NOVA IMS
Information Management School

MDSAA

Master Degree Program in

Data Science and Advanced Analytics

Integration of External APIs into LLM-Based Agents for Enhanced Functionality

Comparing a Price Estimator to a Products Deal Finder

João Duarte Justo Mirotes de Almeida Machado

Master Thesis

presented as partial requirement for obtaining a Master's Degree in Data Science and Advanced Analytics

NOVA Information Management School
Instituto Superior de Estatística e Gestão de Informação
Universidade Nova de Lisboa

NOVA Information Management School
Instituto Superior de Estatística e Gestão de Informação
Universidade Nova de Lisboa

Integration of External APIs into LLM-Based Agents for Enhanced Functionality

Comparing a Price Estimator to a Products Deal Finder

by

João Duarte Justo Mirotes de Almeida Machado

Master Thesis presented as partial requirement for obtaining the Master's degree in Data Science and Advanced Analytics, with a specialization in Data Science

Supervised by

Fernando Bação, PhD, NOVA Information Management School

July, 2025

STATEMENT OF INTEGRITY

I hereby declare having conducted this academic work with integrity. I confirm that I have not used plagiarism or any form of undue use of information or falsification of results along the process leading to its elaboration. I further declare that I have fully acknowledged the Rules of Conduct and Code of Honor from the NOVA Information Management School.

Lisbon, July 2025

ABSTRACT

Large-language models (LLMs) have progressed from research curiosities to multi-tasking fully autonomous AI agents, but their reasoning is still restricted by frozen parameters and the text in their prompt pane. This thesis investigates how the controlled exposure of LLMs to external application-programming interfaces (APIs) may bypass that restriction and overall increase the scope of field applications for these technologies. To accomplish this task, investigate and curate an Amazon product listings dataset, testing it in four different LLM-based configurations. From a zero-shot GPT-4o-mini baseline to a multi-agent Deal Finder that orchestrates both a fine-tuned GPT-4o-mini model, a RAG model, RSS scrapers, ensemble regression and a Pushover push-notification API. The results confirm that the targeted API integration, even light-weight ones, significantly enhance LLM decision making and action capabilities. They moreover uncover new engineering trade-offs like rate limits, quality of retrieval, ensemble coordination, leaving to future work the possibility to scale API-centric agents from prototypes to production systems.

KEYWORDS

LLMs; AI Agents; Agentic Systems; API Integration; Multi-Agent Systems; Products Deal Finder; GPT Models; RAG; RSS Feeds

Sustainable Development Goals (SDG):

TABLE OF CONTENTS

1. Introduction 1
1.1. Background 1
1.2. Research Question and Objectives 1
1.3. Outline 2
2. Literature review 3
2.1. Theoretical Background 3
2.1.1. Evolution of LLM Reasoning 3
2.1.2. Interleaving Actions with Reasoning 3
2.1.3. Reflexion for Self-Feedback 4
2.1.4. Multi-Agent Systems 4
2.2. Implementation Frameworks: Low-Code/No-Code vs. Hard-Coding 5
2.3. Tools and External APIs 5
2.4. Real-World Performance and Benchmarking 6
3. Methodology 8
3.1. Extracting & Curating an Amazon Dataset 8
3.1.1. Data Cleaning and Item Abstraction 9
3.1.2. Data Curation 9
3.2. Price-Predictor: Fine-Tuning an LLM 12
3.2.1. OpenAI Models as Price Predictors 12
3.2.2. Fine-tuned GPT-4o-Mini 14
3.3. Multi-Agent Deal Finder 16
3.3.1. RAG & Ensemble Agents 17
3.3.2. Framework Agents 19
3.3.3. User Interface 19
4. Results and discussion 21
4.1. Predictive-Performance Results 21
4.2. Analysing Worst Predictions 23
4.3. REST API's Analysis & Impact 24
5. Conclusions and future work 25
5.1. Conclusion 25
5.2. Future Work 26
Bibliographical References 27

LIST OF FIGURES

Figure 2.1 – Hierarchical multi-agent architecture ... 4
Figure 3.1 – Distribution of prices before curation ... 10
Figure 3.2 – Count of data points per category before curation ... 10
Figure 3.3 – Distribution of prices after curation ... 11
Figure 3.4 – Count of data points per category after curation ... 12
Figure 3.5 – Results of GPT4o-mini in 250 items of the Test Set ... 13
Figure 3.6 – Results of GPT4o in 250 items of the Test Set ... 14
Figure 3.7 – Results of the Fine-tuned GPT4o-mini in 250 items of the Test Set ... 15
Figure 3.8 – Diagram of the multi-agent system ... 16
Figure 3.9 – 3D Plot of the Vector DB ... 17
Figure 3.10 – Results of the RAG GPT4o-mini in 250 items of the Test Set ... 18
Figure 3.11 – UI Gradio Interface ... 20
Figure 4.1 – Results of the Ensemble in 250 items of the Test Set ... 22

LIST OF TABLES

Table 4.1 - Performance of LLMs with Different Metrics..21
Table 4.2 - 10 Predictions for Each Model with the Highest Absolute Error 23

LIST OF ABBREVIATIONS AND ACRONYMS

LLM	Large Language Model – A family of very large neural networks trained on massive text corpora to understand and generate human-like language.
API	Application Programming Interface – A defined set of routines and protocols that let one software component request services or data from another.
REST	Representational State Transfer – An architectural style for building networked applications that relies on stateless, cacheable client-server interactions—typically over HTTP.
RSS	Really Simple Syndication – A standardized XML feed format that lets users automatically receive updates from websites or podcasts.
UI	User Interface – The visual and interactive elements (buttons, menus, layouts, etc.) through which a person interacts with a digital product.
UX	User Experience – The overall perception, ease, and satisfaction a user has while using a product or service, encompassing usability, accessibility, and emotion.
VDB	Vector Database – A specialized database optimized for storing and performing similarity search on high-dimensional vector embeddings.
LoRA	Low-Rank Adaptation – A parameter-efficient fine-tuning technique that inserts small low-rank matrices into a pretrained model’s layers to adapt it to new tasks.
QLoRA	Quantized Low-Rank Adaptation – An extension of LoRA that combines low-rank adaptation with 4-bit weight quantization, cutting memory use while retaining model quality.

1. INTRODUCTION

Since 2018, where transformer where first introduced, large-language models (LLMs) have moved from research novelties to the foundation of a fast-growing ecosystem of “AI agents” that can do a multitude of tasks such as writing code, draft emails, summarizing content and even power conversational interfaces in consumer devices, browsers and enterprise dashboards alike. Models such as GPT-3, GPT-4 and the open-weight Llama-3 series exhibit an emergent ability to perform zero-shot reasoning, chain-of-thought explanations and multilingual translation without task-specific supervision. Despite these breakthroughs, transformer LLMs still possess a key limitation, during inference the model can only condition on the information available in its frozen parameters and the text supplied in its prompt window. As result, LLMs cannot observe the present moment, execute real-world actions or consume private domain data unless they are explicitly connected to external tools.

1.1. BACKGROUND

A parallel line of research has shown that granting LLMs access to application programming interfaces (APIs) for search, computation and actuation can dramatically improve accuracy, numeric reasoning and task completion rates. Yet systematic, peer-reviewed studies on the engineering trade-offs and empirical performance impact of API augmentation remain scarce. Most published cases describe prototype chatbots or “agent frameworks” in blogposts rather than rigorous experiments.

This thesis aims to close that gap through a focused, reproducible evaluation. Concretely, we contrast a stand-alone fine-tuned GPT-4o-mini with a multi-agent system that orchestrates at least one external API (push-notification service) while also including a vector database and Really Simple Syndication (RSS) scrapers. We will measure evaluation metrics and computational budges, while isolating the marginal value of API integration on accuracy, latency and user-perceived utility.

1.2. RESEARCH QUESTION AND OBJECTIVES

This thesis serves as a practical application for creating an Agent-based system capable of finding product deals on the web. The major purpose is to show how LLMs can be used and fine-tuned for such tasks, along with the creation of autonomous agents capable of using external APIs to leverage pre-made foreign systems, thus increasing the capabilities and range of those same Agents.

The central question guiding this work is: How does the integration of external APIs into large-language-model agents enhance their decision-making and action-taking capabilities, and what challenges and opportunities arise?

This overarching question decomposes into four measurable objectives. First, to quantify the limitations of a standalone GPT-4o-mini when tasked with dollar-level price prediction across varied product categories, using metrics such as mean absolute error (MAE) and root mean-squared log-error. Second, developing a reproducible architecture for API orchestration that includes vector retrieval via a Chroma database, web-scraping agents that gather candidate deals through RSS feeds, and a messaging service (Pushover) for real-time user alerts. Finally, testing the performance between the single LLM and the multi-agent system on accuracy, response latency and downstream user value (e.g., percentage discount captured).

Collectively, these objectives generate both quantitative evidence and qualitative guidelines for practitioners to build API-enhanced agents and thus will showcase how the integration of external APIs into LLM-based agents provides enhanced functionality.

1.3. OUTLINE

The first chapter introduces the reasoning behind this work by exposing and clearly defining the research question and objectives that we purpose and that are used as the baseline for the remaining chapters. The second chapter focuses on the literature review, showcasing prior work on LLM reasoning evolution, open and closed-source trade-offs, ReAct, agentic architectures and large-scale tool benchmarks. The third chapter describes the methodology where the practical project we set ourselves to create is described in detail. From data selection, data cleaning (slot-reservoir sampling), model fine-tuning, RAG construction, ensemble regression, the creation of the agentic framework and the selection and impact of external APIs. Chapter four Results & Discussion reports numeric and visual comparisons, including tables and scatter plots, showing LLM performances. Lastly, the fifth chapter, conclusion & future work, is where the thesis comes to an end by discussing everything that was developed, proposing alternatives and work that could have been expanded upon with a big focus on the API integration itself.

2. LITERATURE REVIEW

This chapter provides an analysis of the current literature around LLM-based agents and external API integration, as well as the observed limitations that are consistently shown across the different papers. It begins with a brief theoretical background on some LLM techniques such as system prompting and model reasoning, then transitions to some of the major frameworks and architectures that enable tool usage, thus moving from classic LLMs to LLM-based Agents. We also discuss multi-agent solutions, culminating in an examination of ongoing challenges related to security, privacy, real-world applicability, and critically, how and why API integration helps address LLM limitations such as hallucinations and static knowledge.

