A Natural Way of Building Financial Domain Expert Agents

Gagandeep Singh Kaler*
Dimitrios Vamvourellis*
Deran Onay*
Yury Krongauz*
Stefano Pasquali*
Dhagash Mehta*

Abstract

Financial experts possess specialized knowledge that is not easily accessible, making the acquisition of such expertise a time-intensive process. Leveraging Large Language Models (LLMs) to emulate financial domain experts offers a promising solution, which can offload routine responsibilities from human experts, allowing them to focus on more strategic tasks. However, developing a GenAI agent that matches the capabilities of a financial domain expert requires more than just LLMs with Retrieval-Augmented Generation (RAG) capabilities. The agent must interact with domain-specific data sources, perform complex analyses, and understand niche terminologies and processes. We propose a *natural way* of developing of GenAI-powered financial domain experts by following a zero-shot approach. Our agent's memory layer has complimentary capabilities to few-shot prompting and provides a *natural way* of remembering information as it interacts with domain experts.

This paper is presented as a case study where we propose a comprehensive framework for building financial domain expert agents. Our approach involves iteratively enhancing a basic LLM with data extraction layer, coding capabilities, and a memory layer to perform complex analyses. We show how addition of each layer to our LLM agent improves its performance and also address the necessary safety and governance processes to ensure the robustness and accuracy of production ready agent. We also introduce a custom dataset (having roots in the financial domain) for evaluating the agent's performance in numerical analysis and multi-step reasoning, providing a clearer picture of the agent's capability to mimic a financial domain expert.

Keywords

GenAI, Large Language Models, Memory, Finance, Natural agent, Zero-Shot, Financial domain expert

ACM Reference Format:
Gagandeep Singh Kaler, Dimitrios Vamvourellis, Deran Onay, Yury Krongauz, Stefano Pasquali, and Dhagash Mehta. 2024. A Natural Way of Building Financial Domain Expert Agents. In *Proceedings of 5th ACM International Conference on AI in Finance (ICAIF '24).* ACM, New York, NY, USA, 10 pages. https://doi.org/XXXXXXX.XXXXXXX

*BlackRock Inc., New York City, NY

ICAIF '24, November 14–17, 2024, Brooklyn, NY

ACM ISBN 978-x-xxxx-xxxx-x/YY/MM
https://doi.org/XXXXXXX.XXXXXXX

1 Introduction

Financial experts possess specialized knowledge of the domain, tools, and processes that are not easily understood by everyone. For a novice, acquiring these expertise can be an intensive process requiring significant time commitment from the domain expert. Additionally, from a compliance perspective, it is challenging to identify gaps in quality control and governance processes, as understanding the domain itself is complex, let alone recognizing what is missing.

Owing to their broad capability, leveraging LLMs to emulate a financial domain expert could be considered. This approach can significantly offload the responsibilities of human experts, allowing them to focus on more complex and strategic activities. Additionally, this should improve understanding for general users, which could help identify and address possible compliance gaps. However, LLMs themselves are plagued with numerous issues such as hallucinations, lack of long-term memory, and inability to follow complex instructions (missing steps in the middle) [2, 13, 33]. In addition to these issues, substantial time and effort is required during prompt engineering (including creating relevant few-shot examples) to enhance their performance [22]. These challenges are significant in any domain, but addressing them becomes even more critical in the financial sector, where accuracy and precision are paramount [18].

Developing a GenAI agent at par with a financial domain expert requires capabilities beyond understanding financial terminologies. Therefore, complementing our LLM agent with RAG capabilities is insufficient [18, 31]. At a minimum, it needs to interact with domain-specific data sources and perform complex analyses before it can respond to a user's request. It must understand terminologies and processes specific to the niche domain it aims to specialize in, rather than hallucinating or misinterpreting meanings from different domains. It also needs the ability to follow long processes precisely to answer questions requiring complex analysis.

We propose a comprehensive and streamlined approach that addresses the aforementioned shortcomings and provides a *natural way* of developing GenAI-powered financial domain experts. In this paper, we follow an iterative improvement methodology, starting with a basic LLM agent and progressively enhancing it with tools and features until it can emulate our financial domain expert.

Working with financial datasets requires additional considerations around safety and governance. Therefore, for the layers we

discuss, we also elaborate on the steps needed to make our agent safe, robust, and production-ready [3, 18, 20].

Datasets used to assess performance of LLMs are primarily on language based reasoning [16]. To evaluate our LLM agent's performance in a given financial domain, we created a custom dataset focused on precise numerical analysis and multi-step reasoning. The questions in this dataset are relevant to the provided financial database schema, enabling combined performance assessment on appropriate tool calling capabilities and multi-step numerical analysis. Therefore, the performance of our LLM agent on this dataset should give a clearer picture of it's capability to mimic a financial domain expert.

2 Methodology

Gaining expertise in any domain requires extensive investment of both time and effort. Instead of training novices on the intricacies of a domain, we propose a framework for creating domain expert LLM agents. Our framework targets a *natural way* of developing financial domain expert. So instead of spending extensive efforts initially in the prompt engineering phase, we let the LLM agent learn from its interaction with the domain expert. Allowing for a more *natural way* of bestowing knowledge from a domain expert to our LLM agent.

Developing a monolithic mega agent presents several challenges and disadvantages [6]. Therefore, we demonstrate an iterative agent approach, progressively adding layers of capability to transform any LLM into a financial domain expert. We begin by introducing knowledge of domain-specific datasets in the form of SQL database schema and complementary data extraction tool. We then add coding capabilities to perform additional analytics on extracted data. Introducing a memory layer helps bridge the gap from a general coding agent to a financial domain expert. This memory layer provides complementary features to few-shot examples, where the LLM agent is able to automatically learn generalized processes as a domain expert interacts with it. In the memory layer we also incorporate a keywords databases to accurately interpret the meanings of terminologies and processes specific to a given financial domain, avoiding hallucinations from different domains. This iterative approach makes our domain expert agent more modular, providing easy optimization capabilities and decoupling benefits.

2.1 Data Extraction Layer

The data source for a given financial domain can exist in various formats. Some of the most prevalent ones are SQL databases and API endpoints. For the purpose of this case study, we have chosen SQL databases. However, without loss of generality, API endpoints or any other data source can also be accommodated within the framework.

We begin by incorporating domain-specific SQL database schema information into our system prompt. Additionally, we include few example records per table to provide the LLM with an understanding of the type of data it will be working with. As part of the system prompt, the LLM is instructed to generate SQL statements to answer user queries. We introduce a corresponding tool responsible for extracting data using the LLM-generated SQL statements. The extracted data is then sent back to the LLM for further inference, allowing it to provide the final answer to the user's original questions.

