Chapter 2

Introduction to LLM-Based Agents

Eduardo Bezerra

Abstract

This chapter provides an introductory yet comprehensive exploration of large language model (LLM)-based agents, tracing their evolution from early statistical language models to modern, tool-using systems capable of reasoning and acting in complex environments. We review the key architectural advances that enabled emergent capabilities, introduce prompting techniques and interaction patterns such as ReAct, Plan-and-Act, and Pre-Act, and explain how these patterns coordinate multi-step reasoning with external tools. Core mechanisms including tool calling, Retrieval-Augmented Generation (RAG), and Text-to-SQL pipelines are examined. To bridge theory and practice, the chapter is accompanied by a suite of Jupyter notebooks that demonstrate complete end-to-end workflows across the topics presented. The aim is to equip the reader with both the conceptual understanding and hands-on skills necessary to design, implement, and critically assess LLM-based agents in real-world applications.

Agents represent an exciting and promising new approach to building a wide range of software applications. Agents are autonomous problem-solving entities that are able to flexibly solve problems in complex, dynamic environments, without receiving permanent guidance from the user.

Jennings & Wooldridge (1998)

2.1. Introduction

According to the AIMA Book [Russell and Norvig 2021], an *intelligent agent* is an agent that acts rationally. That is, it selects actions that are expected to maximize its performance measure, based on the percept sequence and its built-in knowledge. The quest to such agents has been a central theme in AI since its inception. Early efforts, such as the Logic Theorist [Newell et al. 1956] and Shakey the Robot [Nilsson 1984], focused on *symbolic reasoning* and planning in structured environments. These systems embodied the classical view of agents as entities that perceive, reason, and act to achieve goals. With the advent of the large language model (LLM) era, interest in AI agents has experienced a strong resurgence. Unlike earlier symbolic systems, modern agents are increasingly built using *connectionist paradigms*, in which reasoning is powered by LLMs instead of symbolic logic.

The goal of this chapter is to provide a comprehensive yet accessible introduction to LLM-based agents. We aim to equip the reader with a conceptual understanding of their architectures and interaction strategies, and to demonstrate how they can be used to perform structured tasks over databases and other tools. The chapter is accompanied with hands-on examples in the form of Jupyter notebooks.

This rest of this chapter is organized as follows. Section 2.2 reviews the evolution from statistical language models to modern neural-based LLMs, highlighting the scaling advances that enabled emergent capabilities. Section 2.3 transitions from standalone LLMs to agentic systems, defining LLM-based agents, clarifying “structured tasks,” and motivating tool-using architectures. Section 2.4 introduces prompting techniques and interaction patterns that structure multi-step reasoning and tool use. Section 2.5 details the tool-calling mechanism, including tool registration, structured message exchange, and orchestration between the LLM and external systems. Section 2.6 presents Retrieval-Augmented Generation (RAG) as a strategy to ground outputs in authoritative external data. Section 2.7 describes the complete Text-to-SQL pipeline, covering intent parsing, schema linking, value grounding, SQL generation, execution and correction, and answer presentation with interactive refinement. Section 2.8 presents a set of Jupyter notebooks that illustrate the full range of concepts and techniques discussed in the chapter, enabling hands-on experimentation beyond individual components. Finally, Section 2.9 offers concluding observations, discusses limitations and ethical considerations, and highlights directions for further study.

2.2. From Statistical Language Models to Neural-based LLMs

By definition, a *language model* is a system trained to understand and generate human language by learning patterns from large amounts of text data[1]. It predicts what *tokens*, which are small units of text (a whole word, part of a word, or even punctuation), are most likely to come next in a sequence. Then the predicted token can be appended to the sequence, and another token can be predicted. By repeatedly

[1]Although the term “language model” has recently been extended to other modalities such as images, audio, and video, this chapter focuses exclusively on agents that interact with language models for text.

performing this prediction, the model can generate entire sentences, paragraphs, or even full documents. For this reason, a language model is a type of *generative model.*

Early approaches to build language models were mainly probabilistic in nature [Rosenfeld 2000]. These models learns conditional probabilities of co-occurrence of items by counting the frequency of token sequences in a large body of text, called a *corpus*. It was the era of the n-gram models. For example, a bi-gram model (an n-gram with $n = 2$) estimates the probability of the next token (w_i) based only on the previous token (w_{i-1}), i.e., $\Pr(w_i|w_{i-1})$. A tri-gram model ($n = 3$) considers the two preceding tokens, i.e., $\Pr(w_i|w_{i-2}, w_{i-1})$. In general, an n-gram model learns these probabilities by counting the frequency of token sequences in a large body of text, called a *corpus*. For a bi-gram model, the probability would be calculated as:

$$\Pr(w_i|w_{i-1}) = \frac{\text{count}(w_{i-1}, w_i)}{\text{count}(w_{i-1})}$$

In a statistical language model, its *context window* refers to the portion of preceding text the model can consider when predicting the next token. In this case, the context window is fixed to the last $n-1$ tokens. For example, a trigram ($n = 3$) model predicts the next token using only the previous two tokens as context. Tokens outside this window have no influence on the prediction

The probabilistic approach allows the model to predict the most likely next token in a sequence. For instance, after seeing the sequence “the cat”, a trigram model might assign a high probability to tokens like “sat” or “is”, and low probability to tokens like *barks*. These probability values are computed based on how often those pairs appeared in the training corpus. Earlier probabilistic approaches, such as n-gram models, represented language through discrete frequency counts and conditional probabilities. While effective in certain domains, they were limited by sparsity issues and short context windows. With the advent of the Deep Learning era [Bezerra 2016, LeCun et al. 2015], neural network-based approaches for building language models began to appear in the literature [Bengio et al. 2003, Mikolov et al. 2010, Vaswani et al. 2017]. These new approaches eventually gave rise to what is current known as *Large Language Models* (LLMs).

LLMs are language models with a very large number of parameters, trained using self-supervised learning on vast amounts of text, and capable of leveraging much longer context windows. It is difficult to pinpoint exactly when the adjective “large” began to be consistently attached to the term “language model”. Although the expression appears as early as 2018 in reference to models such as BERT [Devlin et al. 2019], it rose to prominence and became the default terminology as model scale increased dramatically, particularly following the release of GPT-3 in 2020. However, the meaning of “large” in LLMs continues to evolve, as each new generation of models surpasses the previous in size and capability.

While the concept of context window is common to both pre-neural and modern neural approaches, its interpretation differs across generations of models.

In statistical language models, each prediction step simply uses the most recent $n-1$ tokens available. In modern neural LLMs, the context window is much larger (thousands or even hundreds of thousands of tokens). Here, the window is a shared "token budget" that includes both the input prompt and the model's generated output. For example, with a 4,096-token context window, if the prompt uses 1,000 tokens, the model can generate up to 3,096 tokens before earlier tokens start to be dropped from consideration (a phenomenon sometimes called *contextual drift*). In both cases, the size of the context window limits how much prior information the model can use at once, directly affecting its ability to maintain coherence, recall details, and follow multi-step instructions.

In modern LLM-based applications, the term *prompt* refers to the input provided to the model at inference time to elicit a desired output. The prompt can contain instructions, examples, constraints, or context information, and it is processed along with the model's learned parameters to determine the generated response. The design of prompts (i.e., what information to include, in what order, and with what phrasing) has a direct impact on the model's behavior and output quality.

A related concept, particularly relevant in multi-turn or agent-based setups, is the *system prompt* (sometimes called a "system message"). This is an instruction or set of instructions, often provided at the start of a conversation, that defines the model's overall role, objectives, tone, and operational constraints. Unlike user prompts that change with each interaction, the system prompt persists as part of the context across turns, acting as a stable guide for the model's responses. In frameworks for build LLM-based applications, the system prompt is explicitly separated from user messages; in these frameworks, it often embeds information about available tools, policies, or domain-specific knowledge.

By the time we discuss prompting techniques in Section 2.4 and tool registration in Section 2.5, both concepts, prompt and system prompt, will serve as foundational elements. They are key to understanding how natural language is transformed into structured model behavior, whether in a simple question-answer scenario or in a complex, multi-step agent workflow.

The context window sets a hard limit on how much information an LLM can use at once. However, scaling up model architecture, data, and compute has empirically revealed a complementary phenomenon, one in which entirely new capabilities emerge. An important observation in the study of LLMs is the emergence of qualitatively new capabilities as model scale increases. This idea echoes Philip W. Anderson's 1972 essay *More Is Different* [Anderson 1972], which argues that increasing the scale and complexity of a system can give rise to qualitatively new phenomena, irreducible to the behavior of its smaller components. In the natural sciences, for example, the principles of chemistry emerge from, but are not reducible to, the laws of quantum physics, and biology in turn exhibits properties not explainable by chemistry alone. Recent studies of large language models reveal similar patterns: as the number of parameters, the amount of training data, and the compute budget increase, new abilities often appear abruptly rather than gradually [Wei et al. 2022a, Kayhan et al. 2023]. Examples include in-context learning, multi-step reasoning, and code generation. These emergent capabilities mirror Anderson's insight that "more" can indeed be fundamentally "different".

The rapid appearance of these emergent capabilities has fueled an intense period of innovation in model architectures and training strategies. Each successive generation of LLMs not only increased in size but also integrated new techniques, such as improved pretraining objectives, instruction tuning, and reinforcement learning from human feedback, that amplified their reasoning abilities and practical usefulness. This acceleration is evident when looking at the key milestones in LLM development. Figure 2.1 presents a timeline that summarizes the chronological evolution of large language models (LLMs) and LLM-based agents from 2018 to 2025. On the left-hand side of the timeline, we highlight key LLM milestones, beginning with GPT-1 and GPT-2 developed by OpenAI, followed by T5 and PaLM, and later models such as LLaMA, Claude, Gemini, Qwen, DeepSeek R1, and GPT-5. These models reflect the rapid progression in model scale, training data, and emergent properties.

