Understanding Agentic AI Systems for Data Analysis: Techniques, Conceptual Framework, and Considerations

Simon C. Wang
University of Maryland, College Park

Abstract

Large language models (LLMs) and agentic AI systems have demonstrated powerful capabilities in automating data analysis tasks—from planning to code generation. However, these systems often use a wide range of AI techniques, from zero-shot prompting to multi-agent orchestration, leading to confusion around the operational definition of 'agentic' in practice. In this paper, we survey representative agentic data analysis systems and distill a simplified conceptual framework for understanding their key components. We introduce core techniques (e.g., in-context learning, tool-use, planning), map them to practical systems, and provide guidance for choosing the right approaches based on task complexity and system requirements. We also present a case study on event sequence analysis using agents, illustrating practical tradeoffs and design challenges.

1 Introduction

In the domain of data science, there have been many works exploring the application of LLMs in various capacities, across a diverse set of use cases. Recent systems labeled as 'agentic' vary widely in complexity, from single LLM prompts to advanced multi-agent frameworks using planning, tool-use, and dynamic orchestration. With all of this complexity, it becomes very challenging and unclear how to navigate or compare these systems as a whole, much less their individual components and workflows. In this work, we aim to survey some of the most popular approaches and papers in the space, extract the underlying techniques and components, organize them into a high-level guiding framework for agentic data analysis systems, and provide preliminary guidance and considerations for applying LLM-based approaches to data analysis in general.

2 Basic Techniques and Concepts of Agentic Systems

In this section, we will provide a brief overview and definition of some of the basic concepts that have led to and now make up the components of more complex agentic systems. Starting from basic LLMs, to strategies that improve LLM performance on specific tasks, then discussing the definition of an “agent” in the modern context as well as some of the innovations that enable agentic workflows.

In the context of these modern agentic systems, the key component is the **LLM**. First developed based on the transformer architecture and attention mechanism in 2017 (Attention Is All You Need), the demonstrated versatility of generative AI to perform a wide array of tasks and seemingly gain deep understanding through its training data has led to an explosion of generative AI applied to every domain possible. In particular, OpenAI’s GPT models (Improving Language Understanding by Generative Pre-Training) have created an almost cultural shift in the incorporation of generative AI into society, especially with the widespread popularity of the ChatGPT product.

2.1 Basic LLM Strategies

An LLM on its own is in essence a next-token predictor, and while it can demonstrate capabilities out of the box with enough training data, many real-world applications need more task-specific performance. To improve this specialized performance of LLMs, there have been many innovations and techniques proposed.

The simplest approach is to explore different ways of modifying the prompt, which is the input we provide the LLM. The most basic approach is **zero-shot prompting**, where we simply tell the model what we want in natural language. Next, there have been several areas of research in improving model performance through the prompt. **Prompt engineering** refers to a set of best practices to prompting through empirical observation, including specific word and phrasing choice, tone, and more (A Prompt Pattern Catalog to Enhance Prompt Engineering with ChatGPT). **In-context learning** at its basis refers to providing one or more examples to the LLM within the prompt, demonstrating the desired behavior ([2005.14165] Language Models are Few-Shot Learners). This is also sometimes referred to as “few-shot” or “many-shot” learning, referencing the number of examples provided as opposed to “zero-shot” where none are provided.

To improve model knowledge of specific domain material or large documents, the technique of **retrieval augmented generation (RAG)** was introduced ([2005.11401] Retrieval-Augmented Generation for Knowledge-Intensive NLP Tasks). This approach converts a corpus of documents into a vector database, then provides the model with important context based on the similarity with the task described in the prompt. In particular, this is especially effective when the task requires specific domain knowledge that might not have been represented in the training data of the model.

Finally, in general the most expensive method to improve LLM performance is **fine-tuning**. This involves providing the model with a dataset of hundreds or thousands of examples defining desired behavior, then training the model on these examples, modifying some or all of the model weights ([1810.04805] BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding). While this method is more time-consuming and expensive, this can provide more robust and consistent model behavior due to the actual modification of model weights. Fine-tuned models can be saved and distributed, which has led to the creation of many open-source task-specialized models that can perform well with minimal computational resources.

2.2 Techniques Enabling Agentic Workflows

More recently, the concept of AI agents and agentic systems has been applied to almost every task domain. However, the definition of what an agent is in the context of LLM-centric AI and what makes an agentic system still has much uncertainty. There have been many innovations and developments that have progressed the field of generative AI from simple prompting based approaches, to providing LLMs with more control to act on their own.

In this work, we define an **agent** as a modular LLM-based entity that has the ability to autonomously make decisions about its next actions based on goals, intermediate context, and access to available tools or functions.

Transitioning from prompt improvement and model fine-tuning, researchers discovered that approaches to encourage LLMs to **reason via prompting** could greatly improve performance. One of the earliest works found that using in-context examples to encourage a model to think step-by-step in a **chain of thought** significantly improves benchmark performance on reasoning tasks ([2201.11903] Chain-of-Thought Prompting Elicits Reasoning in Large Language Models).

Next, an integral component to any agent or agentic system is **tool-use.** This simply refers to the approach of providing LLMs with access to non-generative tools to accomplish tasks or parts of tasks. For example, a common approach was proposed in [2205.12255] TALM: Tool Augmented Language Models and [2302.04761] Toolformer: Language Models Can Teach Themselves to Use Tools where the LLM can make decisions about what tool to use, then access the tool through API calls.

Finally, a major innovation that improved agents was the introduction of **self-reflection and monitoring**. [2210.03629] ReAct: Synergizing Reasoning and Acting in Language Models first explores having models generate reasoning traces during task execution, improving its own reasoning approach to tasks and decision-making by thinking more about why it performs actions. In [2303.11366] Reflexion: Language Agents with Verbal Reinforcement Learning, LLM-based agents were asked to reflect on task feedback signals, using this information to improve decision-making in subsequent tasks. These approaches were shown to provide significant improvements, especially on tasks like complex reasoning and coding.