2.1.1 Drawbacks. Adding access to the data source is a necessary first step. However, relying solely on this capability presents several drawbacks. LLMs are capable of generating complex SQL queries with numerous sub-queries and joins (along with using Turing complete features) to answer atypical user questions. However, the rate of error (both syntactic and logical) increases with increasingly complex SQL queries, hinting at its limitations when employed for complex analysis [29]. Using simpler queries and sending extracted data back to the LLM for further analysis can help mitigate these limitations to some extent, but it introduces data compliance issues. Confidential financial data should not be exposed to a remote LLM. While capable locally-hosted LLMs can be used, they come with large infrastructural requirements, which introduce additional challenges. As LLMs are advancing at mind boggling rate, being stuck to an out-dated local LLM in a non-optimal choice [17]. Therefore, framework designs that use confidential data and can still leverage remote LLMs is desirable. [1]

2.1.2 Limited Dataset Approach. This approach of including schema information within system prompt works well when schema information is limited. Even though the context length of LLMs is increasing by the day, this approach should be avoided for large database schema given cost and data similarity implications [28]. Introducing large database schema without any pre-processing can result in the LLM agent choosing wrong dataset because of data similarity (based on similarity in name, datatype or example records). Or choosing different datasets with similar contextual information between different runs. Tackling large databases is challenging and alternative approaches needs to be employed. Creating multiple niche agents operating with a dataset's subset and following a multi agent approach could be a solution. Limited dataset exposure can make our niche agents more tractable. Alternatively, dataset subset selection can be a precursory step before feeding control to the LLM agent. We let the user choose between different approaches available in literature to tackle large dataset problem [7, 8, 26].

2.1.3 In essence. Domain experts perform multiple steps of complex analysis on extracted data before providing final answers. Even though there has been significant advancement in improving the performance of SQL-enabled LLM agents, such performance gains are usually attributed to time intensive prompt engineering with appropriate few-shot prompting or fine-tuning LLM models on dataset [7, 8, 26]. Given our goal of zero-shot agent development, we soon realize that atypical analysis is beyond the scope of a SQL-enabled LLM agent. Also, data confidentiality requirements needs to be met. Therefore, we introduce additional capabilities to our agentic framework.

2.2 Scripting Layer

Incorporating a programming language capable of running analyses performed by financial domain experts is essential for building our domain expert agent. We need to brings in capabilities like data

[1] Remote LLMs can still have confidentiality issues through exposed database schema and human chat messages. But keeping the raw data within the bounds of our system should help with data licensing and localization requirements.

visualization and data analysis using advance libraries (support of which is current lacking in SQL). To simplify the framework, we opt for scripting languages over compiled languages. Given its widespread use in data science, we have chosen Python. Additionally, Python's popularity ensures that trained LLMs have encountered a substantial corpus of newer sample code fragments, which should result in improved performance of generated code [25].

2.2.1 Loosely Coupling with Data Extraction Layer. Integrating a scripting layer atop our data extraction layer enables the LLM to decompose complex problems into simpler data extraction and scripting tasks. This division enhances problem tractability, as each system addresses more manageable sub-problems [15].

To connect the two systems, we employ the following design for the SQL tool (*DataFrameFromSQL*) and the scripting tool (*PythonScript*):

DataFrameFromSQL	
Field	**Description**
select_query	An SQLite SELECT statement.
df_columns	Ordered names to give the DataFrame columns.
df_name	The name to give the DataFrame variable in downstream code.
PythonScript	
Field	**Description**
code	Python script code.
df_names	The names of the DataFrames that are already present in memory and the code will use.

Table 1: Class Fields and Descriptions

```
class DataFrameFromSQL(BaseModel):
    select_query: str = Field(...)
    df_columns: List[str] = Field(...)
    df_name: str = Field(...)
class PythonScript(BaseModel):
    code: str = Field(...)
    df_names: List[str] = Field(...)
```

(NOTE: Field description are included in Table 1.)

The LLM can invoke the *DataFrameFromSQL* tool multiple times to gather all necessary datasets. Each dataset is assigned a name using the `df_name` parameter, which is subsequently utilized in the *PythonScript* tool via the `df_names` parameter.

2.2.2 Handling Errors. Despite significant advancements in LLMs' code generation capabilities, a considerable portion of generated code still contains syntax, runtime, and logical errors. To address these issues, if we encounter error during generated code execution, we append the error message to the response, prompting the LLM agent to retry with error information. This approach allows the LLM to refine its code incrementally rather than restarting from scratch. There are a good number of error handling techniques, like simply retrying, writing accompanying unit-tests, etc. [1, 32]. The approach we choose provides a good balance of simplicity, accuracy, latency and token usage. Once the LLM successfully self-corrects, we eliminate all error correction dialogues to minimize unnecessary token usage. Retaining correct code ensures more accurate responses to related follow-up questions.

In practice, we integrate the error correction capability holistically with the LLM agent itself. This allows the LLM agent make retry attempts for not only errors in generated scripting code, but also for data extraction layer and tool call parsing. As up until the launch of OpenAI's Structured Outputs, we cannot guarantee tool call schema is respected [21].

2.2.3 Safety and Security. Executing generated code without safeguards poses significant security risks. Malicious code can introduce substantial operational hazards. Therefore, it is crucial to implement measures to prevent and mitigate the execution of harmful code. We address this through the following solutions, which can be used in conjunction with each other:

- *Deploying a separate adversarial LLM to validate that the generated code adheres to predefined safety rules.* This LLM is explicitly programmed with validation rules and provided with the generated code to ensure compliance. Although this incurs a slight performance cost, the safety benefits justify its integration into any code generation framework [19].
- *Running generated code in isolated Docker containers.* This sandbox environment, devoid of elevated privileges (also known as `sudo` access), read/write, or network access, limits the potential impact of malicious code. The framework transfers only the generated code and associated `DataFrames` to the container, which solely performs code execution. The container's output is then type-validated before being serialized back to the host machine. This design ensures validation of both inputs and outputs.
- *Parsing the generated Python script as an abstract syntax tree before execution.* With this step we can prevent importing certain packages/modules, providing finer control over the generated code for enhancing security [14].

2.2.4 Advantages.

- Loosely coupling the data extraction and scripting layers allows the framework to reuse named `DataFrames` already available in memory when addressing related questions, instead of re-extracting data again.
- Error handling through self-correction (with error context) improves overall performance, as we will also see in section 3.
- The Python scripting layer enables the framework to generate graphs/plots when responding to queries, enhancing understanding by engaging visual cortex [11].
- Owing to its popularity Python supports a huge selection of advance data analytics libraries, which can easily support domain specific analysis needs.
- Combining data extraction and scripting layers facilitates complex analyses which was difficult to achieve in a zero-shot manner from the data extraction layer alone.

2.3 Memory Layer

Adding a scripting layer to our LLM agent makes it sufficiently capable of performing complex analysis. However, to bridge the gap between a general coding agent and a financial domain expert, our

LLM agent needs to answer domain related questions like "What's the most likely factor to cause recession?". It should decompose such questions into a complex process, composed of a series of steps. The agent then needs to follow this process precisely and combine outputs from multiple steps to provide the final answer.

