Introduction to LLM-Based Agents

Short Course - SBBD 2025

Eduardo Bezerra – CEFET/RJ

Outline

Instructor

Instructor

Prof. Eduardo Bezerra
CEFET/RJ

- Senior Professor and Researcher in AI and Data Science
- Interests: LLM-based agents, data science, applied AI
- Active in PPCIC graduate program (CEFET/RJ)

ebezerra@cefet-rj.br
github.com/AILAB-CEFET-RJ

Didactic Material

- **Lecture notes:** detailed PDF (30 pages) with all concepts.
- **Slides:** concise and didactic version for class.
- **Jupyter notebooks:** hands-on examples to experiment.
- **GitHub repository:** `sbbd2025_course`

All resources are freely available.

Github repo

https://github.com/AILAB-CEFET-RJ/sbbd2025_course

Which Year?

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 (????)

From Statistical Language Models to Neural-based LLMs

Historic Context of Language Models

1950s	**Chomsky** Formal grammars
1980s	Statistical approaches *n*-gram models
2000s	Word embeddings Continuous vectors
2010s	Transformers (2017) BERT, GPT-2
2020s	GPT-3, ChatGPT LLM-based agents

What is a Language Model?

A language model is a system trained to predict the next word (or token) in a sequence, based on the context of the words that came before.

What is a Language Model?

- A **language model (LM)** is trained on massive text data.
- It learns statistical patterns of language.
- Core task: predict the next token.

Next token prediction

From Prediction to Generation

- Step-by-step prediction enables building coherent sequences.
- Produces **sentences, paragraphs, full documents**.

Token-by-token $\Rightarrow$ full text

From Prediction to Generation

- Step-by-step prediction enables building coherent sequences.
- Produces **sentences, paragraphs, full documents**.

The $\rightarrow$ cat $\rightarrow$ sits $\rightarrow$ outside

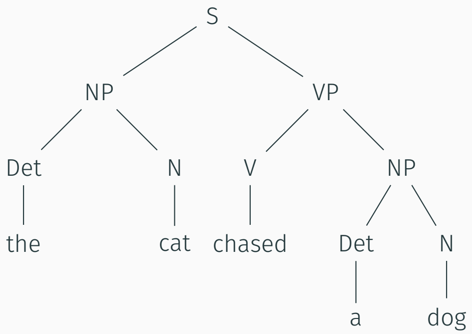
Token-by-token $\Rightarrow$ full text

- Since generation is achieved by chaining predictions, a language model is naturally a generative model.

Chomsky's LMs (Formal Grammar)

Grammar Rules

$$S \rightarrow NP\ VP$$
$$NP \rightarrow Det\ N$$
$$VP \rightarrow V\ NP$$
$$Det \rightarrow \text{"the"} \mid \text{"a"}$$
$$N \rightarrow \text{"cat"} \mid \text{"dog"}$$
$$V \rightarrow \text{"chased"} \mid \text{"saw"}$$



Statistical LMs

Early language models were **statistical**.

Statistical LMs

Early language models were **statistical**. That means:

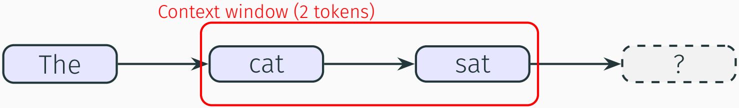
- They relied on counting token co-occurrences in large corpora.
- Used n-grams: probability of next token depends only on last $n-1$ tokens.

Statistical LMs

Early language models were **statistical**. That means:

- They relied on counting token co-occurrences in large corpora.
- Used n-grams: probability of next token depends only on last $n-1$ tokens.

Example: $\Pr(w_i \mid w_{i-1}, w_{i-2})$ for a trigram model.



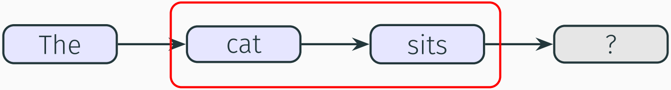
Limitations of Early Language Models

Short Context
Predictions depend only on a limited *n*-gram history.

Data Sparsity
Many valid sequences are rare or unseen in training data.

Poor Generalization
Limited ability to capture long-range dependencies and meaning.

