial Intelligence Engineering (LM-32)

From Proto-Insights to Validated Business Insights: An LLM-Based Multi-Agent Architecture for Insurance Analytics

Relatore:
Prof. Simone Calderara

Candidato:
Riccardo Reale
Matricola 196412

ANNO ACCADEMICO 2025/2026

Contents

List of Figures

List of Codes

1. Introduction

1.1 Research Context and Motivation

In recent years, thanks to the evolution of Artificial Intelligence methods and the widespread adoption of Large Language Models (LLMs), the opportunities in the field of data analysis and business intelligence have changed dramatically. Nowadays, the insurance industry can use increased data availability for a better comprehension of risk phenomena and more efficient data-driven processes.

Specifically, as far as Motor Third Party Liability Insurance (MTPL) is concerned, insurance companies can access plenty of data about policies, claims, customers, and insured vehicles, which is used as the source material for activities like pricing, underwriting, and portfolio management. Among the most important analyses carried out in this respect, there is the identification of relevant patterns within portfolios due to the strong competition and the information complexity of the environment.

Several techniques have been used for discovering correlations and recurring behaviors in the data and for the segmentation of portfolios, including data mining algorithms like clustering, association rule mining, and exploratory analysis. As a result, a large number of patterns were generated and evaluated. However, the results achieved in this way should not be considered valid insights but mere signals.

The key issue here is the fact that a correlation, considered by itself, is not sufficient to qualify as knowledge. On the one hand, there might be some underlying factors that are responsible for the apparent correlations. On the other hand, it might be that those patterns arise from limited samples of the data, just enough to become statistically significant. These reflections are very common when speaking about unsupervised techniques since their results are usually not easy to interpret.

Consequently, the major challenge here is represented by the fact that, despite the availability of pattern discovery techniques able to generate a large number of proto-insights from the data, the conversion of those findings into meaningful and actionable business insights to support the decision-making process is difficult to achieve.

1.2 Research Problem

Generative AI's greatest capabilities in recent years have proven to be reasoning, text generation, and external tool orchestration. Indeed, with the paradigms of agentic AI and tool-augmented reasoning, it has become possible to build systems not only able to generate natural language text but also capable of interacting with databases, analytical tools, and even complex computational pipelines.

These capabilities are crucial to the field of insurance analytics because generating insights from data requires:

- semantic understanding of the domain;
- ability to interpret data and carry out quantitative validation.

As is known, the major drawback of Generative AI technology is the production of errors and false information, i.e., hallucinations, which leads to drawing unjustified inferences. From a control perspective, it is difficult to fully understand the reasoning process of these models in particularly information-sensitive areas, such as insurance.

In this regard, the central research problem of this thesis arises: the transformation of proto-insight into validated and meaningful insights. By proto-insights, this thesis refers to signals directly obtained from data mining algorithms, which are statistically described by pattern-level metrics but are not yet interpreted, validated and refined as business insights.

Therefore, besides the automatic discovery of correlations in the data, the challenge here is represented by the necessity to:

- generate analytical hypotheses compatible with the domain;
- quantify the importance of the phenomenon identified;
- support traceability and controllability of the refinement process;
- avoid arbitrary or unjustified inference.

1.3 Thesis Objectives

The aim of this thesis is the design and development of a schema-aware agentic system that is able to automatically convert proto-insights obtained from unsupervised techniques into validated

business insights in the context of insurance analytics.

From the architectural point of view, the proposed system is designed around a clear separation between:

- the generation/refinement of the analytical hypotheses component, referred to as Insight Hunter Agent (IH);
- the quantitative/statistical validation of the generated hypotheses component, referred to as Data Analyst Agent (DAA).

Hence, the multi-agent paradigm implemented through the Strands Agents SDK framework is particularly suitable for this scenario because it supports tool-based interaction, agent state management and improved traceability during the refinement process.

Specific objectives of the thesis include:

- the design of an agentic process for automatically carrying out insight refinement;
- the definition of suitable YAML files containing enough information for proper semantic grounding;
- the development of prompt engineering techniques to be applied iteratively and in a systematic way;
- the use of batch-based reasoning by the IH over groups of proto-insights;
- the use of SQL-based validation for data retrieval and statistical analysis by the DAA;
- the use of various methods to limit hallucinations and find a balance between creativity and actionability.
- the implementation of logging and traceability mechanisms for inspecting the refinement iterations and tool interactions;

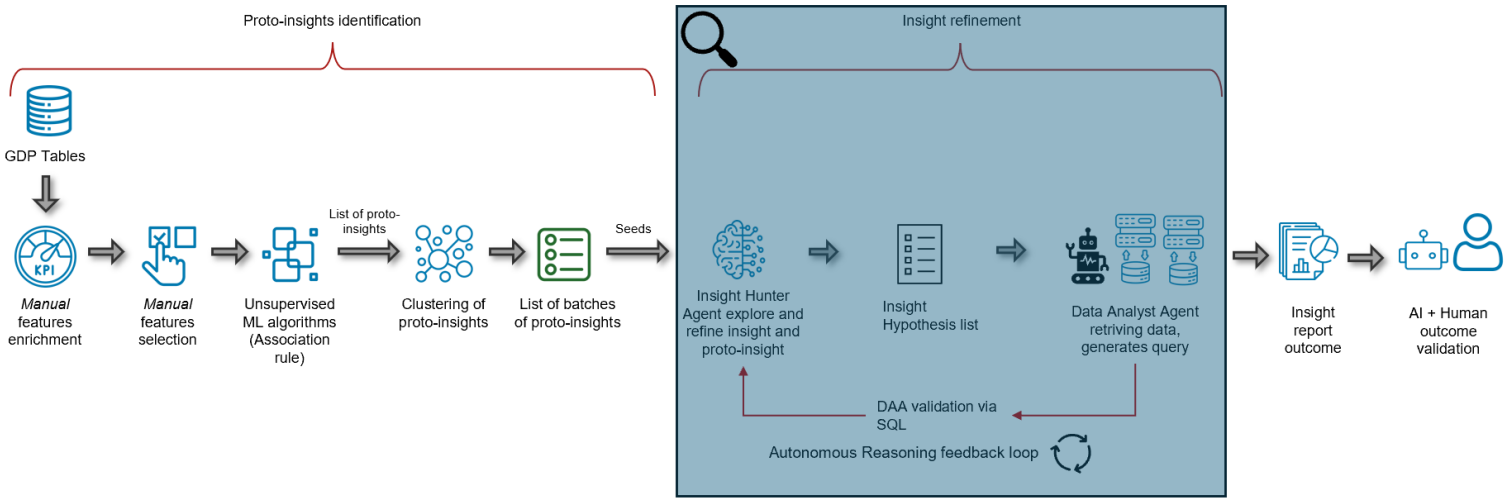

Figure 1.1: The workflow shown in the picture represents the process of transforming raw data and generating proto-insights through unsupervised machine learning algorithms, up to the structuring of insights validated and refined by the agentic system.

Note: The highlighted section represents the part of the workflow on which this thesis is primarily focused.

2. State of the Art

2.1 Pattern Mining and Proto-Insight Modelling

Chapter 1, introduced some machine learning techniques like association rule mining and clustering. Such techniques belong to the class of unsupervised learning algorithms and do not have any predefined labels or targets in their learning process. The main goal of theses approaches consists of automatic identification of some latent patterns.

- Clustering allows us to identify homogeneous groups of observations with similar properties, therefore pointing out hidden structures in data. In this thesis, and in general in insurance, it is used for portfolio segmentation, behavioral profiling or identification of various risk groups.

- Association rule mining involves automatic identification of statistical dependencies among the variables. Rules extracted with this algorithm look like:

$$X \Rightarrow Y \tag{2.1}$$

This rule means that a combination of events X and Y is statistically correlated. The quality of rules is commonly assessed according to support, confidence, and lift metrics.

Support measures how frequently events X and Y occur together:

$$\text{Support}(X \Rightarrow Y) = P(X \cap Y) \tag{2.2}$$

Confidence measures the probability that event Y occurs given that event X has occurred:

$$\text{Confidence}(X \Rightarrow Y) = \frac{P(X \cap Y)}{P(X)} \tag{2.3}$$

Finally, lift measures the strength of the association between events X and Y compared with what would be expected under statistical independence:

$$\text{Lift}(X \Rightarrow Y) = \frac{P(X \cap Y)}{P(X) \cdot P(Y)} \tag{2.4}$$

In the field of insurance, the lift is actively controlled as lift greater than 1 means positive correlation between two events whereas the values below 1 denote negative correlation.

These metrics describe the statistical interestingness of a rule. However, they do not prove that the pattern is stable, causal, or relevant from a business perspective.

For the purposes of the current thesis, these techniques are used for extracting statistical relations between insured people based on particular features like car properties, geographical areas, payed premium, and performance criteria such as claim frequency, severity, or loss ratio.

Among the main limitations of the unsupervised pattern mining techniques for decision-making, we should name business-related aspects. The first limitation is concerned with the issue of interpretation. Extracted rules demonstrate statistical dependency between particular variables, but they give no guarantee concerning causality or practical significance of the discovered pattern.

Another difficulty consists of the facts that many statistical patterns could be caused by:

- spurious correlations;
- confounding variables;
- inadequate sample size of particular group;
- statistical noise.

Therefore, statistical patterns cannot be considered reliable sources of business knowledge.

Table 2.1: Comparison between statistical patterns, proto-insights, and validated insights.

Level	**Description**	**Main Limitations**
Statistical Pattern	Association or correlation automatically detected in the data	Lack of interpretation and validation
Proto-insight	Preliminary signal potentially relevant to business	Requires refinement and quantitative verification
Validated Insight	Information interpreted, verified, and contextualized	Requires a complex analytical pipeline

Translating proto-insights to validated insights becomes a non-trivial task and it cannot be solved with classical unsupervised techniques alone.

2.2 Large Language Models

Nowadays, Large Language Models (LLMs) become more and more relevant components in contemporary Artificial Intelligence systems owing to the possibility of performing operations with natural languages and reasoning. The success of the large language models of our time is associated with their impressive performance and usage in companies providing analytics services; among those models, we can name GPT, Claude, and Gemini.

From the point of view of their functioning mechanisms, LLMs are based on Transformer architecture designed by Vaswani et al. in the famous *Attention Is All You Need [1]* paper published in 2017. The revolutionary nature of the architecture does not consist only in it but also in the possibility of introducing self-attention that replaces previous sequential models and deals with distant word-to-word relations.

Self-attention mechanism can be mathematically described with the use of the following equation:

$$\text{Attention}(Q, K, V) = \text{softmax}\left(\frac{QK^T}{\sqrt{d_k}}\right)V \tag{2.5}$$

where (Q), (K), and (V) are the query, key and value matrices, and (d_k) stands for the number of key vectors' dimensions.

Widely spread and generalizing capabilities of GPT and Claude come from pre-training methods implying huge amounts of text data necessary to create LLMs' ability of understanding semantics, reasoning and code generation, planning, etc. It should be pointed out that the application area of the models exceeds their natural language management because more recent systems may also be integrated with multimodal capabilities or external tools, but this thesis focuses primarily on their role in language-based reasoning, tool use and analytical hypothesis formulation.

Further developments such as large-scale few-shot language modelling [2] and instruction tuning with human feedback [3] contributed to improving the ability of LLMs to follow instructions, interact conversationally and adapt to different tasks.

In terms of insurance analytics, LLMs are very useful when dealing with situations of

interpretation of multiple data and understanding of its context and formulation of analytical hypotheses.

Hallucinations, discussed in the introduction chapter as the creation of facts that seem true but lack any evidence from the provided data, represent a critical point in industries requiring data interpretation because the problem of excessive reliance of human beings on LLMs becomes topical, and human intervention is needed to prevent any possible problems.

Moreover, LLMs gain knowledge during the process of training and do not include the information regarding corporate structures and their characteristics. It resulted in the creation of strategies focused on integrating LLMs with external tools and giving rise to the era of Tool-Augmented Reasoning and Agentic AI.

2.3 Agentic AI and Tool-Augmented Reasoning

2.3.1 From Large Language Models to Agents

In the classic LLM setup, the model gets a prompt and produces a response. As a consequence, the whole pipeline involves one generation phase during which the model interprets the provided context and produces some output.

The move from pure LLMs towards the concept of agents is a notable step in the development of artificial intelligence, especially in the domain of enterprise analysis and operations.

Under the agentic paradigm, the language model itself is just a part of a greater whole. The task is no longer simply generating text, but reaching some goal. To do so, the model has to reason within the evolving context, invoke tools where appropriate, and integrate newly acquired information into the existing knowledge base. The intelligence of the system relies not only on the reasoning capabilities of the model itself, but also on the procedures that let the agent update its state and make decisions based on the observations and results.

The above-mentioned distinction is precisely what separates agents from pure language models. While a typical LLM is designed to generate text in response to prompts, an agent is meant to perform more elaborate tasks that involve reasoning over several steps. As discussed before, an LLM can only rely on the information available in its training data. Agentic reasoning provides a solution to this problem by giving the model access to external tools. With those, an agent can reason over company databases, run empirical validation, execute database queries, retrieve information from external sources, or search the web. An agentic framework can thus

be considered to be superior to a standalone model because it allows the latter to reason beyond its parameterized knowledge.

Another defining property of an agentic framework is that of iteration. At each iteration step, the agent analyzes its current state, reasons about the goal, decides which tools are needed, and invokes them accordingly. Upon receiving results, the agent changes its context and updates its state. If there is no explicit stop condition, it repeats the described process until the goal is met.

It is important to understand that the notion of an agent refers to various systems with specific purposes that can be defined via certain design decisions. One of those is the system prompt which defines the purpose of the agent and constraints on its behavior. Second, the agent has access to a variety of tools which let it perform queries, fetch information, execute code, etc. Finally, the agent has a state that keeps accumulating information at each iteration step.

For the purposes of the current thesis, an agentic framework is absolutely crucial. It allows the model to analyze proto-insights, reason about them with the help of its built-in tools, formulate hypotheses, and validate them using external tools. As will be explained later, this is one of the key aspects that enables systematic insight refinement.

2.3.2 Reasoning patterns: Chain-of-Thought, ReAct, and ReWOO

With the advent of agents, attention has been devoted to the development of new reasoning strategies.

One of the earliest works is the paper *Chain-of-Thought Prompting Elicits Reasoning in Large Language Models, written by Wei et al. [4].*

It describes the role that the language model plays in the process, as well as how the reasoning type and relation to external tools affects the outcome. Chain-of-Thought is a strategy that highlights how intermediate steps make reasoning more effective than one-shot reasoning, wherein the output of the model is provided immediately. Therefore, in use cases where the complexity of the problem needs to be evaluated holistically, the impact of this method on large models is significant. The prompt is not anymore something requested by the model, but becomes the direction of the reasoning. In addition, Chain-of-Thought makes clear how the design of reasoning patterns becomes one of the core components of the agentic framework.

The only problem of Chain-of-Thought is the fact that it still is an internal reasoning strategy, that does not take into account external factors. Indeed, the logic is based solely on the information contained in the prompt and in parametric knowledge; in addition, it does not

interact with any external tool, and, consequently, cannot check results. Thus, it is crucial to create a reasoning architecture capable of accessing new information and checking data against independent sources.

One of the solutions to this issue is the introduction, in another paper, of a new reasoning paradigm called ReAct, described by Yao et al. in *ReAct: Synergizing Reasoning and Acting in Language Models [5]* .

The proposed reasoning pattern can be expressed in the following loop:

$$\text{Observation}_t \rightarrow \text{Thought}_t \rightarrow \text{Action}_t \rightarrow \text{Observation}_{t+1} \tag{2.6}$$

Here, the agent observes the current situation, performs intermediate reasoning, executes an action, and produces a new observation, on which it will again conduct reasoning and follow the same loop, until a certain end condition is met. Of course, this pattern proves to be more flexible and observable, compared to Chain-of-Thought, but also more accurate, since it gives to the model the possibility to self-correct the outputs.

Consequently, the main innovation that was introduced is the more intimate connection between reasoning and tools, that now becomes a core step of reasoning. The context is updated with observations, and information external to the prompt is not anymore a limitation. Tests performed using this logic proved to be more effective on tasks such as question-answering, fact-verification, and interactive decision-making, where alternating between reasoning and action makes the process more interpretable.

ReAct turns out to be the reasoning pattern best suited to our purposes, as it underlies the principle of refinement. The Insight Hunter, indeed, does not interpret a proto-insight using a unique reasoning step, but alternates reasoning, call to external tools, observation, and refinement in a loop.

Another approach to reasoning is the *ReWOO pattern, Reasoning Without Observation [6].*

What changes with respect to ReAct is the reasoning-to-tool relation. With ReWOO, a reasoning pattern is first developed, with placeholders; then, the placeholders are replaced by actions performed and new observations inserted by the tools, after they were called by the model.

The limitation addressed with this approach is the constant switching between reasoning and calls to external tools, characteristic of ReAct. Although the model becomes very flexible, the unpredictable growth of the context and the continuous calls to tools increase the cost in terms

of tokens. This is prevented in ReWOO thanks to the distinction between the planning and the tool execution phases; thus, the model becomes much more efficient and modular, since there is the possibility to anticipate what information it will need.

For the purposes of this thesis, ReAct is more suitable than ReWOO because the refinement process requires repeated alternation between reasoning, tool use and observation. To correctly refine the proto-insight produced by the Insight Hunter, indeed, it is necessary to alternate reasoning and tool usage, and observe the results. It will be explained in detail how the computational costs were balanced with proper prompting techniques and particularities of agentic frameworks, making ReWOO dispensable.

The aforementioned reasoning patterns represent the conceptual basis of agentic refinement.

2.3.3 Tools, Tool-Augmented Reasoning and Function Calling

As already said, tools constitute one of the distinctive elements of an agent, compared to an LLM. Indeed, the limitations imposed by pre-training the model have led researchers to develop agents that complete tasks using external tools, in order to overcome their constraints: these include recognizing the need for such tool, selecting the right one, requesting and executing its function, getting a result, and performing the expected operation.

An external tool is usually an external function that provides a model with a certain service. For instance, this can be a query to a database, interpreting a script, or a call to an API endpoint. As regards our case study, it involves interacting with another agent.

The model does not act directly in all these tasks; indeed, it invokes an external tool that executes the corresponding operation. Each tool consists of two parts: the definition and the execution.

The definition corresponds to the part of the tool seen by the model; it is necessary to provide an instruction on how the tool should work. Usually, this involves structuring the tool, e.g. with a JSON schema; then, the model requests a query compatible with this structure, after having observed the name of the tool and its functionality, together with a schema of the inputs needed. Indeed, this is needed to prevent cases in which the model produces invalid queries.

Then, the execution corresponds to the actual action taken by the tool. ReAct described the importance of observing the environment in order to reason about the next step, and perform actions. Once a request with the name of the tool and the inputs needed have been created, the model executes it, checks the correctness, and returns a new observation of the environment.

The advantage of separating the model from the execution environment is that Chain-of-Thought keeps the reasoning process inside the model, while tool-augmented reasoning separates the model decision from the external operation executed by the tool. In addition, the use of an input schema makes the process more controllable and easy to execute; indeed, it avoids heuristic parsing, which could be risky, and makes tracing possible.

The limitations of the use of pure parametric knowledge emerge in situations that involve the application of knowledge belonging to other areas. For example, in *Toolformer: Language Models Can Teach Themselves to Use Tools [7]*, we can read how a language model can learn to use tools to make calculations, answer questions, use search engines, translate text and check dates. What is relevant here is not the specific procedure used to train the model, but the use of external tools to acquire knowledge that is otherwise unobtainable.

Similarly, other papers have underlined the central importance of integrating the use of tools to enable LLMs to execute more advanced operations. In particular, we refer to *Tool Learning with Large Language Models: A Survey [8]*, in which tool learning is explained using a taxonomy of different phases, which includes: task planning, tool selection, tool calling, and response generation. This precise organization of the process demonstrates how an agent must not only understand the nature of the task, but also find and select the right tool, perform the correct invocation, and use its result appropriately.

Other recent articles have focused precisely on function calling, which implies dealing with the interaction between the language model and the external system, which may prove to be difficult. Indeed, in papers such as *ToolACE: Winning the Points of LLM Function Calling [9]*, the attention is directed towards the importance of the quality of the data and of the infrastructure to support complex and accurate function calling.

The critical issues in the use of external tools arise during the initial phase, i.e. when they are presented to the model. Indeed, the model can mistakenly perform operations because it invokes them with invalid input or too soon. This problem can derive from the similarity between the different tools, or from ambiguities regarding their names, leading to cascading errors and unintended actions. Unreliable data may also be introduced to the context by the use of tools; for this reason, their design needs to include safety measures, such as guardrails, whitelists, validation of inputs, and permission controls.

In this work, external tools play an essential role in the process, as they establish the link between linguistic reasoning and verifiable operations needed to generate insights.

2.3.4 Agent Harness and Control Infrastructure

Apart from the language model and tools themselves, there are yet some other things to be taken into account while designing a LLM-based agent, namely, agent harness and control infrastructure. As the name suggests, the latter defines what the model does, what tools it uses, what information goes into the context, what is done with outputs and even under what circumstances the agent stops.

In other words, agent harness stands for the layer between the language model and its environment. It covers a vast range of tasks that relate to observing the conversation, imposing limitations, managing context, controlling the flow of the conversation, etc. This includes system prompts, tools, tool definitions, context management, run/conversation status, guardrails, output verification and shutdown.

Taking all this into account, we may provide the following definition of agent harness. Agent harness refers to a part of the system that relates to executing the agent within the application. As it follows, the major task of harnessing is to ensure the execution of the system by connecting the model with tools, data, limitations and verification rules. Needless to say, apart from making inferences, it is also important to control their channel and scope.