2.1. THEORETICAL BACKGROUND

2.1.1. Evolution of LLM Reasoning

Early language models were primarily autoregressive text predictors, used for straightforward and simpler tasks like producing fluent output but with limited "reasoning" capacity. Over time, techniques such as chain-of-thought prompting (Wei et al., 2023) explicitly encouraged LLMs to articulate intermediate reasoning steps, whether through zero or one-shot prompting ways, thereby improving performance on tasks requiring multi-step logic like math word problems or abductive reasoning.

In chain-of-thought prompting, the LLM is shown examples where the final output is preceded by a natural language "explanation." This approach has proven beneficial in boosting interpretability, as end-users can read how the model arrived at its conclusion. However, a key limitation is that chain-of-thought text becomes part of the prompt, consuming context window space. For tasks involving long queries or multi-modal data, the prompt can grow unwieldy (Shinn et al., 2023).

2.1.2. Interleaving Actions with Reasoning

A related strand of research, exemplified by ReAct (Yao et al., 2022), weaves actions (e.g., external API calls, database queries, or code execution steps) into the chain of reasoning. The model alternates between "thought" steps (internal reasoning) and "action" steps (calling a tool), with each action's result then used to inform subsequent thoughts. ReAct has shown improvements over pure "chain-of-thought" approaches by grounding intermediate steps in external evidence. Nonetheless, carefully orchestrating multiple interleaved calls in complex real-world systems, where each call might have a unique authentication scheme or data format, remains a non-trivial design challenge.

2.1.3. Reflexion for Self-Feedback

Another noteworthy approach is Reflexion, where the LLM logs its own mistakes and "reflects" on how to avoid them in subsequent attempts (Shinn et al., 2023). This method side-steps expensive retraining or gradient updates in favor of textual self-feedback, potentially boosting performance over iterative trials. Although Reflexion-based frameworks have succeeded in code generation tasks, robustly handling external API feedback, especially when the returned data may be inconsistent or incomplete, remains an open research question.

2.1.4. Multi-Agent Systems

While single-agent systems can embed or call multiple APIs, many researchers have proposed going further by creating multi-agent frameworks. Individual agents are powerful on their own, as they can create subtasks that better split the overall goal, use tools as previously mentioned and learn through their interactions. The combined behavior of multi-agent systems in theory increases the potential for accuracy, adaptability and scalability. In such architectures, as seen in the hierarchical multi-agent architecture in Figure 2.1, each agent might be an LLM specialized in a certain function, like a Supervisor Agent that decomposes a main task into sub-tasks and delegates them accordingly, or a Retriever Agent that fetches external documents or queries from APIs, or even a UI Agent that uses specific tools to design a full front-end user interface for an application.

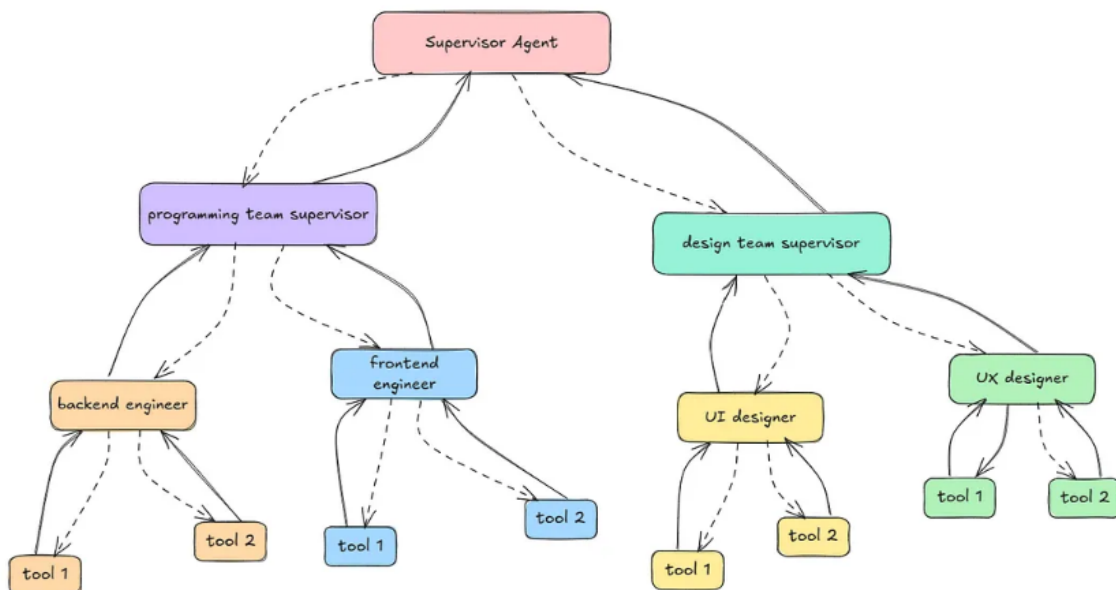

Figure 2.1 – Hierarchical multi-agent architecture

Studies on multi-agent systems indicate that specialization and partition of roles can reduce cognitive load on a single model, potentially improving reliability and interpretability. However, each additional agent introduces new overhead in the form of communication, possible misalignment, and the risk that an early error can propagate through subsequent steps. These concerns are the main reason why such systems are yet to be used as large-scale solutions in real-world scenarios and as such are mainly used for simpler or straight-forward applications in most of the major companies.

2.2. IMPLEMENTATION FRAMEWORKS: LOW-CODE/NO-CODE VS. HARD-CODING

When developing any type of GenAI based application, a practical question that arises, especially when dealing with API calls, is how and where to create the apps themselves. Nowadays, with the number of frameworks and tools available throughout the web developers can choose to either produce their application.

First, through Low-Code/No-Code Platforms such as Zapier, N8N, or using various “prompt orchestration” libraries that provide simple interfaces for chaining together LLM calls and external APIs. These platforms often include built-in modules for memory management, prompt templates, or tool invocation. By reducing boilerplate coding, they can help developers quickly produce results (e.g., hooking an LLM up to a search API or email-sending service).

Second, through Hard-Coded Integrations, by writing custom Python or JavaScript code for each tool or API call, leveraging direct HTTP requests, custom authentication, and specialized logging. This approach is more flexible (developers can fine-tune exactly how calls occur) but also more time-consuming and potentially error-prone, particularly if the system must scale to many endpoints or incorporate robust error handling (Shen et al., 2023).

Choosing between these approaches often depends on domain requirements for security (on-prem vs. cloud), compliance (GDPR or HIPAA constraints), and developer expertise in prompt engineering or backend integration.

2.3. TOOLS AND EXTERNAL APIS

A core limitation of vanilla LLMs is that they rely on static knowledge captured at training time. Consequently, they cannot natively access real-time data, updated content, or specialized computation that occurs outside of their training corpus (Mialon et al., 2023). Integrating external APIs addresses these limitations in diferent ways:

1. Real-Time and Dynamic Data, by using APIs that provide updated information (e.g., weather, stock prices, news) or specialized domain knowledge (e.g., legal databases), allowing LLMs to respond with up-to-date facts
2. Reduced Hallucinations that happen when LLMs sometimes fabricate plausible sounding but incorrect details. Querying authoritative API endpoints (e.g., knowledge graph or an enterprise database) can ground the LLM's responses in factual data, thus mitigating hallucinations (Schick et al., 2023).
3. Action-Oriented Outputs rather than merely producing text. An LLM augmented with APIs can perform tasks like sending emails, scheduling events, updating databases, or orchestrating IoT devices. This transforms the LLM from a passive language model into an active agent capable of taking actions on a user's behalf

Overall, external APIs expand an LLM's operational boundaries, enabling functionalities ranging from simple data lookups to sophisticated multi-step processes. However, enabling these capabilities raises new challenges, including latency, authentication, versioning, and security constraints (Qin et al., 2023). All these topics are addressed in depth in the methodology chapter, where we showcase a practical application of going from using an LLM to try and predict a price product, based on data that the model is trained on, to a deal finder where we agents that can coordinate between each other and are tasked with specific goals, along with using of some external APIs, this results in something much more powerful in terms of capabilities and extends the scope of what LLMs can do.

In recent years, large language models like LLaMA from Meta have achieved remarkable success in natural language tasks. However, their capabilities in leveraging external tools, such as APIs, remain limited as previously stated. Unlike state-of-the-art (SOTA) closed-source LLMs like ChatGPT, which excel in tool usage, open-source LLMs have struggled to integrate such capabilities effectively. This is where frameworks like ToolLLM (Qin et al., 2023), which can explore integration with over 16,000 real world RESTful APIs, pushing beyond single-call or single-domain tasks excel. The approach includes a specialized training set, ToolBench, designed to teach the model to interpret documentation and compose multi-step calls. Although the zero-shot transfer to unseen APIs is a promising approach, the authors of the original paper also acknowledge the difficulty of migrating these techniques to truly unstructured, unpredictable production environments, which once again reinforces the idea that many of these advancements so far are still far from being reliable enough for businesses.

2.4. REAL-WORLD PERFORMANCE AND BENCHMARKING

A crucial step which we have yet to discuss is the use of different benchmarks in agentic applications, used for evaluating real-world performance and success rates of such programs.

A seminal contribution here is Toolformer (Schick et al., 2023), which employs a self-supervised approach. The LLM sees unlabeled text data and inserts hypothetical “tool use” annotations wherever it predicts they might improve performance, such as translating a snippet of text or verifying a fact via a web search. The model is then trained to decide when these calls are helpful. Experimental results have shown that Toolformer can significantly reduce factual errors and produce more truthful outputs. Yet, the approach is tested primarily on curated tasks, leaving open questions about scaling to thousands of APIs, handling ephemeral tokens, or dealing with real-time streams of data.

Among the first benchmarks designed to specifically test an LLM’s skill in calling external tools, exists ToolQA (Zhuang et al., 2023) and Gorilla (Patil et al., 2023). ToolQA focuses on whether the model can parse the correct usage of a small set of provided tools, while Gorilla measures the model’s correctness in picking and calling AI-related APIs. Both highlight how advanced LLMs can struggle when forced to rely on dynamic knowledge, further underscoring the complexities of real-world integration.

Benchmarks like ToolQA (Zhuang et al., 2023), AgentBench (Liu et al., 2023), or Gorilla (Patil et al., 2023) provide valuable insights into whether an LLM can parse API documentation, select the right endpoints, and compose correct queries. Nonetheless, they typically assume a relatively stable environment with minimal concurrency and do not measure the latency or reliability of external calls. Moreover, they fail to evaluate aspects like rate-limiting, frequent endpoint updates, or partially broken documentation, common occurrences in real-world software ecosystems.