The advances in architecture, scale, and training that led to neural-based LLMs not only improved language modeling accuracy but also unlocked capabilities that go beyond text generation. These capabilities have made it possible to design systems where the LLM is one component in a larger framework, **an agent**, capable of reasoning, planning, and interacting with external tools and environments.

2.3. From LLMs to LLM-based Agents

Although autonomous agents have a long history in AI (see Section 2.1), their resurgence in the LLM era began in late 2022 with the release of ChatGPT, and accelerated in early 2023 with frameworks like LangChain and open-source projects such as AutoGPT and BabyAGI, which showcased how LLMs could plan, use tools, and act autonomously. We define a *LLM-based agent* as a system that is built around a Large Language Model. LLM-based agents are also known as *generative agents.* By being built around a LLM, we mean that the LLM can be considered the "brain" of the agent. This analogy is appropriate for several reasons. First, the LLM processes information from the agent's environment and the user. Second, the LLM uses its understanding of the current context to decide on a course of action. Lastly, the LLM creates natural language outputs, whether to interact with a user or to command other tools.

In the context of AI agents, a *structured task* is a problem or goal that can be broken down into a series of well-defined, discrete steps with clear inputs and outputs. These tasks are typically predictable and follow a specific, predetermined workflow. A LLM-based agent extends its underlying LLM with features such as structured decision-making processes and access to external tools. These capabilities transform the underlying LLM (which is a passive text generator) into a active agent that can perceive, reason, and act in the real world, making them suitable for complex, multi-step structured tasks that require both intelligence and execution [Sapkota et al. 2025].

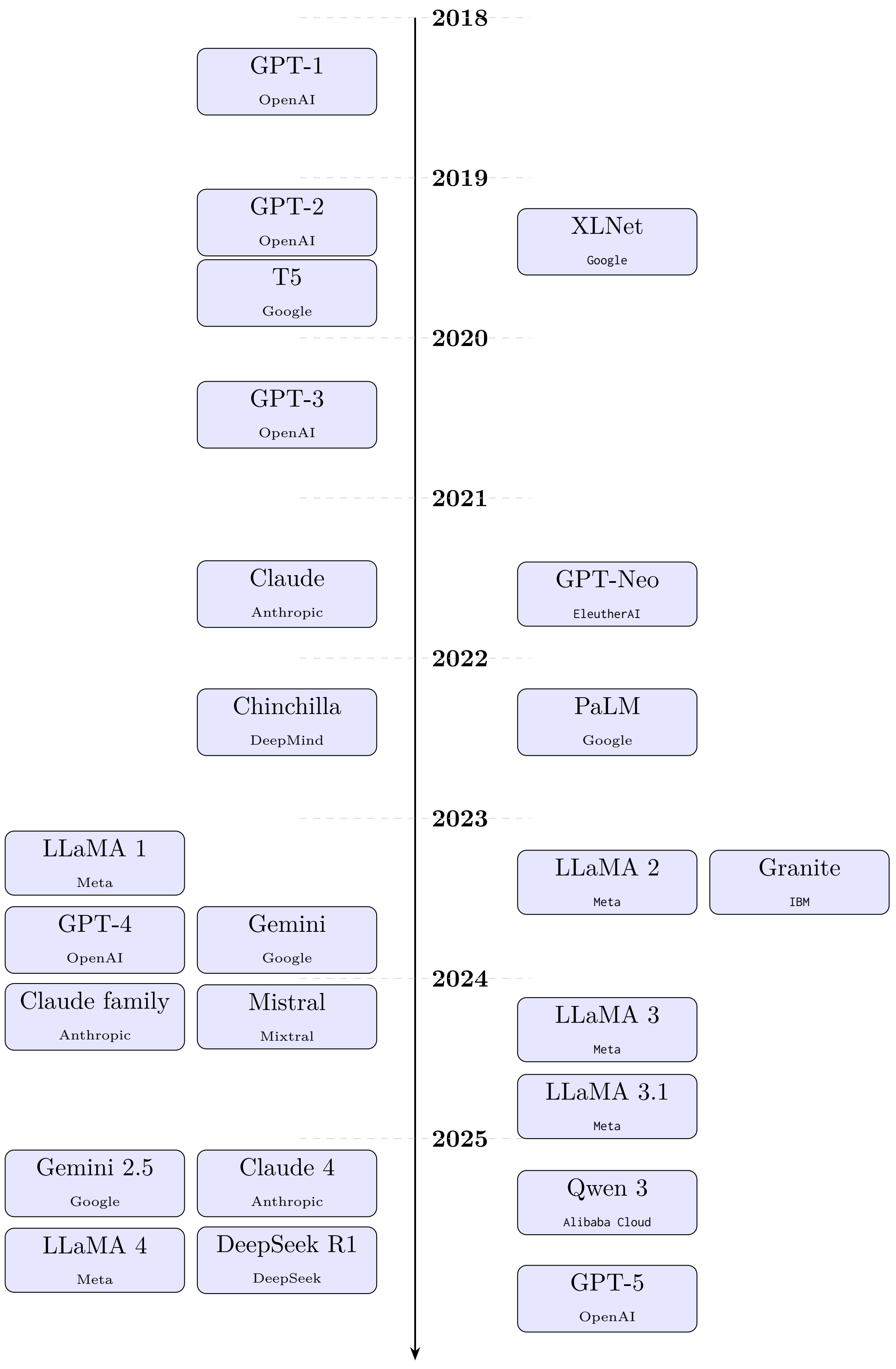


Figure 2.1: Timeline showing the evolution of selected Large Language Models from 2018 to 2025.

2.4. From Prompting Techniques to Interaction Patterns

The idea of guiding a language model's behavior through its input existed in earlier NLP systems, such as sequence-to-sequence models for translation and summarization, but it was often framed in terms of task-specific inputs rather than "prompting". With the release of GPT-2 in 2019, and especially GPT-3 in 2020, the effectiveness of well-crafted natural language prompts in steering large, general-purpose models became widely recognized, sparking a wave of research and practice in *prompting techniques*. Table 2.1 summarizes some notable prompting techniques.

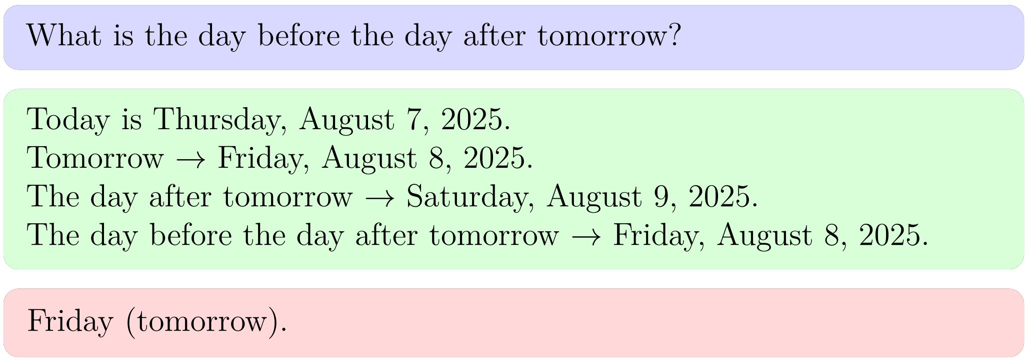
Technique	Description
Zero-shot	Ask the model to perform a task without giving any examples.
Few-shot	Provide a small set of examples to guide output format and style.
Chain-of-thought	Instruct the model to reason step-by-step before answering.
Role	Assign a role/persona to influence tone and perspective.
Self-consistency	Sample multiple reasoning paths and choose the most common answer.
Style/format	Specify required style, tone, or format in the prompt.

Table 2.1: Common prompting techniques for a single LLM invocation.

Most of the prompting techniques listed in Table 2.1 are self-explanatory. One notable exception is *Chain-of-Thought* (CoT), which was the first prompting technique to be formalized in a high-profile paper [Wei et al. 2022b]. CoT refers to phrasing a single prompt in a way that elicits step-by-step reasoning from the model. CoT prompting can be explicit (instructing the model to reason step-by-step) or implicit (designing a question that naturally triggers reasoning). Modern instruction-tuned LLMs often display CoT-like reasoning without the explicit phrase, but the explicit form remains important when one needs to reliably control output style or reasoning depth. Explicit CoT is typically achieved by adding an instruction such as "Let's think step by step" to the prompt, or by supplying intermediate reasoning examples. An illustration of explicit CoT in use is provided in Figure 2.2; note, however, that if only the question "What is the day before the day after tomorrow?" is provided to a modern LLM (without the "Let's think step by step" part), it will most likely generate the reasoning trace along with the final answer.[2]

While simple prompting techniques (Table 2.1) can be effective for direct questions or single-step reasoning, they are often insufficient for tasks that require

[2]This behavior is largely due to the fine-tuning process used in modern instruction-tuned LLMs, which often includes examples of step-by-step reasoning in domains such as mathematics, logic puzzles, and riddles. During reinforcement learning from human feedback (RLHF), annotators tend to reward answers that include clear intermediate reasoning, making such outputs more likely even when no explicit "Let's think step by step" instruction is given.


What is the day before the day after tomorrow?

Today is Thursday, August 7, 2025.
Tomorrow → Friday, August 8, 2025.
The day after tomorrow → Saturday, August 9, 2025.
The day before the day after tomorrow → Friday, August 8, 2025.

Friday (tomorrow).

Figure 2.2: Example of explicit Chain-of-Thought reasoning applied to a simple logic puzzle.