Adding memorization capability to our LLM enables this functionality. We use a vector database where the keys are domain-related questions or process headings, and the related step-by-step process to follow to get the answer is linked in its metadata. To find relevant chunks of memory we employ a two tier process.

- First, a user's query is generalized and searched for within the vector database. During generalization, we ask the LLM to strip away specifics and come up with a few variations of generalized questions. This generalization allows us to get better match in the vector database where the domain-related questions and processes are also saved in a generalized manner. If a match is found, we use the metadata to complement the user's question with a process that must be followed to provide an answer. [2]
- If we do not find a match for the question being asked, we try to find how to solve part of the question. At this step, we ask LLM to divide the original question into an a series of smaller processes which needs to be solved to get the final answer. These smaller processes are searched for in the vector database, to find how to solve part of the question. Eg: If we ask "Excluding all periods of recessions, is current unemployment higher or lower than last 10 years average?", then the LLM will try to find "Determine Recession Periods" and "Calculate Unemployment" inside the memory layer.

In summary, we try to find the process which can be used to answer the question asked. If we are successful in finding the process then we proceed ahead by including this information with our question. If we are not successful then the question is broken down into smaller processes which are looked for inside the memory layer. So, we can use memory to solve part of the question if not whole, and let the LLM agent decide on how to solve the rest of it (based on current chat context or SQL schema information).

2.3.1 Adding Entries to the Vector Database. The entries in the vector database (or the memory layer) are learned through interaction with the domain expert. We start with an empty database and let the domain expert use the LLM agent to perform analysis. The domain expert, with knowledge of different systems, asks directed questions like "Compare today's fit data against last year's and tell me if the fit values are within 1 standard deviation of last year's values." The expert can continue asking numerous simpler questions to build up their analysis, while keeping an eye on the accuracy of conducted analysis.

Once the human expert has built up their analysis by using LLM agent provided outputs to simpler questions, the expert can ask the LLM agent to remember their analysis. [3] This is when a *SaveIntoMemory* tool is called, which triggers a series of steps in the memorization process:

- The LLM agent summarizes the entire history of interaction between the human expert and itself as a series of steps to be followed to get to the final answer.
- These steps are summarized in a generalized fashion, where any specifics are replaced with variables. This is analogous to creating a function with variables in contrast to remembering a fixed recipe.
- The LLM agent then creates a related generalized *process heading* based on the chat history, so it can be used to refer to this process. Alternatively, the human expert can provide their own *process heading* for more precision.
- Finally the generalized *process heading* is added to the vector database with its metadata linking to the generalized summarized steps which should be followed to get the final answer.
- While saving entry to the vector database, if there's a match with an existing entry then the user is notified. This allows the user to choose between rewriting memory (if the new process is better), or ignoring new entry (if the old process is similar), or provide a different *process heading* (if they want to keep both). Depending on the needs of the system, more sophisticated memory models can be employed to enhance this layer and users can take inspiration from existing literature [5, 24, 30, 33].

This is a more *natural* process of memorization when compared to creating few-shot examples, which can provide similar functionality. The interaction of domain expert with the LLM agent is similar to interaction with a capable novice, who remembers complex processes when asked to. This is in line with our goal of a *natural way* for building LLM agents, where no time is invested up front for prompt engineering or creating few-shot examples. But, the LLM agents capability keeps expanding as domain experts interact with it.

2.3.2 Domain Specific Knowledge Layer. Adding the layers of capabilities so far allows our LLM agent to perform complex analysis as a domain expert would. However, there is still a gap that needs addressing before we can truly claim that our LLM agent can become a domain expert. In any domain, there are keyword terminologies (both internal and publicly known) which should be used precisely to perform correct analysis. If not provided with the exact meaning, our LLM agent might hallucinate or infer meaning from other domains. We address this issue by adding a question rephrasing capabilities which:

- Ask our LLM to identify keywords whose meaning cannot be exactly inferred from chat history or data schema.
- Finds the meaning of these keywords in the keyword databases. We use a combination of both traditional database and a vector database to first try to exactly match keywords, post which remaining are searched for in a semantic fashion within the vector database.

[2] As a simple enhancement, we don't solely rely on vector similarity. We also use a separate LLM to decide if the vector matched processes can be used to answer the original question. If so, then this LLM should select the most relevant process. But if the LLM decides the processes are not relevant, then we go ahead without any modification to the original question. This allows us to enhance the relevance of our vector matched selections, while spending less time on optimizing vector database using other techniques.

[3] A concrete example is shown in appendix B.

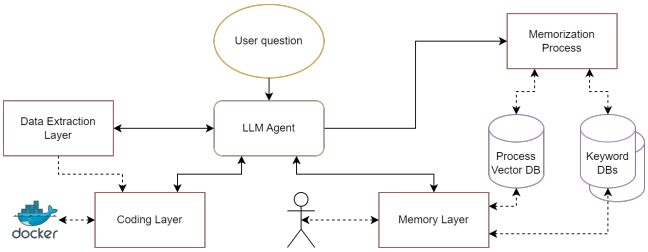

Figure 1: Architecture of GenAI Powered Financial Domain Expert Agent.

- Rephrases the user's question with the found meaning so there is less ambiguity in how the user's question needs to be interpreted.

If the rephrased question still has unknown keywords or if the databases have no additional information about the requested keywords, then the LLM agent asks the user for clarification instead of hallucinating meaning. This creates a human-in-the-loop LLM agent, where a new keywords are also introduced in the databases as humans provide more context [12]. Addressing hallucinations is an important facet of building reliable LLM agents, which is a critical requirement for building financial domain expert agents [18].

2.4 Bringing it All Together

We start with the architecture as described in Figure 1 with no information added to memory databases. The only piece of domain specific knowledge the LLM agent has, is the provided SQL database schema. At this stage, the LLM agent is capable of leveraging data extraction and coding layers to perform simpler analysis. A domain expert uses these capabilities to performing analysis as they usually would for their day-to-day work, correcting the LLM when required. The interaction continues with follow up questions till the expert is satisfied with the final result of their multi-step analysis. Then the domain expert asks the LLM agent to remember the process, which triggers the memorization process. This allows the LLM agent to performing more complex multi-step analysis for simpler questions (or processes) saved in its memory. At any point if the LLM agent is unable to infer meaning of certain keywords it looks into the keywords databases (which starts empty) and reframes user's question by injecting found meaning, so there is less ambiguity. The LLM agent gets back to the user for clarity if the meaning cannot be found in its memory and stores this meaning in keywords databases for subsequent interactions.

Building expert agents requires working with domain-specific knowledge. Instead of manually feeding in complex processes or the meanings of keywords, we let our LLM learn from its interaction with a domain expert. This allows for a more *natural* interaction with the LLM agent instead of investing efforts into prompt engineering or creating relevant few-shot examples upfront.