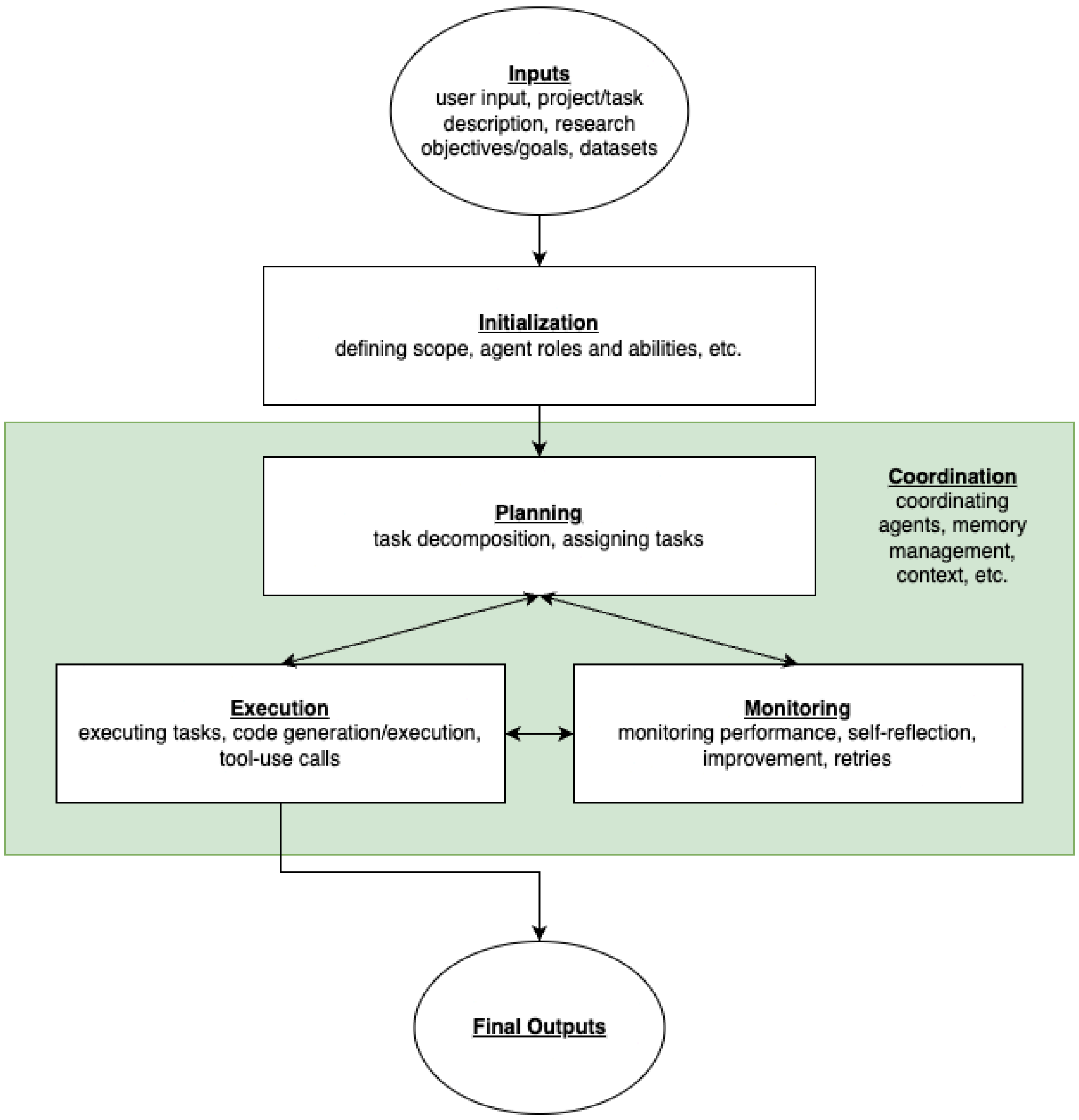
Modern agentic systems are typically **multi-agent** which introduces multiple LLM-based agents with diverse interactions and management strategies. This often requires a combination of initialization, planning, coordination, execution, and monitoring. In general, multi-agent workflows seek to use human work organization as a model, showing promise in performance on complex tasks through task decomposition, collaboration, and refinement.

3 A Conceptual Framework for Agentic Systems

To organize and simplify the complicated space of agentic systems, we propose a high-level framework inspired by project management to map the layers and components of most approaches. Note that this is a conceptual model designed to abstract common system components, **distinct from software implementation frameworks** like AutoGen or LangChain. In defining the components of the conceptual agentic system framework, we draw inspiration from The Project Management Lifecycle which describes the phases of a project as a cycle. This workflow consists of a cyclical sequence of: initialization, planning, execution, and monitoring.

While this framework is applied to human workers, intuitively, the agents in our LLM-based systems map well to it due to the goal of many agentic systems which is to mimic the collaborative workflow of actual humans.

As such, we propose the following agentic system conceptual framework with components closely aligning with the stages of the project management cycle:


Inputs
user input, project/task description, research objectives/goals, datasets
Initialization
defining scope, agent roles and abilities, etc.
Planning
task decomposition, assigning tasks
Coordination
coordinating agents, memory management, context, etc.
Execution
executing tasks, code generation/execution, tool-use calls
Monitoring
monitoring performance, self-reflection, improvement, retries
Final Outputs

This conceptual framework has five components, with the addition of coordination in contrast to the original project management framework.

The **initialization** component refers to the phase in which the system or user defines the scope of the task, defines the roles of the agents, and provides any important input like datasets or research contexts.

The **planning** component handles the decomposition of tasks, subsequent assigning of tasks, and any other optimizations to create a plan of execution.

For many agentic systems, the **coordination** component is integral to managing the one or more agents and their functionalities. To manage this, features such as memory management, shared contexts, API handling, and more are necessary. Overall, this component allows for the entire system of many complex parts to communicate within itself, and make sure that intermediate and final inputs and outputs are handled correctly.

Execution is where the actual work of completing tasks is done, and typically involves simple LLM calls, code generation and execution, tool-use, and other operations necessary for agents to complete their assigned tasks.