🟦 User 🟩 Reasoning 🟧 Action ⬜ Observation 🟥 Final Answer

multiple reasoning steps, integration of external information, or stateful decision-making. In such cases, the interaction between the model and its environment must follow a structured process rather than a single prompt-response exchange.

An *interaction pattern* is a structured template that defines how a language model agent engages in multi-step interactions (encompassing reasoning, tool use, and memory updates) to perform complex tasks. By specifying the order and format of the model's outputs, such patterns promote consistent behavior and facilitate integration into larger systems. Modern LLM-based agents employ these patterns to coordinate reasoning with external tools (e.g., search engines, database interfaces, and APIs), interpret results, adapt their behavior, and decide whether to invoke additional tools or finalize their responses. The emergence of these interaction patterns has been a key catalyst for the renewed interest in building AI agents in the LLM era.

Examples of interaction patterns include *ReAct*, *Plan-and-Act*, *Pre-Act*, and *Tree-of-Thought*. Table 2.2 summarizes a selection of prominent interaction patterns for LLM-based agents, ordered chronologically by their initial publication date. This timeline helps illustrate how research has progressively explored different strategies for structuring model-environment interaction, from the introduction of *ReAct* in late 2022 to more recent approaches such as *Plan-and-Act* and *Pre-Act* in 2025.

Month/Year	Interaction Pattern	Reference
October/2022	ReAct	[Yao et al. 2022]
May/2023	Tree-of-Thought	[Yao et al. 2023]
March/2025	Plan-and-Act	[Erdogan et al. 2025]
May/2025	Pre-Act	[Hu et al. 2025]

Table 2.2: Timeline of selected interaction patterns with references.

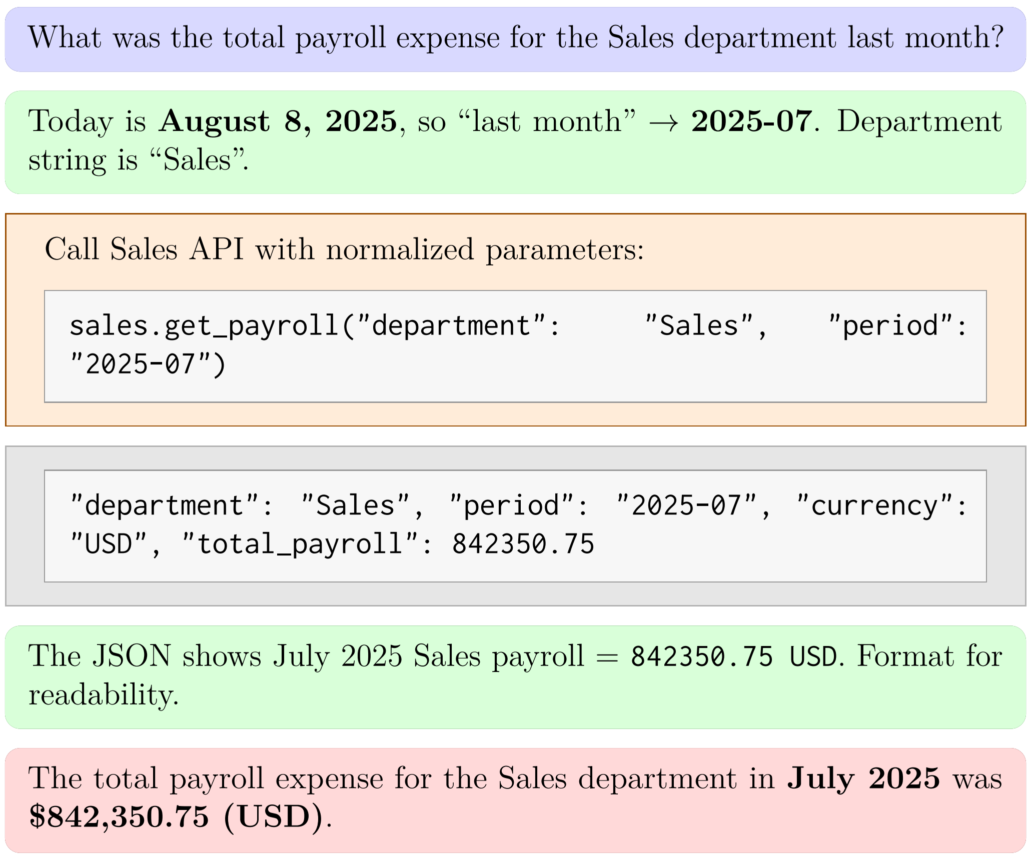
The ReAct (Reason + Act) pattern is designed for scenarios in which large language models (LLMs) must interleave reasoning steps with the use of external tools to accomplish complex, goal-directed tasks [Yao et al. 2022]. The "Reason" component refers to the model's internal reasoning process, similar to the Chain-of-Thought (CoT) approach. The "Act" component corresponds to moments when the model decides to invoke an external tool (such as a search engine, calculator, or database query) to obtain intermediate results that support the final answer. Because LLMs lack direct perception of their environment, they must be explicitly informed (by the agent) about which tools are available. Once equipped with this knowledge, the model can determine, for each prompt, which tools are required and how to integrate their outputs into its reasoning process.

In general, the reasoning process of a model that uses ReAct consists of three stages: Thought, Action, and Observation.

- *Thought*: In this stage, the model generates an internal reasoning step, similar to CoT, to plan its next move. This step may be triggered directly by the original prompt or by the observation of results produced by a previously invoked tool.
- *Action*: Based on the plan, the model may decide it is necessary to interact with an external tool (such as a search engine, calculator, or code interpreter). In this case, the LLM generates a structured instruction indicating which tool should be called and with what parameters.
- *Observation*: The model receives the tool's output and incorporates it into its subsequent reasoning.

Figure 2.3 illustrates the ReAct pattern through an example in which an LLM-based agent handles a business query about payroll expenses. The sequence of steps can be read from top to bottom, following the colored boxes and their legends. The **Query Box** contains a natural language question about the Sales department's payroll for "last month", illustrating how such queries often omit technical specifics. The two **Reasoning Boxes** depict the agent's internal thought process. In the first reasoning step, the agent interprets "last month" relative to the current date `(August 8, 2025)`, concluding that it refers to `July 2025 (2025-07)`. The **Action Box** shows the agent's reasoning (Call Sales API with normalized parameters) and the textual specification of the tool to be invoked. This specification is then used by the agent to make the actual API call. The **Observation Box** contains the raw API response in JSON format, representing the structured data returned by the external system. In the second reasoning step, the agent analyzes this response and prepares it for presentation. Finally, the Answer Box provides the human-readable output (generated by the LLM), synthesizing all the information into a concise, clear answer.

ReAct operationalized the concept of an "LLM-based agent" by giving LLMs a structured way to think, act, and adapt in real time, turning abstract reasoning into practical, tool-augmented workflows. More recent patterns, such as Plan-and-Act separate planning and execution into distinct LLM modules for improved performance on long-horizon tasks [Erdogan et al. 2025]. Pre-Act further enhances this by continually refining multi-step execution plans during task execution [Rawat et al. 2025].



What was the total payroll expense for the Sales department last month?

Today is **August 8, 2025**, so "last month" → **2025-07**. Department string is "Sales".

Call Sales API with normalized parameters:

```
sales.get_payroll("department":     "Sales",    "period":
"2025-07")
```

```
"department": "Sales", "period": "2025-07", "currency":
"USD", "total_payroll": 842350.75
```

The JSON shows July 2025 Sales payroll = `842350.75 USD`. Format for readability.

The total payroll expense for the Sales department in **July 2025** was **$842,350.75 (USD)**.

Figure 2.3: ReAct interaction pattern example showing reasoning, action, observation, and answer for a fictional Sales API query.

🟪 User 🟩 Reasoning 🟧 Action ⬜ Observation 🟥 Final Answer

A detailed discussion of these and other emerging patterns is beyond the scope of this chapter; readers are encouraged to consult the corresponding papers for in-depth explanations and examples.

Memory Management in LLM-Based Agents

In the context of LLM-based agents, *memory* refers to mechanisms that allow the agent to retain and reuse information across interactions. While the underlying LLM has no persistent state beyond its current context window, the agent architecture can maintain external memory to simulate continuity over time. Two common forms are:

- **Short-term memory**, typically implemented as a rolling conversation history or working buffer, storing recent messages, tool outputs, and intermediate reasoning steps. This supports coherence within multi-turn tasks and is often truncated or summarized to stay within context window limits.

- **Long-term memory**, implemented via external storage (e.g., databases, vector stores), enabling the agent to recall facts, user preferences, or prior task results across sessions. This is particularly important in personalized assistants and domain-specific agents.

Memory management involves deciding *what* to store, *how* to represent it (raw text, embeddings, structured records), and *when* to retrieve or summarize it. Poorly managed memory can lead to context overflow, forgotten constraints, or irrelevant retrievals. In interaction patterns such as ReAct or Plan-and-Act, effective memory use supports informed decision-making by grounding reasoning steps in prior context.

Although not the focus of this chapter, memory is a critical enabler for sustained, context-aware behavior in LLM-based agents and directly complements the tool calling, RAG, and prompting strategies discussed later in the chapter.

2.5. Tool Calling

In the agent paradigm, a *tool* is any external capability that an LLM can invoke to perform reasoning steps or access information beyond its pretraining. Examples include APIs, search engines, calculators, and SQL databases. *Tool calling* refers to the mechanism by which an LLM interacts with such external functions, APIs, or services by generating *structured messages* with the appropriate parameters. As discussed in Section 2.4, this process often relies on an interaction pattern, such a ReAct.