It might also be helpful to discuss the notion of evaluation harness. Contrary to the former concept, it refers to analysis and comparison of the agent and/or model within the application. Whereas agent harness is aimed at making the agent function, the aim of the evaluation harness is assessing the performance, accuracy and other characteristics of the agent compared to others.

Similar to how it is outlined by Xi et al. in the survey *The Rise and Potential of Large Language Model-Based Agents [10]*, language model in an agentic system might be treated as the brain of the agent. Nevertheless, the agent's brain is not capable of working alone; instead, it forms a bigger system consisting of the model, the perception of environment and interaction with it. It is also worth mentioning other aspects of the agentic system, including cognition, planning, use of tools and reaction to environmental factors. All these facts prove the idea that an agent and its model are two different components of one system.

Having said that, it should be stressed that several aspects are important regarding agential systems. First of all, the model in agentic system plays an important role as far as reasoning is concerned: interpretation of language, comprehension of the task, hypothesis formation and action selection are all performed due to this model. Nevertheless, the use of different models may lead to completely different agent's behavior depending on the environment. For instance,

a bigger model utilizes a greater amount of context and is capable of doing a more complex reasoning; however, it also costs more than a smaller model. The latter, in turn, requires precise constraints, good tools and a more detailed context regulation.

As it is apparent from the foregoing, another important characteristic of the agentic system is the balance between its model, harness and task.

The next limitation in question is context management. Given that agents must comprehend the conversation, they use a limited context window consisting of system prompt, messages, tool definitions, call results, input from the user and the information obtained from the environment. This fact implies that one of the most pressing problems nowadays is developing appropriate strategies of compacting the context via summarization or deleting the information. The process seems quite easy in case of a complex task; otherwise, lack of strategies may result in poor quality of reasoning. Meanwhile, too much or poorly arranged context may produce quite the opposite effect.

Another limitation, related to agentic system, refers to tool registry and whitelist. Specifically, there is a huge difference between an agent that can modify the data stored in the database and an agent which can merely view the schema of this data. That is why it is important to pay attention to tools that are provided, described, permitted and their parameters. It is advisable to apply the principle of least privilege: the agent should be allowed to use only those tools that are absolutely necessary for performing the task.

Guardrails constitute the next important aspect of agential system. Just as it was with the previous limitations, guardrails are meant to constrain agent's behavior before, during and after generation of information. These limitations might refer to input, output and tool calls. Examples of guardrails include request blocking if it is out of the domain, preventing unauthorized tool calls, validating function arguments, ensuring the absence of sensitive data and requiring response validation. Nowadays, it is rather easy to install guardrails on the tool itself by performing validation before the execution of a tool and after it.

Guardrails are especially relevant when dealing with prompt injection as it becomes possible in the system based on language models. If the user tries to include sensitive data or instructions in the prompt, the agent might be tricked into ignoring them, calling wrong tools and sharing sensitive data.

Another aspect that is extremely important in the context of the present research is verifications. Indeed, an agent's ability to solve complex tasks is just the beginning; there must

be a verification system that will confirm the results obtained. There are numerous types of verification, which include checks for argument schema, fact-checking, comparing results to retrieved documents, queries and statistics or human review. Such verifications are of critical importance in enterprise environment as the agent outputs may influence strategic or economical decision-making process.

Verification might be considered another type of harness as it ties the agent to reality. The problem with agents that have to generate something creative without being determined by anything is that linguistic plausibility gets mixed with empirical validity. This is why it is especially important to use verification tools or develop special models/agents dedicated to this purpose.

As it was demonstrated in the present thesis, a lot of iterations are required for accomplishing the task by a lot of agents. Naturally, it sometimes becomes important to put limits to budget or time of task execution as well as preventing an infinite loop. One may set a strict limit such as a certain amount of iterations and impose a more flexible restriction such as the absence of further evidences in further calls. Thus, this aspect also illustrates the necessity of using agent harness as a tool for controlling the degree of restriction.

All this allow to understand that the quality of agent depends not on one but several factors. More specifically, it is determined by the balance between the features of the chosen language model and the structure of the harness.

Again, it must be emphasized that such a balance is not universal. A flexible harness helps the agent be creative and autonomous, but it may also create conditions for the emergence of hallucinations and other erroneous behavior. Conversely, a rigid harness eliminates errors, but it makes impossible to investigate non-obvious paths. Hence, when designing an agent, it is necessary to choose between freedom and restrictions, creativity and reliability, explorability and control.

2.3.5 State, Memory, and Feedback

To carry out a complex task, the LLM agent requires some form of memory to keep all necessary information that could be used to make decisions. Particularly for complicated tasks, multiple reasoning steps, tool calls, and verification are required so that the agent could remember and evaluate its own actions and reasoning process. In this case, it is important to distinguish between state and memory. State refers to the current state of the operation, meaning information gathered

throughout the process of the specific run, i.e. the initial prompt, messages received, tool calls made, answers received, hypotheses, etc.

Conversely, memory stands for the past experience that is stored so that it could be recalled anytime in the future. This distinction is important because some pieces of information, such as recent validation results and the list of attempted hypotheses, belong to the former category. Other information, instead, is meant to be stored more permanently, such as domain knowledge, reusable feedback, and lessons learned from previous runs.

Notably, in the paper *A Survey on the Memory Mechanism of Large Language Model-Based Agents [11]* one could find more comprehensive discussion on memory types and roles played by them.

Specifically, there one would find more information concerning distinctions between short-term and long-term memory. The first kind is related to model context window, recent messages, and run status, while the latter one relates to information gathered through additional means, such as databases, vector stores, document stores, and retrieval systems. As was mentioned above, such distinction is important for addressing the problem of the model context window limitation that prevents storing all the information produced by the agent.

Apart from the above, it is important to mention that feedback plays a critical role in improving the agent's performance. Here, the feedback is provided by the environment, tool, verification function, another agent, and people, both as participants in communication and evaluation of the process outcomes. Here, as would be demonstrated in this thesis, the improvement process is influenced by many factors that might seem confirming a certain trend from a certain angle yet also need to be considered a potential cause for failure.

As for the work done by Shinn et al., entitled Reflexion: *Language Agents with Verbal Reinforcement Learning [12]*, here the authors discuss the idea of reflexion and introduce the novel paradigm when an agent could develop its own behaviors not by upgrading its model weight but through verbal feedback and episodic memory. There, agents would reflect on their actions and then save this reflection into the memory buffer to be used later.

Last but not least, it is important to point out distinctions between stateful agents capable of creating a reasoning trajectory based on observations and stateless agents capable of producing just one-shot response to the prompt inappropriate for complicated tasks. Although there is no guarantee of success for stateless agents, one needs to consider the process of designing memory and state along with the rest of components of the harness system.

2.3.6 Multi-Agent Systems and Role Specialization

Multi-agent systems represent a further evolution of agents based on Large Language Models.

So far, it has been demonstrated what the key components that characterize an agent are. However, it is sometimes necessary to build a multi-agent infrastructure capable of solving complex tasks that require heterogeneous and transversal skills. Planning, generation, verification, and synthesis are just some of the potential capabilities that, if entrusted to a single agent, can lead to a drop in performance or simply the failure of the task itself. For this reason, multi-agent systems are designed to cooperate together, each specialized in its own area, to achieve the final goal. "Cooperate" means that each agent maintains a certain degree of autonomy and that overall behavior emerges from the interaction between the components.

A second key word is "interaction", as the added value of a MAS (Multi-Agent Systems) depends not simply on the number of agents but on the way they communicate and distribute responsibilities. This information exchange is achieved through messages, tool calls, state sharing, specialized roles, or orchestration mechanisms.

The multi-agent paradigm therefore addresses the problem of task complexity among multiple components by assigning roles, so each agent has its own dedicated prompt and a set of tools for communicating with other agents and the outside world. In this way, complex workflows can be broken down into more interpretable interactions between specialized components.

An important reference in this direction is AutoGen, proposed by Wu et al. in their paper *AutoGen: Enabling Next-Gen LLM Applications via Multi-Agent Conversation* [13]. From this paradigm, a modular, scalable, and controllable system can be derived. The paper explains how LLMs can be used to structure conversations between agents, for example through message exchanges, code execution, and output criticism, enabling collaboration towards the completion of complex tasks.

The resulting architectures are mainly divided into centralized and decentralized structures. In centralized systems, we typically have an orchestrator that decides which agent should perform a subtask, collect and combine results, manage operations, errors, and much more; the only drawback is that it places significant responsibility and dependency on a single agent.

In decentralized systems, however, there is no single orchestrator who decides everything. Therefore, cooperation and interaction with the outside world is managed by individual agents, achieving greater robustness and modularity but increasing the complexity of coordination and little control over the workflow.

Beyond this distinction, there are hierarchical structures, teams, coalitions, or more dynamic organizations, in which agents can collaborate permanently or temporarily depending on the task at hand.

In a context like insurance analytics, the objective is to move toward a controlled role-specialized structure, where the Insight Hunter acts as the central refinement agent and the Data Analyst Agent provides empirical validation. Each component can be designed and tested according to its specific responsibility, improving traceability because it is easier to determine which component produced a given intermediate result and which constraints or permissions should be adjusted.

Multi-agent systems are susceptible to errors when there's a dense exchange of messages, interfaces to manage, and numerous state transitions. The system can become more difficult to orchestrate, debug, and evaluate. This cascade could lead to duplications, inconsistencies, and the loss of relevant information, but above all, it could result in unpredictable costs for additional model calls, context updates, and, in some cases, new calls to external tools.

When developing agents that require creativity, architecture plays a fundamental role. One or more agents may interpret the same objective differently, potentially conflicting, and if they share the same foundation model, the same weaknesses may recur at different points in the workflow. For this reason, multi-agent design requires a balance between specialization and simplicity.

This principle is very important for the work presented here. Transitioning from proto-insights to validated business insights requires both interpretive reasoning and empirical validation. At the implementation level, this leads to a logical and semantic division of labor: the first phase requires domain knowledge and the ability to formulate hypotheses, while the second requires data access and statistical verification capabilities. In the workflow illustrated in 1.1 applies the multi-agent principle to this specific context, using role separation to make the refinement process more controllable and traceable.

2.4 Agentic Frameworks: LangChain, LangGraph and Strands Agents SDK

2.4.1 From Agentic Principles to Frameworks

The previous sections discussed LLM-based agent systems, the interaction between the model, prompts, tools, state, memory, feedback, control mechanisms, and, in more complex cases, multiple specialized agents. The following section dives deeper into these aspects and illustrates concretely how this interaction occurs and the software infrastructure capable of managing the agent's operational cycle.

Frameworks are therefore needed that provide abstractions and components capable of connecting LLM with external tools, modeling the system's behavior by acting on system prompts, context, tools, run state, etc.

The use of the right framework depends on the use case, but from a general perspective, it provides technical support for replacing human intervention on specific recurring mechanisms that agent systems require. For example, to obtain the output, one must consider the information being observed, the tools, the interpretation of the results, stop conditions, and much more. Without an adequate framework, this type of activity would have to be managed manually, increasing the risk of errors and difficulty tracking.

When talking about framework types, reference is generally made to those oriented towards a broad ecosystem of reusable components, useful for rapidly building LLM-based applications. Others, however, are more explicit and controllable, allowing the designer to manage process transitions and states at a lower level. Still others are more model-driven, granting the model greater autonomy and responsibility in the decision-making process, while remaining within the constraints established by the designer. This last case is the one covered in this thesis. Insight Refinement, by definition, requires an exploratory component that starts primarily from the model, without too many rules imposed by the designer; only later, during the validation phase, is a more deterministic approach established. For this reason, the framework used must support both autonomy and control.

In the first phase of the project, the primary task was to find the most suitable framework. The references were LangChain, LangGraph, and Strands Agents SDK. These tools are real ways to design LLM-based systems.

2.4.2 LangChain

Langchain is a well-known framework for building LLM-based agentic systems. As mentioned previously, as a framework, it offers reusable abstractions to connect different language models, prompts, tools, retrievers, memory components, and application logic. It is now one of the most popular and places the language model at the heart of its logic.

The official documentation describes LangChain as an open-source framework that offers predefined agentic architectures and integrations with various models and tools [14]. Most importantly, it offers a way to standardize the interface to models: different providers, such as OpenAI, Anthropic, Google, and others, have different APIs, and consequently, different response formats. Langchain does just that: it provides the ability to interact with them more uniformly. This provides a reliable and simple method for replacing or comparing different models with minimal code changes.

Langchain leverages its ability to combine language models and tools to build complex systems on which iterative processes can be easily implemented until the task is achieved. The documentation defines tools as an extension of what agents can do, allowing them to perform many types of actions, such as those mentioned in 2.3.3 section. Technically, they are callable functions with well-defined inputs and outputs, exposed to the model so that it can decide when to invoke them and with which arguments.

The generic nature of LangChain was one of the main reasons why it was not adopted as a framework in the work presented. More explicit control over state, workflow transitions, or overall orchestration was needed, so it became natural to consider more specialized tools, such as LangGraph. Indeed, the documentation for LangGraph describes it as a lower-level framework suitable for more advanced needs that combine deterministic and agentic workflows [15].

Therefore, Langchain was the first useful reference considered, but it wasn't suitable as the final design choice. It introduces the way to deal with model abstraction, tools, agents, etc.; however, the need to observe and evaluate the degree of control of insight refinement throughout the workflow shifted attention to LangGraph and, subsequently, the Strands Agents SDK.

2.4.3 LangGraph

Unlike LangChain, however, LangGraph solves a more specific problem. Even though LangChain is flexible in terms of implementation, it is more focused on the aspect of process orchestration.

The purpose of LangGraph is to allow designers to describe their applications' behavior as a graph, where the various workflow stages are clearly marked out.

From a conceptual standpoint, there are three main components of LangGraph: state, nodes, and edges. As for state, it refers to the shared state of the application at a certain stage of execution; nodes refer to functions or operations which accept state, execute some actions, and produce updated state; finally, edges define which node should be executed next through either a fixed or a conditional transition. Thus, this framework is known for being much more controllable, readable, and customizable than many others. In a graph, the designer can clearly control state transitions, the tools used, and when the process stops. According to the documentation, LangGraph is particularly convenient for building applications with such features as stateful agents, stateful memory, durable execution, and human-in-the-loop [15]. Durable execution means that you can save the state of the workflow and resume execution from there, whereas human-in-the-loop means that a human user can inspect, approve or modify intermediate states or outputs during the workflow.

The framework was initially considered for this project due to the possibility of modeling the Insight Refinement process using graphs: batches of proto-insights as input data, search for information in the Data Catalog, generation of a hypothesis, verification of the hypothesis by the Data Analyst Agent, and generation of the final insight.

Of course, such a methodology would have made this process more reproducible and controllable. However, this was not the only challenge that had to be addressed in the context of this thesis. The Insight Hunter was expected to analyze different statistical phenomena and formulate analytical approaches which could not be entirely pre-planned.

That is why the LangGraph architecture was not used as the basis for the PoC implementation. It would have been the right choice for more deterministic processes, but this particular task required something different: a framework suitable for exploring the behavior of a model-driven agent. Such a comparison makes it possible to understand the main architectural dilemma of this thesis.

2.4.4 Strands Agents SDK

With respect to model-driven agentic systems and the trade-off between determinism and exploratory reasoning, the Strands Agents SDK appears to be the proper framework for the realization of this thesis. According to the library documentation, Strands is an open-source

SDK for building agents with respect to a language model, prompt system, a suite of tools and an agent loop [16]. The described features make it a proper choice in case an application requires autonomous reasoning but at the same time should adhere to a certain set of limitations specified by the developer.

The first task when implementing a Strands agent consists in determining its capabilities, roles, available tools and constraints. With regard to the scope of application, the model is free when making decisions. However, it is guided by a system prompt, tool descriptions, rules and logging. The distinction between Strands and the graph-based orchestration framework lies in the fact that the model reasoning path is not predefined by a fixed graph in the former case.

Agent loop becomes the main execution engine of Strands-based applications. At the beginning of the execution, the model receives the current state of conversation including its history, system prompt and tool descriptions. Throughout the process of reasoning, the model can either provide a response to the question posed or initiate the execution of a tool. After the execution of the latter, its outcome becomes an observation to add to the conversation history. The process ends once a conclusive answer is generated.

The significance of tools within the Strands SDK is obvious. Using the platform, developers are able to attach agents to tools that allow linking language-model reasoning to calling other functions. The proposed solution works by means of having the model trigger the function execution. Then, the output of the function becomes an observation and is added to the conversation history. In turn, it can serve as input for reasoning.

Another characteristic of Strands that is related to the Insight Hunter discussion is the handling of the state and conversation history during the whole agent's execution. In this case study, this is required because Insight Hunter needs to store all the information collected during the process of refining a set of proto-insights, which includes the input data, hypotheses formed, schemas accessed through the Data Catalog, reports produced by the Data Analyst Agent, as well as all decisions made during the previous iterations.

In the Proof of Concept implemented, this particular function performs the role of run-level memory. The agent does not rely on persistent long-term memory or RAG mechanism; it stores only the data required to complete the current run of refining proto-insights. This is aligned with the experimental nature of the project, whose goal was to prove that it is possible to refine proto-insights using metadata, tool outputs, and validation reports during a single execution.

Apart from the previously mentioned features, traceability can be seen as an advantage of

the proposed solution. The possibility of observing the execution of agents is crucial in case of a business analytics tool. Apart from being capable of generating plausible outcomes, the tool should allow tracing the use of its components. In addition, it is important to be able to trace the inputs used to execute a tool, obtain the output data and use it during reasoning. Traceability becomes an architectural characteristic in this respect.

There is one more feature of Strands that became useful while implementing the project. Specifically, Strands supports multiple model providers. The feature allowed testing several models to find out which one would perform better under certain conditions. The choice of the provider is determined by its performance in terms of conducting reasoning, generating valid output, handling long context and executing tools correctly. The agentic logic remains independent from the model in such a way to avoid reimplementation if needed.

As compared to LangGraph, Strands is less strict in defining a workflow of an agent. In contrast, the framework gives much freedom to the model when selecting a reasoning path. On the one hand, this approach is consistent with the purpose of this thesis because insight refinement requires exploration. At the same time, the reasoning should be properly monitored and controlled. This task is addressed via logging and observation capabilities. Prompt, tool descriptions and outputs should be specified in this case.

Thus, controlled autonomy becomes the key to successful execution of insight refinement in this scenario. The Insight Hunter needs to be provided with enough autonomy to generate hypotheses and explore the data, while control over the reasoning process is required to ensure the validity of outputs and avoid mistakes. The Strands framework supports this balance, but the final degree of control also depends on prompt design, tool definitions, logging and validation constraints.

3. Insurance Data Context and Proto-Insight Generation

3.1 MTPL Domain and Analytical Scope

The project domain presented in this thesis concerns the Motor Third Party Liability (MTPL) branch, i.e., mandatory insurance for civil liability arising from the use of motor vehicles.

Insurance companies carefully evaluate their portfolios to monitor their profitability, and to do so, they increasingly leverage cutting-edge systems capable of considering numerous factors and producing results that are difficult to achieve through human analysis alone. The system developed in this thesis follows this direction by exploring how an agentic workflow can support the interpretation and refinement of portfolio signals.

MTPL portfolio analysis is based on the integration of different types of information, or rather, features, which can be more or less useful in different aspects of the analysis. Specifically, aspects related to the policy, the customer, the insured vehicle, the observation period, and the contract's technical performance are considered.

It is therefore clear that the dataset collects data and information that allow for portfolio analysis not only in aggregate form, but also through specific segmentations. For example, it is possible to compare groups of policyholders with different personal details, vehicles with different technical attributes, policies belonging to specific companies, or segments characterized by different premium levels and observed risk.

Unsupervised pattern mining techniques focus on identifying recurring relationships between attributes, with the aim of highlighting potentially relevant portfolio segments. This underscores the importance of targeted and timely preprocessing, feature selection, and cleansing to improve the quality of proto-insights and, consequently, the patterns identified.

As mentioned in previous chapters, the generation of proto-insights is mainly based on association rule mining, while clustering is used to group similar rules and reduce redundancy in the resulting output, they are algorithms that strongly depend on the quality of the selected features and the input dataset.

3.2 Dataset Structure and Feature Families

The dataset used is organized according to a relational structure, such that the information is distributed across detail tables, bridge tables, and descriptive dimensions. The reason for this logical division lies in the need to represent the insurance portfolio from multiple perspectives.

The core information consists of the tables relating to the policies and insured risks.

Below is a brief description of the main families of features mentioned above, which are useful for explaining and better understanding the insights that will be analyzed later.

- **Contractual and Policy Characteristics:** These variables describe the details of the insurance contract, the observation period, the line of business, the company, the annual premium, the gross premium earned, and other information useful for contextualizing the insurance position. From these features, it is possible to understand, for example, the MTPL level compared to other risk levels or sub-levels.

- **Features Related to the Insured Risk:** In this study, we focused only on MTPL, but the dataset also contains other motor levels and sub-lines of business. This filter is important during preprocessing because it allows risk features to isolate the correct technical perimeter and prevent phenomena belonging to other coverages from being interpreted as signals specific to MTPL.

- **Claims-Related Features:** This is one of the most important feature families, as it describes the loss events associated with policies. It includes information such as the number and cost of the claim, the date of the accident and its reporting, and the status of the claim. This allows for the calculation of indicators such as frequency, severity, and loss ratio, which will then be used in the statistical validation phase.