In the **monitoring** component, many systems employ a variety of strategies to evaluate performance, perform iterative improvements, and potentially retry tasks.

It is important to note that while there is heavy inspiration from the project management lifecycle to organize these components, unlike in the original lifecycle framework, agentic systems are often non-linear and interleaved (as opposed to the more defined sequential nature of the project lifecycle work). In particular, the components of planning, execution, and monitoring can be interleaved or concurrent.

For instance, the execution of a task could trigger dynamic monitoring, causing re-planning, which re-defines the tasks for agents and re-starts the flow of execution. Furthermore, throughout the entire process of planning, execution, and monitoring, the coordination component must continuously manage various aspects of the system to make sure agents can function correctly. Thus, this conceptual framework **primarily aims to organize components conceptually**, and does not imply a deterministic sequential execution order or strict workflow.

4 Survey

In this section, we analyze four representative systems, each of which applies agentic workflows to data analysis in distinct ways. For each system, we will first summarize the basic design and goal of the system. Then, we will map its components to the conceptual framework defined in the previous section, describing in detail for each component: techniques used to implement the component, why these techniques were chosen, and trade-offs or limitations to consider for the technique(s) chosen.

4.1 Data Interpreter - Data Science

Source: Data Interpreter: An LLM Agent for Data Science

4.1.1 Summary

Data Interpreter is an agentic system that enables LLMs to perform end-to-end data science workflows, including data preprocessing, feature engineering, model training, and evaluation. The system is built around two key modules:

Hierarchical Graph Modeling: LLMs dynamically decompose a high-level data science goal into a directed acyclic task graph, where nodes represent subtasks and edges represent dependencies.

Programmable Node Generation: Each subtask is further broken down into executable actions, allowing for iterative code generation, tool selection, execution, and verification.

Data Interpreter emphasizes dynamic adaptability: the system reflects on execution results, updates and restructures its task graph based on runtime feedback, and continuously optimizes its approach. Experimental results show improvements over previous baselines such as InfiAgent-DABench and MATH.

4.1.2 Conceptual Framework Mapping

Initialization. During initialization, Data Interpreter simply asks the user to enter a project requirement in the form of a task-oriented input (e.g. “analyze dataset and build predictive model”) along with a dataset if relevant.

This approach was chosen to allow for maximum flexibility by not enforcing fixed agent roles or pre-defined workflows. Instead, we rely on the system and graph-based approach to adapt to the task, which allows the Data Interpreter system to support a wide range of data analysis tasks.

However, while the lack of role and workflow definition allows for flexibility, it increases the difficulty of debugging or modularizing behaviors to allow for interpretability and modifications. Furthermore, since the system almost entirely relies on the LLM’s capability to infer appropriate tasks and plan execution strategies from an open-ended prompt, there is an increased risk of hallucinations and inefficient or incorrect approaches to the tasks.

Planning. To facilitate the core planning component of Data Interpreter, the system employs a graph-based approach. Using **in-context learning (ICL)** with many demonstration examples, LLMs are prompted to decompose the input project description into a hierarchical task graph in the form of a directed acyclic graph (DAG). Each node in the graph represents a task along with the needed attributes to execute it. Task nodes are further broken down into multiple actions, which results in an action graph with action nodes. This directed graph then becomes the plan of execution for the system.

ICL is chosen as the technique for this component in order to keep the system more adaptable, and also to improve efficiency over more expensive and resource-intensive methods like fine-tuning and reinforcement learning.

While ICL is a good method for improving task-specific performance of LLMs, its effectiveness can also heavily depend on prompt quality or quantity, on top of the LLM’s own reasoning abilities. Through their experimentation, the authors demonstrated good performance with their ICL-based approach, however it could be possible to further improve performance by investing more time into fine-tuning a model. The risk of relying on LLMs for decomposition could be redundant tasks, overlooked dependencies, or over-fragmentation causing the agents to lose the higher-level scope of the research goals.

Execution. For the execution component, Data Interpreter implements graph executors that traverse the task and action graphs to “execute” the graphs. In particular, action nodes represent executable (by the graph executor) code snippets that can contain data transformations, function calls, or other operations needed to complete the task. When executing a task node, Data Interpreter retrieves from a set of tools the most suitable tools for the task. An LLM is tasked with ranking the most suitable tools in reference to the metadata of the task node. After the most appropriate tools are chosen, the system then chooses a tool for the task from the top-k of this list.

This implementation of tool-use allows for good flexibility, with the system dynamically choosing the best tool for the task rather than specific agents having access to specific tools. This improves performance by allowing the LLMs to leverage specialized tools for more difficult tasks rather than depending on potentially unreliable LLM base knowledge.

While the ranking of tools provides adaptability, it is also potentially a significant tradeoff in reliability. The ranking of tools and therefore proper tool selection relies heavily on the quality of metadata descriptions, which are also generated via LLMs in other components. This can lead to volatility and propagation of errors, because bad descriptions can lead to suboptimal or incorrect tool usage, and therefore compromise the quality of responses in that task node. If one task node is erroneous, subsequent dependencies in the graph will also be negatively affected.

Monitoring. Data Interpreter leverages the graph-based workflow to automatically improve execution strategy through iterative graph refinement. During execution, nodes are monitored by LLMs tasked with reflection, updating node statuses to identify failures and provide feedback. Failed or suboptimal tasks trigger mechanisms that can reconstruct branches of the graph based on dependencies, aiming to improve strategy based on the runtime feedback.

This type of self-reflection allows an agentic system to be more robust in failure handling, thinking about why failures occurred and attempting to improve its own performance without manual intervention. Furthermore, the graph-based approach allows the system to only refine the necessary parts of the graph without redoing every action in the plan, which improves efficiency.

While this self-reflection can improve performance iteratively, the reflection and retry mechanism still adds overall computational overhead and latency. Despite the beneficial property of partial graph restructuring, there is still the potential of a failure that causes large portions or even the entirety of a complex graph to be restructured, which would be very expensive both in computation and time. Furthermore, with automatic reflection and refinement, there is no guarantee that the improvements will actually converge to better performance.