Building LLM agent layer by layer instead of as a monolithic mega-agent also provides easy decoupling and optimization benefits:

- Decoupling sub-agents allows for easy replacement of modular parts to adapt to different use cases [15]. For example, interacting with an API instead of an SQL database, or using different language than python, or using different memory models for process memorization etc.
- Individual layer's prompt optimization using DSPy [10]. DSPy allows for algorithmic prompt optimization by relying on example dataset. Optimization monolithic mega-agent (with several tools and functionality) is much harder in both prompt optimization and example dataset creation. Prompt optimization for decoupled sub-agents in their own isolated context is more tractable. So more focus can be made on the layer whose performance needs improvement. This allows us to quickly improve our decoupled agent's performance for faster deployment.

3 Performance Testing and Benefits

3.1 Dataset Creation

Performance assessment of our financial domain expert is dependent on defining the domain within which it is tested. This consists of two parts. First, is the domain specific SQL database. Second, is the set of questions against which its accuracy will be measured.

3.1.1 SQL Database. Our database contains multiple tables of macroeconomic data extracted from publicly available FRED dataset (See appendix A.1) [4]. There are several tables including time-series data for Consumer Price Index (CPI), Gross Domestic Product (GDP), money supply, rates, popular stock index and unemployment. Using public datasets should allow interested user to verify LLM agents performance as described in this paper.

3.1.2 Performance Evaluation Dataset. To effectively evaluate our LLM agent's performance, it must be tested on appropriate datasets. Unlike traditional LLM applications, which often rely on datasets for language-based reasoning, financial domain expert agents require testing focused on precise numerical analysis and multi-step reasoning. To address this, we developed a dataset specifically designed for our needs. For reference, we draw inspiration from articles, where they created their own datasets to assess their domain specific performance [9, 27]. The questions in this dataset are relevant to the provided database schema, enabling combined performance assessment on appropriate tool calling capabilities and multi-step numerical analysis. The performance of our LLM agent on this dataset should give a clearer picture of its capability to mimic a financial domain expert.

Performance evaluation dataset contains 30 questions related to these macroeconomic datasets. (See appendix A.2). These questions require multi-step analysis before coming to the correct answer and are inspired by practitioners use of such macroeconomic time-series.

3.2 Performance Comparison

We test the performance of our LLM agent under different sets of features available to it.[4] The performance is divided into four parts, specifically:

[4]All testing was performed using *gpt-4o* (with temperature set to 0) for chat model and *text-embedding-3-large* for embedding model.

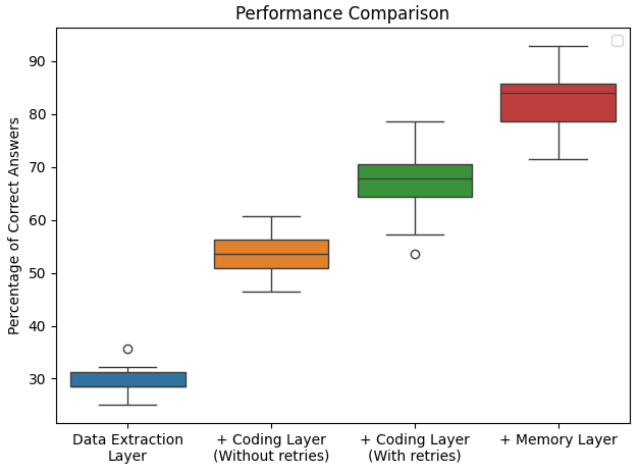

Figure 2: Performance comparison

- *Data Extraction Layer*: The LLM agent has access only to the data extraction layer. It can run analysis on the extracted data to reach the final answer. There are data leak concerns with this approach as highlighted in section 2.1. But for the sake of performance testing on public dataset we ran this analysis.
- *Data Extraction Layer + Coding Layer (with no retry)*: In this part, we introduce coding layer without retries when it encounters errors. The LLM agent does not have access to the output data (from data extraction and coding layers), as architecturally it's displayed to the user directly from the python code.
- *Data Extraction Layer + Coding Layer (with retries)*: Same as previous part, but we let the LLM agent retry few number of times in case of syntactic or runtime errors. As highlighted in section 2.2.2, we include error message to let the LLM agent use this information to correct the error made.
- *Data Extraction Layer + Coding Layer (with retries) + Memory Layer*: Finally we run performance comparison by adding memory layer. In this step relevant memory chunks containing step-by-step process to solve all or part of the question is added inside the chat context, as described in section 2.3.

Figure 2 shows a box plot of percentage of performance evaluation questions (as described in section 3.1.2) answered correctly over several runs. [5] We look at the overall performance on our evaluation questions to look at the capability of LLM agent under different stages of capability. From Figure 2 we can see that:

- With just *Data Extraction Layer* we are able to successfully answer ~ 29.29% of time. From appendix A.2 we can see the kind of questions the LLM agent has to answer are challenging for a simple SQL agent. It has been shown in literature that the performance of SQL agent can be improved by including relevant few-shot examples or by fine-tuning the LLM [23, 29]. But given the zero-shot architecture chosen for the development of domain experts, we were bound to perform sub-optimally here.
- When we introduce *Coding Layer*, our performance improves to ~ 53.57%, as the LLM agent is able to use coding skills to divide the larger problem into simpler data extraction and coding problems. We can also see that introducing retries (on syntactic or runtime failures) leads to further improvement in performance to ~ 66.79%, which is inline with our expectations.
- Introducing *Memory Layer*, improves our expected performance to ~ 82.86%. 9 out of 30 questions are aided partly by this *Memory Layer*. The *Memory Layer* only provides partial help while solving these 9 questions, where only part of the question is aided by the provided memory. The LLM agent still has to do further multi-step analysis to come to the final result. See appendix B for details on this included memory chunk.

One might argue that this is an unfair comparison. But the goal of this comparison is to show the marginal benefits each iterative layer brings, within the realm of zero-shot agent creation. Improving capability of any layer will push the frontier of performance for all subsequent iterative layers. So, if the developer improves capabilities of *Data Extraction* layer, by including few-shot examples or fine-tuning LLMs or using other state-of-the-art techniques, the capabilities of subsequent layers will also increase leading to better overall performance with all layers combined.

The *Memory layer* provides complementary functionality to few-shot prompting, where the agent learns about processes to solve questions and meanings of keywords. Within our dataset we have used only one chunk of memory, which describes recession calculation process. To keep the comparison between "without *Memory Layer*" agents and "with *Memory Layer*" agents fair, we include the following textual description for "without *Memory Layer*" agents: *A recession is typically defined as two consecutive quarters of negative real GDP growth. Recession starts at then start of second downfall, and ends at the period before GDP growth resumes.* To mimic a financial domain expert, our LLM agent needs to perform precise calculations respecting exact dates and periods. Describing the aforementioned textual description as a series of steps (as shown in Figure 5 and described in appendix B) leaves less ambiguity on how it is to be interpreted. Thereby helping improve accuracy over more complex tasks.