Before an LLM-based agent can decide that a tool should be invoked, it must be made aware of its existence, capabilities, and usage parameters. This process is known as *tool registration*. This is typically achieved by providing the model with a structured description of each tool (often in JSON or another machine-readable schema) at the start of the interaction or during an initialization step. The specification usually includes the tool's name, purpose, input parameters (with types and constraints), and expected output format. These descriptions are embedded into the model's system prompt or provided via an API that supports dynamic tool registration. By incorporating this metadata into its context, the LLM can reason about which tool is relevant to the current task, how to construct valid calls, and how to interpret the returned results. Examples of tool registration for a Search API and an SQL executor are shown in Listings 2.1 and 2.2, respectively.

```json
{
  "name": "search_knowledge_base",
  "description": "Find documents in the domain-specific KB",
  "parameters": {
    "query": { "type": "string", "description": "Search terms" },
    "top_k": { "type": "integer", "description": "Max results", "default": 5 }
  }
}
```

Listing 2.1: Example of tool registration with search API details.

Tool registration itself is an instance of a *structured message exchange*. The agent sends the LLM a machine-readable description of each tool, following a fixed schema (often JSON) that encodes metadata such as the tool's name, purpose, input parameters, and output format.

```
{
  "name": "execute_sql",
  "description": "Run SQL queries on the market_data database",
  "parameters": {
    "sql": { "type": "string", "description": "Valid SQL query" }
  },
  "schema": {
    "tables": {
      "companies": ["company_id", "name", "sector", "market_cap"],
      "stock_prices": ["company_id", "trade_date", "close_price"]
    },
    "relationships": [
      "companies.company_id = stock_prices.company_id"
    ]
  }
}
```

Listing 2.2: Example of tool registration with database schema details.

In general, a *structured message* is a machine-readable data object used for communication either within the agent (for example, between the LLM and its orchestration layer) or between the agent and external systems. It follows a predefined, consistent format (typically JSON or similar) with explicit fields for metadata, parameters, and content, enabling reliable parsing, execution of tool calls, and integration of results across multi-step interactions. Listings 2.1 and 2.2 are examples of structured messages that an agent can send to its underlying LLM.

Figure 2.4 presents a schematic view of the tool calling mechanism. The diagram shows a nested architecture in which the agent (depicted as the outer green container) manages the entire process. Within the agent, the LLM serves as the reasoning component. External tools (e.g., search API, SQL executor, code interpreter) are separate systems that the agent can invoke.

In step 1, a human user, a system task, or even another agent provides the agent with an original prompt, for example: "*Identify the top 10 renewable energy companies by market capitalization in 2025, retrieve their average stock price over the past week, and present the results in a table*". Upon receiving this prompt, the agent processes it, informing the LLM of the relevant tools, policies, and context. This information is passed as a structured message. The LLM interprets the message and decides on the next steps, which may include calling a tool.

In the example from Figure 2.4, the LLM first decides to call a search engine. It returns to the agent the specification for invoking this tool, including the textual query: "*Top 10 renewable energy companies by market capitalization in 2025*" (see Listing 2.4a). The agent executes the search (step 2a), receives the results (step 2b), and appends them to a new structured message for the LLM. Reasoning over this updated context, the LLM determines that it needs information from another tool, this time, an SQL engine. The agent again receives the tool call specification, invokes

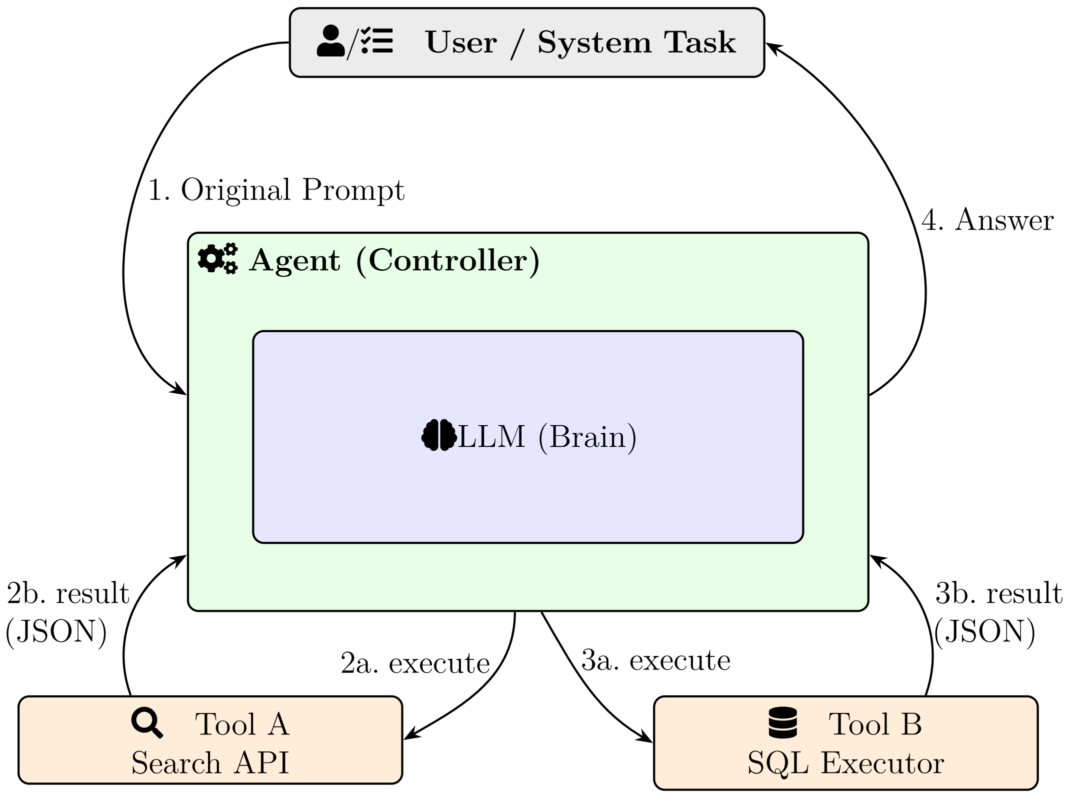


Figure 2.4: Agent-first orchestration with top user/system task source. The task feeds an incoming prompt to the Agent; after iterative reasoning and tool use, the Agent delivers the final answer. The LLM interprets messages, decides next step, and produces structured output. The agent receives prompt, adds tools/policies/context and builds structured messages for the LLM.

the tool (step 3a), receives the result (step 3b, see Listing 2.4b), and appends the resulting data to a new structured message for the LLM. With this enriched context, the LLM reasons one final time and decides to produce the final answer to be generated (see Listing 2.4c). The agent delivers the final answer back to the user/system (step 4).

```sql
SELECT company,
       AVG(close_price) AS avg_price_last_week
FROM stock_prices
WHERE company IN ('NextEra Energy', 'Iberdrola')
  AND trade_date >= CURRENT_DATE - INTERVAL '7 days'
GROUP BY company;
```

Listing 2.3: SQL query generated by the LLM in step 3.

It should be clear from the example of Figure 2.4 that it is not the LLM that directly calls the tool. The LLM's job is to decide that a tool is needed and then generate a structured output (like a JSON object) that describes which tool to

(a) Search API Result

```
{
  "type": "tool_result",
  "tool_name": "Search API",
  "parameters": { "query": "Top 10 renewable energy companies by market
  ↪  capitalization in 2025" },
  "output": [
    { "company": "NextEra Energy", "market_cap": "150B USD" },
    { "company": "Iberdrola", "market_cap": "90B USD" }
  ]
}
```

(b) SQL Executor Result

```
{
  "type": "tool_result",
  "tool_name": "SQL Executor",
  "parameters": {
    "sql_query": "SELECT company, AVG(close_price) ..."
  },
  "output": [
    { "company": "NextEra Energy", "avg_price": 78.52 },
    { "company": "Iberdrola", "avg_price": 12.37 }
  ]
}
```

(c) Final Answer

```
{
  "type": "final_answer",
  "summary": "Average stock prices",
  "data": [
    { "company": "NextEra Energy", "avg_price": 78.52, "currency": "USD" },
    { "company": "Iberdrola", "avg_price": 12.37, "currency": "USD" }
  ],
  "sources": [
    "https://example.com/nextera",
    "https://example.com/iberdrola"
  ]
}
```

Listing 2.4: Structured messages exchanged during an agent execution (e.g., ReAct or Plan-and-Act): (a) Search API result, (b) SQL executor result, (c) final answer produced by the LLM.

use and what arguments to pass to it. The agent (or the orchestration layer in the application) is the component that receives this structured output from the LLM. It then parses that output and actually executes the function or API call associated with the tool. One analogy that can be made is the following: the LLM is the *brain* that reasons and decides; it figures out what needs to be done. The agent is the body that takes the brain's instructions and performs the physical action in the real world. This separation of concerns is fundamental to how tool-calling works in modern LLM applications. It ensures that the LLM, which is a powerful reasoning engine, isn't burdened with the task of execution, while the agent, which is a reliable and predictable executor, can handle the actual interaction with external systems. It's a robust design pattern that maximizes the strengths of both components.

While the example in Figure 2.4 illustrates the general mechanics of tool calling, it abstracts away many of the task-specific details involved in deciding how to use a particular tool. In practice, each type of tool may require its own reasoning strategy, input transformation, and post-processing steps. For instance, generating a search query for an information retrieval module involves different considerations than producing a valid SQL statement for a database engine. The next two paragraphs highlight these differences by previewing how the tool-calling process unfolds in two concrete scenarios that will be explored in detail later in this chapter.

In step 2, the LLM returns to the agent the specification for invoking the Search API, including the textual query "*Top 10 renewable energy companies by market capitalization in 2025*" (Listing 2.4a). Determining this query from the user's original prompt requires retrieving and ranking relevant information, a process that follows the Retrieval-Augmented Generation (RAG) paradigm. The details of this approach will be presented in Section 2.6.