Recent efforts propose extending existing benchmarks or creating new ones that incorporate elements such as: version drift, which consist on the automatic update in API endpoints/parameters to test an LLM adaptability; latency simulation where we use tools that introduce random delays to mimic real network conditions; adversarial testing where malicious prompts attempt to breach security protocols; and finally multi-agent orchestration that is done by using performance metrics that track not only the final answer quality but also cost, concurrency overhead, or cross-agent communication load. Such expansions, though still nascent, are essential for bridging the gap between small-scale prototypes and robust real-world systems (Shinn et al., 2023).

3. METHODOLOGY

This study attempts to compare the problem-solving style of a single fine-tuned large-language-model (LLM) with that of a multi-agent system that orchestrates external APIs, memory and specialized sub-models in a realistic production-like setting. Part of the implementation of the system is drawn from the codebase provided in the course "LLM Engineering: Master AI, Large Language Models & Agents" by Ed Donner on Udemy. The explanations of this course served as a foundational reference for a few components of this work, including the use of some utility libraires to better manage the LLMs. The course license permits educational and personal use, under which this work was developed.

Using the course as a foundation our objective was to see whether we could enhance the capabilities and performance of LLM-based Agents by using external API's. To accomplish this, the chosen task is objectively measurable, rewards structured knowledge, and has an unambiguous target consisting of predicting the retail price of a product from nothing but its textual listing. Price estimation fulfils those design constraints for three reasons. First, the output is a single scalar value, so evaluation reduces to familiar regression metrics rather than subjective rubric scores. Second, online marketplaces expose millions of human-written descriptions, giving us a deep and diverse training signal. Third, shoppers instinctively understand dollars and cents, making any error analysis directly interpretable by non-experts. The task also eliminates the "fuzzy grading" problem that is present in many generative benchmarks, where every prompt has exactly one correct answer at the granularity of the nearest dollar, so downstream metrics remain unambiguous.

The remainder of this chapter is split between two main parts:

- Price-Predictor: a single fine-tuned LLM that attempts to predict different product prices with different ranges based on pre-trained data.
- Deals-Agent Finder: a team of specialized LLMs together with a retrieval-augmented memory and web APIs, wrapped around a Deal Agent Framework.

3.1. EXTRACTING & CURATING AN AMAZON DATASET

The dataset used for this project was the raw scraped data from the public McAuley-Lab/Amazon-Reviews-2023 corpus on Hugging Face, a 2023 dataset of Amazon marketplace data that couples review text, rich item metadata, price fields, and category labels. Competing datasets that were considered, either redact price (eBay), omit technical details (Rakuten), or prohibit text redistribution altogether.

Because fashion, beauty and media items often lack consistent price signals or use size/format-dependent pricing and due to the dimension of the Amazon Reviews dataset, we

restricted the analysis to eight “hard-goods” categories which resulted in roughly 2.8 million candidate products.

The chosen categories were Automotive, Electronics, Office Products, Tools & Home Improvement, Cell Phones & Accessories, Toys & Games, Appliances, Musical Instruments, as they provided more than enough variety of different product types and prices.

3.1.1. Data Cleaning and Item Abstraction

To prepare our dataset for training, we implemented an Item class that transforms each raw JSON record into a well-structured example containing a product description paired with its price. The Item class also makes sure that each data record goes through three stages which are explained below.

First, the text is thoroughly cleaned by removing repetitive boilerplate phrases like “Batteries Included? Yes/No,” and HTML elements/tags are stripped and discarded along with long alphanumeric sequences such as stock-keeping unit codes, which add unnecessary tokens without adding meaning. Second, we enforce a length requirement to ensure sufficient context, and so any listing under 300 characters is discarded, those that remain are tokenized using Meta’s Llama-3 tokenizer, and entries with fewer than 150 tokens are removed for being too sparse. Listings exceeding 160 tokens are truncated so that, after adding our question prompt (“How much does this cost to the nearest dollar?”), no input exceeds 180 tokens. Finally, we make a “prompt wrapping”, where we wrap the cleaned and length-controlled text between the question and an answer line, creating prompts like “How much does this cost to the nearest dollar?” followed by the product title and “Price is $189.00.” This format clearly separates the instruction from the target (product price), and during evaluation, which will be discussed later, we omit the price so the model must predict the missing value.

In the prompt wrapping mentioned above, the first line acts as an instruction, while the final line supplies the supervised target variable and the helper method we created returns a masked version (Price is $) for evaluation. All these techniques combined allow us to transform the raw data from the Amazon dataset into the perfect data points needed for our task.

3.1.2. Data Curation

Before testing our data in different scenarios, we had to further refine it and prepare it, as left unaltered, the corpus was extremely skewed, seen in Figure 3.1 below. Around six in every ten items cost less than forty dollars and another concern was that there was a big discrepancy in the number of data points we had between the different categories, more specifically we had a lot more Automotive data (around 911 000 items) than any of the other category of products, shown in Figure 3.2. All this would result in a model that when trained on that

distribution would learn to predict prices in the $19-$29 range but would hesitate on anything pricier than a certain higher value, making it accurate for a certain type of product but unreliable for others.

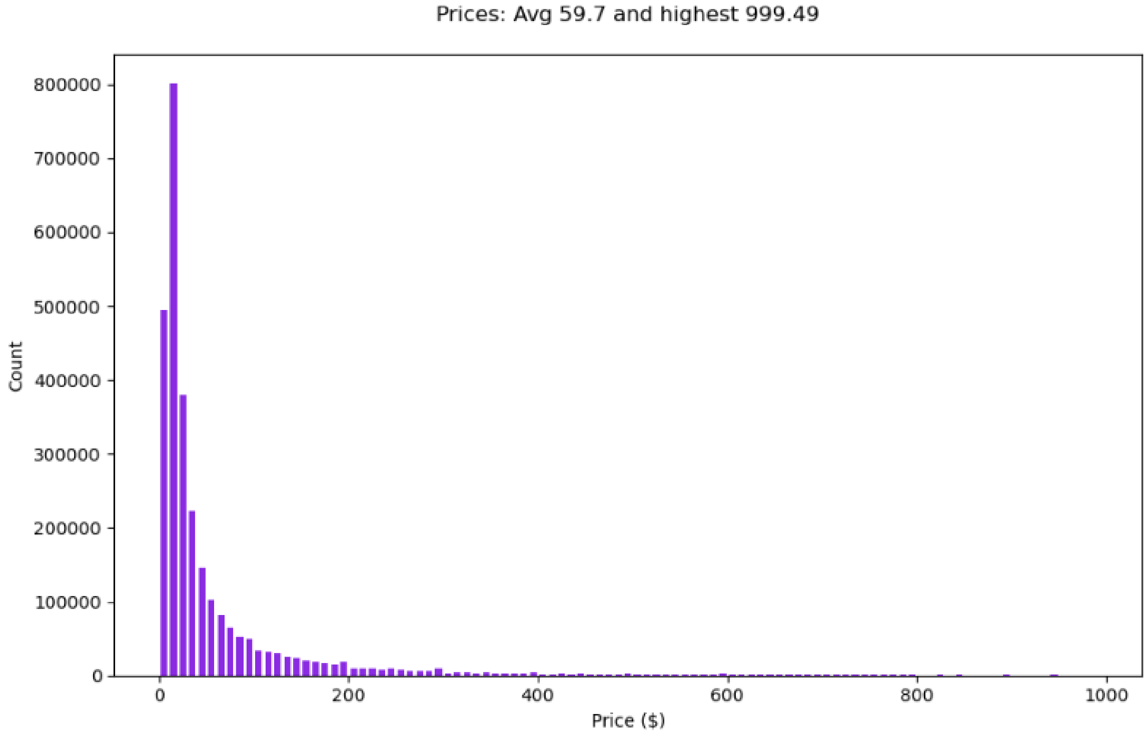

Figure 3.1 – Distribution of prices before curation

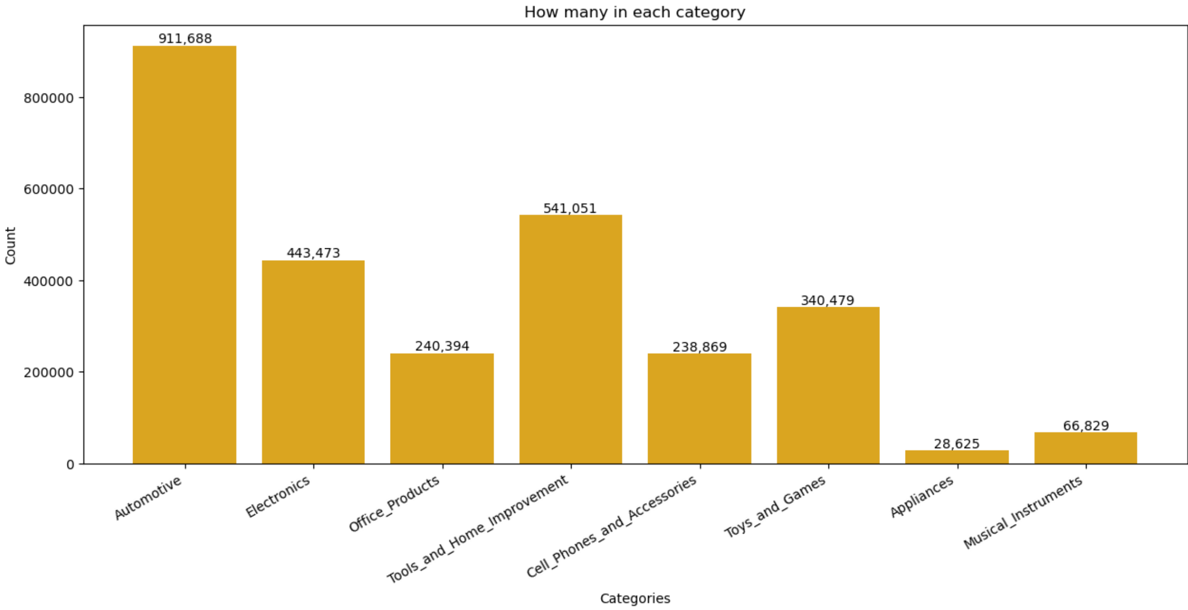

Figure 3.2 – Count of data points per category before curation

To flatten the landscape a “slot-reservoir” technique was adopted. Each whole-dollar price between $1 and $999 owned a slot. For slots below $240, at most 1 200 prompts survived, and within those caps, entries that were not automotive were approximately five times more likely to be kept than automotive ones. Slots above $240 kept every candidate because expensive items were already sparse enough.