3.2.1 Complex vs Simpler Tasks. Questions in our performance evaluation dataset are divided into two categories, complex and simpler. Even though both categories require multi-step analysis to get to the final answer, complex questions have at least one of the following characteristic:

- They require analysis on different datasets, where analysis output of one dataset is used to run analysis on second dataset, before final answer can be produced.
- They need to follow precise step-by-step numerical, date and period based calculations to get the final answer. Questions related to recession calculations fall under this category.

[5] For every tested question, we passed answer format requirement (as part of the prompt) to the LLM agent. The output from our LLM agent is passed to a secondary LLM along with answer format requirements for reformatting if required. So not adhering to output format does not count as a failure. We also do a manual validation of the outputs post this process. The final results are then matched exactly, except for floats which are matched till 2^{nd} precision digit.

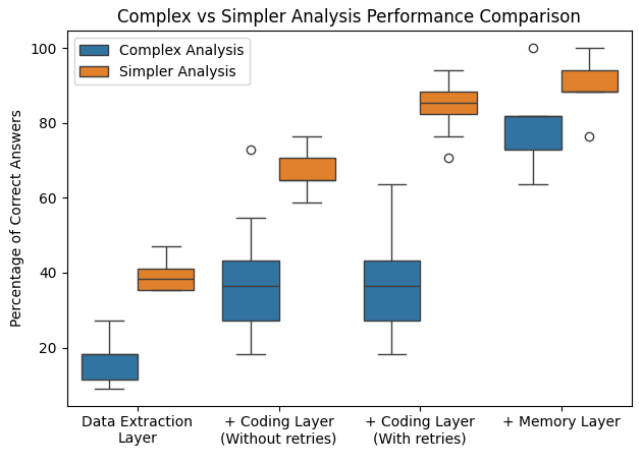

Figure 3: Complex vs Simpler Analysis Questions Performance Comparison

Summarized By	Correct Answers (%)	Failure (%)
LLM Agent	100%	0%
Human	~ 19.75%	~ 18.52%
Human (with nudging)	~ 27.16%	~ 16.05%

Table 2: LLM Agent Summarized vs Human Summarized Accuracy on Long Form Analysis.

From Figure 3 we can see that performance improvement for *Simpler Analysis* increases significantly when moving from *Data Extraction Layer* to *+ Coding Layer (without retries)* to *+ Coding Layer (with retries)*, because of improved capability and error correction. But there is insignificant change between *+ Coding Layer (with retries)* and *+ Memory Layer* and we are able to get over ~ 84.12% accuracy. Highlighting that a good fraction of simpler questions can be answered accurately without any memory layer. This result is significant as we need a base set of capability, which is used (as building blocks) to create more complex processes, which in turn can be saved into memory to add capability to answer more complex questions.

For Complex Analysis, we see insignificant change between *+ Coding Layer (without retries)* and *+ Coding Layer (with retries)*, as complex problems are more likely to be wrong without following a precise (process possibly stored in memory). So being wrong post successful code execution gives similar performance to failing to produce results. But we see a significant increase in performance when *+ Memory Layer* is added. Going from ~ 36.36% to ~ 79.09% accuracy with addition of memory. Demonstrating the advantages of using memory layer for complex analysis, even when used to solve a part of the process (see appendix B for details on included memory chunk). This benefit becomes even more pronounced when the entire process can be loaded from memory, as we will see in next section.

3.3 Long form analysis

We describe long form analysis as the ability to follow a series of steps precisely to get at the final answer. Recession calculation process stored in memory layer is a good example of such long form analysis. Table 2 shows comparison between *LLM agent summarized steps* and *human summarized steps* for long form analysis based on recession calculation process. LLM summarized steps are exactly as used in memory layer (see appendix B). Human summarized steps come from human messages which were sent to the LLM agent to create new memory (see appendix B.1).

From Table 2, we can see the following:

- LLM Agent summarized results are always correct across several runs with different parameters. This also gives confidence that memory used for exact calculations should see significant reduction in hallucinations. So, if we had created memory for all 30 questions in our dataset, we should see much better results than using memory for part of our multi-step analysis (as done currently).
- Human summarized results are accurate ~ 19.75% of the time and fails ~ 18.52% of time (despite several retries). On analyzing the results, we found the incorrect answers were mostly off by 1 period of analysis.
- Human summarized steps with nudging was used to try to fix the 1 period deviation resulting in wrong answer. This helped improve the correct results to ~ 27.16%, showing improvements attributing to prompt-engineering. [6]

Even with human summarized steps with nudging, the results are still far inferior to what LLM agent summarized process highlighting sub-optimal prompt-engineering. This highlights well known drawback of LLM where steps in the middle are missed [13].

Given our goal is development of domain expert in zero-shot realm, our focus has been on avoiding prompt-engineering. Even though this analysis was done for a single use-case, for our goal and use-cases LLM agent summarized steps is a much preferred alternative. More analysis needs to be performed for a conclusive decision, but we keep that option open for future work.

Based on our test, we found that LLM agent's summarizing of the process in its own words leads to much better performance. This memorization process is superior both in terms of learning and reproducing. It replaces time-intensive prompt optimization with interactive chat with a responsive LLM agent. The agent can handle longer processes without forgetting steps in the middle, compared to human-summarized steps [13].

4 Conclusion

In this case study we looked at a *natural way* of building financial domain expert agents. We demonstrate how capabilities of a basic LLM can be improved by adding data extraction layer, coding layer and a memory layer. We show how iteratively adding each layer of capability improves the performance and also address the necessary safety and governance processes to ensure robustness and accuracy of the agent.

[6] With prompt engineering we should be able to get much better results than what we have here. But given we are working under the realm of zero-shot prompting, we did not invest more time engineering our prompt for a singular case, when memory layer gives us a clear win.

For *natural* development of our LLM agent, we took a zero-shot approach to avoid the intensive process of creating few-shot examples upfront. Our memory layer provides complimentary capabilities to few-shot prompting. Which allows the LLM agent to remember interactions with domain expert, improving its capabilities on similar tasks. Thereby, providing a more *natural* approach for creating expert agents from a general coding agents.

For our zero-shot LLM agent, we saw that introducing coding layer (in addition to data extraction layer) results in improved accuracy by diving a complex task into shared responsibility between data extraction and coding layers. Coding layer also brings in capabilities like generating graph/plots and the ability to use advance data analytics libraries. These feature-set are part of the arsenal used by domain experts in their day-to-day analysis. Therefore, enriching out LLM agents with such capabilities, brings them a step closer to emulating a domain expert. The memory layer shows clear benefits when performing complex analysis by achieving higher accuracy in such tasks, even when memory is supporting just part of the multi-step analysis. These benefits should become more pronounced when using memory layer for all steps in the multi-step analysis. We also saw benefits of the memory layer, where LLM agent summarized steps are less prone to missing steps in the middle, even though more evidence is needed to conclusively back this claim, and is left for future work. Overall, based on the results of our LLM agent on the financial domain dataset, we can claim that the architecture presented in this paper presents several advantages and it presents a *natural way* of building financial domain expert agents.