- **Customer or Policyholder Features:** This category includes variables such as age and customer type, as well as geographic information. These features greatly assist pattern mining algorithms, as they significantly impact the construction of segmentations and the identification of risk behaviors related to specific profiles.

  Perhaps the most important feature in this section is the one that distinguishes the policyholder from the vehicle owner. A clear example is the case of a parent with a long insurance history and a newly licensed child using the vehicle. Without this distinction, the

pattern mining process could associate the observed risk with the wrong profile, leading to misleading segment interpretations.

- **Insured Vehicle Features:** This category includes, for example, the type of vehicle, its use, its age, engine power, and its value. These variables are relevant because they influence both the probability of a claim and the average cost of claims.

- **Time Features:** Another essential category is time. The agent was required to explicitly report this information, both to identify whether a specific insight identified a trend of some kind and to verify the periodicity or recurrence of suspicious episodic phenomena

- **Geographic Features:** These features were extracted from the customer's or vehicle owner's postal code and, through further derivations, allow the geographic dimension to be introduced into the insight. This allows highlighting the existence of interesting differences between different urban areas and population density levels.

- **Derived or Behavioral Features:** The last features considered are the most complex to manage at the agent system level, as they derive from specific behaviors or calculations and do not fall within the categories of variables directly observable in the dataset. Some examples include the number of active policies associated with the customer, vehicle changes during the current year, fuel changes compared to the previous year, deviation of the customer's age from the median, and the indicator relating to accidents that occurred over the weekend.

3.3 Insurance Technical Indicators

The features discussed in the previous section have illustrated the multiple perspectives from which an insurance portfolio can be analyzed and the potential for generating different types of insights. However, in order to determine whether a proto-insight is actually relevant from a technical and business perspective, it must be evaluated through specific insurance indicators.

In the MTPL domain, some indicators play a central role in the interpretation of portfolio performance. In particular, exposure, claim frequency, claim severity, average premium, gross earned premium, loss ratio, and pure premium are used to assess whether a given segment shows a meaningful risk or profitability profile.

For this reason, this section introduces the main indicators used during the validation phase. These indicators are important because they allow the Data Analyst Agent to translate the hypotheses generated by the Insight Hunter into quantitative checks based on the available data.

- **Exposure:** exposure represents the share of risk actually observed during a given period. It is necessary when comparing policies that may have different inception dates, cancellation dates, or observation lengths. For a single policy, exposure can be expressed as the fraction of the year during which the policy was active.

$$\text{Exposure}_i = \frac{\text{Observed Days}_i}{\text{Days in the Period}} \tag{3.1}$$

At segment level, total exposure is obtained by summing the exposure of all policies belonging to the segment.

$$\text{Exposure} = \sum_{i=1}^{N} \text{Exposure}_i \tag{3.2}$$

- **Claim frequency:** claim frequency measures how often claims occur with respect to the amount of risk actually observed. The number of claims is therefore normalized by exposure, making the indicator comparable across segments with different portfolio sizes or observation periods.

$$\text{Claim Frequency} = \frac{\text{Number of Claims}}{\text{Exposure}} \tag{3.3}$$

This indicator is useful for identifying segments in which claims occur more frequently than expected, independently from the average cost of each claim.

- **Claim severity:** claim severity represents the average cost of observed claims. It is calculated as the ratio between the total cost of claims and the number of claims.

$$\text{Claim Severity} = \frac{\text{Total Claim Cost}}{\text{Number of Claims}} \tag{3.4}$$

This indicator allows distinguishing between segments with different claim-cost dynamics. For example, two segments may have a similar loss ratio, but one may be driven by many low-cost claims, while the other may be driven by fewer but more expensive claims. Claim severity is therefore essential for understanding whether the technical issue is related to the frequency of claims or to their average cost.

- **Gross earned premium:** gross earned premium represents the portion of premium earned by the company during the observation period. It is more appropriate than written premium when comparing premiums and claims over a defined time interval, because it only considers the part of the premium that corresponds to the risk actually observed.

$$\text{Gross Earned Premium} = \sum_{i=1}^{N} \text{Earned Premium}_i \tag{3.5}$$

- **Average premium:** average premium measures the average earned premium per unit of exposure. It is useful for understanding whether a segment is priced differently from the rest of the portfolio.

$$\text{Average Premium} = \frac{\text{Gross Earned Premium}}{\text{Exposure}} \tag{3.6}$$

This indicator is particularly important when assessing whether the premium level of a segment is consistent with its observed technical risk.

- **Loss ratio:** loss ratio is one of the most important indicators for evaluating insurance profitability. It measures the share of earned premium that is absorbed by the cost of claims.

$$\text{Loss Ratio} = \frac{\text{Total Claim Cost}}{\text{Gross Earned Premium}} \tag{3.7}$$

A high loss ratio means that a significant portion of the premium collected is being absorbed by claims, signaling a potential profitability problem. The prompt guided the Insight Hunter to consider this indicator when assessing the technical relevance of a hypothesis, without reducing the value of an insight only to loss-ratio improvement.

- **Pure premium:** pure premium estimates the expected claim cost per unit of exposure, independently from the premium actually charged. It is useful when assessing whether a given segment presents a level of technical risk that is consistent with the observed premium.

$$\text{Pure Premium} = \frac{\text{Total Claim Cost}}{\text{Exposure}} \tag{3.8}$$

Equivalently, pure premium can be expressed as the product of claim frequency and claim severity.

$$\text{Pure Premium} = \text{Claim Frequency} \times \text{Claim Severity} \tag{3.9}$$

This relationship is important because it shows how the expected cost of a segment can increase either because claims occur more frequently or because each claim is more expensive on average.

The indicators described above are intrinsically connected. For example, the loss ratio depends on both the cost of claims and the premium level, while pure premium can be decomposed into claim frequency and claim severity. During the validation phase, the Data Analyst Agent uses these indicators to break down the phenomenon suggested by the proto-insight and to verify whether the hypotheses formulated by the Insight Hunter are empirically supported by the data.
‘‘‘

3.4 Proto-Insight Generation Pipeline

This section explains the first part of the work pipeline, namely the one concerning the generation of proto-insights. This phase corresponds to the preliminary part of the workflow and represents the starting point of the agentic system developed in this thesis.

The starting point is the insurance database, which contains all the data relating to the families of features listed above. The next step is preprocessing, which had four main objectives:

- **Integrate the various relational entities.** Since the data was not collected in a single table and the selected features are highly heterogeneous, it was necessary to link the tables using keys to obtain a consistent analytical representation at the policy/risk level.

- **Build derived features.** As mentioned previously, not all the features required for the analysis were directly available, so by writing specific functions, features such as `owner_is_holder` were constructed to account for cases in which the policyholder and the registered vehicle owner do not coincide.

- **Define the correct analytical scope.** Here, the necessary filters were applied to narrow the analysis to Italy, MTPL, passenger cars, private use, and retail, and both active and inactive policies.

- **Make the variables usable for pattern mining.** Many variables were binned, transformed into classes, quantiles, and other categories.

For example, numerical variables such as premiums, age, density, or other indicators can be represented through ordered classes or quantile groups. This transformation allows continuous quantities to be converted into interpretable conditions, usable by algorithms such as association rule mining.

The preprocessing phase therefore has more than just a technical function; it directly impacts the quality of the proto-insights generated.

Once the enriched dataset has been constructed, unsupervised pattern mining techniques are applied. Association rule mining was primarily used to identify frequent itemsets and generate association rules. This phase provides rules that identify non-obvious recurrent combinations of features, which may indicate insurance phenomena worthy of further analysis.

Each rule is accompanied by statistical metrics (support, confidence, lift) designed to highlight the strength of that particular signal; not to be confused with business relevance.

The pipeline also includes the clustering phase, which uses this second unsupervised algorithm to group the association rule outputs into clusters attributable to a similar phenomenon. This makes the Insight Hunter's input more interpretable and suggests a common analytical direction.

The result is a list of proto-insight batches. A batch may also correspond to an entire cluster, provided that the number of proto-insights does not exceed the predefined maximum size. Each batch contains these preliminary patterns expressed as association rules, accompanied by the corresponding statistical metrics that do not necessarily distinguish between correlation and causality.

The final step of the first part of the pipeline, known as "proto-insight identification", is a set of JSON files of the following type:

```
{
  "batch_id": "batch_example_01",
  "proto_insights": [
    {
      "insight_type": "association_rule",
      "result": {
        "explanation_rule": "customer_region = lombardy and customer_age between 30 and 45
            -> engine_power_kw between 80 and 110",
```

```
        "metrics": {
          "support": 0.072,
          "confidence": 0.640,
          "lift": 1.310
        }
      }
    },
    {
      "insight_type": "association_rule",
      "result": {
        "explanation_rule": "customer_age between 55 and 70 and fuel_type = diesel ->
            vehicle_age between 10 and 15",
        "metrics": {
          "support": 0.058,
          "confidence": 0.590,
          "lift": 1.180
        }
      }
    }
  ]
}
```


In the agentic system, a proto-insight is interpreted as this JSON, which does not contain any hypotheses but represents a structured statistical signal that must be interpreted, contextualized, and then transformed by the agent to extract its business utility.

The role of the Insight Hunter, described in the next chapter, is precisely to start from these raw signals and use them as seeds for the refinement process.

The architectural motivation behind this input representation is discussed in the next chapter, where the JSON batch is treated as the interface between the preliminary pattern-mining pipeline and the agentic refinement system.

4. Insight Refinement Architecture

4.1 Motivation and general vision of the architecture

In the previous chapter, attention was focused on the part of the system aimed at extracting statistical patterns from the input and interpreting them as proto-insights. From this point onward, this thesis introduces its main contribution: the transition from patterns to insights and, more specifically, the stage of Insight Refinement. Indeed, after this introduction, the rest of this thesis will focus on the agentic architecture developed to interpret, elaborate, and validate the initial set of proto-insights, as exemplified in Figure 1.1.

Since the goal of the entire analysis process is to discover patterns with potential technical or business relevance for the insurance portfolio, the initial proto-insights should be characterized by a sufficiently high degree of relevance. The purpose of the elaboration stage is thus not to extract valuable insights from weak and irrelevant patterns but rather to elaborate on signals that may potentially carry valuable information. To this end, the process of refinement consists in investigating the implications of the signal, validating the conditions under which the pattern appears statistically valid, and analyzing its possible meaning in the insurance field.

As a result, an insight is too complex a concept to rely merely on natural language translation of an association rule. Rather, an insight requires more complex and multifaceted analytical processing, involving the progressive enrichment of the statistical signal with domain knowledge, data analysis, and reasoning skills. On this note, this process bears similarities with that performed by an expert analyst who starts from an intuition and, by analyzing and refining it, makes it more useful for the business.

While the above architecture has become the final one adopted in this thesis, this was certainly not the case from the start. Initially, in fact, the first architecture considered was based on a single agent, named the Insight Hunter, managing the whole reasoning workflow. While such a simple architecture proved helpful at an early stage as an experimental benchmarking exercise, its limitations quickly became evident as the project progressed.

The first major issue of the initial architecture was the concentration of functions inside a single agent. Even though reasoning through ReAct was already implemented, the agent was still supposed to interpret proto-insights, perform information lookup tasks, develop hypotheses,

validate them quantitatively, and synthesize the results. This resulted in unsatisfactory reasoning quality, especially since the goal was that of generating structured insights comparable to those produced by human analysts.

A second important limitation was related to interpretability and debugging. Indeed, whenever an error occurs, it becomes very hard to determine the cause of the problem. This might stem from the proto-insight itself, from the development of the hypothesis, from the choice of the right metric, from quantitative validation, or even from the synthesis performed by the model.

These limitations become even more obvious as soon as the input evolves from one single proto-insight to multiple proto-insights. In early prototypes, indeed, monitoring the behavior of the agent was relatively easy since the agent executed everything triggered by a single pattern. At a later stage, on the other hand, it became necessary to pass multiple proto-insights, making the process more complex since results were now expected to come from the combination of multiple signals, and not from the analysis of a single one.

In light of this issue, an intermediate multi-agent architecture was considered, consisting of three fundamental agents: the Explorer Agent, the Orchestrator Agent, and the Validator Agent.

The purpose of the first was that of analyzing analytical opportunities offered by the proto-insights, retrieving information relevant to the process and developing better refined versions of the input signals. The role of the Orchestrator Agent was instead that of managing the workflow by determining the optimal stopping points according to three criteria: execution budget, maximum execution time, and the quality of the insights generated. Finally, the last component of the system was the Validator Agent, whose purpose was that of validating hypotheses through quantitative analysis via various methods of quantitative analysis.

Even though this architecture was promising in terms of modularity within agentic systems, the degree of complexity that it implied resulted in being inappropriate for the Proof of Concept. This because of some serious issues related to debugging, overhead cost due to orchestration, and difficulty in controlling the output generated by the different agents along the way.

This motivated the adoption of a less complex architecture based on two main modules, namely the Insight Hunter Agent and the Data Analyst Agent.

The Insight Hunter manages the interpretative and generative processes involved in the refinement of insights. Specifically, it receives batches of proto-insights, examines the signals, and calls the tools needed to properly interpret the input. In particular, it uses the Data Catalog to ground its reasoning in the available data schema and to formulate hypotheses that are coherent

with the insurance domain. In this way, the generated hypothesis can be passed to the Data Analyst Agent, the module in charge of verifying the hypotheses, either by data lookup and retrieval, calculation of metrics and technical indicators, and statistical analysis.

This architecture was designed to balance autonomy, control and traceability. Autonomy is necessary since insight generation involves exploratory reasoning. This means that, on the basis of batches of heterogeneous proto-insights, the system needs to explore various analytical perspectives. At the same time, the process of exploration needs to respect two other key aspects: it must remain aligned with the domain and express the reasoning performed as measurable hypotheses.

This trade-off between exploratory reasoning and empirical validation will constitute the core of this architecture.

4.2 Insight Hunter as the Central Refinement Agent

Having defined the overall architecture's structural design, it is possible to move on to analyze the exact role of the Insight Hunter within the final system in more depth. As noted earlier, the task of the Insight Hunter is to perform the interpretative and generative aspects of the refinement process. Therefore, its role is not to validate data but to convert proto-insights into hypotheses and then verify the resulting analytical claims via the Data Analyst Agent.

In general terms, the Input for the Insight Hunter includes the batches of proto-insights, which have been created as outputs in the earlier stage. During the refinement loop, the Insight Hunter formulates analytical hypotheses to be validated by the Data Analyst Agent. At the end of the process, these validated and refined findings are synthesized into the final insight.

However, it is impossible to use proto-insights immediately because they neither give any explanations regarding the reason for importance, nor prove any economical or business relevance, or provide any further empirical support.

As seen from the description above, the agent's role consists not only of generating hypotheses in natural language. It has to organize the analytical meaning of the task, finding out whether the signals found require further analysis and how they may be formulated correctly. Thus, the Insight Hunter should be characterized by the ability to find and combine exploratory reasoning with domain and validation constraints.

On the one hand, the Insight Hunter needs to show some creative capacity, allowing it

to discover new, unexpected directions, which may be followed during further analyses. On the other hand,it has to be aware of the boundaries it faces while generating hypotheses. For example, it has to be sure that the hypotheses it is generating are still consistent with the available data, insurance industry, and capabilities of the Data Analyst Agent.

As stated previously, the Insight Hunter is also a stateful component, operating during a single refinement run of the entire refinement process. While executing tasks, the Insight Hunter gradually accumulates the context from the following sources: input batch, Data Catalog, hypotheses generated thus far, and reports returned by the Data Analyst Agent. Such temporary memory allows the agent to refine its own reasoning after receiving new data instead of trying to generate the solution at once.

It is important to note that the final output is not supposed to be generated after interpreting proto-insight at once. The Insight Hunter should be able to develop an entire trajectory from one hypothesis to the next. Namely, it is expected to develop an initial hypothesis, receive a quantitative feedback, decide whether the evidence is enough for further research, and make a decision on what further steps should be undertaken.

Further sections explore the relationships allowing the Insight Hunter to perform its task successfully. Firstly, it is vital to analyze the design of the input representation. Then, the interaction with the Data Catalog is described as the source providing context for the hypotheses to be created. Lastly, the discussion focuses on interactions with the Data Analyst Agent and their role in the refinement process.

4.3 Input Representation Design

In contrast to the previous section, where the JSON batch structure was described, attention is now focused on a slightly different problem: instead of examining how the architecture should be designed so that the insights generated in the preliminary pipeline become validated, the discussion considers how a particular design affects the agent's ability to reason about proto-insights. While this issue may seem tangential and secondary at first sight, the problem of representation actually has significant implications for the agent's reasoning abilities.

From the very beginning of the project, the design of the input representation became critical because this decision would affect the amount of information available for reasoning, the level of interpretative autonomy, and the extent to which the refinement process is already predetermined.

Initially, various possibilities were explored regarding how proto-insights could be represented for the Insight Hunter. Some options included proto-insights as plain textual rules, as JSON objects with some level of sophistication, as increasingly enriched documents, and as graph elements. Obviously, these options would all expose various degrees of structure to the agent, thus shifting the ratio between guided interpretation and autonomous reasoning.

The adopted solution had to address two conflicting requirements. On the one hand, the input needed to contain some structure because the agent needed to recognize statistical signals and follow patterns. On the other hand, it should not constrain the agent too much by imposing a predetermined interpretation of a pattern. In other words, the goal was not to provide an explanation to the agent but rather to give it a structured starting point.

One of the early hypotheses concerned a graph-based representation of proto-insights. This design intuition matched the nature of the problem quite well, because the patterns discovered by the preliminary pipeline establish certain relationships between various elements of the input. By modeling these relationships in a graph, the agent would be given more freedom to navigate between variables.

In theory, it could have represented the transition from a relational database to a graph representation using three different levels. The first level would be known as the schema graph and represent structural information about the database, such as tables, keys, and relationships between entities. This component would allow to map out the database structure and determine which relationships could be used when navigating the knowledge base.

The second level would be called the feature graph and represent analytical variables and their transformations. Unlike the schema graph, the feature graph would not refer to physical database tables but rather to analytical features used in the domain, such as customer age, vehicle characteristics, premium categories, geographical factors, or any other technical indicators. Additionally, the feature graph would allow to capture various buckets and classes associated with individual features.

Finally, the third level would be named the pattern graph and represent the relationships between features (or, possibly, buckets). Patterns would be stored in the form of links between nodes enriched with statistical metrics such as support, confidence, and lift. With such a model, it could query the system to get access to certain patterns related to particular features, segments, or technical indicators.

In this alternative design, the graph would not necessarily have been provided to the Insight

Hunter in its entirety. Rather, it would have represented the underlying information layer, stored and managed through a graph database such as Neo4j. During each reasoning step, the agent could have queried this infrastructure through a dedicated tool or API and retrieved only the subgraph relevant to the features, segments, or patterns currently under analysis.

The graph representation offered many advantages over traditional relational databases. First, a graph allowed to overcome several drawbacks related to the relational approach, one of which was that some analytical variables (for instance, age) can be represented differently in practice. When reasoning analytically, customers' age groups are often discussed (such as young vs. mature drivers), and as many buckets as needed can be created without adding extra columns to the relational database.

Secondly, using a graph would make it possible to access graph portions, rather than the whole graph, that are relevant to the current reasoning step through some API or a special querying tool. Instead of exposing the model to the entire knowledge structure, the corresponding part of the graph would be retrieved to help explore features, segments, or patterns.

Finally, using graphs would enable the storage of different kinds of relevance associated with each pattern. A pattern might be strong from a statistical perspective but uninteresting or even trivial from a business point of view. It could be a weak pattern, but still useful for pricing, underwriting, or portfolio management. Thus, a more sophisticated version of this architecture would allow additional dimensions to be associated with each node in the graph.

Although there were many arguments in favor of the graph representation, this architecture was eventually discarded. It would have required the implementation of another pipeline and the development of mechanisms for keeping the nodes updated with fresh information. Also, to use such a representation, an extra infrastructure layer based on Neo4j or a similar technology would have had to be introduced. Overall, implementing this solution would have required a significant amount of resources that could have been better spent elsewhere.

During development, storing proto-insights in a document-oriented representation was also explored. To do that, a special database resembling a DocumentDB- or DynamoDB-style database would have been created. In this case, each proto-insight would evolve through different states, such as initial, updated, enriched, and reanalyzed versions. Such a representation was very suitable for reflecting the evolutionary nature of the process, as it stored historical information about the evolution of each insight.

Again, although this solution seemed promising, it was abandoned due to its unnecessary

complexity. Using a document database implied additional complications related to persistence, versioning, and retrieval. But perhaps more importantly, this architecture would have diverted attention from solving the primary problem: whether an agentic system can generate valuable insights from proto-insights.

As a further step, the idea of pre-processing the proto-insights before delivering them to the Insight Hunter was considered. For instance, the construction of a single aggregated proto-insight from the whole batch was considered in order to make things easier for the model. Although this would have simplified the input representation, it was not a good idea because it contradicted the primary advantage of this project, namely the ability of the Insight Hunter to analyze various signals and derive a general direction.

Therefore, the final input representation was gradually reached, based on the JSON batch format. In comparison with graph-based representations, it was relatively light, and in comparison with document-oriented databases, it was easier to maintain. The main drawback of the JSON approach, compared with these architectures, was the lack of advanced information structuring.

At the beginning stages of the project, richer JSON formats were used, including various filters, conditions, segments, meta-information, and even some hints for performing validation. This design choice appeared appropriate at first because it provided a lot of structure for reasoning. Yet it raised several problems, the main one being that this excessive structure led to higher input sizes and consumed context-window capacity. Moreover, it distracted the model's attention from what was really relevant.