After balancing, the working data set was reduced to 408 635 prompts, showing that the mean price rose from $59.7 to $220.5, as seen in Figure 3.3 and automotive’s share falls from roughly one-third to just under one-quarter, also seen in Figure 3.4. The results were far from optimal as there was still quite a bit of discrepancy between category sizes, but still, we managed to make the Automotive category a less dominant one while raising the prices mean and keeping a big enough data pool.

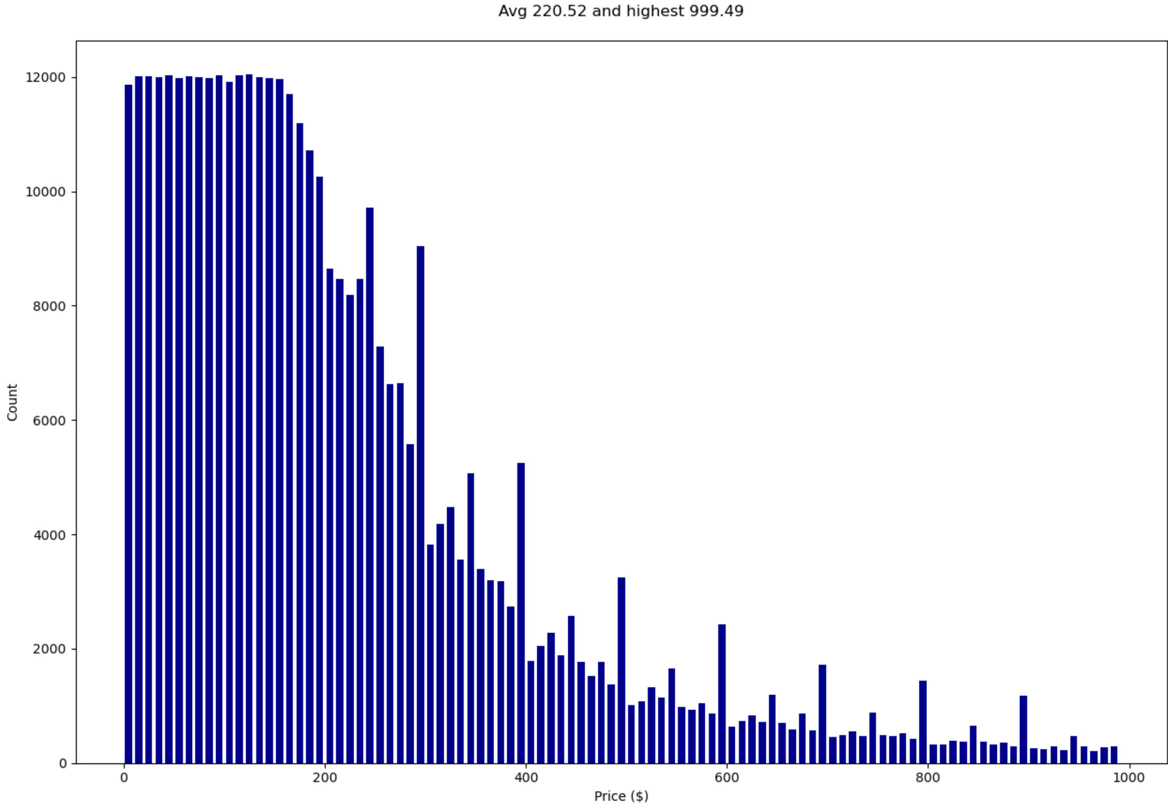

Figure 3.3 – Distribution of prices after curation

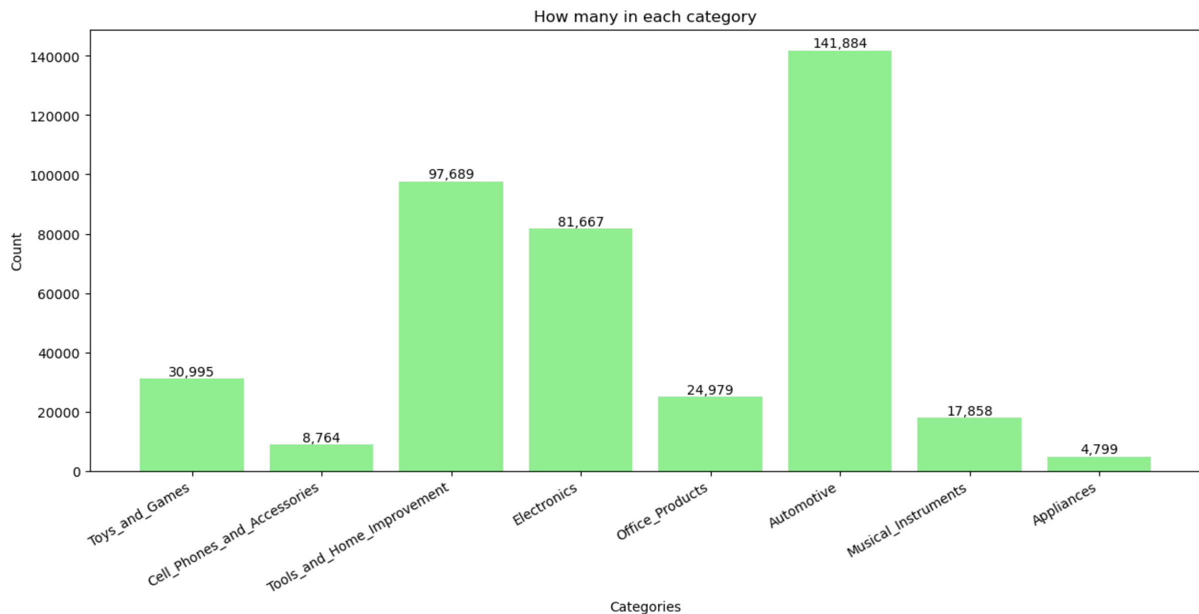

Figure 3.4 – Count of data points per category after curation

Finally, after curating the dataset, we also spit it as it's usually done in classic machine learning projects. Normally, we would use 5 to 10% for testing purposes, but our initial goal with cleaning this dataset was to fine-tune an OpenAI close-source frontier model, and we had far more data than needed at this point. So, we randomly shuffle the data under a 42 seed and split the dataset between 400 000 items for training and 2 000 for testing. Finally, for reproducibility purposes, we pickled both datasets. This way we wouldn't have to re-run all the codes each time we wanted to use our training and test sets.

3.2. PRICE-PREDICTOR: FINE-TUNING AN LLM

3.2.1. OpenAI Models as Price Predictors

Before diving into fine-tuning LLMs we chose to test our data in two frontier close-source models. We ended up choosing GPT-4o-mini due to accepting chat-style messages and at the same time being extremely inexpensive for large-batch evaluation, while its larger sibling, GPT-4o, was included merely to check whether sheer parameter count meaningfully alters the error pattern or not.

An evaluation loop was also created, encapsulated by a small utility class named Tester and what this class did was that for each of the 250 randomly chosen test examples it first fed the product prompt to a predictor function (a thin wrapper around the model call) and then recorded the model's guess and the ground-truth price. Finally, it accumulated three summary

statistics that were chosen based on the task we had: the mean absolute error, the root mean-squared log-error, and the proportion of green "hits".

Both GPT-4o variants used the same three-step template: a system instruction telling the model to "reply only with the price, no explanation," a user query that embeds the cleaned listing after the question "How much does this cost?", and an assistant primer, "Price is $," that pushes the first generated token to be a digit. This structure delivered a single numeric output (price) that the evaluation loop could easily ingest without costly parsing, while also decreasing needless explanations and trimming token usage. Fixing the wording of the query kept the task identical across models so accuracy differences reflected architecture rather than prompt drift and prefixing the dollar sign biased the model toward a deterministic numeral.

The results showed that GPT-4o-mini typically missed the true price by about eighty dollars and produced a neat diagonal "swarm" for items under $200 before spreading out sharply for high-ticket goods, while also showing a significant number of extreme outliers, as seen in Figure 3.5 below.

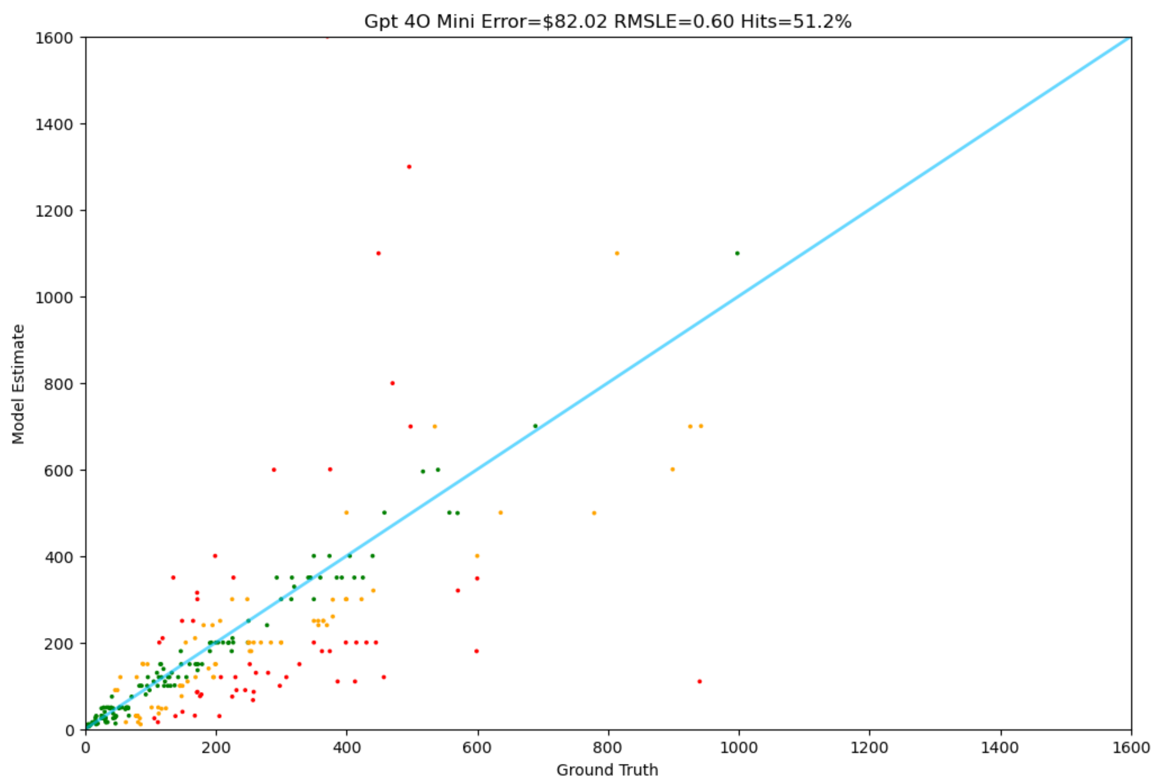

Figure 3.5 – Results of GPT4o-mini in 250 items of the Test Set

While upgrading to GPT-4o narrowed the average miss by only a small margin, though it did tighten the extreme outliers, Figure 3.6. The results showed that sheer parameter count helped, but did not erase the information deficit inherent in a short, stand-alone description.