Coordination. The management of agents is determined by the graphs created in the planning component. The nodes in these graphs contain important information on task status, execution feedback, and dependencies. To coordinate this complex system, information and context is passed along the directed edges of the graph, to make sure each agent receives the necessary information and resources to execute the task at each node.

Graph-based coordination was chosen to enable flexible task ordering and recovery by leveraging the dependencies and structure of the DAG. By traversing the directed edges, the system can maintain and coordinate a complex flow of information and task execution.

The main challenge with the graph-based approach is the maintenance of graph consistency, especially when the monitoring component could cause dynamic changes to its structure. Furthermore, especially for complex or large project descriptions, graphs can become very difficult to debug in the event of an error in the system.

4.2 Qualitative Data Analysis

Source: Can Large Language Models Serve as Data Analysts? A Multi-Agent Assisted Approach for Qualitative Data Analysis

4.2.1 Summary

This work proposes an LLM-based multi-agent model designed to automate various types of qualitative data analysis tasks, including thematic analysis, content analysis, narrative analysis, discourse analysis, and grounded theory. This implementation uses a more simple approach in contrast to Data Interpreter, deterministically specifying sequential workflows pre-defined for each use-case.

Each agent in the system is a specialized instance of an LLM, designed to perform a distinct task. The system processes diverse data inputs such as discussion links, uploaded documents, text prompts, and interview transcripts, and outputs results in multiple formats like CSVs or documents. For each qualitative analysis method, different numbers and types of agents are deployed in a fixed workflow to collaboratively process the input data.

The results demonstrate that the system significantly accelerates the data analysis process while improving scalability and consistency, with practitioner evaluations reporting 87% satisfaction with the system's performance.

4.2.2 Conceptual Framework Mapping

Initialization. The initialization component of this system is completely manual and controlled by the user of the system. Users provide input data or documents, and also text prompts specifying their desired goals. The user is also asked to manually select the qualitative analysis type of their task, which tells the system which predefined agent workflow to use.

This approach was chosen both for simplicity and for robust performance. By allowing users to specify the analysis type of their research goals, the system will always follow the corresponding agentic workflow. Furthermore, for the developers of the system, it becomes much easier to debug and make modifications to the system due to the deterministic nature of each workflow.

As a trade off, however, this approach greatly limits the flexibility and adaptability of the system. For tasks that are more complex, the ideal approach could actually be a combination of analysis types. Additionally, the manual specification of analysis type depends on the user correctly configuring and identifying their task type, which could be unreliable.

Planning. As mentioned before, this agentic qualitative analysis system relies on pre-defined agent workflows. In particular, there are five types of qualitative analysis included in this work: thematic analysis, content analysis, narrative analysis, discourse analysis, and grounded theory. For each type of analysis, there is a corresponding static workflow chosen by the researchers. For example, the narrative analysis workflow consists of four agents in sequential order: a summary agent to summarize the story, a coding agent to generate initial codes for the task, then two agents that generate subcategories and categories.

Again, this technique of pre-defined workflows is chosen to guarantee consistency. With these clearly mapped workflows to analysis tasks, researchers can be confident that the task will be completed through a known series of analysis steps tailored towards the specified type.

While more consistent, this approach also introduces rigidity, preventing the system from dynamically adjusting its workflows or adapting to tasks that aren’t clearly restricted to a single analysis type. This means that this system is only suited for tasks in which the analysis goal is already very clear, which isn’t always the case for some data analysis contexts where the task is more exploratory.

Execution. Each agent in these workflows is an instance of an LLM, and through a simple system prompt each LLM is assigned its role. The specific prompts are not shared in the paper, but a standard system prompt would be something like “You are tasked with generating initial codes from a summarized text.” for the coder agent. Then, the execution of tasks relies simply on the inherent capabilities of each LLM, without any additional training or tool-use.

This approach of using system-prompting to assign LLMs roles was chosen because it is the most straightforward and intuitive. While the approach is basic, many works have shown that simply telling LLMs their roles can have surprisingly good results, with minimal complexity in programming or training.

However, although system prompts can elicit promising behaviors from LLM agents, this approach is still less effective for more complex tasks. It is entirely reliant on LLM abilities, which can be very inconsistent and unreliable especially when specific domain knowledge is required. Without tool-use or additional task-specific improvement techniques like RAG or in-context learning, LLMs can be very susceptible to hallucinations and misunderstandings of tasks, leading to significantly degraded result quality.

Monitoring. This work does not implement any monitoring or reflection techniques. Users simply manually review the final outputs after the workflow is complete.

Foregoing a monitoring component again simplifies the system, and also reduces computational cost as well as latency incurred by the additional mechanisms to monitor and potentially retry portions of the workflow.

While this approach definitely saves time and costs, it sacrifices the fault tolerance of the system. Without automatic failure-detection, mistakes made by the system will have to be manually inspected by the user, and in the event of an error the whole workflow would have to be manually rerun which would incur additional costs anyways. Furthermore, this approach misses out on the self-improving nature of other systems, which can lead to higher quality results with deeper insights or understanding.

Coordination. In terms of coordination, this system does not need much additional implementation. All workflows are simple sequential handoffs, which just require the management of the previous agent’s output becoming the input to the next agent in the sequence. In this way, agents can coordinate simply through the context of these prompts that are propagated through the sequential flow. While not discussed in-depth in this paper, it is implied that the multiple agents do share access to the input documents or datasets, but there is no mention of a shared data structure or memory enabling them to collaborate on or modify any shared context.

This simplicity keeps the coordination of the system very straightforward, and minimizes the complexity of handling extensive communication between agents.

Again, the rationale for choosing this approach is also one of the main downsides of the system. By reducing communication complexity, the benefits of multi-agent collaboration which have been shown in other works is sacrificed. There is no negotiation or feedback loops allowing agents to work together and be creative in their problem-solving. Furthermore, the strictly sequential nature causes errors made at any point in the pipeline to be propagated downstream without opportunity for correction.