Short Context Limitation

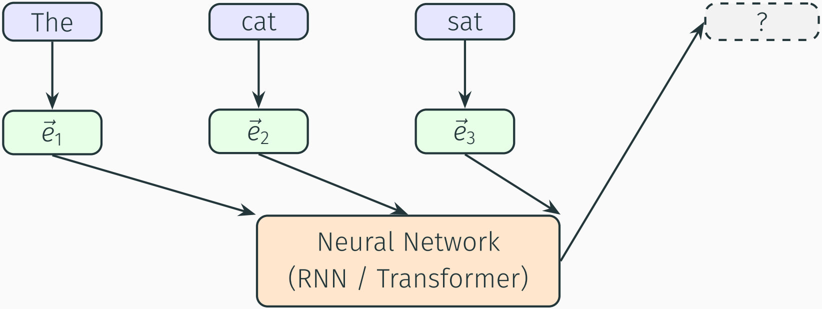


Trigram model: prediction of the next token uses only the last 2 words.

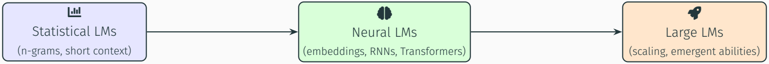
Neural LMs

Neural language models overcame the limits of n-grams:

- Tokens mapped to **embeddings** (dense vectors).
- Context modeled via **neural networks** (RNNs, LSTMs, Transformers).
- Capture longer dependencies and generalize better.
- Enabled the scaling path toward LLMs.



From Statistical to Neural to LLMs



- **Statistical:** count-based, limited context.
- **Neural:** embeddings + deep networks capture longer dependencies.
- **LLMs:** massive scale unlocks emergent capabilities.

Emergent Capabilities

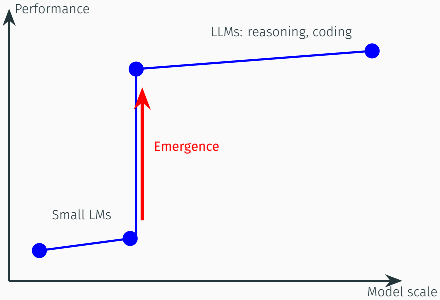
Massive scale does not just improve accuracy;

Emergent Capabilities

Massive scale does not just improve accuracy; at sufficient scale, LLMs suddenly exhibit new abilities Wei et al. [2022], *Emergent Abilities of Large Language Models*.

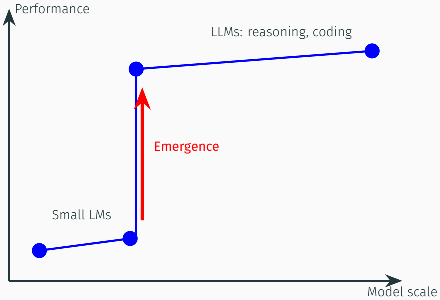
Emergent Capabilities

Massive scale does not just improve accuracy; at sufficient scale, LLMs suddenly exhibit new abilities Wei et al. [2022], *Emergent Abilities of Large Language Models*.



Emergent Capabilities

Massive scale does not just improve accuracy; at sufficient scale, LLMs suddenly exhibit new abilities Wei et al. [2022], *Emergent Abilities of Large Language Models.*



"I know Kung Fu!"

Emergent Abilities of LLMs: Hype and Criticism

Original Claim (Wei et al. [2022])

- As model size grows, **new abilities appear suddenly**, not gradually.
- Example: multi-step reasoning, in-context learning, code generation.

Emergent Abilities of LLMs: Hype and Criticism

Original Claim (Wei et al. [2022])

- As model size grows, **new abilities appear suddenly**, not gradually.
- Example: multi-step reasoning, in-context learning, code generation.

Recent Criticism (Schaeffer et al. [2023])

- Apparent "emergence" may be an artifact of evaluation metrics (threshold effects).
- Performance often improves **smoothly**, but looks abrupt when measured with accuracy or pass/fail metrics.
- Some abilities may reflect better prompting or training data, not sudden leaps in capability.