Finally, this kind of rich input representation made the system too mechanical and therefore limited its exploratory capabilities. Another challenge was linked to the relationship between the Insight Hunter and the Data Analyst Agent: if a proto-insight contained too many filters and various validation-oriented elements, then the former did not know what to do next. Should these proto-insights be forwarded to the latter? Or should they be modified? Or should the interpretation of the pattern be reconsidered?

This ambiguity blurred the boundary between hypothesis formulation and validation, which meant that the input had to be simplified to ensure that the agent knew what needed to be done next. Thus, the adopted input representation is shown in Code **??**.

As can be seen, this representation does not give much information about the underlying structure, nor does it track the evolutionary history of a proto-insight. It only specifies what kind

of algorithm detected the pattern, provides the actual rule, and gives a few necessary statistics. This solution was quite effective because it reflected the core of the problem in the most efficient way: it was lightweight, interpretable, and controllable.

Thus, the chosen input representation represents a compromise between two extreme strategies. On the one hand, too little structure implies that the model can only paraphrase the original rule. On the other hand, too much structure may imply that the interpretation of a pattern is predetermined, thus removing the element of creative freedom. The JSON representation makes it possible to strike a balance between these two extremes.

4.4 Semantic Grounding through Data Catalog and YAML

After discussing the structure of the input representation, it is time to consider another crucial architectural relation, namely the interaction between the Insight Hunter and the Data Catalog. In general, a Data Catalog is the element of the software architecture that describes and structures the information contained in the database. In contrast to the database, that is where the operational records reside, the catalog includes metadata, such as descriptions of tables, columns, types, textual information, relations, key identifiers, examples, etc.

In the context of the current implementation, the role of the Data Catalog is particularly significant due to the nature of the agent: since the Insight Hunter is based on an LLM-based agent, it is capable of coming up with the most plausible hypotheses. However, at the same time, it is unable to figure out which actual variables and derived features exist in the data domain, which tables are related to one another, and how certain variables should be understood or used. Hence, it requires an external controlled description that would provide it with the information necessary for forming hypotheses.

In the architecture used within the scope of this thesis, the Data Catalog takes the form of YAML files accessible to the agent through special tools. The purpose of the latter is to make sure that the Insight Hunter has an opportunity to find necessary information about the available tables and their contents. Such information includes columns' names, data types, textual descriptions, examples, and information about how the table should be treated. Hence, the catalog acts as an external semantic layer providing the agent with the information needed for forming hypotheses.

Thus, the role of the catalog in this context can be viewed as semantic grounding. While

the goal of the catalog is not to verify hypotheses quantitatively, it provides necessary means for making sure that the concepts used in the hypothesis exist in the available data or can be generated from there. As has already been mentioned above, this layer of the system provides information that can be analyzed, while its validation is left for the Data Analyst Agent.

This implies that consultation of the catalog becomes an integral part of the agentic cycle: after receiving the batch of proto-insights, determining the potential directions, and finding the available concepts from the catalog, only after that can the model proceed to formulating a particular hypothesis. As a result, the risk of introducing variables that could be used to formulate hypotheses within the context of the domain but that do not really exist within the database is considerably reduced.

This is particularly significant in the case of insurance data, which usually does not contain any additional descriptions. In such cases, a particular field might mean one thing to a person with sufficient domain knowledge but be understood as something else by a machine without any prior knowledge in that area. The catalog addresses this challenge by structuring the raw information about the data domain.

Furthermore, it becomes especially useful when it comes to considering derived features, as several key variables used throughout the project were not presented in the database as originally provided. For instance, among such variables are the indicator of whether a client is a vehicle owner, in contrast to being the insured party, various territorial categorization schemes, population density, and behavioral variables. All these variables need to be created beforehand through feature generation techniques.

By structuring them in the catalog, it is ensured that they can be used as meaningful concepts to base analytical hypotheses on while maintaining the link with actual information from the original database. Therefore, in addition to the fact that a properly constructed catalog is able to help in reducing the risk of errors, it is capable of making the generated hypotheses more precise.

For instance, a Data Catalog allows forming a hypothesis by connecting contractual information, customers' characteristics, vehicles' attributes, and technical indicators. In turn, this means that this layer of architecture enables the Insight Hunter to expand its analytical capacity without violating the constraints posed by the available information.

One more implication of this approach is schema-level traceability of the resulting hypotheses. Given the fact that the catalog helps the agent determine which concepts to use to generate

hypotheses, tracing the connection between different portions of the schema helps identify where these concepts come from. The goal is not just generating hypotheses but making sure that their generation process can be inspected later on.

As shown in Figure 4.1, this type of interaction between the Insight Hunter and the Data Catalog takes place through special tools enabling the former to access metadata provided in the catalog. It does not substitute quantitative reasoning but facilitates the process.

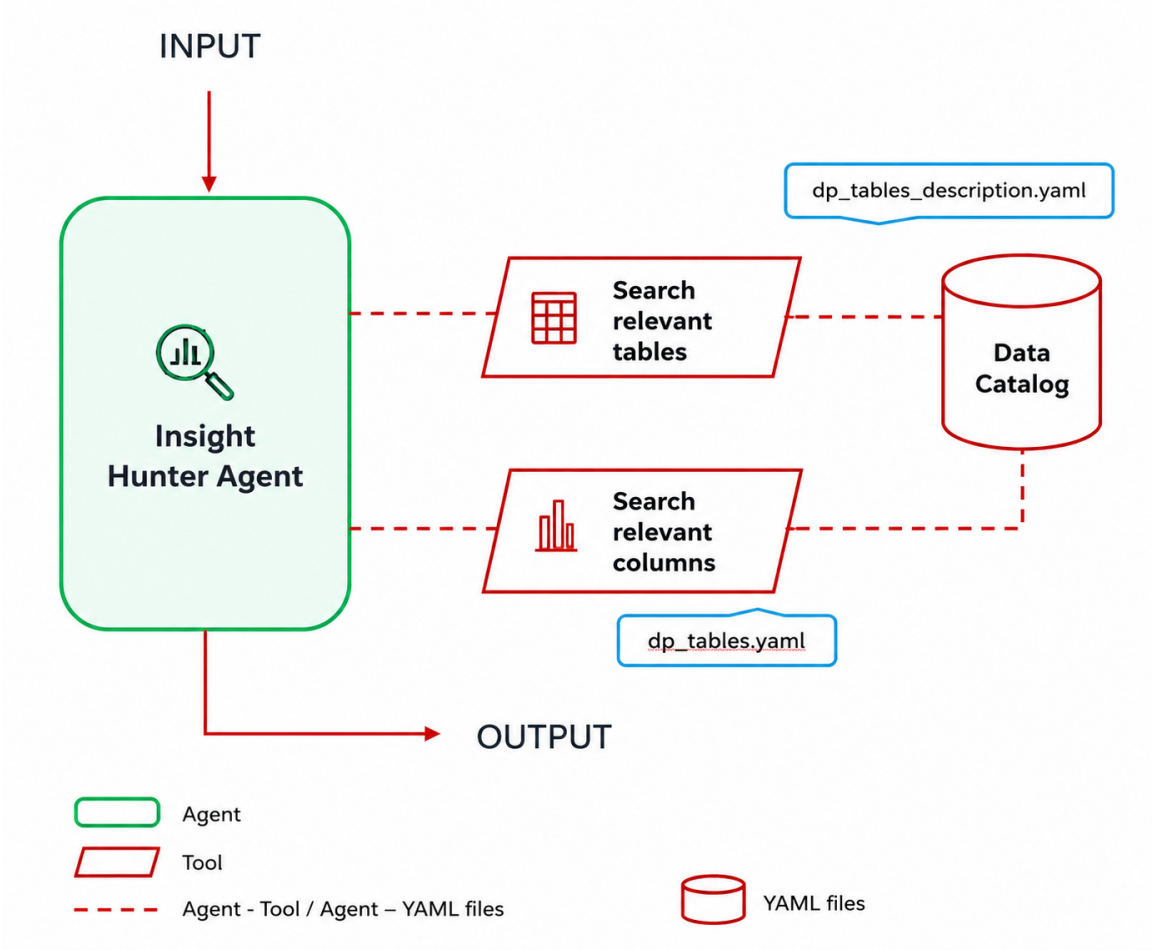

Figure 4.1: Interaction between the Insight Hunter Agent and the Data Catalog through tools for retrieving relevant metadata.

In summary, the Data Catalog is the semantic layer of the architecture helping the Insight Hunter reason about the provided data without being too detached from the available information. As a result, it makes the process of reasoning more transparent and precise. This layer helps trace which parts of the reasoning belong to the reasoning process and which are grounded in the information provided in the catalog.

The following chapter will be focused on the other major architectural relation concerning

the Insight Hunter, the interaction with the Data Analyst Agent.

4.5 Interaction between Insight Hunter and Data Analyst Agent

In addition to the semantic grounding provided by the Data Catalog, another important multi-agent relationship that needs attention is the interaction between the Insight Hunter and the Data Analyst Agent. This interaction is crucial for understanding how an analytical hypothesis created by the first agent is subsequently turned into quantitatively validated results.

The Data Analyst Agent acts as a validation component of the architecture and from this perspective it is essentially the empirical counterpart of interpretive reasoning performed by the Insight Hunter. This division of roles was necessary to prevent the overlap between the two components.

From the standpoint of the Insight Hunter, the Data Analyst Agent interface can be seen as a callable function. After the creation of the hypothesis, it is possible to send it to the Data Analyst via the corresponding tool interface. Such a design follows the pattern of tool-augmented reasoning presented in Section 2.3.3. Instead of performing quantitative analysis directly, the language model delegates it to a special-purpose component and then analyzes its output.

Interactions via the tools in this architecture have been shown in Figure 4.2. The Insight Hunter has the capability of accessing the Data Catalog through semantic grounding and also invoking the Data Analyst Agent through `daa_tool`. This is followed by SQL-based validation done by the Data Analyst Agent in the analytical database, after which a report is generated.

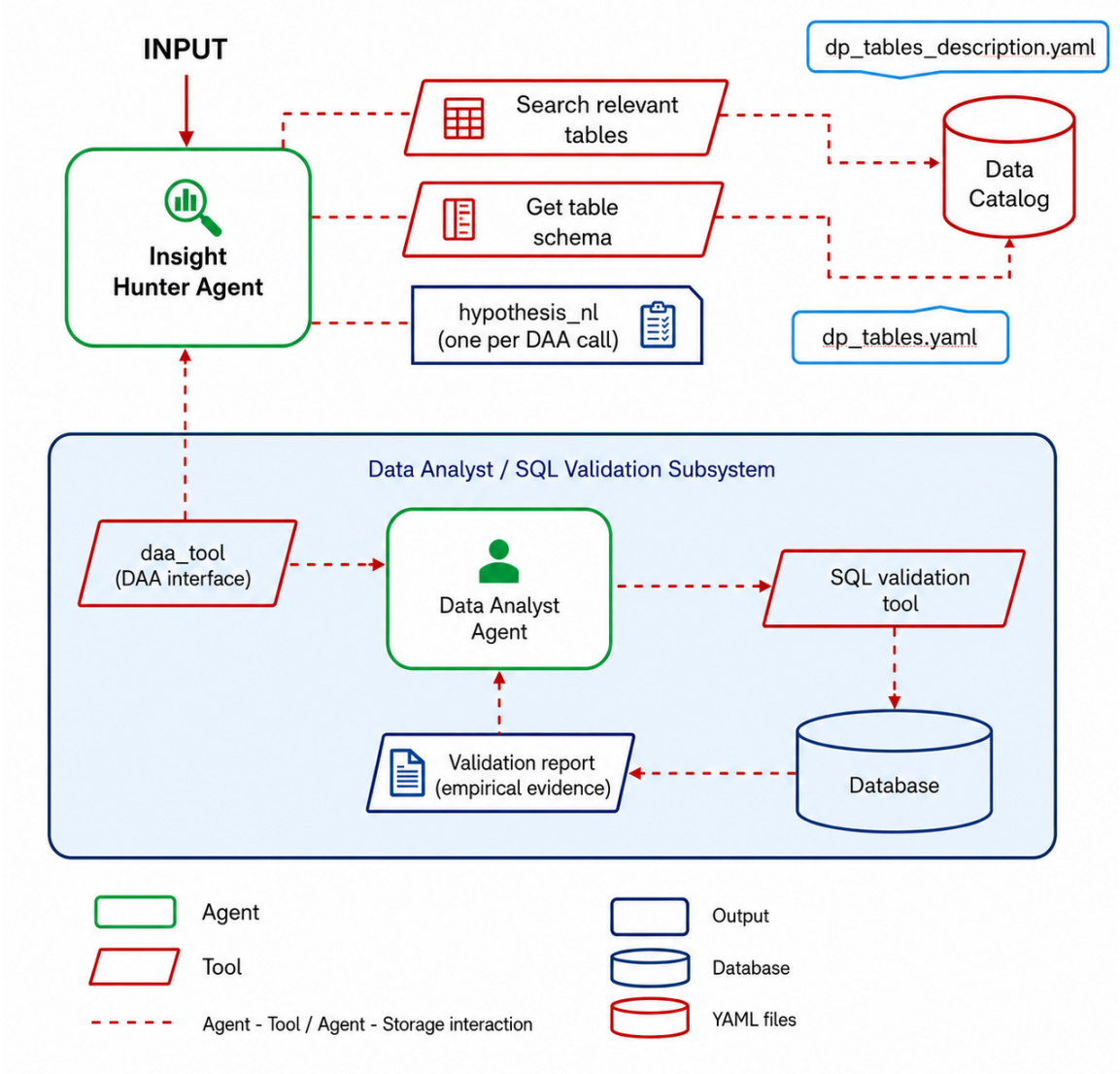

Figure 4.2: Tool-based interaction between the Insight Hunter, the Data Catalog and the Data Analyst Agent during the refinement workflow.

There are many options related to the input payload sent to the Data Analyst module, each carrying certain advantages and disadvantages. In particular, one of the early design solutions included transmitting richly structured information about hypotheses, proto-insights, various filters, and conditions, in addition to the hypothesis itself. However, such a design approach was not appropriate for the current architecture because it made it difficult to delineate between interpretation and validation stages.

For this reason, the final design adopts a simpler interaction model: from the perspective of the Insight Hunter, the payload is intentionally limited to one self-contained natural-language hypothesis, expressed through `hypothesis_nl`. Additional technical context may be added by the backend, but the analytical formulation remains the responsibility of the Insight Hunter.

Thus, the task of analytical formulation now falls on the responsibilities of the Insight Hunter. Specifically, this agent is expected to determine what needs to be tested and under what condition, according to which expected analytical direction. Meanwhile, the Data Analyst Agent will receive this formulation and perform quantitative analysis. The formulated hypothesis, in turn, acts as the bridge connecting interpretation with validation.

This design improves hypothesis-evidence traceability. With every interaction with the Data Analyst, it will be possible to unambiguously trace back which hypothesis was tested and what empirical evidence resulted in it. Sending many hypotheses or complex payloads to the validation function would make it much more complicated to relate specific numbers to particular analytical questions raised by the Insight Hunter.

The report created by the Data Analyst Agent contains the empirical evidence necessary for further refinement of the analytical problem by the Insight Hunter. Depending on the nature of a specific hypothesis, it may contain insurance-related technical indicators such as the loss ratio, claim frequency, claim severity, premium metrics, exposure, the number of policies, etc. Therefore, the Data Analyst produces quantitative data rather than simply evaluates the initial analytical direction.

The outcome of this interaction does not necessarily have to be positive in every case. A validation report may confirm a given hypothesis, partially support it, contradict it, or provide no strong evidence at all. Nonetheless, all of these outcomes can be used for further refinement of the analytical question because supportive evidence can strengthen the current analytical direction, while contradictory or weak evidence can motivate the Insight Hunter to narrow the scope, reformulate the hypothesis or explore an alternative explanation and search for the reasons behind.

At the architectural level, this interaction creates a validation feedback mechanism: the report produced by the Data Analyst Agent becomes an observation that the Insight Hunter can use to confirm, revise or reject the current hypothesis. The full iterative logic of this mechanism is discussed in the next section.

For this interaction to be effective, it is important to have a properly formulated hypothesis. When the hypothesis is formulated vaguely, in terms of “this segment”, “the same period” or “similar policies”, then the validation report may differ from the actual analytical problem. That is why the hypothesis needs to be clearly defined, consistent and valid for a quantitative check.

It also means that the interaction between the two parts is not only technological, but also

linguistic. Even though the Data Analyst Agent works with the help of a special interface, the quality of the validation depends on the way in which the hypothesis is formulated.

4.6 Iterative Refinement Loop, Exploration Threads and Stop Rules

The validation performed by the Data Analyst Agent is not designed as an independent final step. Instead, the interaction between the Insight Hunter and the Data Analyst Agent is organized as an iterative refinement process.

Such an iterative scheme is crucial for two reasons. First, the meaning of the statistical pattern in terms of the business reality is unlikely to be immediately clear after just one iteration. Even if the hypothesis has been confirmed by the Data Analyst, the reason behind the detected signal usually remains unclear. For example, the low profitability of the segment can be caused by high claim frequency, high claim severity, insufficient premiums, or some extremely costly outliers. For this reason, the agent needs an iterative procedure to decompose the phenomenon and identify the root cause.

The refinement process starts with the hypothesis formulation. At this point, the candidate hypothesis is identified based on the proto-insights batch. As soon as it is generated, it is passed as the `hypothesis_nl` parameter of the Data Analyst Agent call. Then, the latter agent creates an empirical report that becomes the new observation included in the agentic reasoning process.

However, the Insight Hunter does not have to accept the returned report as the final answer right away. It needs to evaluate whether the results confirm, disconfirm, or fail to comment on the stated hypothesis. Depending on the output, the agent has the ability to continue with the same analytical approach, narrow down the investigated segment, include benchmarks, control for the potential influence of other variables, and even stop the exploration process.

Thus, this scheme represents another mechanism of contextual learning within a single reasoning chain. The agent's internal parameters do not change but the reasoning path evolves along with the incoming data. Every iteration enriches the context of agentic reasoning with new evidence, including quantitative metrics, segment comparisons, technical indicators, time-dependent analysis and signal stability assessments.

Another important element of the refinement process is the notion of an exploration thread. Every hypothesis starts a new analytical trajectory that is supposed to stay consistent with the

original analytical direction. This rule will help to avoid the randomness of the exploration process that would make it difficult to find a relation between the hypothesis, the obtained report, and the final insight.

In the same exploration thread, the Insight Hunter may investigate the phenomenon from different angles. For example, it can test the hypothesis by comparing the segment in question with the benchmark, evaluating the claim frequency and claim severity, assessing the sufficiency of premiums, checking for temporal stability, or searching for an outlier group within the selected segment. The refinement process thus allows building a logical and coherent exploration of the selected phenomenon.

The described process requires setting up explicit stopping rules to limit unnecessary usage of tools. If an agent with access to analytical functions can keep on generating reports endlessly, it will lead to the unnecessary consumption of time, computational resources, and context length. At the same time, an excessively long chain of iterations will complicate the process of tracking and interpreting the process of reasoning.

Within the presented implementation, the stop rules serve two complementary goals. First, the process should not be stopped too early when meaningful validation questions remain. On the other hand, the process cannot be extended infinitely. Thus, the Insight Hunter prompt includes the minimum and maximum number of Data Analyst Agent calls for each hypothesis. While the former is required to prevent superficial analysis, the latter will prevent the use of too many tools and uncontrolled increase of the working context.

However, apart from these numerical limits, there is also a criterion for stopping the process of refinement. The idea is that the Insight Hunter should evaluate whether the information gathered from the previous validation has opened a new meaningful analytical question. If the answer is positive, the thread continues. Otherwise, the exploration should be stopped since there is no analytical value in further tool calls.

It is necessary to emphasize that in the context of this task, the Data Analyst should not be treated as a means to generate random reports. Every call must have a specific purpose to clarify the original signal and to provide more insight about the underlying phenomenon. If a hypothesis has already been supported, additional calls should not simply repeat the same validation, but should investigate a new mechanism, benchmark or explanatory dimension.

The loss ratio analysis example is very helpful here. If the hypothesis relates to the profitability of a segment, having only the loss ratio data would not be sufficient. Instead, the

agent should understand what drives the high loss ratio: claim frequency, claim severity, low premiums, and/or expensive outliers. To do so, the Insight Hunter might need several validation steps, but all of them have to add some value to the analysis.

Temporal stability may serve as another dimension of the analysis. Since the current database utilizes `dt_period` values to represent the time period, the Insight Hunter may need to understand whether the signal under consideration exists only during one period or across several periods. This concerns pricing, underwriting, and portfolio management because temporal stability is usually more valuable than a sporadic pattern.

Finally, it is important to note the role of stopping rules in preserving the statistical robustness of the analysis. As the agent might try to find increasingly narrow segments in the hope of making the signal even stronger, this may reduce the sample size and lower the reliability of the findings. That is why the process of refinement should avoid overrefinement. The purpose of the agent is not to identify a niche with an exceptional signal but to discover a real phenomenon.

This means that the iterative refinement process increases the chances of being able to track the reasoning trajectory. Each iteration includes the investigated hypothesis, the generated report, and the decision made by the Insight Hunter. It allows the reconstruction of the reasoning trajectory that led to the final result and the understanding of the reasons behind the choice of a certain analytical path. The requirements to record the trajectory at prompt level are described in the next chapter, while here the focus is on the architectural role of the loop.