4.3 DB-GPT - Data Interaction

Source: [Demonstration of DB-GPT: Next Generation Data Interaction System Empowered by Large Language Models | Papers With Code]() (demo)

[[2312.17449] DB-GPT: Empowering Database Interactions with Private Large Language Models]() (paper)

4.3.1 Summary

DB-GPT is an open-source, production-ready system that enables natural language-driven database interaction and generative data analysis. It introduces a multi-agent framework, retrieval-augmented generation (RAG) pipelines, fine-tuning for Text-to-SQL, and strong privacy protection through local deployment capabilities. DB-GPT supports building complex workflows through its custom-designed Agentic Workflow Expression Language (AWEL), which leverages DAG-based orchestration inspired by Apache Airflow.

In addition to traditional tasks like Text-to-SQL and QA over databases, DB-GPT is capable of advanced applications such as autonomous generative data analytics. It supports both local private LLM deployment and cloud-based usage.

4.3.2 Conceptual Framework Mapping

Initialization. The DB-GPT system notably uses retrieval-augmented generation (RAG) to create a knowledge base of relevant information for the data analysis task. This RAG pipeline approach first encodes documents into a vector database format, retrieves relevant knowledge using cosine similarity, then finally injects the top K retrieval results into the context of the LLM, improving task-specific performance through in-context learning. For more complex operations such as Text-to-SQL, they choose to fine-tune open source models on task-specific training sets (like Spider which includes inputs of dataset descriptions and natural questions, paired with outputs of the expected SQL). Lastly, to initialize the system, there are several predefined agent roles such as Data Analyst, Software Engineer, and Database Architect. Each of these agent roles is defined with a specific system prompt describing their desired behavior, along with access to different tools and plugins depending on their task. Additionally, if desired, users can define custom agent roles as well.

By employing a combination of agentic strategies to different parts of the system, this work does well in choosing techniques that improve each unique aspect of the system as much as possible. The RAG pipeline approach enhances LLM knowledge beyond its original training data, improving performance and relevance to the specific context of the data analysis task being approached. By using the combination of knowledge retrieval and in-context learning, the system can improve the task-specific performance of the LLMs without the cost of a full fine-tuning. In contrast, they identified through experimentation and previous works that in-context learning was insufficient especially for more complex tasks like Text-to-SQL. In this case, fine-tuning on a large dataset of examples ensures much higher robustness in behavior and performance. Finally, the flexibility to use predefined agent roles or define custom ones achieves the best of both worlds in terms of ease of task assignment and tailored performance.

In terms of trade-offs, each technique has its individual drawbacks. RAG-based approaches naturally rely on the quality of the retrieval approach, and if irrelevant context is retrieved the downstream task accuracy could be significantly impacted. Fine-tuning is a resource-intensive approach, and while providing better adherence to the training data use, could make the model less adaptable to situations not covered by the fine-tuning dataset examples.

Planning. DB-GPT supports both automatic and manual task decomposition. For natural language queries, a specialized planning agent (role defined through system prompt) is used to generate a plan by decomposing the task into subtasks and assigning them to specialized agents. Alternatively, developers can manually define multi-agent workflows using AWEL (Agentic Workflow Expression Language), a DAG-based declarative language inspired by Apache Airflow. This dual-mode planning provides flexibility: end users benefit from autonomous planning via LLMs, while developers can fine-tune or debug workflows with complete control.

The automatic task decomposition approach is much simpler, allowing more casual users with simpler tasks to describe their goal in natural language and have an LLM agent figure out a plan on its own. This also allows more dynamic adaptability in the system, where the planning agent can adapt its strategies to a diverse array of possible tasks. For more complicated tasks or for users desiring granular control over the workflow, AWEL generates a DAG where nodes are operators represented by an LLM agent. The developer can then design intricate workflows by specifying dependencies through edges, carefully controlling the flow of information in addition to custom agent configurations.

Each of these approaches has their own unique tradeoffs. The agent-based automatic planning makes task decomposition more simple for the user, but leaves the system vulnerable to LLM hallucinations or inefficient planning. If the planning agent makes an error in its strategy, this failure would propagate downstream to later agents without much ability for the user to mitigate. In contrast, the AWEL graph-based approach heavily depends on the proficiency of the developer that constructs the workflow. The low-level control is good for when the developer is very certain of how the task should best be approached, but also makes the performance much more sensitive to mistakes in factors like subtask granularity, multi-agent strategy, agent roles, and more. Furthermore, this systematic approach can make the system more rigid. While beneficial for very specialized and complex tasks, a carefully tailored system with intricate dependencies might not transfer well to many other types of tasks.

Execution. Agents in the DB-GPT system are associated with nodes, which contain details describing the task that is to be completed. These agents have access to various tools and capabilities as defined in the initialization stage, such as SQL generation, code execution, and more. As the workflow follows the dependencies in the graph of tasks, agents can leverage their tools to execute complex actions beyond the LLM's basic capabilities. For specific tasks like Text-to-SQL translation, the system will use specialized fine-tuned LLMs (e.g. Qwen) to complete the task.

Tool augmentation approaches like this greatly improve and extend LLM-based agent's reach to difficult tasks like real-world data interaction, visualization creation, and more. This effectively extends the abilities of the system past the inherent limitations of LLMs, which is especially important for a domain like data analysis in which specific knowledge and manipulation of data is crucial. Fine-tuned models, like mentioned before, typically outperform in-context-learning approaches especially on specialized tasks like SQL generation.

While tool-use improves model capabilities, the access to plugins and external APIs also introduce risk around security or tool misuse. For example, despite being given access to a robust tool specialized for generating visualizations, this approach still relies on the LLM to use it correctly (e.g. providing accurate values for a chart, choosing the appropriate type of chart). Furthermore, fine-tuned models could require careful re-training to stay aligned to evolving database formats, new research contexts, or simply changes in desired behavior.

Monitoring. DB-GPT does not support automatic reflection or feedback-based improvement mechanisms. However, it does support adaptive learning via user feedback post-execution. In addition to user interactions, this feedback can be used to adjust agent behavior, retrain or fine-tune models, and overall improve the system for future runs.

This approach was chosen to simplify runtime operations and separate the feedback-mechanism from the execution workflow. Additionally, depending on user feedback ensures that future modifications to the system are rooted in feedback that is closely aligned with what the user wants specifically.