Emergent Abilities of LLMs: Hype and Criticism

Original Claim (Wei et al. [2022])

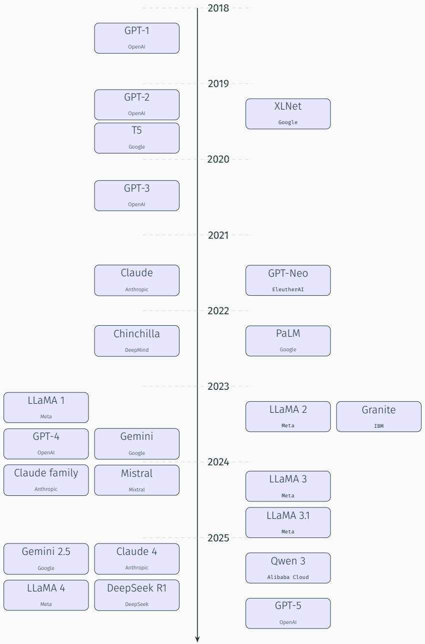
- As model size grows, **new abilities appear suddenly**, not gradually.
- Example: multi-step reasoning, in-context learning, code generation.

Recent Criticism (Schaeffer et al. [2023])

- Apparent "emergence" may be an artifact of evaluation metrics (threshold effects).
- Performance often improves **smoothly**, but looks abrupt when measured with accuracy or pass/fail metrics.
- Some abilities may reflect better prompting or training data, not sudden leaps in capability.

Regardless, LLM progress has been impressive in the past few years...

LLMs Timeline



Demo Time

From LLMs to LLM-based Agents

From LLMs to LLM-based Agents

An LLM is, in essence, a **probabilistic text generator**: it predicts the next token.

From LLMs to LLM-based Agents

An LLM is, in essence, a **probabilistic text generator**: it predicts the next token.

Yet at scale, LLMs exhibit reasoning-like abilities (planning, inference, language understanding).

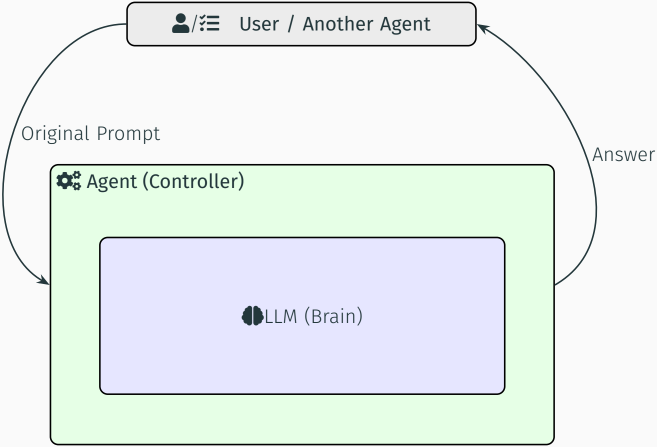
From LLMs to LLM-based Agents

An LLM is, in essence, a **probabilistic text generator**: it predicts the next token.
Yet at scale, LLMs exhibit reasoning-like abilities (planning, inference, language understanding).

An LLM-based agent combines:

- These **reasoning-like abilities** of the LLM,
- **Control logic** to orchestrate steps and manage context,
- Access to **external tools** (search, databases, APIs),
- Mechanisms for **perception and action**, so reasoning can affect the real world.

LLM-based Agent



Demo Time

From Prompting Techniques to Interaction Patterns

Prompting Techniques: Examples

Zero-shot

Prompt:

```
What is 47 + 35?
```

Output:

```
82
```

Few-shot

Prompt:

```
Examples:
12 + 7 = 19
5 + 9 = 14
23 + 18 = 41

Now, what
is 47 + 35?
```

Output:

```
82
```

Chain-of-Thought

Prompt:

```
What is 47 + 35?
Let's think
step by step.
```

Output:

```
First, add tens:
40 + 30 = 70
Then, ones:
7 + 5 = 12
Sum:
70 + 12 = 82
Answer: 82
```

From Prompting to Patterns

- Prompting techniques (Zero-shot, Few-shot, Chain-of-Thought) steer a **single LLM call**:
- But complex tasks require multi-step reasoning and tool use.

From Prompting to Agents

Examples of complex tasks that require multi-step reasoning and tool use:

- Answering questions over a large knowledge base (search + reasoning)
- Text-to-SQL (NL → SQL query → execution → formatted result)
- Planning a trip (dates, flights, hotels, budget)
- Solving math word problems (step-by-step reasoning + calculation)
- Data analysis (retrieve data → transform → summarize)

Interaction Patterns

- **Interaction patterns** structure reasoning + action.
- Ensure consistency and integration with external tools.