In step 3, the LLM returns to the agent the specification for invoking the SQL Executor, including the query presented in Listing 2.3. Determining this SQL expression is itself a non-trivial process that involves translating natural language into an executable query. The specific techniques and challenges of this Text-to-SQL transformation will be discussed in Section 2.7.

2.6. Retrieval-Augmented Generation

One significant challenge in LLM-based agents is *hallucination*, the tendency of the model to produce outputs that sound plausible but are factually incorrect or unsupported. Hallucinations often arise when the model is asked about information that was likely absent, outdated, or only partially represented in its training data. This includes recent events, niche technical domains, and proprietary datasets. A widely adopted strategy to address this limitation is Retrieval-Augmented Generation (RAG) [Fan et al. 2024], in which the agent retrieves relevant, authoritative information from external sources at query time and feeds it to the LLM. By grounding the model's generation in fresh, domain-specific evidence, RAG both mitigates hallucination and enables the agent to handle queries beyond the scope of its original training corpus.

In practice, RAG works by retrieving relevant documents or passages from a knowledge base and inserting them directly into the LLM's context window. This integration allows the model to ground its output in concrete, up-to-date evidence, improving factual accuracy and reducing unsupported content. In the earlier tool calling example (Figure 2.4), this corresponds to the Search API step, where the agent formulates a query, retrieves results from an external source, and passes them to the LLM before generating the next reasoning step.

2.6.1. Phases of RAG

A RAG pipeline is typically composed of two phases: *Indexing* and *Retrieval and Generation*. The first phase is executed offline as a form of pre-processing, while the second is performed online, since the language model needs to retrieve relevant information in real time to build context for generating a response. Each of these phases involves several steps, as described below.

Indexing. The steps of this phase are depicted in Figure 2.5. In the *Data Loading* step, documents are collected from targeted sources such as PDF files, web pages, or other forms of unstructured information. In the subsequent *Chunking* step, each document is divided into smaller segments to facilitate processing. Various strategies can be employed for splitting documents, depending on the structure and nature of the document (see Section 2.6.2). Then, each chunk is mapped to a vector representation, using a process usually known as *embedding*. A vector embedding is a numerical representation of text data in a continuous, high-dimensional space that captures semantic or structural similarity. There are several embedding frameworks to map text data to a vector representation, such as BERT, RoBERTa, and BERTimbau. In the last step (*Vector Storage*), these vectors are stored in a *vector database*, which is a specialized data store designed to efficiently index, store, and retrieve high-dimensional vector embeddings based on similarity search. This indexing phase is typically performed offline; afterward, the collection is ready to be queried at runtime so relevant chunks can be injected into the LLM's context window to ground answers.

Retrieval and Generation. The steps of this phase are depicted in Figure 2.6. It starts with *Query Embedding*, where the user's query is transformed into a vector representation using the same embedding model employed during indexing. In *Retrieval*, the system searches the vector database using similarity search (e.g., semantic search or dense retrieval methods such as DPR) to identify the top-k most relevant chunks. These retrieved chunks are then combined with the original query in *Prompt Construction*, forming a context-rich prompt for the language model. Finally, in *Generation*, the LLM processes this augmented prompt to produce a grounded, accurate answer, leveraging both the retrieved evidence and its own reasoning capabilities.

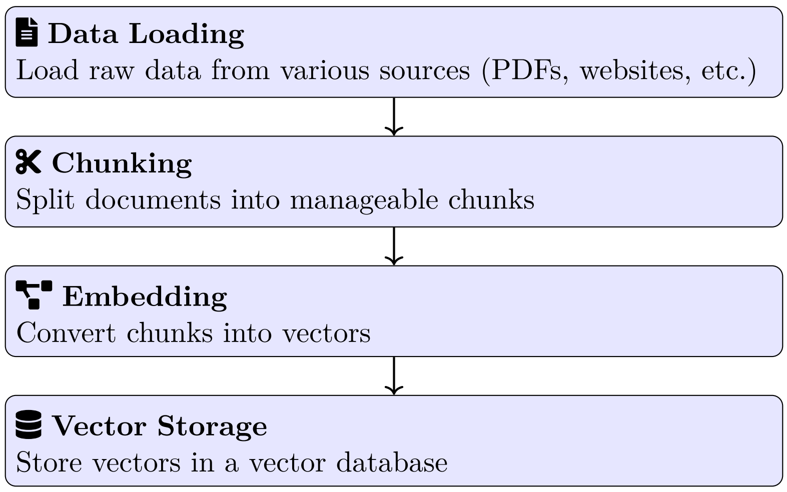

Data Loading
Load raw data from various sources (PDFs, websites, etc.)
Chunking
Split documents into manageable chunks
Embedding
Convert chunks into vectors
Vector Storage
Store vectors in a vector database

Figure 2.5: Phase 1: Data Indexing (Offline). This phase prepares a knowledge base for use in RAG. It begins with *Data Loading*, where raw data is collected from various sources such as PDFs, websites, or internal documentation. In the *Chunking* step, the documents are split into smaller, manageable segments to improve retrieval granularity. These chunks are then passed through an *Embedding* step, which converts them into high-dimensional vector representations using a pre-trained model. Finally, the resulting vectors are stored in a *Vector Database*, enabling efficient semantic search during the retrieval phase.

2.6.2. Chunking Strategies

In the indexing stage of RAG (see Figure 2.5), the strategy to break each document in the knowledge base into chunks is very important. In a document chunking strategy, several factors directly influence retrieval quality and downstream LLM performance:

1. **Chunk Size Trade-offs**. Chunks that are too small risk losing important context, leading to incomplete or ambiguous matches during retrieval. Chunks that are too large may exceed the model's context limits or dilute relevance by mixing unrelated information. The ideal size balances semantic completeness with retrieval precision, often expressed in tokens or characters.

2. **Overlap Strategy**. Overlapping content between consecutive chunks helps preserve context across chunk boundaries, ensuring that relevant information appearing near the edge of one chunk is still captured in the next. Too much overlap, however, increases storage and retrieval costs without proportional benefits.

3. **Metadata Preservation**. Alongside the chunk text, associated metadata (e.g., source document ID, section headings, timestamps, authorship) should be stored and linked to each chunk. Preserving metadata enables filtered or faceted retrieval, traceability of sources, and better grounding of the final generated response.

There are four commonly used approaches to chunking [Fan et al. 2024,

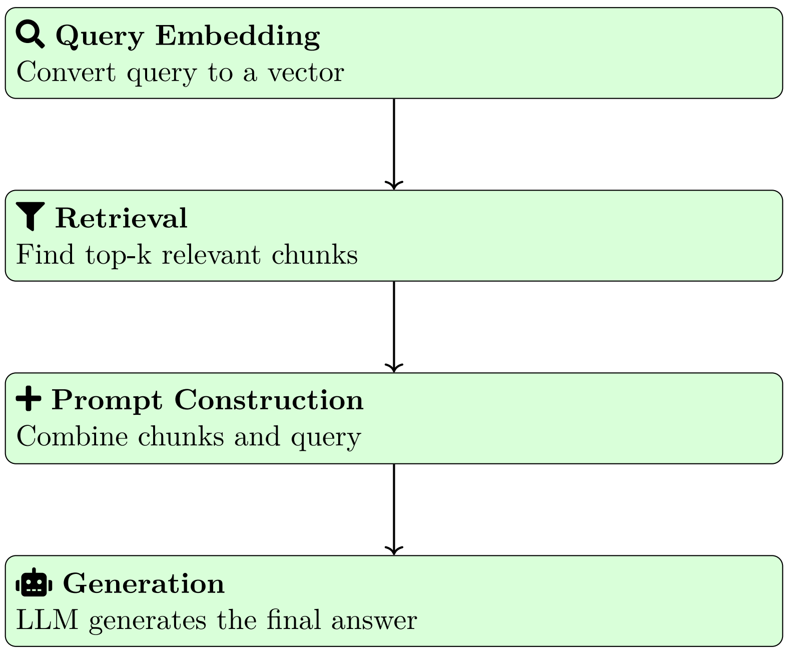

Query Embedding
Convert query to a vector
Retrieval
Find top-k relevant chunks
Prompt Construction
Combine chunks and query
Generation
LLM generates the final answer

Figure 2.6: Overview of the *Retrieval and Generation* phase in a RAG pipeline. This online phase begins with *Query Embedding*, where the user's natural language query is converted into a dense vector representation. In the *Retrieval* step, this vector is used to perform a similarity search over a vector database to retrieve the top-k most relevant text chunks. These chunks are then combined with the original query during the *Prompt Construction* step to form the context window. Finally, in the *Generation* step, the language model uses this enriched prompt to generate a grounded and context-aware response.

Arslan et al. 2024]: *Fixed-Size Chunking*, *Recursive Character Text Splitting*, *Semantic Chunking*, and *Structure-Based Chunking*. The following paragraphs describe each one of these approaches.

Fixed-size chunking is the most basic approach to splitting text, where the content is divided into consecutive segments of a predetermined size, typically measured in characters or tokens, without regard for linguistic or semantic boundaries. For instance, a document might be split into 500-character chunks with an overlap of 50-100 characters to preserve some context between segments. This method works well for uniformly structured data, such as simple logs, raw transcripts, or other unformatted text streams, but its simplicity comes at a cost: it can interrupt sentences or paragraphs mid-thought, fragmenting ideas and potentially reducing retrieval quality.

Recursive character text splitting is a more sophisticated and widely recommended strategy for general-purpose text, designed to prioritize natural boundaries before falling back to fixed-size segments. It uses a hierarchy of separators, such as paragraph breaks ("\n\n"), line breaks ("\n"), sentence endings ("."), and spaces, to divide the text. The splitter first attempts to segment by paragraphs, then by sentences, and so on; if a chunk remains too large, it defaults to a fixed-size split. This method is particularly effective for documents with varied structures, like reports, articles, or books, as it preserves paragraphs and sentences intact whenever possible. However, because it still relies on syntactic markers, it may fail to capture semantic relationships that span across these boundaries.