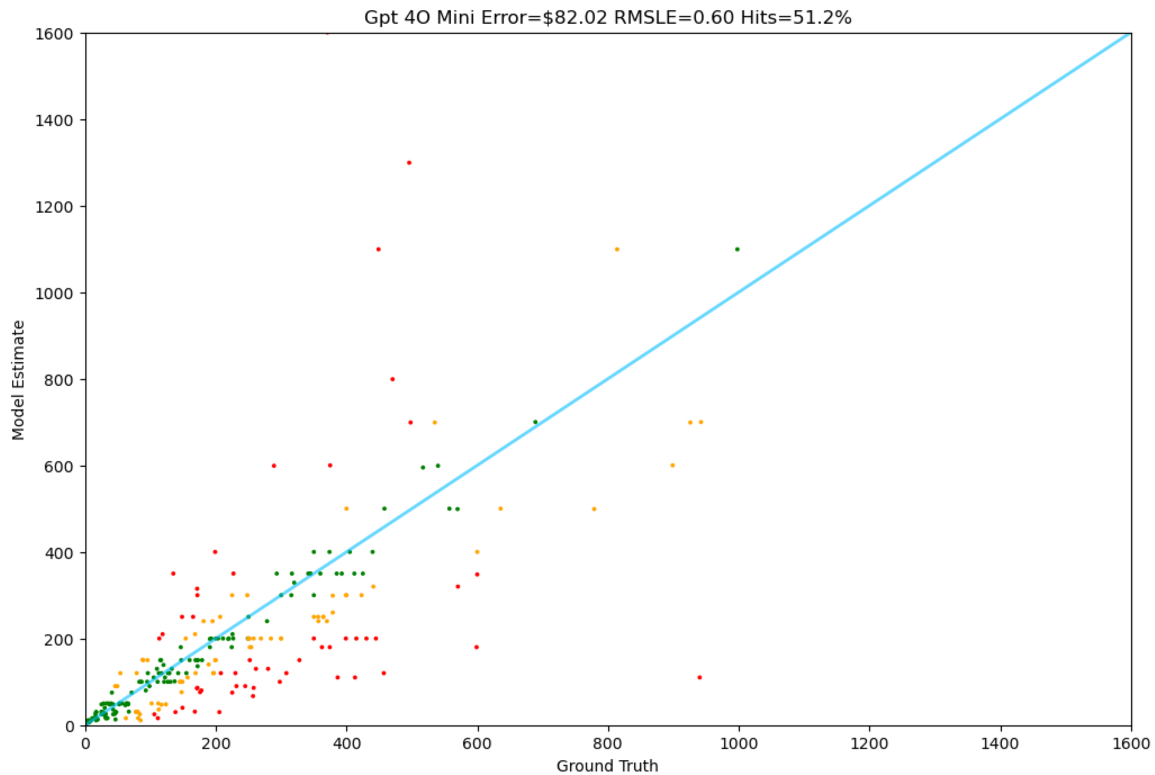

Figure 3.6 – Results of GPT4o in 250 items of the Test Set

3.2.2. Fine-tuned GPT-4o-Mini

The next goal was to fine-tune the GPT4o-mini model and to do this we first checked the OpenAI documentation, where it's stated to use fifty to one-hundred chat samples for instruction-following tasks. Because our prompts were short (roughly 180 tokens), from the original 400 000 prompts, we adopted a round number of 500 for training and an additional 50 for validation. OpenAI's default hyper-parameters, and a single training epoch kept the exercise minimal.

After the fine-tune finished the new model was passed through the same Tester class as the previous ones, to be tested using the same criteria. The outcome was underwhelming, as seen in Figure 3.7 the average error grew to a little over one hundred dollars and the scatter collapsed toward a narrow horizontal band around $225. The wider root mean-squared log-error confirmed that variance, not just bias, had increased. In practical terms the model seems to have over-fitted to the small, still-skewed slice of examples and lost the general price intuition its zero-shot parent possessed.

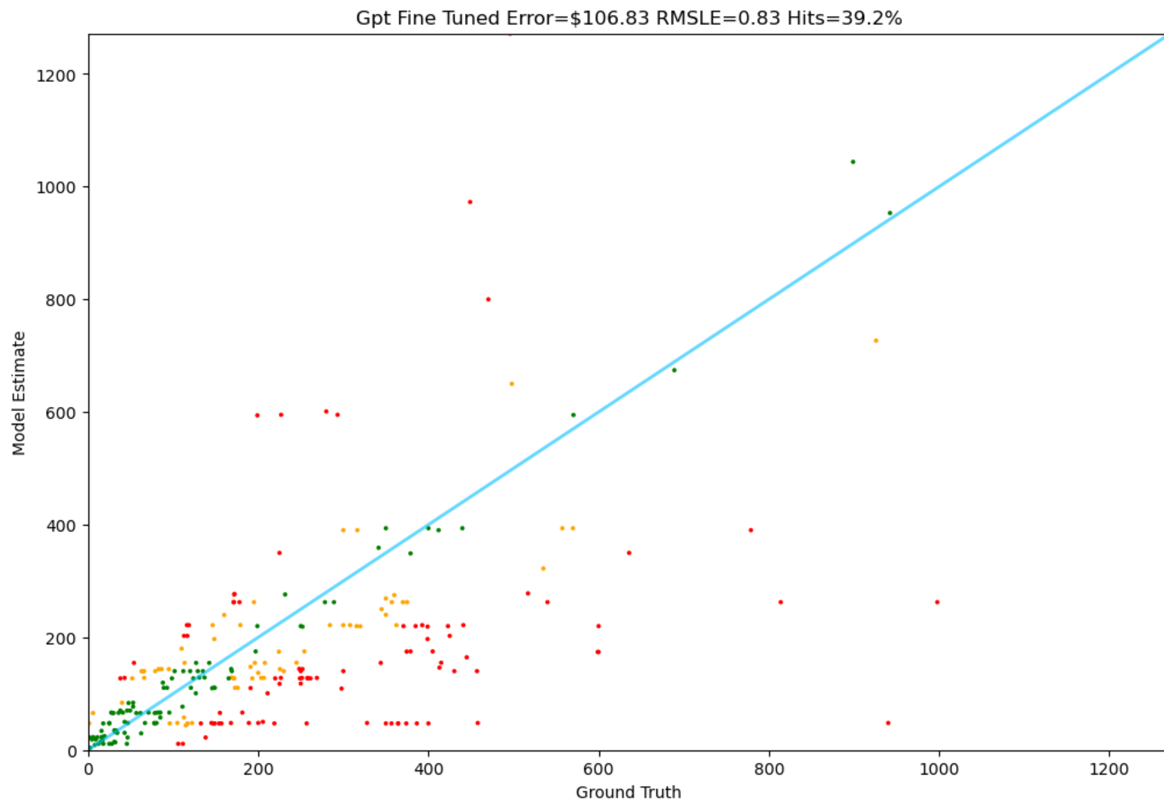

Figure 3.7 – Results of the Fine-tuned GPT4o-mini in 250 items of the Test Set

The negative result was instructive in showing us that when supervision is shallow, fine-tuning a closed-weight model can worsen useful priors. The likely cause was that this wasn't an optimal problem for fine-tuning an OpenAI model and that by training with the new prompts, we made it lose some context and capabilities that it had learned during pre-training. Remedying this would either require a much larger labelled set (financially costly if every call returns frontier-model logits) or an open-weights model that we can train in-house on the entire 400 000-prompt corpus.

An alternative approach would be to fine-tune an open-weights model such as Llama-3-8B (or 70B) on the entire 400 000-prompt corpus, since we would be able to control the weights, we would potentially avoid the sample-size bias problem with GPT-4o-mini that we previously had and could train cheaply on pre-emptible A100 or TPU-v5e hardware in Google Cloud. In addition to this, parameter-efficient adapters like LoRA or its 4-bit variant QLoRA, would allow us to train these massive models in "simplified versions" with less parameters in local machines.

3.3. MULTI-AGENT DEAL FINDER

Unfortunately, we got underwhelming results on the few-shot fine-tune model, but this showed us that creating a large frontier close-source model with a few hundred labels without tweaking the number of parameters of the model itself was not enough to significantly improve its performance for the pricing task. Nonetheless these models were perfectly capable of working as price-predictors and being the basis of chatbots and AI tools for the purpose of answering price-related questions.

The LLMs tested until this point were limited to pure price-prediction with no additional capabilities and so we began working on a new framework, seen in Figure 3.8. One that was capable of not only estimating product prices, but also finding good product deals and alerting us to these findings through a mobile app.