However, implementing monitoring post-execution does sacrifice the benefits of real-time reflection. Most importantly, the lack of real-time error detection and improvements makes the system much more susceptible to task failures and hallucinations. Furthermore, the manual nature of this adaptive learning approach lacks consistency, meaning that the quality of user feedback and how it is used to modify the system can lead to drastically different performance for different users.

Coordination. To manage the coordination of multiple agents, the DB-GPT system archives the entire communication history of agents within a local storage system. Coordination of information and context flow between agents is implemented using AWEL, generating a directed acyclic graph (DAG) which specifies intricate relationships and dependencies through edges. This approach was inspired by big data processing concepts like Apache Airflow.

The DAG-based orchestration enables clearly-defined workflows, allowing users to directly control and create complex multi-agent collaboration. Furthermore, storing full conversation histories allows for improved debugging and interpretability, allowing users and researchers to manually inspect all agent communication.

While these approaches greatly improve the structure and controllability of the system, they also introduce complexity and memory overhead. For example, for large workflows, storing entire communications in local storage can take up very large amounts of memory space. Additionally, DAG models can also suffer from issues like looping, where if not carefully designed, a workflow can get stuck in an infinite cycle and cause many complications to the performance of the system and difficulty of debugging.

4.4 AutoGen - Agentic Framework

Source: AutoGen: Enabling Next-Gen LLM Applications via Multi-Agent Conversations

While the previous examples focused on specific agentic systems designed for particular data analysis tasks, there also exist more general-purpose frameworks aimed at building flexible, reusable agentic workflows across domains. These frameworks, such as AutoGen and LangChain, provide abstractions for initialization, planning, coordination, execution, and monitoring — aligning closely with the agentic system structure described earlier.

In this section, we highlight AutoGen as a representative example of a multi-agent framework that can be adapted for various data analysis workflows. An important note, in contrast to our conceptual framework, AutoGen provides a software framework for implementing agent-based workflows.

4.4.1 Summary

AutoGen is a general-purpose framework designed to enable the construction of multi-agent LLM systems through structured conversations between agents. Rather than building rigid pipelines, AutoGen provides flexible agent classes and dynamic orchestration mechanisms, allowing developers to define complex collaborative workflows where multiple LLMs (or tools) interact to solve tasks.

Agents in AutoGen can take on different roles, maintain memory, exchange information with each other, call external tools, and revise plans based on feedback — making it highly adaptable to data analysis tasks that require multi-step reasoning, code generation, iterative refinement, or complex coordination.

4.4.2 Framework Mapping

Initialization. In the initialization component, AutoGen uses explicit role definition, allowing developers and developers to define their own custom agent types (e.g. “PlannerAgent”, “CoderAgent”, “CriticAgent”) and specify their capabilities (e.g. tool access, system prompt, communication permissions). The framework also has various features to facilitate management of inputs of many formats, support for LLM APIs, and extensive documentation guiding developers on how to initialize their desired workflow using many popular workflow templates or lower-level customization for those that need more control.

This approach was chosen to maximize modularity and control for developers, so that roles can be easily defined or modified without rewriting the whole system. In this way, researchers and developers can quickly specify the workflow and roles they want, then easily make adjustments to experiment quickly with different approaches.

In terms of trade offs, the main drawback is that this approach requires more extensive and careful design upfront by the developer. Rather than a predefined system that the developer can immediately apply to their problem, more effort is needed to define agent roles and specify the system workflow through code. While the ease of agent definition is advantageous for experimentation and research, it also means that poorly designed roles could lead to weaker performance.

Planning. As described in the initialization section, all agents in AutoGen can be custom-defined by the developer. Therefore, any agent can be involved in the planning component. However, typical practice is to define a “PlannerAgent”, which is asked to decompose complex tasks into subtasks. This can be done dynamically through conversation with the developer, other agents, or simply sequentially.

This approach is chosen to maximize flexibility, allowing the developer to deploy agents that can plan and modify plans at any point in their agentic workflow. This can enable more creativity and collaboration in a dynamic way between agents, rather than rigid pipelines.

While this approach offers increased flexibility, the quality of task planning can also heavily depend on many factors such as initial prompting, model capabilities, and more. This can cause undesirable unpredictability, with agents potentially proposing redundant or even incorrect subtasks that don’t effectively achieve the overall goal. However, due to the flexibility of the AutoGen framework, this also means that developers can use any combination of techniques to improve model performance, like in-context learning, fine-tuning, RAG, or more capable models. To the more experienced developer, this can be a benefit as it removes the restrictions of the predefined workflows in other agentic systems.

Execution. Most AutoGen workflows leverage techniques like tool-use, LLM-based code generation, as well as traditional LLM prompting and reasoning ability. Agents can be given the ability to invoke external APIs for tools, generate and run Python code, or trigger other types of executions through executors (e.g. file-system interaction, database queries, real-world interactions). Agents can also reason about what tools or actions to use to complete their assigned task.

By equipping agents with tool-use and external access, AutoGen extends LLM capabilities to allow for greatly improved task-execution power and performance. Beyond the inherent knowledge and ability of the LLM, when tasked with a difficult operation the agent can choose to invoke a more suitable tool for the job.

With tool-augmented agents, there are several important risks to consider, including tool failures, compatibility issues, and security risks. Depending on the types of tools we give the agents access to, there can be significant complexity if the agents use the tools incorrectly, or potentially execute operations in an unexpected way that is harmful to other aspects of the system and workflow. For example, given access to a database and code execution tools, an agent could modify or delete parts of the database that are important for other parts of the task, causing the result to be incorrect and causing actual harmful change to our data.

Monitoring. To implement monitoring, the most common practice is to define specialized agents to assess task outcomes at various points in the workflow, identify issues, and suggest improvements. Once this monitoring system identifies a failed tasks, the agent can trigger re-execution with updated strategies after reflecting on the feedback of the failure.

These monitoring agents are an intuitive way to allow for dynamic error detection without human supervision, which can often save time and effort. Furthermore, if implemented well, the reflection can allow the system to self-improve and become more reliable over time, without having to retrain models or redefine agents.