Consequently, the final insight should not be perceived as a hypothesis formulated at the last step of the analysis. On the contrary, it represents the entire reasoning trajectory, which includes the initial signal from the proto-insight batch, schema information obtained from the Data Catalog, quantitative validations performed by the Data Analyst Agent, and the results of intermediate reasoning steps. In other words, the final insight arises from the reasoning process in its entirety, not from its last step.

In this way, the iterative refinement loop may be considered the most important compromise between the components that were discussed earlier. The agent has quite a wide degree of freedom for exploration and reasoning, but it is limited by stopping rules, works within the insurance-domain boundaries, respects the constraints of statistical robustness, and requires quantitative validation. The next chapter will explain how these architectural principles are used in the construction of the Insight Hunter prompt.

5. Prompt Engineering and Reasoning Design

5.1 Evolution of the Insight Hunter Prompt

While the previous chapters introduced LLM-based agents, tool-augmented reasoning and control mechanisms, this chapter focuses on prompt engineering as the component that translates the architectural requirements of the refinement workflow into concrete behavioral rules.

Prompt engineering proved to be one of the key processes involved in developing the Insight Hunter Agent within the scope of this thesis. In case of LLM-based agents, the prompt is responsible not only for describing the tasks assigned to the agent, but also for shaping the its identity and role, its constraints, the way in which it should use of the tools available to it, and the way in which it should interact with the other agents that constitute the system.

Thus, the development of the prompts became an integral part of the research rather than an accompanying task to perform. Even slight changes to the prompt resulted in major changes in the way in which the agent interpreted proto-insights, utilized the Data Catalog, made hypotheses, and communicated with the Data Analyst Agent. Therefore, the prompt became one of the key tools through which the agent could be guided towards a certain type of behavior. During the process of the project realization, the Insight Hunter prompt underwent multiple stages, from relatively simple prompts to increasingly complex ones that introduced new elements into the prompt itself, such as domain constraints, rules on precision of hypotheses, rules for working with tools and output formatting requirements. This process represented the transition from general-purpose text generation to specialized agentic reasoning.

One of the key changes in this process related to the generation and verification of candidate hypotheses. For example, in the earlier stages of development, the prompt allowed one to specify that the Insight Hunter should generate a certain number of hypotheses. This helped to check whether the generative abilities of the agent would work well without using the Data Analyst Agent. However, once the latter was integrated into the workflow, the prompt had to become more disciplined. Multiple hypotheses could still be explored during a run, but each validation call had to be centered on one self-contained `hypothesis_nl`. In this way, each DAA report could be traced back to a specific analytical question.

Finally, the design of the prompt was influenced by modifications of the models that were

applied in the process of the creation of the prompt as well. The replacement of the prompts that relied on OpenAI to those relying on Claude proved that a prompt could not be considered a fully autonomous construction since various models might respond to instructions and tool usage rules in their own way. Thus, the subsequent version of the prompt was redesigned using a more clear structure based on role, context, constraints, tool use rules, and output requirements.

Another relevant difference concerned the visibility of model-generated reasoning. During the initial experiments with OpenAI models, the internal reasoning trace was not directly exposed by the provider. For this reason, earlier versions of the prompt attempted to make the analytical process inspectable by requiring the model to produce explicitly labelled reasoning sections. However, these sections did not provide direct access to the model's internal chain of thought. Rather, they were prompt-elicited rationales, namely textual explanations generated according to a structure imposed by the prompt. Although this approach made the output easier to inspect, it also made the reasoning more mechanical, since the model was required to fill predefined categories instead of freely developing the analytical trajectory. This distinction is consistent with the fact that OpenAI reasoning models do not expose their raw chain of thought, although summaries of the reasoning may be provided [17].

5.2 Prompting Strategies in the Final Insight Hunter Prompt

The final Insight Hunter prompt utilizes a combination of prompting techniques whose theoretical underpinnings were presented in Chapter 2. The goal of this subsection is not to reproduce the behavioral rules reviewed below, but rather to demonstrate how these techniques are specifically realized in the prompt. Specifically, the structure of the prompt includes role conditioning, task decomposition through staged reasoning, tool-oriented reasoning, contextual examples, critique scaffolds, and the reiteration of critical constraints.

Similarities between this prompt and the ReAct paradigm discussed in Section 2.3.2 can be seen in the prompt blocks regulating access to the Data Catalog and the Data Analyst Agent [5]. Rather than generating the final insight in one step, the prompt allows the Insight Hunter to retrieve schema information, formulate a working hypothesis, obtain empirical evidence, and determine the next analytical step based on the returned observation. However, the current prompt does not introduce a new definition of the ReAct loop, which has already been discussed in Sections 2.3.2, 4.5, and 4.6 from both theoretical and architectural perspectives. The specific

contribution of the prompt is the translation of the general reasoning paradigm into domain-level instructions concerning the observations to be collected and the way in which they should affect the refinement process.

Another technique that can be identified in the current prompt is in-context calibration through contrastive demonstrations. As mentioned in Section 2.2, few-shot language modelling allows examples to be included in the context and influence the model's behavior without requiring changes to the model parameters [2]. In the case of the Insight Hunter prompt, the `EXAMPLES` block contrasts a naive hypothesis with a more rigorously formulated hypothesis. These examples do not simply define the desired output format. Instead, they present the analytical transformation required from the agent: the replacement of generic associations with falsifiable comparisons, the decomposition of technical outcomes into their potential drivers, the control of confounding factors, and the connection of validated phenomena with decision-relevant implications. The contrast between undesirable and desirable formulations is especially valuable, since it defines not only what should be produced by the agent, but also which types of seemingly plausible behavior should be avoided.

The `CRITICAL_THINKING` and `NOVELTY_MOVES` blocks provide an additional critique scaffold for the agent. They require the agent to question causal interpretations, seek alternative explanations, explore different segmentations, and check whether the observed risk is already reflected in pricing. These instructions do not represent empirical validation itself, which is performed by the Data Analyst Agent. However, they affect the way in which the questions submitted for validation are formulated. In this way, the current prompt distinguishes between two levels of control: the linguistic and conceptual critique performed by the Insight Hunter and the quantitative validation performed through the DAA.

Some high-priority requirements are intentionally reiterated throughout different prompt blocks. For example, full-batch reasoning is stated both in the general context and in the orientation and refinement phases, while the one-hypothesis-per-call requirement is provided both in the DAA interface and in the hypothesis rules. Scope constraints are also restated close to the formulation requirements. Thus, the current prompt design does not involve the complete duplication of the prompt. Rather, it can be interpreted as selective reiteration, in which a constraint is repeated close to the point at which it becomes operationally relevant. Recent work on prompt repetition reports performance improvements for LLMs in several non-reasoning configurations [18]. However, the current work did not experimentally compare versions of the

Insight Hunter prompt with and without such reiteration. Therefore, selective reiteration should be considered a deliberate design choice intended to reinforce critical requirements, rather than an independently demonstrated source of performance improvement.

Overall, the techniques described above demonstrate the dual purpose of the prompt: providing enough flexibility to generate non-trivial analytical directions, while offering contextual examples, critique questions, and reiterated operational constraints that reduce ambiguity in tool use and hypothesis refinement. The following sections examine the rules used to achieve this balance.

The later Claude-based configuration provided a different form of reasoning observability. When the corresponding extended thinking functionality was enabled, the model could return dedicated thinking blocks separately from the final answer [19]. The application could therefore capture a model-generated reasoning trace without requiring the prompt to prescribe its complete structure. The prompt continued to define the analytical task, scope, available tools, validation rules and required output, but it no longer forced the model to organize every intermediate reasoning step through rigid labels. In this sense, the transition made it possible to move from a prompt that attempted to script the form of the explanation towards one that primarily constrained observable actions and analytical artifacts, such as hypotheses, tool calls, validation reports and iteration decisions.

Nevertheless, the exposed thinking blocks were used primarily for development, inspection and debugging, rather than as empirical evidence supporting the final insight. The reliability of the refinement process therefore continued to depend on externally observable and verifiable elements. These included Data Catalog calls, the hypotheses submitted to the Data Analyst Agent, the returned quantitative reports, the decisions recorded after each iteration and the final structured outputs. Consequently, the provider-exposed reasoning trace improved the observability of the agent's behavior, while tool interactions and validation artifacts remained the main basis for system-level traceability.

The next section explores the elements that make up the final version of the Insight Hunter prompt.

At the same time, these prompt-level rules should not be interpreted as deterministic controls. They guide the behavior of the Insight Hunter, but their effectiveness also depends on the implementation choices discussed in the following chapter, such as tool design, state management, output parsing and logging.

5.3 Role Definition and Agent Boundaries

Having analyzed the evolution of the Insight Hunter task description, it is now appropriate to analyze the prompt in detail concerning the definition of its role and the definition of the boundaries of the agent. As discussed in Chapter 4, the prompt distinguishes three concepts: interpretation, semantic grounding, and quantitative validation. It is necessary to mention not only the architectural distinction but also the behavioral one: it should define not only the scope of activity for the model but also impose certain constraints on its behavior.

The first behavioral requirement implied in the prompt refers to the identification of the Insight Hunter's identity and mission. In particular, the agent is identified as an expert in insurance analytics and business intelligence, who needs to convert the input patterns into hypotheses that will be subsequently validated. It is important to notice how crucial such a definition is: the agent cannot simply rewrite patterns but use them as a basis for generating new hypotheses.

Secondly, the distinction is made between the epistemic statuses of proto-insights and hypotheses. Specifically, they are not taken as established facts but need to be critically evaluated. Indeed, while some proto-insights represent some meaningful relationships, others may refer to accidental correlation, confounded data and noise. Therefore, instead of rewriting proto-insights, the agent should interpret them in terms of hypotheses.

Thirdly, the batch level of responsibility for the Insight Hunter is also mentioned. According to the prompt, the agent is supposed to use all the patterns provided in a given batch and analyze the common signals or tensions in order to create hypotheses for further analysis.

In addition to the aforementioned, the prompt also specifies the boundaries of responsibility for the Insight Hunter relative to other agents. According to the definition, the Insight Hunter is responsible for the formulation of hypotheses, while the Data Analyst Agent verifies them. In other words, the former is the only agent capable of changing the field `hypothesis_nl`, whereas the latter validates given hypotheses.

Indeed, as previously discussed, it is crucial to clarify the boundaries of responsibilities of agents in order not to confuse the tasks. Otherwise, the agent might validate the hypotheses it creates and, conversely, analyze rather than verify the hypothesis provided to it. The specification of boundaries made in the prompt preserves the distinction described above.

One of the key components of the developed prompt is the formulation of `hypothesis_nl`

field. As was already mentioned before, the field stands for the natural language hypothesis submitted to the Data Analyst Agent. It is extremely important to ensure that the hypothesis includes only one analytical statement.

It implies very strict rules for hypothesis formulation. The Data Analyst Agent should be able to identify the analytical question being asked, the parts of the hypothesis that need validation, and the type of the quantitative evidence that is required. If the hypothesis asks several analytical questions at once, the validation may become rather difficult since it will be unclear which part of the statement should be verified.

Also, the hypothesis must be self-contained. While the hypothesis is formed by the Insight Hunter based on the proto-insight batch, the Data Catalog and the reasoning done in the course of refinement, it must not include any implicit references to that context. Thus, when the Data Analyst Agent receives a hypothesis, it should be able to interpret it as a separate validation request.

This makes it important to ensure that all the conditions necessary for validation are included in the hypothesis itself. Thus, every hypothesis must include all information needed to clarify the analytical perimeter of the validation and reference to values that comply with the available schema.

That way, each hypothesis must include the conditions of observation. When the hypothesis is formulated using a numerical variable, its value should be stated. In case when there are categorical variables, the hypothesis should include actual values that exist in the data set or are defined in the Data Catalog. Finally, in case when time is important for the hypothesis, the `dt_period` must be included.

Restricting the hypothesis scope is another mandatory condition for a hypothesis creation. By default, all hypotheses are restricted to Italy, MTPL, passenger cars with private use and retail customers, and both active and inactive policies. The constraint does not relate to context but the scope itself. For example, a pattern detected in Italian passenger cars insured under an MTPL policy is not generalizable to other countries and vehicles.

The scope of the insight must also be included in the hypothesis statement. The reason is that it plays an essential role in the understanding of the finding. While some context can help, it must be included in the statement of the hypothesis. This requirement is especially relevant for insurance analytics because a slight change in scope may radically transform the meaning of the findings.

Another rule is natural language and schema grounding. Each hypothesis must use terminology from inspected schema but should not directly mention tables and columns. `hypothesis_nl` is supposed to define an analytical problem for the agent. Hence, the hypothesis must be written in natural language, and schema grounding must be used only when necessary.

At the same time, the hypothesis should not contain references to any expectations regarding the result. The expected quantitative results are allowed only if these exact figures are produced by the agent during the previous rounds of interaction. Otherwise, the agent may produce some false expectations and thus contradict the hypothesis validation process.

The same principle is also relevant for hypothesis refinement. The agent should use its observations during the process, but any invented facts cannot be included in the hypothesis. Hypotheses can be interesting, non-obvious, and insightful but also should be verifiable. In this regard, hypotheses should not go beyond data sources, indicators, and possible comparison types.

To conclude, `hypothesis_nl` is a kind of a contract made between Insight Hunter and the Data Analyst Agent. Here hypothesis defines the analytical question, its scope, observation subject, and direction. Based on the statement, the agent can conduct validation and produce reports. The better the statement of the hypothesis, the better the validation will be.

Such precision rules play a methodological role because they help reduce dependency on context, enhance the control over interaction with the agent, and facilitate the validation process. Nevertheless, they provide an opportunity for the creative part of the hypothesis generation process.

5.4 Batch-Level Reasoning and Proto-Insight Interpretation

Another significant element of the prompt is the requirement to reason over the whole batch of proto-insights. This requirement is connected to the input structure used by the Insight Hunter. Unlike the previous version, where each insight was generated from a single proto-insight, the final version provides the agent with a batch of proto-insights.

The logic here is the following: reasoning based on one proto-insight leads to relatively superficial outputs. If the agent receives only one signal and tries to interpret it, then it is likely to produce an explanation or a rewording of the rule. In turn, such a hypothesis will improve readability, but not necessarily be an analytical idea. The purpose of the Insight Hunter

is to identify possible phenomena by reasoning over multiple statistical signals rather than by rephrasing a single rule.

Thus, the specification concerning the entire batch means that all the proto-insights should be considered. That is, the agent cannot just ignore any of them. However, it does not mean that all of them should be treated the same way. Some signals could be less important and less relevant for business analytics. Nonetheless, each of the received proto-insights should be processed, because it contradicts, supports or enriches another signal.

At the same time, the prompt does not allow the agent to prematurely narrow down the scope of reasoning towards one idea. While formulating the core hypothesis statement during the orientation phase, the Insight Hunter should try to make it as clear and broad as possible. Unnecessary restrictions should not be added to it, because they will lead to artificial limitations on the hypothesis.

Instead, the focus on precision can be made at the latter stages of development when the hypothesis is refined and justified by the results obtained by the Data Analyst Agent. On the contrary, the prompt does not leave any space for uncritical acceptance of signals as valid patterns. Before the creation of candidate hypotheses, the agent is expected to evaluate each of them using metrics like support, confidence and lift.

It allows to consider a pattern to be statistically strong; however, it does not prove its relevance in the specific context of business analysis. In addition, the agent should find out whether the pattern found is feasible according to the rules of insurance business or if it is affected by confounders or purely random factors. It is necessary, since the aim of the batch-oriented reasoning is not pattern collection.

Rather, the objective is to understand what phenomena could explain the identified signals. Some of these phenomena may become the basis for a new insight, whereas others may contradict it and therefore help to determine whether the observed pattern was only coincidental. Finally, the prompt does not allow the agent to evaluate the value of a hypothesis only in terms of financial performance.

While exploring the effects that can be inferred from the signal patterns, the Insight Hunter is also expected to determine how they may influence income, expenses, risks or portfolio structure. This represents a more subtle and indirect way of adding value. Indeed, the impact of such variables will not always be directly related to profitability; however, it may still be crucial for efficient operations.

Once again, this aspect concerns the creative combination of batch signals. When analyzed in combination, two or three signals may be more fruitful than each of them considered in isolation. Nonetheless, the prompt still places strict limitations on batch reasoning. Namely, each proposed hypothesis should be measurable, testable, actionable, reasonable and novel. Otherwise, the agent would generate well-formulated but vague ideas.

5.5 Refinement Procedure, Validation Threads and Stopping Rules

Once candidate hypotheses have been identified by the Insight Hunter, the prompt defines how they should be refined through interaction with the Data Analyst Agent. Therefore, it is important to ensure that the rules provided guide the agent toward the desired type of output, while not becoming a constraint on the potential value of the resulting insight.

In order to facilitate effective collaboration between the agents, the prompt specifies how communication between the Insight Hunter and the Data Analyst Agent should take place. Candidate hypotheses are communicated via the `hypothesis_nl` field and constitute the seeds of possible validation threads. A thread can be defined as a refinement trajectory, starting from one particular hypothesis and progressing through a sequence of DAA calls, validation reports, reformulations, and other intermediate steps.

The same proto-insight batch processed by the Insight Hunter might generate several hypotheses, each of which could become the seed of its own refinement thread. However, the current implementation of the Proof of Concept follows the single-thread strategy. This decision was motivated by both methodological and practical reasons. First of all, before implementing a multi-thread strategy, it was necessary to make sure that a single-threaded run is possible, i.e., a single thread could be completed, its intermediate evidence preserved, and a new synthesized insight generated.

Even though the multi-thread configuration would allow broadening the range of analytics of the system, it would significantly increase its operational complexity. Indeed, if a single thread requires several DAA calls and a number of validation reports, then the multi-thread configuration would lead to much more DAA calls and validation reports, thus significantly increasing the amount of the work of the database and the computational resources used by the Proof of Concept. Moreover, if in the case of a single-thread configuration the run takes hours,

in the case of the multi-thread configuration it would last even longer.

Therefore, the final version of the prompt keeps the interaction pattern under control: each validation request contains one self-contained `hypothesis_nl`, and each returned DAA report is interpreted as evidence for one specific analytical question.

The stopping rules in this context also play a vital role in managing the refinement threads. As will be shown in Chapter 7, the minimum and maximum number of iterations specified in the prompt can significantly affect the behavior of the run in terms of both resource consumption and performance. Specifically, the Insight Hunter is required to communicate with the Data Analyst at least once in order to ensure empirical validation.

It is important to note that besides quantitative characteristics, the stopping rules also imply that certain decisions need to be made. Namely, the agent should carefully examine each iteration's result and define whether it is possible to continue or finish working on the thread.

For this reason, each new validation call should either refine the current analytical direction, explore a genuinely different angle, or stop the thread if the previous evidence does not justify further analysis. In this sense, the prompt requires the Insight Hunter to react differently to supportive, contradictory or weak evidence, rather than forcing every validation result into the original hypothesis.

Transparency is also a key feature of the refinement process. The result of the Data Analyst Agent call needs to contain the information about the iteration number, the analyzed hypothesis, the validation results, numbers, the suggested mechanism and the conclusion on proceeding with this hypothesis.

Following this rule will allow retaining the reasoning pathway that otherwise can be lost due to compression of information or lack of capacity.

Diversification of the thread by exploring alternative angles is yet another requirement imposed on the Insight Hunter. In accordance with this rule, one should not repeat the previous iteration in a similar manner but should look for alternative drivers, segments, mechanisms, decision levers, temporal comparison and even company-level analysis.

In turn, it can be noticed that company-level comparison is automatically considered if it is appropriate since it is required to understand what is general for the whole portfolio and what is relevant only for some companies.

At the same time, one more aspect that should be taken into account by the Insight Hunter is the temporal factor. Thus, if it is found that the hypothesis is validated for some `dt_period`, it

is required to analyze the phenomenon in question during another period in order to understand its temporal instability or stability. Although this rule does not apply to all hypotheses, it is especially relevant in cases when the business meaning of the phenomenon studied depends on the temporal component.

The prompt also provides another rule known as pricing capture. Specifically, this term refers to the estimation of the degree to which a given risk is already reflected in the price. In order to estimate this parameter, it is required to compare the loss ratio of the segment with the loss ratio of the portfolio. If the segment loss ratio is similar to the portfolio loss ratio, the observed risk may already be reflected in the price, and the hypothesis of underpricing becomes weaker.

Finally, one final restriction concerns the use of additional filters. Specifically, it is stated that additional filtering should not lead to extremely small segments containing fewer than 1,000 records.

5.6 Final Insight Output and Traceability

Another aspect discussed in the final prompt relates to the output format and how to build a final insight from the collected evidence. This is relevant since the process of validating a hypothesis does not end when a working hypothesis is supported by validation.The agent must then formulate an insight summarizing the results of all Data Analyst iterations.

An important distinction mentioned in the final prompt is related to the difference between `hypothesis_nl` and `final_insight_nl`. While the former is the working hypothesis presented to the Data Analyst, the latter is the final product of the whole thread and should not introduce a new unsupported analytical claim. In other words, the final insight should not introduce a new unsupported idea; it should be a synthesis of all findings that are empirically supported.

As a result, according to the rules defined in the final prompt, `final_insight_nl` should synthesize findings of all iterations. The reason for such a requirement is that the refinement process can yield partial insights along the way, and the final output should incorporate all those findings instead of limiting itself to the final stage. Thus, the final insight needs to summarize all available evidence collected in the course of the process.

The list of required elements to include in `final_insight_nl` starts with the inclusion of the key metrics provided by the Data Analyst. In particular, this element includes such indicators as

loss ratio, claim frequency, average premium, cost per claim and others, depending on the topic of a particular analysis. The reason why this element should be included is that any metric used in the process of Data Analyst iterations is empirical data, and there is no need for the agent to invent its own numbers.