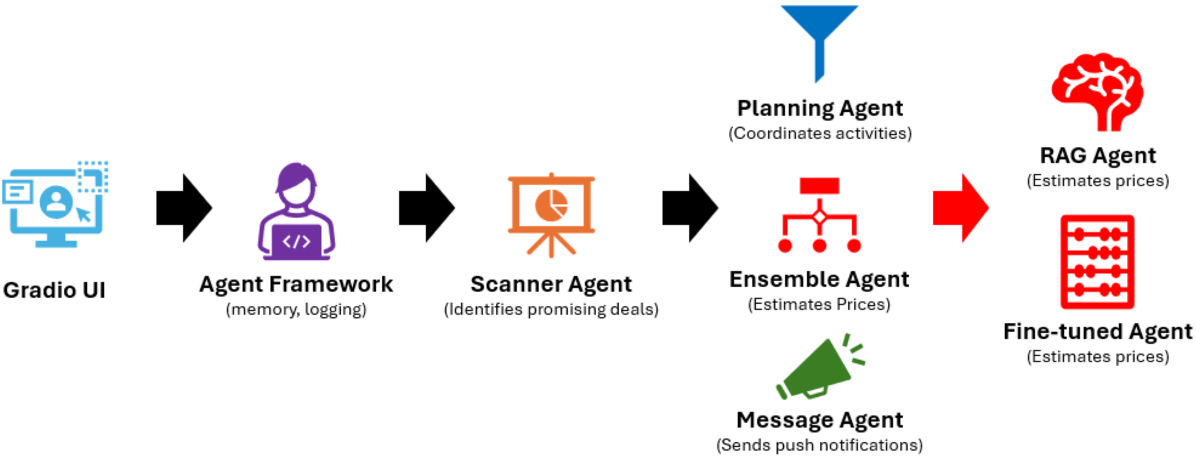

Figure 3.8 – Diagram of the multi-agent system

To accomplish this task, it was decided that we first needed to create another LLM that could collaborate with the previous fine-tuned GPT4o-mini model, to try and improve the overall performance of the price-prediction through the output of a single Ensemble Agent. Only after doing this, we went on to complete the remaining steps of the agentic pipeline including encapsulating the LLMs in agent classes and creating the remaining agents, each with their own task and role. All the steps are explained more in depth in the following subchapters written below.

3.3.1. RAG & Ensemble Agents

We began the first act of the agentic pipeline by creating a second model, this time with a different approach, using a RAG (retrieval-augmented generation) model with the purpose of creating an overall better price estimator. As previously mentioned, a single chat model, no matter how large, can guess only from patterns it internalized during pre-training, so it may know that most USB hubs cost less than forty dollars, yet it will struggle to distinguish a $95 Thunderbolt dock from a $29 four-port USB-C, because both share similar vocabulary like "aluminum", "USB-C". If, however, we were to show the model five short items of actual hubs, together with their prices, it would be able to associate its reasoning in ground truth rather than "gut feeling".

To accomplish this task the adoption of using a retrieval-based model seemed the most reasonable since the balanced Amazon corpus already embodied four hundred thousand price points, so if, at inference time, we could retrieve a few "nearest neighbors" and place them in the prompt, the language model could reason by analogy instead of by statistical guesses alone. So, we opted to use the 384-dimension all-MiniLM-L6-v2 encoder, to transform every curated prompt (minus its answer line) into a vector embedding, the plain text, and two scalar metadata fields (category and price) were written into a Chroma vector databased collection. The choice of the VDB was due to Chroma's server-less, Python-native design allowed us to keep everything in-process, avoid maintaining a separate FAISS index or remote service, and still retrieve millions of vectors with sub-millisecond latency.

We then did a geometric check on the 20 000 datapoints for visualization purposes, where ten thousand random embeddings were reduced to three dimensions with t-SNE, colored by category. The plot showed somewhat distinct islands which reassured us that the encoder was sensitive to both function and price scale, as seen in Figure 3.9 below.

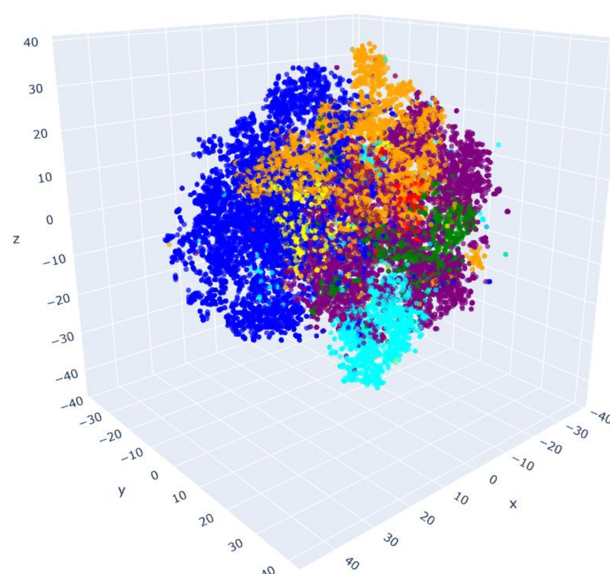

Figure 3.9 – 3D Plot of the Vector DB

Using the created vector DB as an external knowledge base for the GPT4o-mini model and testing its performance as we had previously done with other LLMs (with the Tester class), resulted in a significant increase in the overall performance, with mean the absolute error falling from eighty-two dollars to forty-eight, and the proportion of "green" also increasing significantly, as seen in Figure 3.10.

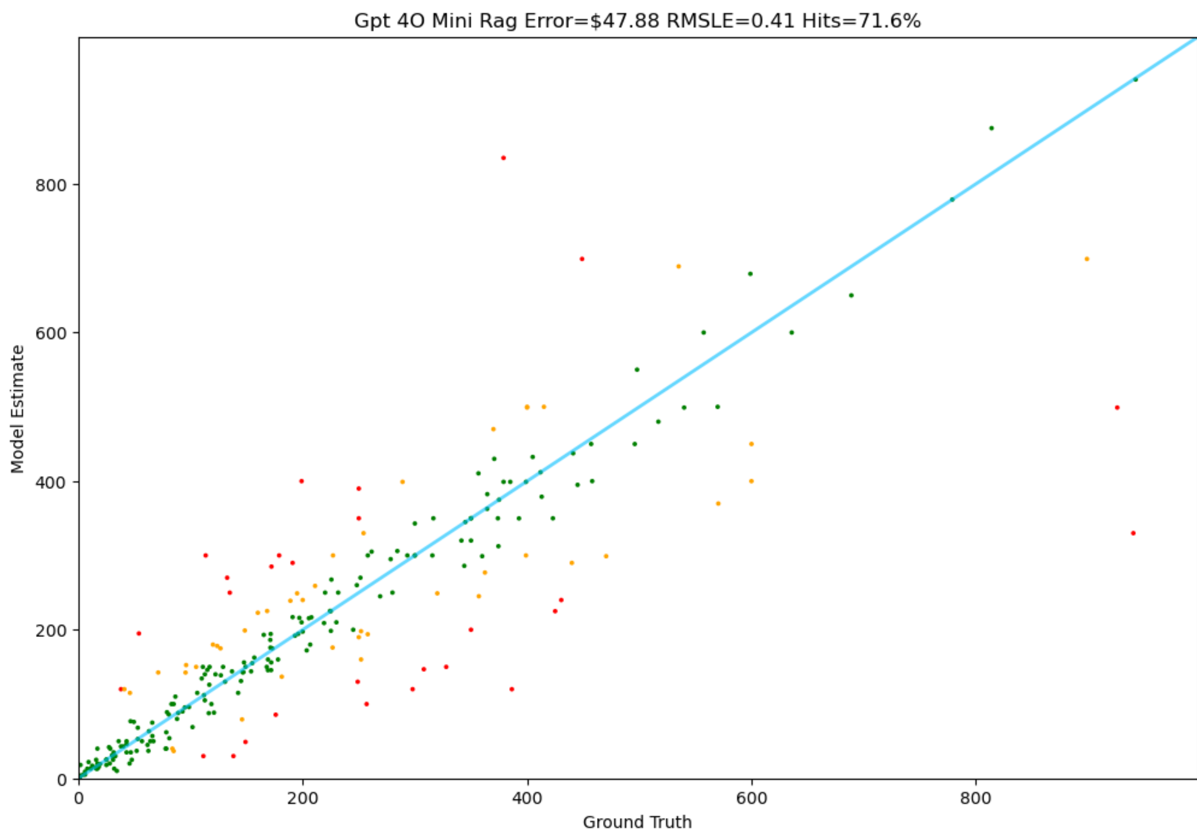

Figure 3.10 – Results of the RAG GPT4o-mini in 250 items of the Test Set

From the previous results a pattern was noticed, the fine-tune GPT4o-mini LLM proved to be able to better handle and squash the outlier data points, while the RAG had the best overall performance against the test data. Thus, as previously mentioned, we decided to leverage the strengths of both models, with a simple Linear Regression Ensemble. This model was done using the input prices of the other two models as input features, together with min and max values of those same input values, in a total weighted average to calculate the estimated prices for each product.

Having completed and tested all models we wrapped them in their own individual agent classes. This allowed us to convert specialized models into modular, composable services, making them observable, schedulable, and upgradable in the same way as every other agent in the workflow that we discussed in the following subchapters.

3.3.2. Framework Agents

To make a deal finder we needed to create tools that could search and scan for products on different websites, retrieve their prices, feed the information to the LLM Agents and compare the scrapped values to the estimated ones. So, we first created a Scanner Agent, that pings five manually chosen RSS feeds from different categories of the website dealnews.com. The Scanner agent then uses a series of functions that curated these feeds, download the raw XML data, scrape the needed information with Beautiful Soup, and consult the persistent memory file to discard any URL that has already been found before and leaving a batch of the best candidate blurbs. These blurbs are concatenated into a single user prompt and sent to GPT-4o-mini alongside a rigid system template that demands exactly five JSON objects. By forcing the model to re-author each description in fluent prose and to extract a numeric price token, we obtain semantically rich yet structurally predictable data that can be used by the remaining agents without further parsing.

The JSON files are then sent to the Planning Agent, which served as a basic strategist and referee. For each deal it invoked the ensemble valuation pipeline explained in the previous subchapter, and the fine-tuned model gives linguistic nuance, the RAG agent injects comparable historical prices, and a local regression smooths outliers across the nearest-neighbors embeddings stored in Chroma DB. The result is a market estimate that can be compared against the scraped sticker price that the Planning Agent then uses to sort the discounts, select the highest valued one, and check it against a dynamically learned $50 profitability threshold.

If the candidate product deal is approved, the Messaging Agent formats a terse notification, product synopsis, current price, estimated fair value, and percentage saved, and dispatches it via a push notification that is sent to the Pushover mobile app along with a chosen ring sound to alert the user of a deal found. Immediately afterward, the same opportunity is written to JSON memory and to the vector store, preventing duplicates and simultaneously enriching the retrieval corpus that powers future ensemble queries.

Finally, the Deal Agent Framework stitches these pieces together, by managing the Chroma connections and streaming the color-coded logs for the UI that we will discuss in the next subchapter and exposes a single entry-point. Through this orchestration the entire pipeline can locate, appraise, and broadcast a product deal in less than half a minute.