While reflection allows the system to improve unsupervised, it can also add latency and cost to workflows. Additionally, reflection can lead to overcorrections if constraints are not carefully applied, which could cause the workflow to actually diverge in terms of performance and accuracy. In other words, if we completely leave reflection and re-execution up to agents that are not well-defined, they could potentially infinitely retry and modify their strategies, eventually costing excessive time and resources or completely losing the scope of the original task.

Coordination. Within the AutoGen framework, there are several classes implemented to handle any important conversational pieces of the system. For example, there are classes to handle messages of various types like AgentEvent, ChatMessage, ToolCallExecutionEvent, TextMessage, and many more. Together these features help manage agent communication and collaboration. There are also features to manage shared memory, chat histories, and persistent context.

The implementation of these classes helps give the developer a very granular understanding and control of the multi-agent coordination. With different types of messages and objects handling every type of agentic communication, the framework is capable of managing very complex workflows while effectively maintaining context and propagation of information.

Like before, the tradeoff of this granularity of implementation is also the risk of sub-optimal execution. If coordination isn't carefully controlled, conversations and multi-agent collaboration could result in inefficiencies like loops, loss of context, and overall complex errors to debug especially for more complicated workflows.

5 Survey Takeaways

As highlighted in the survey of representative systems, there are many tradeoffs and considerations to take into account when deciding what AI approach to use, or what technique to implement the components of a complex system. In this section I will summarize some guidance for approaching the application of LLM and agentic workflow techniques to a variety of use cases.

In the table below, each of the basic techniques mentioned before are described along with their ideal use case(s), along with some additional considerations. The surveyed systems are all combinations of many if not all of these techniques, so this summary of takeaways aims to outline why each technique was used, connect them to example use cases in the surveyed systems, and provide some tradeoff considerations that were mentioned throughout this paper. Overall, the hope is that by providing a clear high-level understanding of this diverse set of techniques, readers can have a better sense of how to apply LLMs to their own usage scenarios, or even develop their own agentic system.

Technique	Ideal Use Case(s)	Rationale	Trade Offs	Example System(s)
Zero-shot Prompting	Simple tasks or queries where minimal adaptation is needed	Fast, simple, cheap, requires no examples or retraining	High variability in outputs, less reliable for complex tasks	Most common general approach, any situation that relies only on base LLM capabilities
Prompt Engineering	Tasks needing more consistent or specific outputs without retraining	Improves LLM adherence to instructions through optimized phrasing	Can be labor-intensive to pick a good prompt, unreliable performance dependent on prompt quality	Implicitly used in all systems in the choice of system prompts, in-context approaches, etc.

In-Context Learning	Teaching LLMs new structured behaviors (e.g., task decomposition, SQL generation)	Adapts LLMs to a specific task or behavior without fine-tuning	Context window limitations (if there are too many examples), less robust than fine-tuning or retraining	Data Interpreter (task graph generation), DB-GPT (general prompting)
Retrieval-Augmented Generation	When external domain knowledge is crucial and not guaranteed in training data	Supplies specific context and knowledge to enhance response quality, especially when domain knowledge isn't included in model training	Critically dependent on retrieval quality, risk of retrieving irrelevant or noisy context that confuses the LLM	DB-GPT (knowledge base crafted from input documents, retrieval for each query)
Fine-Tuning	Specialized, high-structure tasks where consistency and precision are critical (e.g., Text-to-SQL)	Creates highly robust, domain-specific behavior	More expensive and time consuming, dependent on data quality, less adaptable after fine-tuning (to tasks outside of fine-tuning dataset)	DB-GPT (Qwen fine-tuned on Spider dataset for Text-to-SQL)
Reasoning via Prompting	Tasks requiring multi-step logical reasoning, planning, or debugging	Encourages intermediate reasoning steps for better correctness	Longer prompts and response latency, reasoning can still fail which leads to more wasted time	Data Interpreter (asking LLM to reason in terms of a task/action graph)
Tool-Use	Tasks where external operations (e.g., data cleaning, querying, modeling) are needed beyond	Extends LLM capability without retraining, improves reliability by using existing code or tools	LLM can still use the tool incorrectly, or choose the wrong tool for the task	Data Interpreter (nodes in the task graph have ranked relevant tools to access), AutoGen (can assign agents

	LLM generation			with tools through various integrations)
Multi-Agent Collaboration	Complex workflows requiring role specialization, iterative task handoff, or collaboration	Models human work paradigms, supports parallelism and task division, emergent benefits of collaboration	Coordination overhead can be high especially for complex systems with many agents, systems can be harder to manage especially when errors occur	Qualitative Analysis System (Summary, Coder, Pattern Extractor agents), DB-GPT (specifying multi-agent workflows in graph form)
Monitoring & Reflection	Tasks needing dynamic error detection, retries, and output improvement	Enhances reliability by allowing for automatic error correction, introduces self-improvement without human intervention	Increases latency in the system, retries might always be successful which can lead to further wasted work	Data Interpreter (monitoring and feedback triggered graph restructuring)

Across the surveyed systems, a few patterns emerge: in-context learning is frequently used for planning (e.g., Data Interpreter), while execution often combines tool-use and code generation (e.g., DB-GPT, AutoGen). Only some systems implement robust monitoring (e.g., Data Interpreter, AutoGen), while others defer to user review (e.g., Qualitative Analysis). This highlights a trend of escalating complexity aligned with task specificity and reliability needs.

6 Case Study

We now describe a real-world case study illustrating how our framework and survey-informed guidance can support system development for event sequence analysis. In a current project we are working on, we aim to apply LLM-based approaches to event sequence analysis. Through the course of developing a system for this project, we have experimented with various approaches and techniques. In this section, I will provide a walkthrough of our experimental process in choosing techniques, as well as considerations of trade offs and rationale for the choices and changes made.