5 Acknowledgment

The views expressed here are those of the authors alone and not of BlackRock, Inc.

References

[1] Nadia Alshahwan, Jubin Chheda, Anastasia Finegenova, Beliz Gokkaya, Mark Harman, Inna Harper, Alexandru Marginean, Shubho Sengupta, and Eddy Wang. 2024. Automated Unit Test Improvement using Large Language Models at Meta. *arXiv preprint arXiv:2402.09171* (2024). https://arxiv.org/abs/2402.09171

[2] Sourav Banerjee, Ayushi Agarwal, and Saloni Singla. 2024. LLMs Will Always Hallucinate, and We Need to Live With This. *arXiv preprint arXiv:2409.05746* (2024). https://arxiv.org/abs/2409.05746v1

[3] McKinsey & Company. 2024. Capturing the full value of generative AI in banking. (2024). https://www.mckinsey.com/industries/financial-services/our-insights/capturing-the-full-value-of-generative-ai-in-banking

[4] Federal Reserve Bank of St. Louis. 2024. Federal Reserve Economic Data (FRED). https://fred.stlouisfed.org/

[5] Zafeirios Fountas, Martin A. Benfeghoul, Adnan Oomerjee, Fenia Christopoulou, Gerasimos Lampouras, Haitham Bou-Ammar, and Jun Wang. 2024. Human-like Episodic Memory for Infinite Context LLMs. *arXiv preprint arXiv:2407.09450* (2024). https://arxiv.org/abs/2407.09450

[6] Taicheng Guo, Xiuying Chen, Yaqi Wang, Ruidi Chang, Shichao Pei, Nitesh V. Chawla, Olaf Wiest, and Xiangliang Zhang. 2024. Large Language Model based Multi-Agents: A Survey of Progress and Challenges. *arXiv preprint arXiv:2402.01680* (2024). https://arxiv.org/abs/2402.01680

[7] Zijin Hong, Zheng Yuan, Hao Chen, Qinggang Zhang, Feiran Huang, and Xiao Huang. 2024. Knowledge-to-SQL: Enhancing SQL Generation with Data Expert LLM. *arXiv preprint arXiv:2402.11517* (2024). https://arxiv.org/abs/2402.11517

[8] Yiming Huang, Jianwen Luo, Yan Yu, Yitong Zhang, Fangyu Lei, Yifan Wei, Shizhu He, Lifu Huang, Xiao Liu, Jun Zhao, and Kang Liu. 2024. Enhancing Text-to-SQL Capabilities of Large Language Models with Knowledge Injection. *arXiv preprint arXiv:2409.15907* (2024). https://arxiv.org/abs/2409.15907

[9] Aditya Kalyanpur, Kailash Karthik Saravanakumar, Victor Barres, CJ McFate, Lori Moon, Nati Seifu, Maksim Eremeev, Jose Barrera, Abraham Bautista-Castillo, Eric Brown, and David Ferrucci. 2024. Multi-step Inference over Unstructured Data. *arXiv preprint arXiv:2406.17987* (2024). https://arxiv.org/abs/2406.17987

[10] Omar Khattab, Arnav Singhvi, Paridhi Maheshwari, Zhiyuan Zhang, Keshav Santhanam, Sri Vardhamanan, Saiful Haq, Ashutosh Sharma, Thomas T. Joshi, Hanna Moazam, Heather Miller, Matei Zaharia, and Christopher Potts. 2023. DSPy: Compiling Declarative Language Model Calls into Self-Improving Pipelines. *arXiv preprint arXiv:2310.03714* (2023). https://arxiv.org/abs/2310.03714

[11] Jill H. Larkin and Herbert A. Simon. 1987. Why a Diagram is (Sometimes) Worth Ten Thousand Words. *Cognitive Science* 11, 1 (1987), 65–100. https://onlinelibrary.wiley.com/doi/pdf/10.1111/j.1551-6708.1987.tb00863.x

[12] Erran Li, Nitish Joshi, and Kumar Chellapilla. 2023. Improve Multi-Hop Reasoning in LLMs by Learning from Rich Human Feedback. *AWS Machine Learning Blog* (2023). https://aws.amazon.com/blogs/machine-learning/improve-multi-hop-reasoning-in-llms-by-learning-from-rich-human-feedback/

[13] Nelson F. Liu, Kevin Lin, John Hewitt, Ashwin Paranjape, Michele Bevilacqua, Fabio Petroni, and Percy Liang. 2023. Lost in the Middle: How Language Models Use Long Contexts. *arXiv preprint arXiv:2307.03172* (2023). https://arxiv.org/abs/2307.03172

[14] Li Ma, Huihong Yang, Jianxiong Xu, Zexian Yang, Qidi Lao, and Dong Yuan. 2022. Code Analysis with Static Application Security Testing for Python Program. *Journal of Signal Processing Systems* 94 (2022), 1169–1182. https://link.springer.com/article/10.1007/s11265-022-01740-z

[15] Robert C. Martin. 2003. The Single Responsibility Principle. https://solidprinciples.org/docs/single-responsibility-principle/detailed-explanation

[16] Shervin Minaee, Tomas Mikolov, Narjes Nikzad, Meysam Chenaghlu, Richard Socher, Xavier Amatriain, and Jianfeng Gao. 2024. Large Language Models: A Survey. *arXiv preprint arXiv:2402.06196* (2024). https://arxiv.org/abs/2402.06196

[17] Humza Naveed, Asad Ullah Khan, Shi Qiu, Muhammad Saqib, Saeed Anwar, Muhammad Usman, Naveed Akhtar, Nick Barnes, and Ajmal Mian. 2024. A Comprehensive Overview of Large Language Models. *arXiv preprint arXiv:2307.06435* (2024). https://arxiv.org/abs/2307.06435

[18] Yuqi Nie, Yaxuan Kong, Xiaowen Dong, John M. Mulvey, H. Vincent Poor, Qingsong Wen, and Stefan Zohren. 2024. A Survey of Large Language Models for Financial Applications: Progress, Prospects and Challenges. *arXiv preprint arXiv:2406.11903* (2024). https://arxiv.org/abs/2406.11903

[19] Ana Nunez, Nafis Tanveer Islam, Sumit Kumar Jha, and Peyman Najafirad. 2024. AutoSafeCoder: A Multi-Agent Framework for Securing LLM Code Generation through Static Analysis and Fuzz Testing. *arXiv preprint arXiv:2409.10737* (2024). https://arxiv.org/abs/2409.10737

[20] Department of Energy. 2024. Department of Energy Generative Artificial Intelligence Reference Guide. (2024). https://www.energy.gov/cio/department-energy-generative-artificial-intelligence-reference-guide