Semantic chunking is an advanced strategy that prioritizes the meaning of the content over its syntactic structure. It begins by breaking the document into smaller units, such as sentences, and then generating vector embeddings for each. By measuring the similarity between embeddings of adjacent sentences, the method detects "semantic breaks" where the similarity falls below a chosen threshold, signaling a shift in topic and prompting the start of a new chunk. This approach is well-suited for unstructured text or content where the flow of ideas takes precedence over rigid formatting, such as conversational exchanges or creative writing. However, it is more computationally demanding and requires both an embedding model and careful tuning of the similarity threshold.

Structure-based chunking leverages the explicit organization of a document to produce meaningful chunks. For example, a Markdown header splitter uses section headers (e.g., "#", "##") as boundaries, creating chunks that align with sections and subsections, often preserving the headers in the metadata to provide additional context. Similarly, an HTML/XML splitter relies on tags to identify logical components such as titles, paragraphs, or lists, ensuring that related content stays together. This method is particularly effective for structured documents like technical manuals or user guides with clear hierarchical organization. Its main limitation is that it is format-dependent and unsuitable for unstructured text.

2.7. Text-to-SQL

Text-to-SQL refers to the process of converting a natural language query into an equivalent SQL statement that can be executed on a relational database [Shi et al. 2025, Liu et al. 2025, Deng et al. 2022]. The objective is to generate a query that faithfully captures the users's intent and returns the correct results when run against the target data. In the context of LLM-based agents, Text-to-SQL serves as a reasoning capability that enables the agent to bridge the gap between free-form language and the structured syntax of SQL, allowing natural language requests to be transformed into precise, executable commands.

In an agent-based system, interaction with a database typically requires that the SQL interface be registered as an available tool, along with its connection details and schema description. Once this registration is in place, the agent can decide when to use the tool in response to a user request. For example, given the question "*What was the total payroll expense for the Sales department last month?*", the agent may determine that answering it requires querying the database. Following the ReAct or Plan-and-Act paradigms (see Section 2.4), the LLM chooses to invoke the SQL interface, using Text-to-SQL reasoning to translate the natural language request into a valid SQL statement. In this way, the database becomes just another tool the agent can call, much like a calculator or search engine, with Text-to-SQL serving as the mechanism that bridges free-form language and precise, executable queries.

Text-to-SQL has long been recognized as a valuable capability in the database community [Shorten et al. 2025], enabling users to query relational databases without writing SQL manually. Typical applications include: (i) business intelligence dashboards with natural language interfaces; (ii) voice or chatbot assistants that retrieve data from corporate systems; and (iii) self-service data exploration tools for non-technical users. In the LLM era, these use cases are increasingly integrated into multi-tool agent frameworks, where Text-to-SQL operates as the bridge between natural language input and executable database queries.

The key steps of a Text-to-SQL pipeline, illustrated in Figure 2.7, describe the sequential reasoning and processing stages required to transform a natural language request into an executable SQL query and a user-friendly answer. While the specific implementation details vary between traditional rule-based systems and modern LLM-powered agents, the underlying process remains similar: interpret the request, map it to the database schema and values, construct the query, execute it, and refine the result as needed. The following sections describe each stage, highlighting common challenges and considerations for LLM-based agents.

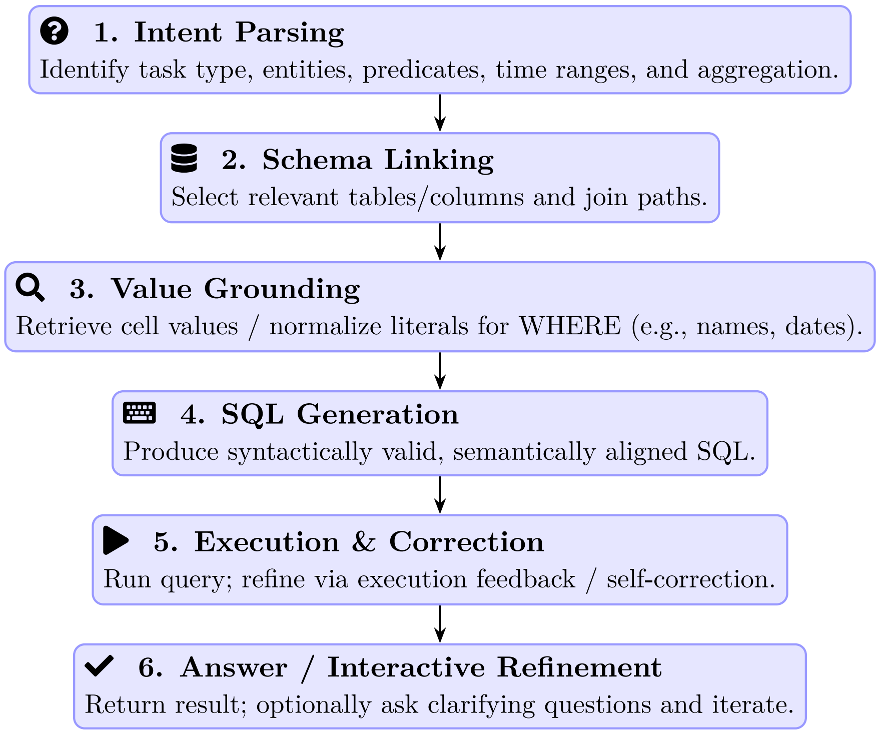

1. Intent Parsing
Identify task type, entities, predicates, time ranges, and aggregation.
2. Schema Linking
Select relevant tables/columns and join paths.
3. Value Grounding
Retrieve cell values / normalize literals for WHERE (e.g., names, dates).
4. SQL Generation
Produce syntactically valid, semantically aligned SQL.
5. Execution & Correction
Run query; refine via execution feedback / self-correction.
6. Answer / Interactive Refinement
Return result; optionally ask clarifying questions and iterate.

Figure 2.7: Revised Text-to-SQL pipeline. Prior to this pipeline, the SQL interface must be registered/available as a tool (connection + schema).

2.7.1. Intent Parsing

The first step in a Text-to-SQL pipeline is to interpret the user's request and determine its underlying intent. This involves identifying the task type (e.g., retrieval, aggregation, filtering), the key entities mentioned, the attributes of interest, and any constraints such as time ranges, conditions, or grouping requirements.

In LLM-based agents, intent parsing often relies on semantic understanding rather than rigid templates, enabling the model to capture nuanced requests such as "*List the top five customers by total sales in Q2*" or "*How many support tickets were closed last week?*".

Accurate intent parsing ensures that subsequent steps operate with a clear representation of what needs to be retrieved, reducing the risk of generating SQL that is syntactically valid yet semantically misaligned with the user's intent.

2.7.2. Schema Linking

Given a parsed intent, an LLM-based agent maps mentions in the request to concrete schema elements (tables, columns, and relationships) and determines feasible join paths. This typically involves name normalization (singular/plural forms, aliases), leveraging foreign keys and primary keys, and disambiguating homonyms (e.g., `name` in multiple tables) using surrounding context. For large or heterogeneous schemas, the agent often performs schema pruning: serializing a subset of the catalog (table/column summaries, data types, brief descriptions) and including only the most relevant pieces in the context window. Effective schema linking yields a compact schema subgraph that preserves necessary joins and candidate columns while excluding distractors, thereby simplifying subsequent value grounding and SQL generation.

This process is often implemented as a two-stage pipeline: initial retrieval followed by LLM-based validation. For example, consider the prompt "*List the top five customers by total sales in Q2*". The agent first retrieves the top-matching schema items (via embedding search):

- `customers (customer_id, name, region)`
- `orders (order_id, customer_id, order_date, total_amount)`

It then confirms their relevance by sending an intermediary prompt to the LLM, asking which schema items are necessary to answer the question.

Figures 2.8a and 2.8b illustrate how intermediate prompts can be used to confirm schema relevance across different domains. The relevant schema elements extracted by the LLM for each intermediate prompt are summarized in Table 2.3, illustrating how the schema linking step narrows the database context to only the fields necessary for accurate SQL generation.

2.7.3. Value Grounding

The third step in the Text-to-SQL pipeline involves mapping abstract references and natural language expressions in the user query to concrete values that exist in the database schema or data. This includes resolving temporal expressions (e.g., "last month" → `2025-07`, "Q2" → specific date range), performing entity disambiguation (e.g., "Apple" → "Apple Inc." as stored in the `company` table), and normalizing literals (e.g., "Jan" → "January", standardizing currency symbols, abbreviations, or capitalization).

Given this question: “List the top five customers by total sales in Q2” and these candidate schema elements:
- customers: customer_id, name, region - orders: order_id, customer_id, order_date, total_amount
Which elements are necessary to answer the question? Return only the relevant ones.

(a) Sales database example

Given this question: “How many support tickets were closed last week?” and these candidate schema elements:
- tickets: ticket_id, opened_date, closed_date, status, assigned_team - teams: team_id, team_name, department
Which elements are necessary to answer the question? Return only the relevant ones.

(b) Customer support database example

Figure 2.8: Examples of intermediate prompts for the schema linking step. In each case, the agent presents candidate schema elements to the LLM, which identifies the subset needed to answer the question.

Prompt (Figure ref.)	**Relevant schema elements**

Figure 2.8a	`customers: customer_id, name`
	`orders: customer_id, total_amount`

Figure 2.8b	`tickets: ticket_id, closed_date`

Table 2.3: Relevant schema elements identified by the LLM for each intermediate prompt in the schema linking examples.