Materiality is another element that should be incorporated into a final insight. The prompt requires the presence of the absolute number of affected policies in case it is provided by the Data Analyst. This will help understand whether the issue in question is a relevant problem or concerns only a small portion of the portfolio. In this case, the prompt mentions that in the insurance domain, the existence of a certain percentage difference is insufficient, especially if it relates to a minor portion of the portfolio.

The temporal element should also be added to each final insight, since the prompt requires the indication of the `dt_period` for which the insights apply and states that the phenomenon should be stable over the period in question. This is done to prevent the situation where the final insight describes some episodic phenomenon as a stable one.

Another required element that relates to the domain specifics is financial impact in EUR, which is needed when it is indicated in the Data Analyst response. Financial impact should be expressed in the form of premium leakage, excess claim cost, underpricing gap or margin delta. Such a requirement is needed to link the results obtained during the Data Analyst iterations to their real-world importance.

Finally, the structure of the output provided by the agent is governed by the prompt's requirement to separate it into two parts: `[[OUTPUT_CANDIDATES_JSON]]` and `[[OUTPUT_REASONING]]`. The former part contains `final_insight_nl`, whereas the latter is responsible for documenting the whole process of obtaining the final insight, including batch parsing, exploration threads, DAA call and evidence collected along the way.

This separation of the output into two parts is relevant because it simplifies both automatic processing of the final insight and its manual inspection. In other words, while the first part provides a structured format of output, the second part provides the possibility of manually checking the trace of the process of obtaining such an output.

According to the requirements formulated in the final prompt, the agent should document its interactions with Data Analyst Agent in the reasoning part. To do so, it should indicate whether DAA was involved, how many payloads were used, how many iterations were conducted and provide a record of iteration reports generated during the process.

The list of required elements is supplemented with the condition that metric formulas should be included only if provided in the Data Analyst report. Otherwise, the agent should use "ND" instead of trying to guess the meaning of any metric. Such a requirement is important to distinguish what information is gathered empirically and what information belongs to the model's parametric knowledge.

6. Implementation of the Agentic Refinement System

6.1 Runtime Architecture and Project Structure

This chapter demonstrates how the architectural modules discussed earlier were translated into executable backend components. Given that the Insight Refinement component is the centerpiece of this thesis, the entire software platform is not covered. Instead, the focus remains on the backend portions that are directly relevant to the implemented refinement workflow.

Initially, the project began as a Proof of Concept. With that in mind, the primary objective at the outset was to get the core elements operational. This involved the Insight Hunter runtime, prompt configuration, tools layer, linkage with the Data Catalog, Data Analyst Agent integration, and management of the agent's outputs.

Consistent modularity and scalability were top priorities from the start. The central runtime module handles creation and execution of the Insight Hunter agent. Meanwhile, the prompt module dictates the agent's behavior. The tools layer enables the agent to interact with the Data Catalog and more via limited permissions. On the other hand, supporting modules deal with scope setup, result parsing, logging activities, and monitoring agent operation.

Therefore, the fundamental setup looks like this:

Code 6.1: Simplified backend project structure.

```
backend/
|-- agent_runtime.py
|-- prompts/
|   `-- IH_prompt.py
|-- tools/
|   |-- catalog_tools.py
|   |-- daa_tool.py
|   `-- storage_connector.py
|-- proto_insight/
|   `-- scope_config.py
```

```
|-- model_providers/
`-- utils/
    |-- output_parsing.py
    `-- logging/
```

This structure reflects the main execution flow of the Proof of Concept. The runtime module is responsible for instantiating the Insight Hunter, associating it with the selected model provider, loading the prompt configuration and exposing the available tools. The prompt module defines the behavioral rules of the agent, while the tool modules provide controlled access to external components such as the Data Catalog and the Data Analyst Agent. The remaining utilities support parsing, logging and inspection of the run after execution.

6.2 Data Catalog and Tool Layer

The first steps in the implementation involved testing the functionality of the agent built with Strands using simple prompts and requests, verifying that the tool and model were called correctly, and finally evaluating the output.

The second step was to build the Data Catalog, exposing the two YAML files discussed in Section 4.4 as two separate read-only tools so that the agent could retrieve catalog information only when needed during the reasoning process.

The choice not to include the YAML files in the Insight Hunter prompt was made to avoid unnecessarily burdening the context window in situations that do not necessarily require access to that information. Maintaining this type of external access therefore allows the agent to invoke the two tools only when truly useful.

The first tool returns the list of available tables along with their descriptions; it is designed to provide only a high-level overview of the domain in question.

The second tool, however, accesses a much more detailed YAML file, meticulously describing table names, corresponding schema, columns, data types, and textual descriptions.

This design separates semantic grounding from quantitative validation. The catalog tools allow the Insight Hunter to understand which data entities exist, how they are described and which variables may be relevant for a hypothesis. However, they do not perform joins, calculations or empirical tests. Quantitative validation is instead delegated to the Data Analyst Agent.

Below is a simplified version of the code that performs the above:

Code 6.2: Implementation of the Data Catalog tools exposed to the Insight Hunter.

```python
from strands import tool

_CACHE = {}

def _load_yaml(key: str, bucket: str):
    cache_key = (bucket, key)


    if cache_key in _CACHE:
        return _CACHE[cache_key]

    data = read_data_catalog_from_yaml(
        file_key=key,
        bucket_name=bucket
    )

    _CACHE[cache_key] = data
    return data


@tool
def dp_search_tables(question: str) -> dict:
    """
    Read-only tool used to retrieve the available table
    descriptions from the Data Catalog.
    """
    raw = _load_yaml(TABLE_DESCRIPTIONS_KEY, BUCKET_NAME)
    return raw

@tool
def dp_get_table_schema(table_name: str) -> dict:
    """
    Read-only tool used to retrieve the schema of a specific table.
    """
    raw = _load_yaml(TABLE_SCHEMAS_KEY, BUCKET_NAME)


```

```
    key_map = {k.lower(): k for k in raw.keys()}
    real_key = (
        table_name
        if table_name in raw
        else key_map.get(table_name.lower())
    )

if not real_key:
    return {"error": f"table '{table_name}' not found"}

table = raw[real_key]

return {
    "table": real_key,
    "location": table.get("location", ""),
    "table_description": table.get("table_description", ""),
    "columns": table.get("columns", {}),
}
```

Using the `@tool` decorator makes these functions available within the Strands agent loop. As explained in previous chapters, tools are often custom functions that the model sees as possible external actions that can be selected during reasoning.

The model decides when a tool is useful, while the actual execution is handled by the deterministic backend code. The returned metadata then becomes a new observation available to the Insight Hunter.

A simple cache was used to reduce unnecessary latency. This avoids loading the same catalog documents multiple times during execution.

The quality of the YAML files and the descriptions associated with the tools was tested by inhibiting calls to the Data Analyst Agent so that Insight Hunter could rely only on the input batch and the catalog files, without statistical feedback from the DAA. This strategy allowed to maximize the separation between semantic grounding and empirical validation and test how far the Insight Hunter could push its creativity with those inputs alone.

The YAML files are loaded via a storage connector. In the prototype, the catalog files are stored in an AWS S3 bucket and accessed at runtime through this connector. The retrieved YAML content is then parsed and converted into Python dictionaries, which can be used by the Data Catalog tools exposed to the Insight Hunter.

Below is an example of the code used:

Code 6.3: Storage connector used to load YAML catalog files.

```
import yaml

def read_data_catalog_from_yaml(file_key: str, bucket_name: str) -> dict:
    response = storage_client.get_object(
        Bucket=bucket_name,
        Key=file_key
    )

    data = yaml.safe_load(response["Body"].read())
    return data
```

6.3 Insight Hunter Runtime and Prompt Assembly

In the implementation, the Insight Hunter prompt is not treated as a single static string. Instead, it is assembled as a modular backend artifact composed of role definition, domain context, scope constraints, tool-use rules, validation requirements and output-format instructions. This design makes it easier to modify specific parts of the agent behavior without rewriting the entire prompt.

From an implementation perspective, these prompt components had to be connected with the available tools and with the execution loop of the agent. The Strands SDK framework made this possible.

It was vital to write the Insight Hunter prompt according to precise rules to ensure high agent performance. The first step was to write it in a modular manner, that is, divided into logical parts, each focused on a specific aspect of the agent's behavior during the various phases of the run: explaining its role, what it had at its disposal, how to use the tools, and when. This made it possible to version the prompt multiple times without rewriting it from scratch.

This modular structure was also useful because some parts of the prompt depend on the execution context. For example, the scope constraints can be generated from a configuration object, so that country, line of business, vehicle type, vehicle use, customer segment, and analysis period are not only hardcoded in the prompt, but can also be represented as structured runtime parameters. Similarly, the temporal block can be generated using the current reference year, allowing the prompt to guide the agent toward the correct `dt_period` logic.

Code 6.4: Simplified logic for assembling the Insight Hunter prompt.

```
def build_insight_hunter_prompt(scope, current_year):
    return "\n".join([
    ROLE,
    CONTEXT,
    CATALOG_TOOLS,
    DAA_TOOL,
    build_scope_constraints_block(scope),
    PRECISION,
    build_dt_period_block(current_year),
    PHASE_0,
    PHASE_1,
    PHASE_2,
    PHASE_3,
    FINAL_INSIGHT_RULES,
    SUMMARY_FORMAT,
    MANDATORY_OUTPUT_ORDER,
    ])
```

The code shown is just a draft of the logic followed in the prompt. It demonstrates how the writing order is important for the agent's understanding. Above all, it highlights how, despite creativity being paramount, specific steps were needed to ensure a minimum of stability in the initial analysis.

Precisely because the prompt is the basis of the entire system's functioning, it can be considered part of the backend configuration rather than static text manually inserted into the agent call.

Once the prompt is assembled, the agent can be instantiated by passing the model, the system prompt, and the tools available during execution. In simplified form, the runtime can be represented as follows.

Code 6.5: Simplified runtime structure for creating the Insight Hunter agent.

```
agent = Agent(
    model=model,
    system_prompt=insight_hunter_prompt,
    tools=[
        dp_search_tables,
        dp_get_table_schema,
```

```
        daa_tool,
    ],
)
```

The code clearly demonstrates the intention to keep the model provider layer separate from the agentic logic. This choice provides the possibility of being able to change the LLM model very quickly, requiring only technical changes at the prompt to adapt it to the model provider. In fact, at the beginning it was chosen to use OpenAI models, while in the second half of the project we switched to Claude models. This transition showed that the prompt could not be treated as completely independent from the underlying model. Different models may react differently to the same instructions, especially in terms of tool use, long-context behavior, strictness in following output formats and sensitivity to detailed constraints.

The runtime therefore acts as the point where all previous design choices become executable. The input batch provides the starting material, the prompt defines the behavior, the tools expose external capabilities, and the model performs the reasoning steps. Strands manages the loop that connects these elements, allowing the Insight Hunter to move between natural-language reasoning and tool-based observations.

Overall, this implementation combines static prompt components with dynamically generated blocks. Stable behavioral and tool-use rules remain reusable across different runs, while context-dependent constraints are injected from the runtime configuration. This approach preserves consistency in the Insight Hunter's behavior while allowing the same prompt assembly logic to be adapted to different analytical scopes and reference periods.

6.4 Data Analyst Agent Integration

Once we completed the integration of the Data Catalog, the next step was connecting the Insight Hunter to the Data Analyst Agent. This step can be considered one of the most complex from an implementation standpoint, as it turned the system from an agent capable of reasoning about metadata into an agent able to perform an external validation of its insights.

Initially, the approach to validation did not prove to be stable. The main reason for such behavior lay in how the Data Analyst Agent was used to validate the hypotheses, through a less efficient, flexible, but harder to control execution of Python-based calculations using a built-in Strands tool called `python_repl`. Although this solution provided some flexibility in terms of implementing validation on different datasets, the process could be considered slow and hard to

track.

That is why, in future iterations, a different direction was taken by implementing validation using SQL queries and metrics calculated based on the given data. By doing that, two main goals were achieved: first of all, this approach aligned the functionality of the Data Analyst Agent with the purpose of the thesis by making the agent not return an analytical reply generated by the model, but rather create reports on the basis of database querying. Secondly, this implementation turned the validation process into something more similar to a proper data analysis, where it is possible to see the actual numbers instead of just a simple Yes or No.

At the implementation level, the Data Analyst Agent is exposed to the Insight Hunter through a Strands tool. Choosing a more structured approach to implementing logic using Strands tools allowed the two layers to be kept separate, with the Insight Hunter responsible for reasoning and formulating hypotheses, while the second layer performs the analysis and validation as part of the refinement process.

Another important thing to highlight here is the use of the decorator `@tool(context=True)`. Unlike the simple approach with the decorator `@tool` without parameters, the latter allows to access the context of the whole agent during the execution of the tool's function. It means that tool is capable of not only processing input and generating some output but also reading or even updating the internal state of Insight Hunter.

Here is a simplified version of the logic implemented by this tool:

Code 6.6: Simplified implementation of the Data Analyst Agent tool.

```
@tool(context=True)
def daa_tool(query: str, tool_context: ToolContext) -> str:
    ih_agent = tool_context.agent

    calls = ih_agent.state.get("daa_calls") or 0
    ih_agent.state.set("daa_calls", calls + 1)

    results = ih_agent.state.get("daa_results") or []

    payload = json.loads(query)

    if isinstance(payload, dict):
        candidates = [payload]
    elif isinstance(payload, list):
```

```
        candidates = payload
    else:
        return "[daa_tool] ERROR: invalid payload"

    output_list = []

    for cand in candidates:
        hypothesis = cand.get("hypothesis_nl")

        question = (
            additional_instructions()
            + "\n"
            + "This is the hypothesis under analysis:\n"
            + hypothesis
        )

        token = get_token(
            email=os.getenv("DAA_EMAIL"),
            password=os.getenv("DAA_PASSWORD"),
        )

        raw_response = ask(token=token, question=question)

        output_list.append(raw_response)

        results.append({
            "insight": hypothesis,
            "report": raw_response,
        })

        ih_agent.state.set("daa_results", results)

    return output_list
```

As we can see from this snippet, the tool receives the request in JSON format and extracts the value from the field `hypothesis_nl`. After this, it creates the question to send to the Data Analyst Agent, gets an authentication token, calls the endpoint and returns the response. The latter is also stored in the state of the Insight Hunter in order to be used in the following refinement steps. It is important to clarify that the SQL generation and execution logic is not implemented

directly inside the wrapper shown in Code 6.6. The wrapper exposes the Data Analyst Agent as a callable tool for the Insight Hunter and forwards the natural-language hypothesis to the external DAA service. The SQL-based validation is then performed inside the Data Analyst Agent, which decomposes the hypothesis into one or more analytical queries, executes them on the database and returns a report containing the resulting metrics and, when available, the corresponding query traces. For this reason, the wrapper code does not explicitly contain SQL statements, even though the validation process described in the reports is SQL-based.

While in the mature version of the prompt the agent is instructed to submit one self-contained `hypothesis_nl` per validation call, the tool itself is more flexible and accepts both a single JSON object and a list of candidate hypotheses. This implementation choice leaves open the possibility of extending the system toward a multi-thread refinement workflow. However, this possibility is not fully exploited in the implemented Proof of Concept. The reason is that the current version mainly relies on the Strands agent state to preserve run-level information, such as the number of DAA calls and the reports collected during the run. Managing several refinement threads would require a more explicit orchestration layer, with persistent storage for thread identifiers, intermediate hypotheses, validation reports, decisions and final thread-level outputs.

Another implementation detail worth mentioning is that the tool allows enriching the request sent to the Data Analyst Agent. Prior to making the call, the tool loads some extra instructions and a YAML file containing descriptions of some key features that may play a role in the validation process. Thus, it allow the DAA to receive not just a hypothesis in plain language, but also a technical background needed to correctly interpret all the variables and produce the correct analysis.

The tool also implements persistence of the report. Each reply from Data Analyst Agent is saved in the report file associated with the current run of the agent. If configured, the content of this report will also be persisted to an external storage. Such a measure is quite essential since it allows the calls to the DAA to be inspected after the end of the run.

From this example, it is possible to see the role of Strands in the implementation clearly enough. The use of the Strands framework is not limited to sending some prompt to the model. What is achieved here with this implementation is the construction of a running environment in which the model is able to choose some tools to work with, receive observations, modify the state, and proceed with further reasoning. In this case, an observation received by the `daa_tool` is not just a value helping in reasoning, but the empirical report affecting the decisions made by

the agent.

This step can be viewed as the moment when the agent becomes fully capable of performing refinement based on data-driven evidence. Earlier in development, the agent had the possibility to reason about its input and metadata. Now, by implementing calls to the DAA in a structured way and storing all this in the running state, the agent is given the chance to verify its insights against factual data

6.5 Output Parsing, Logging and Run Traceability

Besides implementing the Data Analyst Agent, another key consideration when integrating the Insight Hunter related to managing its output and creating run traces.

Since the final response of the Insight Hunter consists of a structured JSON object and a reconstruction of the refinement process leading to the conclusion, utilities capable of separating and handling these components had to be implemented.

Specifically, it was necessary to create a mechanism to separate and extract the JSON object from the final response. To achieve this goal, the final prompt included instructions on how to structure the response: there should be one main response block followed by a reasoning section that includes a reconstruction of the iterations, calls to the Data Analyst Agent, and evidence used to come up with the final insight. Hence, a lightweight utility for parsing responses was needed.

It was therefore required to implement an auxiliary function able to extract contents between two specific tags. Namely, here is a simplified implementation:

Code 6.7: Simplified utility for extracting tagged output blocks.

```
import re

def extract_block(text: str, start_tag: str, end_tag: str) -> str:
    pattern = re.escape(start_tag) + r"(.*?)" + re.escape(end_tag)
    match = re.search(pattern, text, flags=re.S)
    return match.group(1).strip() if match else ""
```

The aforementioned function does not interpret the extracted information semantically, but it enables the relevant parts of the response to be separated. It is used to separate the JSON object from the rest of the output and to obtain the reasoning block. This means that the response generated by the agent is no longer just free-form text, but can be managed by backend software.

In addition to developing a solution for parsing the agent's output, logging was implemented. As already mentioned above, logging is especially important for agentic systems because the final output depends on several stages: messaging, tooling, results obtained from using tools, intermediate observations, and the final output.

As a result, logging provides runtime traceability. A utility based on callbacks integrated into the Strands runtime was therefore developed. These callbacks collect information about each run: run ID, used model, final message, number of cycles, number of tool uses, number of results, and other details. The following simplified data class is an illustration of how this information can be collected:

Code 6.8: Simplified data class for storing agent run logs.

```
from dataclasses import dataclass
from typing import Optional, List

@dataclass
class AgentRunLog:
    run_id: str
    start_time_utc: str
    end_time_utc: Optional[str]
    user_prompt: str
    final_response_text: str
    cycles: List[str]
    tool_uses: List[dict]
    tool_results: List[dict]
    model_id: Optional[str] = None
```

During execution of an agent, the aforementioned callbacks intercept events generated by the agent, allowing the flow of the execution process to be tracked. Whenever a tool invocation occurs, the name of the tool, invocation ID, and input passed to the tool are saved. Once the result arrives, the result ID, result status, and the result itself are saved. Consequently, it is possible to identify which tool invocations were made and what their results looked like.

Here is a simplified implementation of an event parser utility:

Code 6.9: Simplified event parser for tool calls and tool results.

```
def extracttool_events(content):
    tool_uses, tool_results = [], []

```

```
    if not isinstance(content, list):
        return {
            "tool_uses": tool_uses,
            "tool_results": tool_results,
        }

    for item in content:
        if "toolUse" in item:
            tool_uses.append({
                "tool_name": item["toolUse"].get("name"),
                "tool_use_id": item["toolUse"].get("toolUseId"),
                "input": item["toolUse"].get("input"),
            })

        if "toolResult" in item:
            tool_results.append({
                "tool_use_id": item["toolResult"].get("toolUseId"),
                "status": item["toolResult"].get("status"),
                "content": item["toolResult"].get("content"),
            })

    return {
        "tool_uses": tool_uses,
        "tool_results": tool_results,
    }
```

Each run is saved as a JSONL record containing the final message generated by the agent plus information about the events. Thus, a separate run log contains only one JSON object per agent execution. This is a simple and effective solution for a Proof of Concept because logs can easily be stored in one file and processed later.

This implementation aspect was critical for monitoring the behavior of the Insight Hunter because it would not have been easy to understand whether the final result came from the correct use of the Data Catalog, a successful call to the Data Analyst Agent, the formulation of an appropriate hypothesis, or an incomplete response from the model. Thanks to logging and the collection of tool calls and results, it became possible to understand which events led to the final output.

Logging turned out to be useful during testing. For instance, it enabled checking whether the

model had actually performed the queries to the Data Catalog before formulating a hypothesis, whether the payload passed to the Data Analyst Agent was valid, whether the number of DAA calls matched the requirements stated in the prompt, and whether the output block could be successfully parsed.

Overall, the implementation described in this chapter translates the proposed refinement architecture into an executable backend workflow. The Data Catalog tools provide controlled semantic grounding, the Data Analyst Agent tool enables empirical validation, and the parsing and logging utilities make the generated outputs and intermediate events inspectable after the run. These implementation choices are essential for the evaluation discussed in the next chapter, because they make it possible to analyze not only the final insight, but also the sequence of tool calls, reports and decisions that produced it.

7. Evaluation and Results

7.1 Evaluation Strategy

The approach for the evaluation of the proposed agentic workflow was chosen taking into account the specific nature of the thesis contribution. The objective is not to test the accuracy of a predictive model or supervised machine learning method, but about assessing whether the agentic framework allows the insight refinement stage to be reached in a coherent manner.