To briefly provide context for the goal of this project, we aim to leverage LLMs to create a human-AI collaborative system specifically for event sequence analysis. Based on the following event sequence analysis framework [2408.04752] A Multi-Level Task Framework for Event Sequence Analysis consisting of a hierarchical multi-level organization of event sequence analysis approaches based on a survey of many event sequence analysis systems, this work will use LLM-based techniques to assist a data analyst across the entire process of developing an analysis strategy, then eventually executing the steps of that strategy to generate results and/or visualizations.

6.1 Experimentation

6.1.1 Zero-shot prompting

To serve as a baseline for our testing, we simply provided an input of just a sample research question from one of the case studies in the framework paper, and asked the LLM to extract an analysis plan. Without providing additional examples or knowledge, we ask the model to extract an analysis plan.

Results. While this was a very quick and easy way to begin our testing, as expected, the results of this approach were highly unreliable and did not adhere at all to the desired framework.

6.1.2 In-Context Learning and Prompt Engineering

Seeking to improve the task-specific performance of the LLM, we then employed a combination of in-context learning and prompt engineering. To give the model more guidance, we converted the sections of the event sequence framework paper that describe the main components of the framework, adding it to the context of the input prompt to the LLM. Additionally, we experiment with different prompting formats, asking it to return its answer following a specified JSON format (adding extra enforcement using structured outputs with Pydantic). Finally, to further improve adherence to desired behavior, we add several examples to the prompt context in the form of several case studies included in the original paper. These case studies included tables with pairs of research goals as input, and plans that map to the desired framework as expected outputs.

Results. These experiments incorporating in-context learning were done across several iterations, testing out different prompting approaches and strategies for specifying desired output format. Overall, a combination of including paper details with case studies, as well as a strict output format through JSON parsing, seemed to improve results significantly. In particular, the LLM more consistently adhered to the components of the framework, almost always providing complete plans without missing parts. However, testing this approach on some of the example inputs from case studies (making sure to remove that example from the prompt), revealed that the system was still not very reliable in producing results close to the expected output plans.

6.1.3 Multi-Agentic Workflows

At this point in the experimentation, we considered that the single-instance LLM might struggle with managing the complex multi-part tasks that are often present in data analysis. So, we decided to implement several multi-agent workflows using the AutoGen framework. Throughout this phase of testing, we used the highly customizable nature of AutoGen to define many different multi-agent workflows, aiming to decompose the complex data analysis tasks into smaller subtasks that individual agents can complete and collaborate on. This involved several sequential workflows, defining agent roles that extract smaller portions of the analysis plan before passing the context on to the next agent, which extracts another portion of the plan. We also experimented with agent group chats, where several agents with different roles (e.g. framework extractor, critic) including the real user through a user proxy, can complete a complex task through a round-robin conversation approach.

Results. Overall, this approach when combined with the in-context learning and prompting strategies from the previous section did provide some benefits in terms of output quality and consistency. However, the process of creating multi-agent workflows involved significant trial and error as well as complexity in implementation. Some of the attempts actually seemed to perform worse than single-LLM prompting, due to various factors like poorly designed agent roles, or hallucination that was propagated through the multiple agents of a sequential workflow. As mentioned before, multi-agent system performance can rely heavily on the quality of the agent design and coordination, and is vulnerable to complex failure conditions that can be quickly propagated throughout the workflow.

6.1.4 RAG

After the inconsistent success of the multi-agent approaches, we decided to take a step back and consider other potential strategies for improving LLM performance. While multi-agent systems can be great for dynamic tasks requiring complex task decomposition and flexibility, our work actually needs more robustness and controllability. Our original event sequence framework is strongly rooted in the details of over 100 event sequence analysis system papers (non-AI approaches). Therefore, we decided to try implementing a retrieval-augmented generation (RAG) approach by building a knowledge base from the text of all 100+ papers. To carry out this approach, we are experimenting with using GraphRAG [2404.16130] From Local to Global: A Graph RAG Approach to Query-Focused Summarization as well as Open WebUI RAG integration GitHub - open-webui/open-webui: User-friendly AI Interface (Supports Ollama, OpenAI API, ...).

Results. We are still currently in the process of curating our knowledge base of papers, but preliminary testing using GraphRAG with a smaller subset of papers shows significant promise. The goal of using this RAG approach is to ensure the LLM(s) has access to relevant knowledge context extracted directly from papers that are most applicable to the specific research task. By leveraging this corpus of papers, we hope to help the system generate plans that are deeply rooted in the context of the event sequence analysis framework, and provide users with guidance informed by trustworthy sources.

6.1.5 Future work: Fine-Tuning

Beyond experimenting with a RAG-based approach, we have a relatively large dataset of example case studies and expected objective extractions that could potentially be used for fine-tuning. These examples consist of inputs including case study details which describe a domain, dataset, and research goal, and outputs that specify the expected analysis objective (which is the highest level of the event sequence task framework) to be extracted. By leveraging these 100+ examples, we could attempt to fine-tune an open source model to follow this expected behavior.

Results. The goal of this fine-tuning would be to further enhance the reliability of the LLM in terms of adherence to our expected behavior and domain. After fine-tuning this model, it could also potentially be used as a more robust "objective extractor" agent as part of a multi-agent system.

6.2. Takeaways

This case study demonstrates some real-world considerations in applying the basic techniques to a data analysis system. Following the experiments and observed results, we can see that this example closely follows the summary of takeaways from the surveyed systems as well. In attempting to implement our own agentic system, we observed and experienced many of the trade offs described. Overall, we found that the best approach is to start as simple as possible with the most basic technique, then slowly add complexity when needed while carefully considering the needs of our use case.

7 Conclusion

In this paper, we have covered the basic concepts behind most modern LLM-based approaches, provided a conceptual framework for organizing the components of agentic systems, surveyed several of the most popular agentic systems in the domain of data analysis, and demonstrated the survey takeaways through a case study of an ongoing project. All in all, the insights gained from our system survey and case study provide important guidance for practitioners seeking to apply LLM-based techniques to their own use cases. In a space with an overwhelming amount of techniques, concepts, and complexity, our conceptual framework and use-case guidance helps to provide the clarity needed to apply and understand agentic systems with confidence to the domain of data analysis and beyond.