Value grounding also encompasses fuzzy matching techniques to handle variations in naming conventions, such as mapping “NYC” to “New York City” or “AI dept” to “Artificial Intelligence Department” as they appear in the database records. In LLM-based agents, this step often combines retrieval mechanisms (used to search for candidate matches in the database) with the model’s reasoning capabilities to infer the most appropriate mappings based on query context.

Effective value grounding is critical for preventing SQL queries from referencing non-existent or incorrectly formatted values in `WHERE` clauses and `JOIN` conditions. By ensuring values are both valid and properly formatted, it increases the likelihood that generated queries will execute successfully and return meaningful, non-empty results rather than failing due to exact-match errors.

The value grounding step is illustrated in Figure 2.9, where entity names and temporal expressions are mapped to concrete database values and date ranges prior to SQL generation (see Figures 2.9a and 2.9b). In these examples, the mapping assumes the current date is August 2025, so "Q2 last year" corresponds to April 1 to June 30, 2024, and "last month" corresponds to July 1 to July 31, 2025. Such mappings may vary depending on the reference date, calendar conventions, or fiscal year definitions.

Natural language query:
"Show the total sales for Apple in Q2 last year."

Value grounding:
- "Apple" → `"Apple Inc."` (as stored in `company.name`)
- "Q2 last year" → `'2024-04-01'` to `'2024-06-30'`

(a) Entity and temporal grounding with a named company and a quarter-relative period.

Natural language query:
"What was the total payroll expense for the Sales department last month?"

Value grounding:
- "Sales" → `"Sales"` (as stored in `department.name`)
- "last month" → `'2025-07-01'` to `'2025-07-31'`

(b) Department and temporal grounding with a month-relative period.

Figure 2.9: Examples of value grounding: mapping abstract references (entities and time expressions) to concrete database values and ranges before SQL generation.

2.7.4. SQL Generation

The fourth step synthesizes the parsed intent, identified schema elements, and grounded values into a syntactically correct and semantically aligned SQL query. This process involves translating the natural language structure into appropriate SQL constructs, such as mapping aggregation requests to `GROUP BY` clauses with aggregate functions (`COUNT`, `SUM`, `AVG`), temporal filters to `WHERE` conditions with date comparisons, and ranking requirements to `ORDER BY` with `LIMIT` clauses.

The generation must also handle complex query patterns like subqueries for nested conditions, proper `JOIN` syntax to connect multiple tables identified during schema linking, and correct handling of `NULL` values and data types. In LLM-based systems, SQL generation leverages the model's understanding of SQL syntax and semantics, often guided by schema-aware prompting that includes table structures, relationships, and example queries to ensure dialect-specific correctness (e.g., PostgreSQL vs. MySQL syntax differences).

Robust SQL generation produces queries that not only execute without syntax errors but also accurately reflect the user's intent, avoiding common pitfalls such as Cartesian products from missing `JOIN` conditions, incorrect aggregation levels, or inefficient query structures that could cause performance issues on large datasets.

Example 1: Aggregation and Ranking. Consider the prompt: "*List the top five customers by total sales in Q2.*" After **Intent Parsing**, **Schema Linking**, and **Value Grounding**, the system has determined:

- Task type: aggregation + ranking
- Relevant schema: `customers(customer_id, name)`, `orders(customer_id, total_amount, order_date)`
- Grounded values: Q2 = `'2025-04-01'` to `'2025-06-30'`

```sql
SELECT c.name, SUM(o.total_amount) AS total_sales
FROM customers c
JOIN orders o ON c.customer_id = o.customer_id
WHERE o.order_date BETWEEN '2025-04-01' AND '2025-06-30'
GROUP BY c.name
ORDER BY total_sales DESC
LIMIT 5;
```

Listing 2.5: Example SQL generated for the Q2 sales query.

Example 2: Counting Filtered Records. Consider the prompt: "*How many support tickets were closed last month?*" The system has determined:

- Task type: count + filtering by date
- Relevant schema: `tickets(ticket_id, closed_date)`
- Grounded values: last month (August 2025) = `'2025-08-01'` to `'2025-08-31'`

```sql
SELECT COUNT(*) AS closed_tickets
FROM tickets
WHERE closed_date BETWEEN '2025-08-01' AND '2025-08-31';
```

Listing 2.6: Example SQL generated for counting closed support tickets.

2.7.5. Execution and Correction

The fifth step is to execute the generated SQL query against the target database and address any errors that occur. This step forms a critical feedback loop in the Text-to-SQL pipeline. The agent submits the query and receives either a valid result set or an error message from the database engine. If an error is returned (e.g., syntax violation, invalid column name, missing table, or permission restriction), the system must interpret the message and use it to guide a correction.

In LLM-based agents, the error output is typically fed back into the model as an observation. The LLM then analyzes the failure, identifies its likely cause (such as a misspelled column name, mismatched data type, or missing `JOIN` condition), and produces a revised query. This iterative execution-correction cycle continues until a valid query is obtained or a stopping condition is reached. By incorporating this self-repair mechanism, the LLM shifts from being a static code generator to a dynamic, resilient problem-solver capable of adapting to schema-specific constraints and recovering from its own mistakes.

The interaction between SQL generation, execution, and error-driven correction is illustrated in Figure 2.10, which shows how an LLM-based agent iteratively refines queries until a valid result is produced.

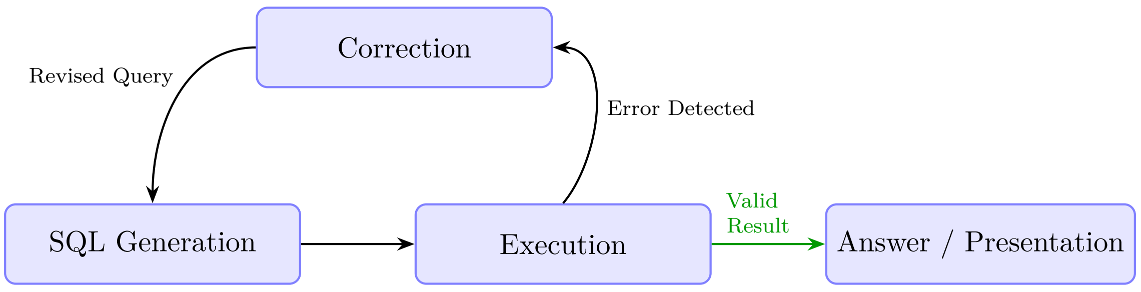


Figure 2.10: Execution and correction loop. Successful execution yields a valid result that flows to answer presentation; errors trigger correction and a revised query that feeds back into SQL generation.

2.7.6. Answer and Interactive Refinement

The final step in the Text-to-SQL pipeline is to present the query result to the user in a clear, natural language format and, when needed, to engage in a dialogue for further refinement. The raw result set returned by the database, typically a table of rows and columns, is often not user-friendly. The agent's role is to interpret this data and produce a concise, human-readable answer that directly addresses the original request. For instance, a result containing a single count might be presented as: "There were 15 support tickets closed last week," rather than displaying a one-row, one-column table.

If the initial request was ambiguous, incomplete, or yielded unexpected results, this step also enables an interactive refinement process. The user can issue follow-up queries (e.g., "*What about the week before that?*") or clarify intent (e.g., "*No, I meant closed by the engineering team*"). The system can then restart the pipeline from an appropriate earlier stage, such as intent parsing or value grounding, while incorporating the additional context, ultimately generating a more precise and satisfactory answer.

This process is illustrated in Figure 2.10, where the agent's response generation is not the end of the process but part of an interactive cycle that can loop back to earlier stages when clarification or refinement is needed.

Figure 2.11 demonstrates how an agent transforms raw database outputs into concise, user-friendly answers and supports iterative refinement based on follow-up questions.

count
15

Agent: There were 15 support tickets closed last week.

(a) Initial query result transformed from raw SQL output to a clear natural language answer.

count
12

User: What about the week before that?
Agent: There were 12 support tickets closed in the previous week.

(b) Interactive refinement after the initial answer.

Figure 2.11: Examples of the answer and interactive refinement step: (a) raw SQL output reformulated as natural language; (b) follow-up interaction to refine the result.

The process of transforming raw SQL results into a concise, user-friendly response is illustrated in Figure 2.12. In this case, the query result is a single count value (15), which, following the approach in Figure 2.11a, is reformulated into the natural language answer "*There were 15 support tickets closed last week*". This example demonstrates how the agent bridges the gap between structured data and conversational output, ensuring that the final response directly addresses the user's original question.

```
The user asked: "How many support tickets were closed last week?"
The SQL query returned this result:
count
15
Respond with a natural language sentence directly answering the question.
```

Figure 2.12: Example of a prompt guiding the agent to transform a raw SQL result into a natural language response.

2.8. Case Studies and Demonstrations

To consolidate the concepts presented throughout this chapter, we provide a collection of Jupyter notebooks that illustrate the main topics covered, from the foundations of large language models to the design and implementation of LLM-based agents with multi-step reasoning and tool use. These notebooks are available at `https://github.com/AILAB-CEFET-RJ/sbbd2025_course`), and are designed to be self-contained and reproducible, enabling readers to experiment directly with the techniques, architectures, and workflows described in the text. The demonstrations include:

1. *From Statistical Models to LLMs.* A hands-on comparison between n-gram language models and small transformer-based models, illustrating differences in context windows, scaling, and emergent capabilities.
2. *From LLMs to LLM-based Agents.* Construction of a minimal agent that uses an LLM as its reasoning core, demonstrating perception, decision-making, and tool invocation in simple structured tasks.
3. *Prompting Techniques and Interaction Patterns.* Experiments with zero-shot, few-shot, and Chain-of-Thought prompting, followed by implementations of interaction patterns such as ReAct, Plan-and-Act, and Pre-Act.
4. *Tool Calling.* Step-by-step examples of tool registration, structured message exchange, and execution via APIs, search engines, and SQL interfaces.
5. *Retrieval-Augmented Generation (RAG).* Building a complete RAG pipeline, including document chunking, embedding generation, vector storage, retrieval, and grounded generation.
6. *Text-to-SQL Pipeline.* An end-to-end example covering intent parsing, schema linking, value grounding, SQL generation, execution and correction, and answer presentation with interactive refinement.