There are two main levels of system evaluation required: one regarding the behavior of the system backend components and another concerning the reasoning itself. On the first level, it is necessary to test whether each of the modules works correctly and whether it is possible to trace the overall flow of the execution. This concerns the agent runtime, the prompt construction module, the Data Catalog tools, the wrapper of the Data Analyst Agent, the output parser utilities, logging of the results and storing of all data related to the executed run.

On the second level, the results produced by the Insight Hunter when performing the full cycle of insight refinement should be examined. This means that the point is not only whether the agent runs successfully and without errors. It should also be assessed whether the formulated hypothesis is precise, backed up with data and consistent with the reports generated by the invoked tool.

The developed test suite included several categories covering different elements of the backend such as prompt creation, setting the scope for queries, execution of tools, YAML file loading, calls to the Data Analyst Agent API, output extraction and logging of runs.

Regarding the qualitative evaluation of the Insight Hunter's work, some example runs of the agent should be considered. They allow the Insight Refinement process carried out by the system to be assessed: how the set of proto-insights is parsed, what hypothesis is formulated, how the Data Analyst Agent is called and what happens next to the result. Such an analysis allows more and less successful refinement sessions to be distinguished.

It follows that the evaluation criteria include both technical and analytical parameters. Thus, a run is successful when the agent acts as expected, uses the available tools, forms a hypothesis that can be verified, receives and processes quantitative evidence in a meaningful way and provides structured output. All other runs provide valuable information about the current limitations

due to the model's work, context limitations, issues with formatting the output of the invoked service, delays in working with an external service and weak validation evidence.

As this work is a proof-of-concept, the evaluation strategy chosen above is appropriate. Firstly, it allows testing whether the proposed architecture can be realized in practice and whether its components interact correctly with each other within the framework of a refinement loop. Secondly, the identification of possible problems allows areas where improvements may be needed to be identified.

At the same time, the evaluation should not be interpreted as a statistically exhaustive benchmark. Its purpose is to assess feasibility, traceability and qualitative refinement behavior in representative experimental conditions.

7.2 Experiment Log and Timing Analysis

For the evaluation of the refinement workflow, the experiment log prepared in the course of development of the Proof of Concept was used. This log was not meant to serve as a basis for a comprehensive benchmark, but instead to document how the various configurations influence the execution of the Insight Hunter and the final quality of the refinement result. Here, a *run* refers to the execution of the entire agentic workflow for a given input, while an *iteration* refers to a refinement step in which the Insight Hunter formulates, evaluates, or updates an analytical hypothesis on the basis of the available evidence. In this section, the expression *DAA query* does not refer to the high-level invocation of the Data Analyst Agent tool made by the Insight Hunter. Instead, it refers to an individual SQL validation query generated or executed internally by the Data Analyst Agent during its validation activity. Therefore, a single refinement iteration may correspond to several DAA queries, because the Data Analyst Agent can decompose one hypothesis into multiple checks, benchmarks, metric calculations, query refinements, or robustness analyses. This distinction is important for interpreting the experiment log correctly: a run may complete a limited number of refinement iterations while still producing a much larger number of internal DAA queries.

The experiment log was divided into three blocks of information: test configuration, test results and timing analysis. Test configuration contains input, max iterations, DAA time limit and execution environment used in the run. As for input, it could consist of either a single proto-insight or of a group of proto-insights. The distinction is important because clusters usually

contained a widely varying number of elements, from small clusters with 6 or 8 proto-insights up to bigger clusters of 31 or 147 proto-insights.

The test result block records the total duration of the run, number of iterations performed, number of DAA queries, number of timeouts in DAA queries and final report generated by the system. This block of information turned out to be quite useful since the max number of iterations stated in the prompt could not always precisely match the real number of iterations performed by the system. Usually the agent was able to respect the specified limits; however, in some early iterations, the agent could either finish the process sooner or exceed the max number of iterations slightly.

The information provided in the timing analysis helps to see what is influencing the run execution time. These factors could include the number of proto-insights, max iterations, DAA time limit or even the validation service's stability. Moreover, it is necessary to consider that all the tests in the log have been performed in the QUAL environment, which is not as performant as the production environment. Thus, the recorded values should be seen as relative to other QUAL experiments, rather than exact measures of the runtime.

	Test Configuration					Test Results					Timing Analysis			
ID	(single protoinsight,	Input	of max iteration	timelimit (minutes)	ent (qual/ prod)	Total test duration	number of iteration	number of DAA	output Report	of DAA queries in	for an Iteration	for iteration	for iteration	for iteration
0	single protoinsight	IF: policyhold	5	5	QUAL	4h	5	148		22	13min	35min	28min	1h 17min
1	single protoinsight	IF: policyhold	5	10	QUAL	2h 46m	4	108	I'll analyze this proto-	17	9min	27min	17min	50min
2	cluster n°70 (12 proto-	**IF: 'changed_**	5	10	QUAL	3h 22min	6	150	I'll analyze this batch	20	9min	17min	9min	32min
3	single protoinsight	IF: policyhold	5	5	QUAL	4h 36min	5	131	I'll analyze this proto-	24	32min	55min	14min	1h 7min
4	single protoinsight	IF: policyhold	10	5	QUAL	5h 6min	6	162	I'll analyze this proto-	28	21min	40min	21min	1h 24min
5	cluster n°34 (6 proto-insight)	IF: 'changed_f	5	10	QUAL	2h 25min	7	159	I'll analyze this batch	15	10min	19min	9 min	36min
6	cluster n°120 (15 proto-	**IF: 'dsc_fuel_**	5	10	QUAL	3h 23min	6	125	I'll analyze this batch	12	8min	33min	18min	1h
7	cluster n°34 (6 proto-insight)	IF: 'changed_f	5	10	QUAL	4h 49min	8	215	I will analyze	21	17min	32min	11min	1h 1min

Figure 7.1: Experiment log and timing analysis.

One of the problems revealed during the experiments was the potential risk of uncontrolled exploration. As already discussed, in the early prompt, the Insight Hunter was encouraged to explore new angles of the topic and act more creatively. On the one hand, such behavior helped to discover non-trivial insights, while on the other hand, it also caused a risk of indefinite refinement. The agent would go on generating and validating additional hypotheses and asking validation services about their validity, without having an appropriate stopping criterion.

To address this problem, the prompt constraints were tightened significantly, particularly

with respect to the maximum number of iterations and the limits imposed on interactions with the Data Analyst Agent. The maximum number of iterations has a direct impact on both the quality of the refined insight and the overall cost of the execution. A too low number of iterations would make the refinement shallow, resulting in a rather primitive interpretation of the proto-insight. At the same time, an excessive number of iterations would increase the duration of the run, as well as the cost of API calls and the number of timeouts.

Therefore, the number of max iterations for the run has to be optimized to balance the exploration of possible angles with the duration of the refinement. To check what would happen, the experiments with single proto-insight and five and ten iterations have been conducted. In particular, there was a run where the max number of iterations was set to five and the DAA time limit was set to five minutes. As a result, the run took approximately four hours to complete, finished in five iterations, requested 148 DAA queries and produced 22 timeouts. This does not mean that the Insight Hunter invoked the Data Analyst Agent 148 times. Rather, the number refers to the internal SQL validation queries generated within the DAA during the refinement process.

Another run has been conducted with five maximum iterations and ten minutes DAA time limit. That experiment took two hours and forty-six minutes to complete, finished in four iterations, generated 108 DAA queries and produced 17 timeouts. The comparison above suggests that simply specifying the maximum number of iterations is not sufficient. The duration of the run depends also on the interaction between the prompt, timeout behavior, number of queries and DAA efficiency in processing the queries.

A further run with ten max iterations and five minutes DAA time limit took five hours and six minutes, completed six iterations, generated 162 DAA queries and produced 28 timeouts. This suggests that an increasing maximum number of iterations does not automatically lead to better results. If the validation remains expensive or unstable, a higher number of iterations would result in an increased number of API calls and timeouts. This was one of the reasons why the final configuration has limited max iterations.

The second major issue related to the refinement process was validation. Initially, the DAA validated a hypothesis using the `python_repl` tool. Although the Python-based validation was very flexible, allowing the DAA to generate new scripts and perform calculations, it had some problems. Firstly, there could be an issue of unpredictably large and potentially erroneous Python code. Secondly, even if the code did not have any syntactic mistakes, it could fail to

properly calculate the indicators.

From a methodological perspective, this experiment demonstrates the following. It shows that while a language model-based agent is suitable for analytical reasoning, generating executable code for calculations is not always safe for the refinement process. A single mistake in a calculation may result in completely different conclusions being drawn. Therefore, it is not recommended for the DAA to rely solely on generated code.

In order to address the problem, a modified DAA approach was adopted, in which hypothesis validation was performed through SQL-based data retrieval and metric calculation. It should be noted that the use of SQL did not mean that all validation steps were pre-defined. The Data Analyst Agent still analyzed the hypothesis independently and determined which queries were required to support, reject, or refine it. However, this validation was performed by generating and executing SQL queries inside the DAA service, rather than by relying on freely generated procedural Python code.

It seems that this modification contributed greatly to the increase of control and effectiveness of validating procedure. SQL language allows to sort and filter the database, aggregate and calculate various metrics directly in the database. Compared with the previous Python-based version, this approach reduced the dependency on freely generated procedural code and made the relationship between hypothesis, retrieved data and calculated metrics easier to inspect. This technique helped to establish relations between hypothesis, data and metrics obtained during the work with the data.

Nevertheless, there were also drawbacks in this approach as well. The problem was that some queries generated by Data Analyst Agent could require too much processing time, retrieve too many data or cause additional load on the whole AWS environment. It would result in spending more time for operations with data. Nevertheless, despite its weakness, it appeared to be more effective compared to the approach with Python scripts.

The difference in performance of the two DAA versions could be clearly seen by comparing the test runs with these DAA versions. For instance, two test runs with cluster number 34, consisting of 6 proto-insights, have taken two hours and twenty-five minutes and four hours and forty-nine minutes. During those runs, seven and eight iterations were performed, respectively, 159 and 215 DAA queries were generated and 15 and 21 timeouts occurred. It can be seen that while the process works, the validation process is relatively expensive.

NEW DAA VERSION														
8	cluster n°34 (6 proto-insight)	IF: 'changed_f	5	8	QUAL	1h 45min	5	62		3	11min	21min	8min	38min
9	cluster n°34 (6 proto-insight)	IF: 'changed_f	10	8	QUAL	3h 50min	9	118		7	14min	25min	11min	48min
10	cluster n°3 (8 proto-insight)	**IF: 'owner_ag**	10	8	QUAL	1h 55min	6	70		3	10min	19min	7min	35min
11	cluster n°100 (31 proto-	IF: 'owner_ag	10	8	QUAL	1h 45min	5	58		2	12min	21min	6min	41min
12	cluster n°98 (147 proto-	**IF: 'owner_is**	10	8	QUAL	1h 53min	5	61		2	14min	23min	5min	42min
13	cluster n°60 (147 proto-	**IF: 'VELOCITA**	10	8	QUAL	1h 52min	5	59		2	13min	22min	6min	40min

Figure 7.2: Comparison between the two Data Analyst Agent versions.

When the new DAA version is used, a similar experiment with five max iterations and eight minutes DAA time limit takes only one hour and forty-five minutes, completes five iterations, generates 62 DAA queries and produces three timeouts. The timing profile is noticeably better: the average duration of iterations is now twenty-one minutes and max iteration duration is thirty-eight minutes. It can be seen that introducing a new version has decreased the number of validation attempts and number of timeouts.

Finally, another experiment with the new DAA was performed with cluster number 34. This time the maximum number of iterations was set to ten and DAA time limit was set to eight minutes. The experiment takes three hours and fifty minutes to complete, makes nine iterations, generates 118 DAA queries and produces only seven timeouts. This result is interesting because it suggests that the new version made the validation process more manageable. At the same time, however, it increases the number of iterations performed.

It also became evident from these experiments that the execution time depended not solely on the number of proto-insights in the input. For instance, a cluster with 8 proto-insights took about one hour and fifty-five minutes, executed 6 iterations, made 70 DAA queries and generated 3 timeouts. A cluster with 31 proto-insights took about one hour and forty-five minutes, executed 5 iterations, made 58 DAA queries and generated 2 timeouts. A pair of bigger clusters with 147 proto-insights took one hour and fifty-three minutes and one hour and fifty-two minutes, respectively. Each executed 5 iterations, made 61 and 59 DAA queries and generated 2 timeouts. Clearly, the input size alone was not the determining factor in terms of feasibility. Prompt constraints, DAA validation technique and query complexity played a greater role here.

Meanwhile, increasing the batch size posed another kind of constraint: the need for more careful management of the context in which the synthesis occurs. An input containing many proto-insights provided more grounds to work with and resulted in potentially richer synthesis. At the same time, it also created a bigger burden on the available context window. It would be

required to combine the prompt, all the Data Catalog information, the generated intermediate reasoning as well as the DAA reports. Once the total context volume grew too big, it could cause the model to ignore or forget something important, generate an unstable output or stop the generation process completely. Based on the experiments, batches up to roughly 150 proto-insights appeared feasible in the tested setting, while larger inputs risked saturating the available context window.

Finally, a contributing factor for the variation in execution times was the fact that the runs were done in the shared QUAL environment and thus depended on external services. Server load and the number of other users interacting with the same environment at any point could influence the duration of a given run. This explained, at least partially, why a correlation between execution time and other factors such as input size or number of iterations was not absolute.

It was observed that the DAA timeouts were not just some kind of technical error. On the contrary, they often served as a good indicator that something went wrong with the validation procedure. For instance, in some runs the validation component insisted on executing rather complex queries, which did not fit into the allotted time frame. This made the run longer and, in some cases, reduced the value of the collected evidence. This finding implied that having too much analytical freedom in the system could eventually prove to be a problem. The agent needed to have enough leeway to find valuable angles but also enough constraints to limit the costly validation paths that would yield no usable evidence.

For this reason, later versions of the prompt introduced a number of behavioral requirements. Specifically, the agent was required to formulate specific and meaningful hypotheses, avoid overly generic or vague analysis, limit the number of validation attempts, adhere to the scope of the analysis and provide a structured output format at the end. Structured output would be facilitated by using JSON code blocks and reasoning summaries throughout the report. This did not make the generation process strictly deterministic, but it helped increase its controllability and predictability.

Therefore, this experiment log suggests that the refinement workflow used in the final version of the Proof of Concept was formed during a series of practical optimizations. Initially, there was a more open-ended setup, with more open-ended exploration. The DAA validation procedure involved flexible Python queries that provided more freedom. Eventually, the agent was asked to perform iterations, formulate specific hypotheses, restrict the number of validation attempts, use only SQL validation and manage timeouts more carefully.

As a result, this experimental phase helped identify a more stable proof-of-concept configuration. There were clear trade-offs between creativity and constraints, with the latter being crucial in terms of collecting usable evidence in an acceptable period of time.

7.3 Report-Based Evaluation of the End-to-End Refinement Flow

After the time analysis, the next evaluation involved the analysis of the refinement reports produced by the Insight Hunter. As mentioned previously, one contribution of the proposed approach is not a supervised classification model. Thus, in addition to numerical results like accuracy or recall rate, the assessment should include an analysis of the output report structure and content. Specifically, the question is whether the workflow is capable of transforming the proto-insight into a more precise, empirical and robust final insight.

To test this ability, two exemplary reports generated by the Insight Hunter have been selected. The two reports do not apply to the same input cluster and therefore cannot be compared using before/after criteria on the same input. Rather, they represent two phases of the Proof of Concept workflow evolution and therefore allow analyzing the change in output traceability.

In particular, the first report refers to an early phase of the development and applies to a batch of 8 association rules concerning MTPL insurance policies. The customer policyholders in the proto-insight belong to an alleged homogeneous group of elderly people, with ages between 75 and 81, with high association to zero-claim events and different loss ratio dynamics depending on fuel-type subcategories. The first hypothesis proposed to be tested by the Insight Hunter concerns the Diesel versus Other fuel types. Specifically, the hypothesis suggests that there is a materially worse loss ratio among the latter customers driven by high severity.

The second report concerns a later development phase and is applied to a set of 55 association rules representing cluster 23. This cluster concerns renewed policies in bonus-malus class 1, associated with a young or newly registered vehicle and Roadside Assistance coverage. The business interpretation initially suggested a high-value and loyal segment with cross-selling opportunities. During the process of refinement, however, the Insight Hunter explores whether the narrative is confirmed.

The difference between these inputs is crucial to understand. The goal of the analysis below is not to prove that a new version of the workflow outperforms an old one on the same business

problem. Instead, it consists in observing how the report structure changed over the time of workflow development. From the earlier report, we can see that the system was already able to analyze and query the input. The later report shows that the same basic functionality improved into a more systematic and controlled refinement process.

The report structure is crucial for the evaluation of agentic workflow performance. Even if the system generates a plausible conclusion, it is impossible to evaluate it if the reasoning path was not recorded in a readable way. Conversely, if the hypothesis testing procedure and final synthesis are documented, then the process becomes more traceable and more comprehensible. For this reason, this section focuses on the form and not only on the content of the reports under analysis.

It can be seen from the earlier report that the workflow is indeed able to follow the logic of hypothesis testing. The agent proposes a first hypothesis, invokes the DAA module, gets quantitative evidence and then changes the course of the analysis when the initial hypothesis turns out to be unsupported. However, the report includes many traces of technical information regarding the internal functioning of the workflow. SQL traces, query errors, query fixes, metric tables and other elements are present throughout the document. While useful for debugging purposes, this level of detail makes the output harder to understand as a pure refinement artifact.

```
[tool_call] redshift_query ...
[tool_result] redshift_query: error

SQL query:
...

Let me fix the syntax for Redshift:
[tool_call] redshift_query ...
[tool_result] redshift_query: success
```

Figure 7.3: Extract from the earlier report containing the traces of technical execution of DAA module.

This is a typical example of the earlier workflow. While the extract above demonstrates its transparency in terms of showing exactly what happens under the hood, it also demonstrates the weakness of the report. It includes too much detail about the technical process that does not help in understanding the analytical process.

Another extract from the earlier report is more relevant to the analytical power of the

workflow. In this case, the Insight Hunter discovers that the data contradict the initial hypothesis and consequently redirect the analysis to another hypothesis.

Result: NOT SUPPORTED (STRONG)

Key numbers: Diesel LR = 60.81%, Other LR = 61.47% (gap = +0.66 pp only); Diesel avg cost per claim = EUR 4,480, Other = EUR 3,989 (Other severity is lower); Hybrid LR = 79.91% (real outlier).

Mechanism: The Diesel vs. Other gap is negligible and the severity hypothesis is directly contradicted.

Decision: The original Diesel-vs-Other framing is abandoned. The data reveals a much more interesting signal: Hybrid vehicles for 75–81 year-olds carry a 79.91% loss ratio driven by very high severity.

Figure 7.4: Extract from the earlier report demonstrating hypothesis rejection and refinement redirection.

It can be observed that the first version of the agentic workflow was able to test the hypothesis, discover that the data contradict it and redirect the analysis to another hypothesis. This represents a good example of analytical thinking. However, the extract also demonstrates some problems with documentation and presentation: it is not easy to distinguish DAA outputs from the reasoning process.

From the later report, we can see that the workflow had already reached a more mature stage in terms of report production. Before actually invoking the DAA module, the Insight Hunter analyzes the input critically to detect potential problems in business interpretations. Such issues may include various denominator effects, baseline effects and inconsistency between business narrative and actual data.

Phase 1 – Critical Pre-DAA Assessment of Cluster 23

Cluster: Renewal + BM_1 + 2026-registered vehicle + Roadside Assistance → approximately 4% of the portfolio, confidence 65.6%, lift 6.45x, avg premium around EUR 1,240.

Critical structural tensions I must probe:

- The "2026-registered vehicle" paradox and its structural implications.
- The lift = 6.45x denominator problem.
- BM_1 at approximately 84% of policies is the modal class.
- Roadside Assistance at approximately 65% take-up across `AUTO_MOTO`.

Figure 7.5: Extract from the later report describing pre-validation critical assessment.

This can be considered a significant improvement because the agent is now able to think critically even before the DAA is executed. In other words, the refinement process becomes more focused. The questions posed to the validation tool become more specific and targeted.

This improvement is due to the design of the prompt used for generating a report. In this particular case, the instructions provided the agent with enough space to think critically, question potential business misinterpretations and make precise statements for testing before DAA validation.

The second improvement concerns the structure of the iteration summaries. In the later report, each iteration includes a summary of the hypothesis tested, result, key numbers, explanation and conclusion.

Iteration 6

Hypothesis tested: The lift = 6.45x for the cluster is inflated by including policies from other lines of business in the denominator, where BM_1 and Roadside Assistance cannot exist, making the within-AUTO_MOTO lift much lower.

Result: SUPPORTED (STRONG)

Numbers:
- Portfolio-wide BM_1 + Roadside Assistance rate: 19.98%.
- Within-AUTO_MOTO BM_1 + Roadside Assistance rate: 56.70%.
- Antecedent rate: 56.96%.
- Within-AUTO_MOTO correct lift: 1.005.

Explanation: The within-AUTO_MOTO lift = 1.005. Thus, there is no significant relationship once the denominator distortion is accounted for.

Decision: All six iterations comprehensively explored the cluster. The one finding worth synthesis is that TPL is the key driver behind premium differentiation between new-vehicle and older-vehicle BM_1 renewals.

Figure 7.6: Extract from the later report demonstrating an iterative approach.