[21] OpenAI. 2024. Introducing Structured Outputs in the API. https://openai.com/index/introducing-structured-outputs-in-the-api/

[22] Kun Qian, Yisi Sang, Farima Fatahi Bayat, Anton Belyi, Xianqi Chu, Yash Govind, Samira Khorshidi, Rahul Khot, Katherine Luna, Azadeh Nikfarjam, Xiaoguang Qi, Fei Wu, Xianhan Zhang, and Yunyao Li. 2024. APE: Active Learning-based Tooling for Finding Informative Few-shot Examples for LLM-based Entity Matching. *arXiv preprint arXiv:2408.04637* (2024). https://arxiv.org/abs/2408.04637

[23] Jane Smith and Emily Johnson. 2023. Fine-Tuning Language Models for Context-Specific SQL Query Generation. *arXiv preprint arXiv:2312.02251* (2023). https://arxiv.org/abs/2312.02251

[24] John Smith, Emily Johnson, and Michael Brown. 2024. Memory Matters: The Need to Improve Long-Term Memory in LLM-Agents. *AAAI Conference on Artificial Intelligence* (2024). https://ojs.aaai.org/index.php/AAAI-SS/article/download/27688/27461/31739

[25] TIOBE Software. 2024. TIOBE Programming Community Index for October 2024. https://www.tiobe.com/tiobe-index/

[26] Siqiao Xue, Caigao Jiang, Wenhui Shi, Fangyin Cheng, Keting Chen, Hongjun Yang, Zhiping Zhang, Jianshan He, Hongyang Zhang, Ganglin Wei, Wang Zhao, Fan Zhou, Danrui Qi, Hong Yi, Shaodong Liu, and Faqiang Chen. 2023. DB-GPT: Empowering Database Interactions with Private Large Language Models. *arXiv preprint arXiv:2312.17449* (2023). https://arxiv.org/abs/2312.17449

[27] Yi Yang, Yixuan Tang, and Kar Yan Tam. 2023. InvestLM: A Large Language Model for Investment using Financial Domain Instruction Tuning. *arXiv preprint arXiv:2309.13064* (2023). https://arxiv.org/abs/2309.13064

[28] Yijiong Yu, Xiufa Ma, Jianwei Fang, Zhi Xu, Guangyao Su, Jiancheng Wang, Yongfeng Huang, Zhixiao Qi, Wei Wang, Weifeng Liu, Ran Chen, and Ji Pei. 2024. Hyper-multi-step: The Truth Behind Difficult Long-context Tasks. *arXiv preprint arXiv:2410.04422* (2024). https://arxiv.org/abs/2410.04422

[29] Wei Zhang and John Doe. 2024. Evaluating LLMs for Text-to-SQL Generation With Complex SQL Workload. *arXiv preprint arXiv:2407.19517* (2024). https://arxiv.org/abs/2407.19517

[30] Zeyu Zhang, Xiaohe Bo, Chen Ma, Rui Li, Xu Chen, Quanyu Dai, Jieming Zhu, Zhenhua Dong, and Ji-Rong Wen. 2024. A Survey on the Memory Mechanism of Large Language Model based Agents. *arXiv preprint arXiv:2404.13501v1* (2024). https://arxiv.org/abs/2404.13501v1

[31] Siyun Zhao, Yuqing Yang, Zilong Wang, Zhiyuan He, Luna K. Qiu, and Lili Qiu. 2024. Retrieval Augmented Generation (RAG) and Beyond: A Comprehensive Survey on How to Make your LLMs use External Data More Wisely. *arXiv*

	index	table	description
0	CPIAUCSL	CPI	CPI Urban - All Items
1	CPILFESL	CPI	CPI Urban - All Items Less Food & Energy
2	UMCSENT	CPI	Consumer Sentiment
3	GDP	GDP	GDP
4	GDPC1	GDP	Real GDP
5	GDPDEF	GDP	GDP Implicit Price Deflator
6	GDPPOT	GDP	Real Potential GDP
7	DSPI	Income	Disposable personal income
8	DSPIC96	Income	Real Disposable Personal Income
9	PCE	Income	Personal Consumption Expenditures
10	PCEDG	Income	Personal Consumption Expenditures: Durable Goods
11	PSAVERT	Income	Personal Saving Rate
12	RRSFS	Income	Real Retail and Food Services Sales
13	IR14270	Money	Nonmonetary Gold Price
14	WM1NS	Money	M1 Money Stock
15	WM2NS	Money	M2 Money Stock
16	DFF	Rates	Effective Federal Funds Rate
17	DGS10	Rates	10-Year Treasury Rate
18	DGS30	Rates	30-Year Treasury Rate
19	DGS5	Rates	5-Year Treasury Rate
20	DTB3	Rates	3-Month Treasury Rate
21	T10YIE	Rates	10-year Breakeven Inflation Rate
22	T5YIE	Rates	5-year Breakeven Inflation Rate
23	T5YIFR	Rates	5-Year, 5-Year Forward Inflation Expectation Rate
24	DJIA	Stock	Dow Jones Industrial Average
25	NASDAQCOM	Stock	NASDAQ Composite Index
26	SP500	Stock	S&P 500
27	CIVPART	Unemployment	Civilian Labor Force Participation Rate
28	EMRATIO	Unemployment	Civilian Employment-Population Ratio
29	UNEMPLOY	Unemployment	Unemployed
30	UNRATE	Unemployment	Civilian Unemployment Rate

Figure 4: FRED Datasets

Use below information to calculate recession. Combine its output to answer the original question:

0. Determine the start of period using today's date information provided. Say it's <period_start>. If no period information is provided by the user then use the entire timeseries.
1. Retrieve Real GDP data for the specified period from the 'gdp' table using the following SQL query:

```
SELECT date, value FROM gdp WHERE gdp_index = 'Real GDP' AND date >=
<period_start> ORDER BY date
```

2. Load the data into a DataFrame with columns ['date', 'real_gdp'].
3. Sort the data by date and convert the 'real_gdp' column to numeric.
4. Calculate the change in GDP between periods using the `diff()` method.
5. Calculate cumulative periods of GDP decline using the following logic:

```
df_gdp['cumulative_decline'] = (df_gdp['gdp_change'] <
0).astype(int).groupby(df_gdp['gdp_change'].ge(0).cumsum()).cumsum()
```

6. Identify when a recession starts (two consecutive periods of GDP decline) using the following logic:

```
df_gdp['recession_start'] = (df_gdp['cumulative_decline'] >= 2) &
(df_gdp['cumulative_decline'].shift(1) < 2)
```

7. Identify when a recession ends (the period before GDP growth resumes) using the following logic:

```
df_gdp['recession_end'] = (df_gdp['cumulative_decline'] >= 2) &
(df_gdp['cumulative_decline'].shift(-1) == 0)
```

8. Filter the data based on <period_start>.
9. Select relevant columns for the final table: ['date', 'real_gdp', 'cumulative_decline', 'recession_start', 'recession_end'].
10. NOTE: THIS IS NOT THE FINAL ANSWER. USE THIS RECESSION INFORMATION TO ANSWER THE QUESTION GIVEN BELOW.