3.3.3. User Interface

Having done the core agentic framework, the next priority was to expose its functionality through a clear, responsive user interface that could render each agent's state transitions and outputs as they ran. Thus, a good UI (user interface) design is not merely aesthetic, but it also shapes how users interpret complex information and decide what to do next, emphasizing clarity and feedback.

Traditional full-stack solutions such as Django give fine-grained control over databases, routing and templating, making them ideal when the interface must be tightly integrated with bespoke back-end logic or when enterprise-grade authentication is required. Flask, FastAPI

and even Node-based stacks like Next.js offer full freedom to craft pixel-perfect, production-scale dashboards. Projects in the agent community often pair React front ends with such APIs for maximum flexibility.

For rapid prototyping, however, the overhead of wiring HTML, CSS and JavaScript can slow iteration and become a heavy time-consuming task. Streamlit removes much of that complexity by turning Python scripts into live apps in a lesser timeframe, making it an attractive choice when quick experiments trump custom styling and when server-side state is simple. Yet an even quicker and simpler approach was to use Gradio which excels at making interactive interfaces with inputs for text, images or audio in only a few lines of code, making it ideal of quick prototyping and Proof of Concept type project such as the one presented in this thesis. Its tight alignment with the machine-learning ecosystem, built-in chatbot and gallery elements, and one-line launch command meant we could spend development cycles refining agent behavior rather than plumbing.

All the qualities mentioned above made Gradio the optimal bridge between the multi-agent back end and end users, but a closer look at the application's internal mechanics makes the choice even clearer. As seen in Figure 3.11 below, the interface was built with a single gr.Blocks context that provides two simultaneous visual threads, an HTML log pane to clearly visualize the agents initiating and a continually updating dataframe for discovered deals.

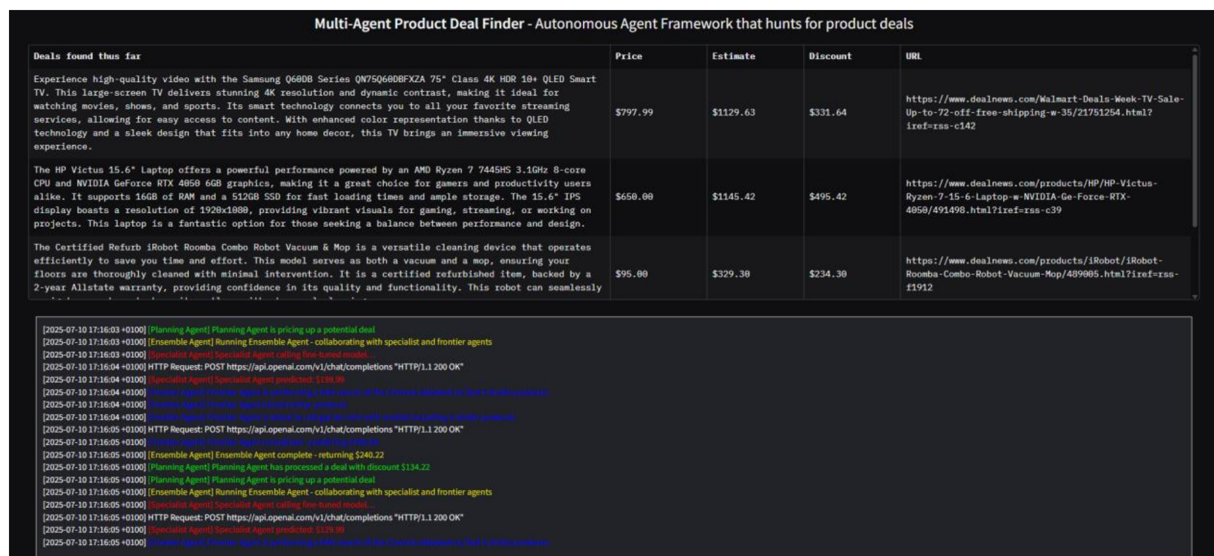

Figure 3.11 – UI Gradio Interface

From a usability perspective the design follows classic UI heuristics, by providing continuous feedback, clear status visibility and direct manipulation, so even non-technical product managers can watch price estimates converge while strolling through color-coded embeddings. In practice this transparency accelerated troubleshooting, so if for example the scanner agent began emitting malformed URLs, the anomaly would appear in the log pane within seconds, allowing an immediate debugging of where the problem lies.

4. RESULTS AND DISCUSSION

Although our thesis purpose was to focus on the impact of external APIs in LLM-based agents, we ultimately limited our experimental surface in this chapter to just three endpoints. First, Section 4.1 examines the sheer predictive power of three model variants that were discussed during the methodology, but this time we dive deeper into the metrics and results obtained, using scatter plots and error tables. Then, Section 4.2 steps deeper into some instances of where the predictions failed and why. Finally, in Section 4.3 we discuss API performance and impact on LLM-based agents, with a focus on the use of the Pushover REST API.

4.1. PREDICTIVE-PERFORMANCE RESULTS

Table 4-1 sits at the heart of this part of the evaluation. We can see the fine-tuned GPT model ("FT-LLM"), the retrieval-augmented counterpart ("RAG-LLM"), and the simple mean of the two ("Ensemble Pricer", or "ENS") on an equal footing. We used 250 random unseen products from the test set we had created and tested the same metric trio of MAE, RMSLE and Hit-Rate@20 %. Plus, we also added latency and cost to the equation, to better measure the LLMs performance across multiple fields.

The results showed that the fine-tuned model had a mean absolute error (MAE) of roughly $98 and barely a quarter of the items landed inside the 20% accuracy band, revealing that task-specific weights alone (in close-source models) were not enough when the range of prices was vast in order of magnitude. Figure 3.7 showed in chapter 3.2.2 of the methodology further reinforces the table results by showing that beyond $400, the red and yellow dots fell beneath the blue identity line, a sign of systematic underpricing resulted from the inability to predict prices of products that are more expensive in the dataset.

Table 4.1 – Performance of LLMs with Different Metrics

Model	MAE_$	RMSLE	Hit_%20	Median_ms	p95_ms
ENS	59.36	0.57	42.10	1302.20	2046.79
FT-LLM	98.46	0.78	23.75	613.42	1123.86
RAG-LLM	54.02	0.42	50.85	716.84	1125.42

RAG-LLM on the other hand was allowed to consult/retrieve five look-alike items from the vector store, and so we saw that its MAE almost halved to $54, RMSLE decreased to 0.42, and more than half of the items landed within the twenty-per-cent accuracy band, showing us an increase in performance when compared to the previous model. Once again, Figure 3.10

shown in chapter 3.3.1 confirmed what the table implied, the cloud tightened around the diagonal and the surplus of deep-green dots in the $100 to $400 range showed that local context and the ability to retrieve similar products in real time did far more for calibration and efficiency than any amount of memorized global statistics.

The ensemble was originally planned to outperform both models, yet as we saw the ENS failed to beat pure RAG, achieving a $59 MAE and a 42 per cent hit-rate. The corresponding scatter (Figure 4.1) contained fewer bigger errors than FT-LLM but also lost some of the razor-sharp alignment that made RAG stand out. A simple mean ends up diluting the stronger opinion, resulting in lackluster results. Weighted or confidence-based blending is left for future work, but we did learn that a second "step" is helpful only if the previous steps disagree in complementary, not redundant, ways.

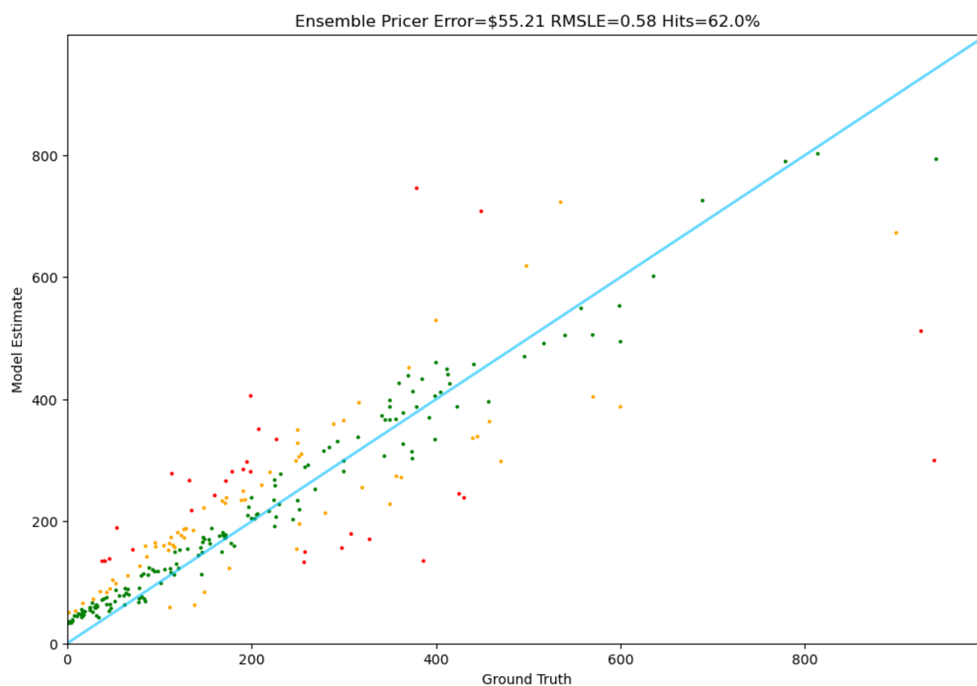

Figure 4.2 – Results of the Ensemble in 250 items of the Test Set

Accuracy however is not the only relevant metric when evaluating LLMs, as latency and cost are also extremely important indicators to measure. While FT-LLM responded very fast, in a median response time of 0.61 s, RAG-LLM needed 0.72 s to embed the query and fetch neighbors, a gap small enough that users could notice. The ensemble increased the median to 1.30 s which is to be expected because it calls both sub-models, doubling token usage and adding roughly 0.2 cents per request, which is not significant enough to be considered a pricier model call.

4.2. ANALYSING WORST PREDICTIONS

Looking deeper into the previous analysis we also searched for the 10 worst predictions (based on absolute error) for each mode, as seen in Table 4.2 below. The results made it clear that for all 3 models the biggest concern was the lack of understanding and predictive power for higher cost products, generally above the $700 mark.