Each notebook follows a consistent structure: defining the required data and tools, walking through each relevant pipeline stage, and allowing readers to modify prompts, parameters, and components to observe their effect. By running these notebooks, readers can explore how the concepts described in the chapter translate into working implementations, reinforcing theoretical understanding with hands-on practice.

2.9. Final Remarks

This chapter provided an introduction to LLM-based agents, tracing their evolution from statistical language models to modern, tool-using architectures capable of reasoning and acting in complex environments. We examined core building blocks such as prompting techniques, interaction patterns, tool calling, retrieval-augmented generation, and the Text-to-SQL pipeline, illustrating how these components integrate into coherent, end-to-end workflows. Beyond theory, the accompanying Jupyter notebooks provide practical demonstrations that enable readers to experiment with these ideas in real-world scenarios. Together, these elements aim to equip the reader with both the conceptual understanding and the hands-on skills necessary to design, implement, and critically evaluate LLM-based agents in diverse application domains.

Looking ahead, the field is poised to advance rapidly along several fronts. Emerging trends such as multi-agent systems (agent collaboration), seamless multimodality, dynamic memory management, and domain-specific fine-tuning will expand both the capabilities and the applicability of LLM-based agents. At the same time, significant challenges remain, including ensuring factual reliability, safeguarding data privacy, maintaining transparency in decision-making, and mitigating bias amplification. Mitigation strategies include grounding outputs in authoritative data (e.g., via RAG), maintaining detailed logs of reasoning steps and tool calls for auditability, applying domain-specific access controls, and performing bias and safety evaluations during development. Embedding such safeguards into the design and operation of agents is crucial to align technical capabilities with legal requirements, organizational policies, and societal expectations. Addressing these challenges will require not only technical innovation but also interdisciplinary collaboration between AI researchers, domain experts, and policymakers. By combining robust architectures with responsible deployment practices, the next generation of LLM-based agents can evolve from promising prototypes to dependable tools that operate safely and effectively in high-impact real-world settings.

Finally, it is worth noting that the quest to build intelligent agents is far from new. As Jennings and Wooldridge observed more than two decades ago [Jennings and Wooldridge 1998], agents are autonomous problem-solving entities capable of operating in complex, dynamic environments without continuous human guidance. While the technological substrate has shifted (from symbolic reasoning systems to large-scale neural models), the essence of this vision remains strikingly relevant. Today's LLM-based agents embody many of the same aspirations outlined in that earlier work, demonstrating that the principles articulated then continue to guide and inspire the development of the next generation of AI systems.

Acknowledgments

This chapter was developed with the assistance of artificial intelligence tools, which were used to support tasks such as drafting, revising, and refining text, as well as suggesting examples and clarifying technical concepts. All content was critically reviewed and edited by the author to ensure accuracy, coherence, and alignment with the intended objectives.

References

[Anderson 1972] Anderson, P. W. (1972). More is different. *Science*, 177(4047):393–396.

[Arslan et al. 2024] Arslan, M., Ghanem, H., Munawar, S., and Cruz, C. (2024). A survey on RAG with llms. *Procedia Computer Science*, 246:3781–3790. 28th International Conference on Knowledge Based and Intelligent information and Engineering Systems (KES 2024).

[Bengio et al. 2003] Bengio, Y., Ducharme, R., Vincent, P., and Jauvin, C. (2003). A neural probabilistic language model. *Journal of Machine Learning Research*, 3:1137–1155.

[Bezerra 2016] Bezerra, E. (2016). Introdução à aprendizagem profunda. In Ogasawara, V., editor, *Tópicos em Gerenciamento de Dados e Informações*, chapter 3, pages 57–86. SBC, Porto Alegre, Brazil, 1 edition.

[Deng et al. 2022] Deng, N., Chen, Y., and Zhang, Y. (2022). Recent advances in text-to-SQL: A survey of what we have and what we expect. In *COLING*, Gyeongju, Republic of Korea. International Committee on Computational Linguistics.

[Devlin et al. 2019] Devlin, J., Chang, M.-W., Lee, K., and Toutanova, K. (2019). BERT: Pre-training of deep bidirectional transformers for language understanding. In Burstein, J., Doran, C., and Solorio, T., editors, *ACL 2019*, pages 4171–4186, Minneapolis, Minnesota. ACL.

[Erdogan et al. 2025] Erdogan, L. E., Furuta, H., Kim, S., et al. (2025). Plan-and-act: Improving planning of agents for long-horizon tasks. In *ICML 2025*.

[Fan et al. 2024] Fan, W., Ding, Y., Ning, L., Wang, S., Li, H., Yin, D., Chua, T.-S., and Li, Q. (2024). A survey on RAG meeting llms: Towards retrieval-augmented large language models. In *KDD'24*, KDD'24, page 6491–6501, New York, NY, USA. Association for Computing Machinery.

[Hu et al. 2025] Hu, S., Kim, S. R., Zhang, Z., et al. (2025). Pre-act: Multi-step planning and reasoning improves acting in LLM agents. *arXiv preprint arXiv:2505.09970*.

[Jennings and Wooldridge 1998] Jennings, N. R. and Wooldridge, M. J., editors (1998). *Agent Technology: Foundations, Applications, and Markets*. Springer, Berlin, Heidelberg.

[Kayhan et al. 2023] Kayhan, V., Levine, S., Nanda, N., Schaeffer, R., Natarajan, A., Chughtai, B., et al. (2023). Scaling laws and emergent capabilities of large language models. *arXiv preprint arXiv:2309.00071*.

[LeCun et al. 2015] LeCun, Y., Bengio, Y., and Hinton, G. (2015). Deep learning. *Nature*, 521(7553):436–444.

[Liu et al. 2025] Liu, X., Shen, S., Li, B., Ma, P., Jiang, R., Zhang, Y., Fan, J., Li, G., Tang, N., and Luo, Y. (2025). A survey of text-to-sql in the era of llms: Where are we, and where are we going? *IEEE Transactions on Knowledge and Data Engineering*, pages 1–20.

[Mikolov et al. 2010] Mikolov, T., Karafiát, M., Burget, L., Černockỳ, J., and Khudanpur, S. (2010). Recurrent neural network based language model. In *INTERSPEECH*, pages 1045–1048.

[Newell et al. 1956] Newell, A., Shaw, J., and Simon, H. A. (1956). The logic theory machine–a complex information processing system. *IRE Transactions on Information Theory*, 2(3):61–79.

[Nilsson 1984] Nilsson, N. J. (1984). *Shakey the Robot.* SRI International, Menlo Park, CA.

[Rawat et al. 2025] Rawat, M., Gupta, A., et al. (2025). Pre-act: Multi-step planning and reasoning improves acting in llm agents. *arXiv preprint arXiv:2505.09970.*

[Rosenfeld 2000] Rosenfeld, R. (2000). Two decades of statistical language modeling: where do we go from here? *Proceedings of the IEEE*, 88(8):1270–1278.

[Russell and Norvig 2021] Russell, S. and Norvig, P. (2021). *Artificial Intelligence: A Modern Approach.* Pearson, 4th edition.

[Sapkota et al. 2025] Sapkota, R., Roumeliotis, K. I., and Karkee, M. (2025). Ai agents vs. agentic ai: A conceptual taxonomy, applications and challenges.

[Shi et al. 2025] Shi, L., Tang, Z., Zhang, N., Zhang, X., and Yang, Z. (2025). A survey on employing large language models for text-to-sql tasks. *ACM Comput. Surv.*

[Shorten et al. 2025] Shorten, C., Pierse, C., Smith, T. B., D'Oosterlinck, K., Celik, T., Cardenas, E., Monigatti, L., Hasan, M. S., Schmuhl, E., Williams, D., Kesiraju, A., and van Luijt, B. (2025). Querying databases with function calling.

[Vaswani et al. 2017] Vaswani, A., Shazeer, N., Parmar, N., Uszkoreit, J., Jones, L., Gomez, A. N., Kaiser, Ł., and Polosukhin, I. (2017). Attention is all you need. In *Advances in Neural Information Processing Systems (NeurIPS)*, pages 5998–6008.

[Wei et al. 2022a] Wei, J., Tay, Y., Bommasani, R., Raffel, C., Zoph, B., Borgeaud, S., Yogatama, D., Bosma, M., Zhou, D., Metzler, D., Chi, E. H., Hashimoto, T. B., Vinyals, O., Liang, P., Dean, J., and Fedus, W. (2022a). Emergent abilities of large language models. *arXiv preprint arXiv:2206.07682.*

[Wei et al. 2022b] Wei, J., Wang, X., Schuurmans, D., Bosma, M., Ichter, B., Xia, F., Chi, E. H., Le, Q. V., and Zhou, D. (2022b). Chain-of-thought prompting elicits reasoning in large language models. *arXiv preprint arXiv:2201.11903.*

[Yao et al. 2023] Yao, S., Yu, D., Zhao, J., Shafran, I., Griffiths, T. L., Cao, C., and Narasimhan, K. (2023). Tree of thoughts: Deliberate problem solving with large language models. *arXiv preprint arXiv:2305.10601.*

[Yao et al. 2022] Yao, S., Zhao, J., Yu, D., Du, N., Yu, W.-t., Shafran, I., Griffiths, T. L., Neubig, G., Cao, C., and Narasimhan, K. (2022). React: Synergizing reasoning and acting in language models. *arXiv preprint arXiv:2210.03629.*