Figure 5: Example of Memory for Recession Calculations

preprint arXiv:2409.14924 (2024). https://arxiv.org/abs/2409.14924

[32] Kunhao Zheng, Juliette Decugis, Jonas Gehring, Taco Cohen, Benjamin Negrevergne, and Gabriel Synnaeve. 2024. What Makes Large Language Models Reason in (Multi-Turn) Code Generation? *arXiv preprint arXiv:2410.08105* (2024). https://arxiv.org/abs/2410.08105

[33] Wanjun Zhong, Lianghong Guo, Qiqi Gao, He Ye, and Yanlin Wang. 2023. MemoryBank: Enhancing Large Language Models with Long-Term Memory. *arXiv preprint arXiv:2305.10250* (2023). https://arxiv.org/abs/2305.10250

A Dataset

A.1 SQL Database

We extracted the macroeconomic time-series from FRED and added them to corresponding tables as indicated in Figure 4 [4]. We have a mix of both long and wide format tables, to make the performance evaluation more challenging.

A.2 Performance Evaluation Dataset

Below we list a sample of 15 questions (out of 30 total) used during performance evaluation. These questions require multi-step analysis before coming to the correct answer and are inspired by practitioners use of such macroeconomic time-series.

- When did the latest recession start and how long did it last?
- What was the unemployment rate at the start of latest recession?
- How many recessions happened in the last 30 years?
- In the last 20 years, what percentage of time did real GDP beat real potential GDP?
- How many times did the GDP cross 20,000 billion?
- Using 6m moving average, how many times did unemployment cross 8%?
- What's the average unemployment for the last 10 years excluding periods of recession? Also exclude 2 quarters before recession start and 2 quarters after recession end, before your analysis.
- Excluding all periods of recessions, is current unemployment higher or lower than last 10 years average?
- In what year did CPI value cross 200?
- In the last 20 years, when was inflation the highest?
- What's the correlation between inflation calculated from CPI with 5 year break-even inflation rate, in the last 10 years?
- In the last 10 years, how many times did the CPI value kept dipping before increasing?
- What is the inflation contribution because of Food and Energy in March 2022?
- What was the latest date when the yield curve got inverted?
- What's the correlation between federal funds rate and unemployment?

B Memory Example

As highlighted in section 2.3, the memorization process summarizes the history of interaction and saves it in a generalized fashion. Figure 5 shows how memory related to recession calculation looks like. This memory is injected into questions requiring any sort of recession calculation, to provide the LLM agent with a recipe for accurately calculation of recession metrics.

Its important to highlight that memory is only being used to help with part of a question and it needs to be used in conjunction with additional analysis to come to the final answer. For example from our performance dataset, recession memory is being used for questions like:

- When did the latest recession start and how long did it last?
- What was the unemployment rate at the start of latest recession?
- How many recessions happened in the last 30 years?
- What's the average unemployment for the last 10 years excluding periods of recession? Also exclude 2 quarters before recession start and 2 quarters after recession end, before your analysis.

For a memory to be pulled into LLM agent's chat context, the LLM agent has to make a decision on what calculations needs to be performed. If it determined "Recession Calculation" as a required piece, it does a semantic search on memory vector database and pulls the matching vector results into the LLM agent's chat context.

Even though we could have asked our LLM agent to create memory for all asked question, we looked at a realistic scenario where memory is created for general processes and it is used to answer questions requiring additional analytics. Thus, we have used only this singular piece of "recession calculation" memory which is used by 9 out of 30 questions. The remaining 21 questions should have identical performance between without memory and with memory. This demonstrates a realistic use-case where the LLM agent becomes smart not on exact tasks, but on any task having overlap with processes already saved in memory.

B.1 Human Messages Used to Create Memory Example

As highlighted before, the memory creation process involves a human expert interacting with the LLM agent. The human expert asks simpler questions and gets a feedback response from the LLM agent as its response to the simpler question. The expert can correct the agent generated output by providing more context, therefore reducing ambiguity in their original statement. The expert can keep using outputs of previous steps to build a more complex processes. Once the expert is satisfied with the outputs of their interaction, they can ask the LLM agent to remember the process. Doing so results in the LLM agent summarizing the entire history of their current interaction and creates a memory record inside the vector database.

The following human messages were provided to the LLM agent to generate recession calculation memory as described above.

- A recession is defined as having two consecutive periods of real GDP decline. Create a table with time-series of real GDP and column indicating the total number of periods of continuous GDP decline from year 2000.
- Continuous decline should go down to 0 when GDP increases.
- Now using the information in continuous decline column, determine when recession started. It happened when when continuous decline becomes equal to 2.
- Also determine when recession ended. Recession ends when continuous decline goes from >=2 to 0. It's the period before the continuous decline value goes to 0.
- Indicate both recession started and recession ended as boolean against date within the table.
- Based on the columns created tell me when the recession started, when it ended and how long was the recession for in months.
- Now remember this process for me against "Recession Calculation".

We can see that during memorization the specifics (like year 2000) were generalized before saving into memory. Memorization also included chunks of SQL and python code to remove ambiguity, which as we see from results of section 3.3 is quite important.

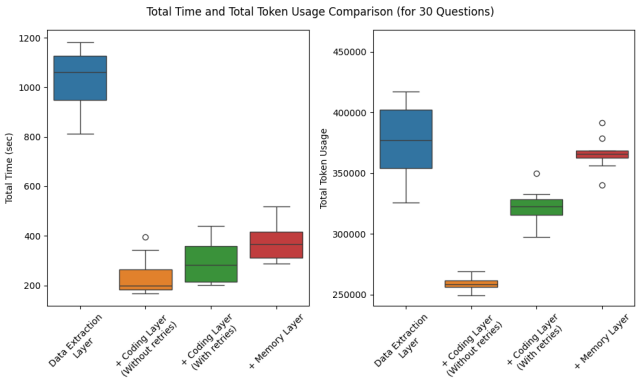

Figure 6: Comparison of Total Time and Total Token Usage (for 30 questions)

C Latency and Token Usage

From Figure 6 we can make the following observations:

- The latency/run-time and total token usage with just *Data Extraction Layer* is high, because we have to send back the extracted dataset for analysis by the LLM. Which is not the case for others as any analysis is done locally in code. It's also worth noting that with just *Data Extraction Layer* there were 2 questions for which tokens sent in request was higher than limit supported token context gpt-4o model, which result in failure. These cases were not included in our token usage plot.
- Our token usage is already quite high in the base case (of *+ Coding Layer (Without Retries)*. This is partly because of including SQL database schema information in memory.