This way of organizing the report makes it evaluable. Each iteration represents a miniature hypothesis test in the report. The agent's report is not limited to describing what happened after the DAA was called. On the contrary, it describes how this evidence changed the state of the refinement process. The explanation is necessary since the final insight cannot be generated through a single step. Instead, it is produced as a result of confirmations, contradictions and redirections.

In addition, the example provided demonstrates another reason for checking the baseline and denominator. The lift of the observed pattern might be high due to the presence of records in the denominator where the consequent is structurally impossible. In this case, the Insight Hunter identifies and empirically verifies the issue. As a result, the originally reported lift cannot be considered as a valid business signal. It should be broken down and analyzed in terms of an appropriate analytical scope.

A third important improvement concerns handling of negative evidence. In contrast to the

earlier report, the later report treats unsupported hypotheses not as errors in the process, but as means to exclude irrelevant explanatory variables. The Insight Hunter rejects the hypothesis that the cluster refers to a loyalty segment, refutes the enrichment of BM_1 and Roadside Assistance, eliminates the idea of higher premiums and shows that the high lift is just a denominator artifact. As a result, the output is much more precise.

This is a very important property of the later workflow. A weak refinement system could transform any proto-insight into an elaborate business narrative. A better system would be capable of rejecting unnecessary explanations. In other words, a strong refinement system narrows the narrative in cases when the evidence does not support it.

In comparison with the earlier report, the final output of the later system is better structured. In addition to the conclusion, it includes a JSON block with the final insight. In that way, it facilitates parsing, storage and analysis of the output.

> *The business impact of this finding is a pricing adequacy review: old-vehicle BM_1 renewals carry higher TPL loads than new-vehicle BM_1 renewals of equivalent engine power. There is no difference between coverage selection, renewal propensity, or engagement.*

Figure 7.7: An example from the later report demonstrating the final structured insight.

As shown by the final output, the refined insight differs significantly from the original cluster description. While initially, the target group seemed to be high-value customers marked by renewal, BM_1 and Roadside Assistance with a young car, the final insight focused on the pricing issue associated with the TPL component and vehicle age. This confirms that the process of refining insights was not a simple transformation of the proto-insights into narratives.

The comparison of the two reports can therefore be interpreted as a qualitative evaluation of the workflow. As we can see, the former report demonstrates that the agent was able to make validation calls and use quantitative evidence. However, it is quite technical, less readable and more difficult to evaluate as a refinement artifact. The latter one provides more precise formulations of hypotheses and more organized validation evidence, makes decisions after each iteration clearer, and finalizes its output in the form of a JSON block.

This improvement can be explained by some changes made in the Insight Hunter prompt and in the structure of the refinement report. The improvement was not due to the introduction of hypothesis generation itself, since earlier versions of the prompt already supported the creation

of multiple candidate hypotheses. Rather, the later version made the refinement process more disciplined by requiring analytical directions to be expressed as explicit hypotheses and by linking each DAA validation call to one self-contained `hypothesis_nl`. Moreover, the agent was prompted to consider scope boundaries, appropriate benchmarks and possible misleading interpretations before accepting the initial interpretation of the cluster. Another benefit was that the interaction with the Data Analyst Agent became easier to trace, due to more organized formulations of hypotheses and validation evidence in the form of refinement iterations. Finally, the prompt included information about the final output format, thus making it possible to separate the refined insight from the reasoning trace and provide it in a JSON block.

These modifications have not only improved the reports' readability but increased their evaluability as well. While in the former case the analyst had to trace the reasoning path from SQL queries, metrics and comments, in the latter one the reasoning path becomes much more explicit and takes the form of refinement iterations.

Overall, the evaluation shows that the contribution of the implemented workflow should not be assessed only in terms of execution time or number of completed iterations. Its main value lies in the ability to make the refinement process more structured, inspectable and evidence-driven. The experiments highlight the operational constraints of agentic refinement, while the report comparison shows that the final workflow is able to transform proto-insights into narrower and better supported insight candidates.

8. Conclusions and Future Work

8.1 Synthesis of Results and Contributions

As previously stated, the goal of this research was to study the possibilities of refining statistical patterns into valid business insights using an agentic refinement workflow. Clusters and association rules are susceptible to confounding factors, denominator effects, lack of interpretability, or insufficient empirical backing. In other words, the key issue addressed by this thesis is not producing new patterns, but transforming existing ones into insights through hypothesis testing and interpretation.

For this reason, the chosen solution consisted of the Proof of Concept of an agentic workflow for pattern refinement. The core of the system was the Insight Hunter, an agent based on the language model that would formulate analytical hypotheses, question them, and progressively refine them in cooperation with other system components. Those other components include the Data Catalog, offering semantic grounding by way of data structure exploration and domain metadata, and the Data Analyst Agent, performing empirical validation through quantitative analysis. It was decided to separate the interpretation and validation phases because they serve different purposes. While the Insight Hunter is responsible for interpretation and hypothesis formulation, it is not required to compute all indicators directly. Conversely, the Data Analyst Agent is responsible for empirical validation rather than for independently redefining the business meaning of the pattern.

In general, the main contribution of this research is the development of an agentic refinement process, which can bridge the gap between statistical patterns and actionable insights. The proposed system neither generates additional statistical signals nor replaces the business analyst. Rather, it helps with the intermediary step of questioning, contextualizing and validating statistical signals to produce an insightful message. This is achieved through a series of interactions between several agents and data sources that can be studied and improved further.

The first contribution is semantic grounding. As mentioned above, the Insight Hunter is not limited to working on the textual description of a pattern. With the help of the Data Catalog tools, it is able to investigate the underlying data and metadata and produce hypotheses that are consistent with the actual data structure. This is especially relevant in a domain like insurance

analytics, where the exact meaning of a variable often depends on the line of business, risk category, observation time span or even the type of insured party. In addition to reducing the risk of invalid claims, it ensures that the hypotheses will be more meaningful in terms of the business domain.

The second contribution is the interaction between reasoning and validation processes. While the Data Analyst Agent works as a validation module, it does not need to receive any complex instructions. It just needs to receive the natural language hypothesis formulated by the Insight Hunter. The `hypothesis_nl` field in the final JSON outcome acts as a contract between the modules and helps to keep the process structured and clear. In particular, it makes sure that the payload sent to the validation module is not ambiguous or too complex.

The third contribution is the design of prompts and interactions between agents. Over the course of development, the prompting strategy evolved from loose instructions into the behavioral specification. As a result, the final version of the prompt did not instruct the agent what task to solve; rather, it described the agent's behavior, scope, goals and interaction with other entities. In other words, the prompt defined how the agent explored the data, how it used the data analysis tools and interacted with the validator, and how it formulated the final insight. In this sense, the prompt acted as a mechanism for bounded autonomy: it allowed the agent to explore alternative analytical directions, while constraining its behavior through scope rules, validation requirements and structured output formats.

Finally, the last important contribution is traceability. In contrast to regular prompts, an agentic system produces complex output that cannot be evaluated without considering the way in which it was generated. In particular, the execution of an agentic process includes multiple steps and tool calls and leads to complex interactions with data, validation reports and other components. Therefore, traceability of an agentic output is essential to understand why certain hypotheses are formulated and how evidence influences the final message. To support traceability, several logging and parsing utilities were built during the project.

The evaluation described in Chapter 7 demonstrated that the agentic refinement workflow is indeed feasible. However, the results show that the effectiveness of the process depends on balancing freedom and control. Initially, the Insight Hunter was allowed to explore numerous analytical angles in order to support its creativity and flexibility. The results of experiments demonstrated that increased exploratory freedom does not necessarily lead to good results; on the contrary, it increases execution time and risks of failures.

The evolution of the data analysis module also influenced the outcome. From the initial flexible Python-based approach to the more constrained SQL-oriented validation process, it helped to ensure a clearer relationship between the hypothesis and validation results. Although SQL still allows for quite flexible data analysis, it ensured a more stable refinement workflow by increasing the transparency of the interaction between the hypothesis and the validation process. The experiment log indicates that this approach led to a reduction in timeouts and execution time.

However, the performance characteristics of the agentic refinement process turned out to be rather complex. While running the process on a batch of N proto-insights, it was initially assumed that increasing N would result in longer processing times. However, this assumption turned out to be incorrect: execution time was influenced by several other factors, including prompt constraints, behavior of the Data Analyst Agent, timeout values, context window size, and even the common execution environment. In other words, the agentic refinement process behaved like a system consisting of multiple components, each of which could influence the others.

As regards the qualitative characteristics, the evolution of reports was tracked over the course of experiments. Although the earliest reports indicated that the Insight Hunter is capable of formulating hypotheses, collecting evidence, rejecting unsupported assumptions and redirecting the analysis towards more productive paths, the reports tended to be too dense and unstructured. Later on, the output became more structured: it included pre-validation critical assessment, hypothesis testing iterations, key numbers and mechanisms, decisions, and a structured insight block at the end.

As a conclusion, this thesis demonstrates that an effective agentic refinement system should not only generate plausible hypotheses. It should also be capable of questioning a pattern, analyzing hypotheses in detail, validating them using quantitative analysis, rejecting invalid interpretations, and retaining the valid portion of the initial signal. It often turns out that an important part of the initial interpretation is false, and that the final insight differs significantly from the original one. In fact, the process of refinement is more about transformation than amplification.

This research has a few notable weaknesses. First, it must be remembered that this thesis provides only a Proof of Concept of an agentic system, not a ready-to-deploy solution. As regards control, various constraints are implemented via prompts, which improves the behavior

of agents and helps avoid random failures, but does not guarantee predictable performance. In turn, this means that the quality of output depends on the quality of proto-insights, the validity of the metadata, quality of validation reports, and ability to summarize evidence without overestimation. Moreover, large inputs and reports may cause problems due to excessive context memory pressure.

Second, the methodology of this evaluation also has some limitations. Since this work does not offer any predictive models, it would be impossible to apply a traditional evaluation approach based on accuracy or similar measures. Instead, the evaluation relied on backend tests, experiment logs, execution time analysis, and output reports. While this strategy was suitable to assess the Proof of Concept and demonstrate how the process evolved, it does not allow estimating the overall performance for all possible proto-insights. A full evaluation would require running multiple experiments on different inputs, possibly comparing different versions of the workflow, and even evaluating the quality of insights manually.

Despite those restrictions, this research provides proof of concept that agentic systems may indeed be useful in a specific part of analytics. Instead of focusing solely on the generative capabilities of LLMs, this thesis shows how a proper combination of components can facilitate a useful interaction. The system is based on semantics, validation, prompting, statefulness and traceability, all of which help to make the pattern refinement process more controllable.

8.2 Future Work

This proof of concept shows the working architecture of an agentic insight refinement system that helps refine proto-insights into insight candidates. As can be seen, the interaction between the Insight Hunter, Data Catalog and Data Analyst Agent may prove useful in the process of refining proto-insights into insights. However, several aspects remain open for further development and improvement.

Firstly, one can increase the amount of controls at the backend. At the moment, Insight Hunter receives constraints through prompts and the code itself. The current backend already supports several control functions, and in the future, it can do even more. For instance, in addition to controlling tools, providing metadata to the Data Catalog, interacting with Data Analyst Agent, managing states, storing output files, and logging the runs, the backend can control such aspects as limiting the number of times DAA calls can occur, rejecting invalid

hypotheses, interrupting the process of refinement once the analytical goal is reached, and disallowing unnecessary validations.

Another important direction concerns the extension from single-thread to multi-thread refinement. In the implemented Proof of Concept, each run follows a controlled single-thread configuration, where one candidate hypothesis is refined through a sequence of validation steps. A future version could allow several candidate hypotheses generated from the same proto-insight batch to open separate refinement threads. This would increase analytical coverage, but would also require a more explicit orchestration layer, persistent thread-level storage and stricter control of database queries, timeouts and intermediate reports.

In the previous chapter, it was shown how switching from Python validation to SQL validation greatly improved the system's performance. Yet, there still are cases when SQL validation may lead to overly complicated queries due to complex joins, aggregations, and comparisons of certain segments. One way to improve the system might be creating a layer that controls queries, evaluates the cost of each validation, simplifies queries with too complex logic, and blocks unnecessary actions.

Quality evaluation is another aspect worth discussing. Although the proof of concept presented here had enough data for evaluation, including backend behavior, log of the experiments, time analysis, and representative sample of reports, it would be useful to develop additional criteria for measuring the quality of the process of insight refinement. Some examples of possible evaluation criteria may include precision, level of empirical support, business relevance, novelty, traceability, DAA evidence compatibility, and output parsability.

Besides, there are numerous open questions related to insights and their management and storage. In the current version of the PoC, agents only retain their states until the end of a run. Yet, in a future version, one may want to introduce an external insights storage where both positive and negative insight instances will be stored. In such a way, the list of positive insights will contain only valid and empirically supported insights, while the negative list will contain rejected hypotheses, false hypotheses, denominator errors, and weak validation paths.

One possible approach to external insight storage may be a vector database. Thus, refined insights, hypotheses, DAA reports and the whole reasoning process may be encoded and saved in the database. Then, at the beginning of the next session, similar insights and hypotheses will be fetched and used for comparing the current task with previously analyzed ones. In this way, the language model would not be directly retrained, but it could use retrieved past cases as

contextual guidance during new refinement runs.

Of course, in order to create a more sophisticated proof of concept, one must add human-to-agent interaction capabilities. At the moment, everything works at the backend level, which means that there is no front-end interface where people can interact with the system. Hence, adding such an interface will be very helpful since it would enable users to review generated proto-insights, validate hypotheses, check DAA reports, interrupt refinement if necessary, and evaluate the utility of insight candidates.

As far as additional ideas for a more advanced PoC go, it is reasonable to give agents the ability to access factual information. So far, the main sources of factual information were proto-insights, Data Catalog, and DAA evidence. In a more sophisticated version of the PoC, agents could receive factual information from such sources as enterprise search, APIs, managed connectors, and even controlled web scraping of regulatory news, market events, macroeconomics, vehicle-market trends, weather, and territorial information. Access to such sources would be limited to approved sources, and retrieved documents would also be stored along with information about their origin.

In addition, one could improve the functionality of the Data Catalog by adding even more available metadata. At the moment, the Data Catalog provides metadata on tables, columns, and their properties; however, this process is associated with problems of context window. Consequently, in the future version, it might be useful to implement selective access to the Data Catalog, with metadata provided based on the hypothesis or compressed metadata included into the context of the agent.

Another point of improvement is extension of the analytical perimeter and its testing. In this thesis, the proof of concept was implemented specifically for the purpose of analyzing MTPL insurance cases with a certain list of indicators. However, in a more advanced version of the PoC, the same architecture could be tested on other lines of business or insurance domains. Thus, more tests need to be performed.

Finally, operational robustness is another factor to consider if this proof of concept is to evolve into a usable tool. Better error handling, external services monitoring, timeout handling, artifact storage and log inspection should be implemented.

8.3 Final Remarks

To conclude this work, the most pressing issue is no longer the replacement of the business analyst by artificial intelligence, but how it can be used to give a different perspective on the data and derived insights. With the developed Proof of Concept, it is possible to see that language models can indeed become very useful precisely in this niche. They are not decision-makers, but can act as machines to process statistical observations, generate hypotheses and check initial assumptions.

The mentioned feature resembles the topic addressed by Marcus du Sautoy in his book *The Creativity Code* [20]. Discussing the opportunities provided by artificial intelligence to become creative, the author does not confine himself to technical aspects of using algorithms but speaks about the possibility to treat them as instruments to rethink what creativity is. Thus, one of the major lessons learned from the analysis of du Sautoy's ideas is that being creative does not necessarily mean having inspiration of one's own. Creativity might be understood as exploring a particular space, combining or reconfiguring existing elements or even changing the rules of the game.

In this context, the idea of creative exploration underlying this research can be seen in a similar light. Insight Hunter itself cannot be driven by inspiration; instead, it explores data-driven patterns from different angles, incorporates domain knowledge, challenges assumptions and discards interpretations not backed up by data. Therefore, machine creativity is not free imagination but a finite exploration of a certain space defined by boundary conditions. In this exploration space, the agent is very helpful indeed.

Finally, the main point about the proposed workflow is how it balances creativity with validation. A purely creative process can lead to the development of a number of plausible stories about the data analyzed that may not actually be true. Conversely, a purely deterministic approach allows validation but limits creativity. Finding an appropriate way in the described architecture consisted precisely in establishing such a balance, where each step made by the agent could be verified, justified and tracked.

And in this respect, the indispensable role played by the human analyst can be understood as follows. As du Sautoy writes, works of art can open "a window into the way another mind works." Similarly, the outputs produced by an agentic system can be considered, technically speaking, as windows allowing us to see how the algorithmic mind operates. However, this

requires opening the window to see what is happening inside, namely, how the insights have been generated. It is especially important in business analysis when it is essential to trace back the origin of results obtained.

Consequently, the significance of the present work should not be interpreted through the ability of a language model to produce some insights independently. To the contrary, the presented research can be seen as the beginning of a process during which the machine explores the space, creates hypotheses and accumulates evidence, whereas the person evaluates the process and applies the resulting insights. Through this process, both machine and analyst are capable of producing something more than mere data-driven patterns, namely, real and actionable insight.

Bibliography

[1] Ashish Vaswani et al. “Attention is all you need”. In: *Advances in neural information processing systems* 30 (2017).

[2] Tom B. et al. Brown. “Language Models are Few-Shot Learners”. In: *Advances in Neural Information Processing Systems* 33 (2020). NeurIPS 2020. URL: https://arxiv.org/abs/2005.14165.

[3] Long et al. Ouyang. “Training Language Models to Follow Instructions with Human Feedback”. In: *arXiv preprint arXiv:2203.02155* (2022). OpenAI. URL: https://arxiv.org/abs/2203.02155.

[4] Jason et al. Wei. “Chain-of-Thought Prompting Elicits Reasoning in Large Language Models”. In: *arXiv preprint arXiv:2201.11903* (2022). Google Research, Brain Team. URL: https://arxiv.org/abs/2201.11903.

[5] Shunyu et al. Yao. “ReAct: Synergizing Reasoning and Acting in Language Models”. In: *arXiv preprint arXiv:2210.03629* (2022). Princeton University and Google Research. URL: https://arxiv.org/abs/2210.03629.

[6] Binfeng et al. Xu. “ReWOO: Decoupling Reasoning from Observations for Efficient Augmented Language Models”. In: *arXiv preprint arXiv:2305.18323* (2023). Reasoning WithOut Observation. URL: https://arxiv.org/abs/2305.18323.

[7] Timo et al. Schick. “Toolformer: Language Models Can Teach Themselves to Use Tools”. In: *arXiv preprint arXiv:2302.04761* (2023). Meta AI. URL: https://arxiv.org/abs/2302.04761.

[8] Changle et al. Qu. “Tool Learning with Large Language Models: A Survey”. In: *arXiv preprint arXiv:2405.17935* (2024). Survey on tool learning with LLMs. URL: https://arxiv.org/abs/2405.17935.

[9] Weiwen et al. Liu. “ToolACE: Winning the Points of LLM Function Calling”. In: *arXiv preprint arXiv:2409.00920* (2024). Function calling and tool-learning data. URL: https://arxiv.org/abs/2409.00920.

[10] Zhiheng et al. Xi. “The Rise and Potential of Large Language Model Based Agents: A Survey”. In: *arXiv preprint arXiv:2309.07864* (2023). Survey on LLM-based agents. URL: https://arxiv.org/abs/2309.07864.

[11] Zeyu et al. Zhang. “A Survey on the Memory Mechanism of Large Language Model-based Agents”. In: *arXiv preprint arXiv:2404.13501* (2024). Survey on memory mechanisms in LLM-based agents. URL: https://arxiv.org/abs/2404.13501.

[12] Noah et al. Shinn. “Reflexion: Language Agents with Verbal Reinforcement Learning”. In: *arXiv preprint arXiv:2303.11366* (2023). NeurIPS 2023. URL: https://arxiv.org/abs/2303.11366.

[13] Qingyun et al. Wu. “AutoGen: Enabling Next-Gen LLM Applications via Multi-Agent Conversation”. In: *arXiv preprint arXiv:2308.08155* (2023). Microsoft Research. URL: https://arxiv.org/abs/2308.08155.

[14] LangChain. *LangChain Documentation*. Official documentation. 2025. URL: https://python.langchain.com/docs/introduction/.

[15] LangChain. *LangGraph Documentation*. Official documentation. 2025. URL: https://langchain-ai.github.io/langgraph/.

[16] Strands Agents. *Strands Agents SDK Documentation*. Official documentation. 2025. URL: https://strandsagents.com/latest/.

[17] OpenAI. *Learning to Reason with LLMs*. Official OpenAI article. 2024. URL: https://openai.com/index/learning-to-reason-with-llms/.

[18] Yaniv Leviathan, Matan Kalman, and Yossi Matias. “Prompt Repetition Improves Non-Reasoning LLMs”. In: *arXiv preprint arXiv:2512.14982* (2025). Google Research. URL: https://arxiv.org/abs/2512.14982.

[19] Anthropic. *Building with Extended Thinking*. Official Claude API documentation. 2025. URL: https://docs.anthropic.com/en/docs/build-with-claude/extended-thinking.

[20] Marcus du Sautoy. *The Creativity Code: How AI Is Learning to Write, Paint and Think*. Fourth Estate, 2019.

[21] Miles Turpin et al. “Language Models Don’t Always Say What They Think: Unfaithful Explanations in Chain-of-Thought Prompting”. In: *Advances in Neural Information Processing Systems* 36 (2023). NeurIPS 2023. URL: https://arxiv.org/abs/2305.04388.