Table 4.2 – 10 Predictions for Each Model with the highest Absolute Error

Model	truth	guess	err_abs
RAG-LLM	940.33	329.99	610.34
RAG-LLM	854.09	249.99	604.1
RAG-LLM	914.97	384.75	530.22
RAG-LLM	661.59	137.45	524.14
RAG-LLM	894.97	375.99	518.98
RAG-LLM	735	245	490
RAG-LLM	378.99	834.98	455.99
RAG-LLM	599.5	149.95	449.55
RAG-LLM	926	499	427
RAG-LLM	756.96	330.99	425.97
FT-LLM	940.33	89.99	850.34
FT-LLM	894.97	85.95	809.02
FT-LLM	854.09	79.99	774.1
FT-LLM	823.93	50	773.93
FT-LLM	859.99	89.99	770
FT-LLM	914.97	174.99	739.98
FT-LLM	812	100.49	711.51
FT-LLM	942.37	249.99	692.38
FT-LLM	929.99	249	680.99
FT-LLM	929.95	249.99	679.96
ENS	940.33	302.909	637.421
ENS	854.09	240.753	613.337
ENS	894.97	335.677	559.293
ENS	914.97	369.173	545.797
ENS	661.59	148.484	513.106
ENS	735	247.691	487.309
ENS	599.5	178.951	420.549

ENS	926	506.415	419.585
ENS	891.49	477.098	414.392
ENS	582.46	169.016	413.444

This shows us that a bias skewed dataset does impact a lot on LLM performance, since after curating the data most of our data points were lower cost products and goods, only a small percentage were very high-cost products. This made it so that the models didn't have enough context to understand and differentiate a higher cost product and does the performance on all 3 models in these instances was poor.

4.3. REST API's Analysis & Impact

As previously discussed, large language model agents become more powerful and even useful when they leave the chat window and interact with the outside world. In our prototype that outward-facing capability was mainly supplied by a single REST endpoint, the Pushover notification API used for the messaging agent, that is executed immediately after the planner agent finds a good deal by comparing the predicted price to the extracted "real" price from the RSS feeds. Although modest in scope a short JSON payload that delivers a push alert to a user device, the call sits on the critical path from inference to user action, so its cost, latency and reliability have a direct influence on the overall system.

The Pushover licensing model is simple. A one-time payment of US $4.99 unlocks unlimited personal use on each receiving platform (Android, iOS or Desktop), after a 30-day free trial. Message quotas are enforced per sending application rather than per user: every app may send up to 10 000 messages per calendar month at no additional charge, and one "team-owned" app may reach 25 000 messages before paid capacity is required. Additional blocks are sold once the free allowance is exhausted. The Teams license on the other hand, intended for organizations, moves to a subscription of US $5 per user per month and inherits the higher 25 000 free quota.

5. CONCLUSIONS AND FUTURE WORK

5.1. CONCLUSION

This thesis is meant as an exposition on the integration of external APIs in LLM-based agents and serves as a guide to the creation of such frameworks in a practical and easy to understand application of a real-time products deal finder. By first quantifying the performance of a stand-alone GPT-4o-mini, then a fine-tuned GPT-4o-mini and then contrasting it with a retrieval-augmented variant and a multi-agent framework endowed with an RSS crawler, a Chroma vector store and the Pushover messaging service, the study produced three main points.

First, the empirical evidence shows that real-time access to structured knowledge is more valuable than marginal increases in parameter count or naive few-shot fine-tuning. When the RAG pipeline supplied GPT-4o-mini with five semantically nearest neighbors retrieved from a 400 000-item Amazon corpus, mean-absolute error in price prediction fell by roughly forty per cent and the proportion of hits inside a 20 % band doubled. The fine-tuned model, in contrast, became too focused on a narrow portion of examples (lower-priced products) and lost much of the priority that makes foundation models useful in the first place.

Second, the case study demonstrates that a lightweight constellation of single-purpose agents can translate predictive gains into actionable utility with only modest latency overhead. Orchestrated through a thin Python layer, the scanner, planning, and messaging agents tracked multiple RSS feeds, filtered duplicates, triangulated fair value and delivered a mobile push alert in well under thirty seconds. Even with such a simple configuration the flow repeatedly found double-digit discounts (above 50%) that a human shopper, limited to one website at a time, would have missed. In a production setting the same architecture could be extended to dozens of feeds, richer knowledge graphs or user-defined watchlists without fundamentally altering the control flow and thus becoming an extremely useful application.

Third, the Pushover REST endpoint, although cheap and simple, still imposed the hard 10 000 messages per month and periodic network jitters limit. These constraints didn't eliminate the proof-of-concept but show that resilient agent deploys must treat every outbound call as on the critical path, subject to the usual monitoring, catching and fallback techniques that apply to any microservice design.

Combined, the experiments validate the central hypothesis posed since the beginning. External APIs do indeed materially enhance an LLM agent’s decision-making and action-taking capabilities, provided that the integration is engineered with explicit attention to data quality, prompt design and runtime budgets.

5.2. FUTURE WORK

There are several points for further investigation that became apparent while building this thesis and which deserve consideration, starting with the fact that data curation could have been substantially more rigorous. While the slot-reservoir approach did help alleviate the highly skewed price distribution in the Amazon dataset, the resultant set still overrepresented cheaper items and the dominant category Automotive. A stratified or adversarial sampling schedule, one that specifically hunts the under-represented price ranges and product groups, would have reduced the systematic underpricing seen on more expensive products.

In terms of language models, fine-tuning a closed-weight OpenAI model proved to underperform for this specific task despite being trained with Amazon data. Conducting the experiment with an open-source model such as Llama-3-8B, fine-tuned through parameter-efficient LoRA or QLoRA layers, would allow full-corpus supervision and gradient-level control, likely regaining the useful priors that the few-shot GPT-4o-mini lost. Likewise, the RAG model was made in a simple wat by retrieving the five nearest neighbors and passing them to the decoder unchanged. Incorporating an end-to-end light-weighted re-ranker like a cross-encoder trained on pairwise price similarity could result in the retrieval of more diagnostically informative embeddings and further decrease the MAE.

The agentic flow could also be further explored and extended. The planer already relies on one immutable heuristic threshold and offloads no sub-tasks, so incorporating it with the convenience of calling tools, scratchpad memory or reflexion loop could allow it to review its own valuations and back-off from grazing deals. Additionally, the architecture included only one foreign service, the Pushover API. Checking redundant alerting channels, running currency converters or real-time inventory checkers would demonstrate how multi-ingredient tool chains impact end-to-end latency, reliability and usability.

Future work would also need to be able to capture individual-agent failure modes, broadcast estimates of uncertainty to the user interface and capture longitudinal outcomes such as click-through, buy conversion and alert fatigue. Merging such person-centric quantities with ablation analyses of API availability would add clarity to under which conditions, and to what extent, external aids are warranted in terms of operational overhead.

BIBLIOGRAPHICAL REFERENCES

Kaplan, J., McCandlish, S., Henighan, T., Brown, T. B., Chess, B., Child, R., Gray, S., Radford, A., Wu, J., & Amodei, D. (2020). Scaling laws for neural language models. arXiv preprint arXiv:2001.08361. https://doi.org/10.48550/arXiv.2001.08361

Liu, X., Yu, H., Zhang, H., Xu, Y., Lei, X., Lai, H., Gu, Y., Ding, H., Men, K., Yang, K., Zhang, S., Deng, X., Zeng, A., Du, Z., Zhang, C., Shen, S., Zhang, T., Su, Y., Sun, H., … Tang, J. (2023). AgentBench: Evaluating LLMs as agents. arXiv preprint arXiv:2308.03688. https://doi.org/10.48550/arXiv.2308.03688

McAuley, J., Pandey, R., & Leskovec, J. (2023). *Amazon-Reviews-2023* [Data set]. Hugging Face. https://huggingface.co/datasets/McAuley-Lab/Amazon-Reviews-2023

Mialon, G., Dessì, R., Lomeli, M., Nalmpantis, C., Pasunuru, R., Raileanu, R., Rozière, B., Schick, T., Dwivedi-Yu, J., Celikyilmaz, A., Grave, E., LeCun, Y., & Scialom, T. (2023). Augmented language models: A survey. arXiv preprint arXiv:2302.07842. https://doi.org/10.48550/arXiv.2302.07842

Patil, S. G., Zhang, T., Wang, X., & Gonzalez, J. E. (2023). Gorilla: Large language model connected with massive APIs. arXiv preprint arXiv:2305.15334. https://doi.org/10.48550/arXiv.2305.15334

Qin, Y., Liang, S., Ye, Y., Zhu, K., Yan, L., Lu, Y., Lin, Y., Cong, X., Tang, X., Qian, B., Zhao, S., Hong, L., Tian, R., Xie, R., Zhou, J., Gerstein, M., Li, D., Liu, Z., & Sun, M. (2023). ToolLLM: Facilitating large language models to master 16 000+ real-world APIs. arXiv preprint arXiv:2307.16789. https://doi.org/10.48550/arXiv.2307.16789

Schick, T., Dwivedi-Yu, J., Dessì, R., Raileanu, R., Lomeli, M., Zettlemoyer, L., Cancedda, N., & Scialom, T. (2023). Toolformer: Language models can teach themselves to use tools. arXiv preprint arXiv:2302.04761. https://doi.org/10.48550/arXiv.2302.04761

Shinn, N., Cassano, F., Berman, E., Gopinath, A., Narasimhan, K., & Yao, S. (2023). Reflexion: Language agents with verbal reinforcement learning. arXiv preprint arXiv:2303.11366. https://doi.org/10.48550/arXiv.2303.11366

Wei, J., Wang, X., Schuurmans, D., Bosma, M., Ichter, B., Xia, F., Chi, E., Le, Q., & Zhou, D. (2022). Chain-of-thought prompting elicits reasoning in large language models. arXiv preprint arXiv:2201.11903. https://doi.org/10.48550/arXiv.2201.11903

Yao, S., Zhao, J., Yu, D., Du, N., Shafran, I., Narasimhan, K., & Cao, Y. (2022). ReAct: Synergizing reasoning and acting in language models. arXiv preprint arXiv:2210.03629. https://doi.org/10.48550/arXiv.2210.03629

Zhuang, Y., Yu, Y., Wang, K., Sun, H., & Zhang, C. (2023). ToolQA: A dataset for LLM question answering with external tools. arXiv preprint arXiv:2306.13304. https://doi.org/10.48550/arXiv.2306.13304

NOVA Information Management School
Instituto Superior de Estatística e Gestão de Informação

Universidade Nova de Lisboa