Interaction Patterns

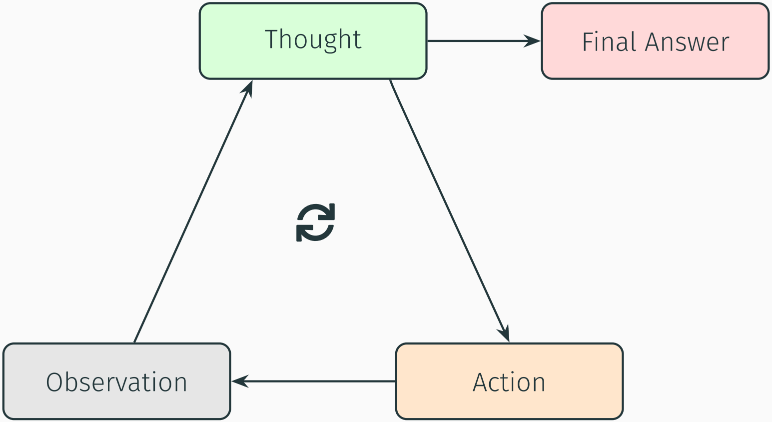
- **Interaction patterns** structure reasoning + action.
- Ensure consistency and integration with external tools.
- Examples:
    - ReAct (Reason + Act)
    - Plan-and-Act
    - Reflexion

Interaction Patterns

- **Interaction patterns** structure reasoning + action.
- Ensure consistency and integration with external tools.
- Examples:
  - ReAct (Reason + Act)
  - Plan-and-Act
  - Reflexion
- We focus on ReAct.

ReAct: Synergizing Reasoning and Acting in Language Models [Yao et al., 2022]

ReAct Pattern



The agent **thinks**, **acts**, and **observes** in a loop,
until enough information is gathered to deliver the final answer.

ReAct in Action

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

Reasoning (LLM): Today is **August 8, 2025**, so “last month” $\rightarrow$ **2025-07**. Department string is “Sales”.

Action (Orchestrator): Call Sales API with normalized parameters:

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

Observation (Orchestrator):

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

Reasoning (LLM): The JSON shows July 2025 Sales payroll = `842350.75` **USD**. Format for readability.

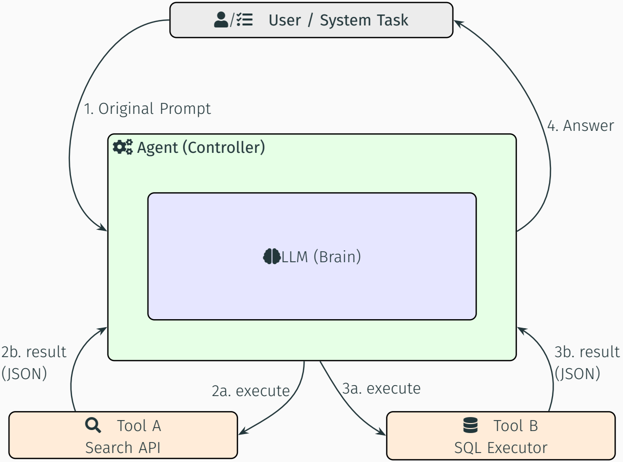
Answer (LLM): The total payroll expense for the Sales department in **July 2025** was **$842,350.75 (USD)**.

User | Reasoning | Action | Observation | Final Answer

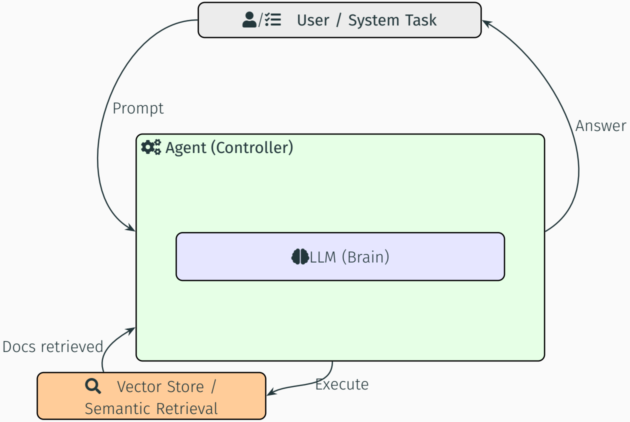
Demo Time

Tool Calling

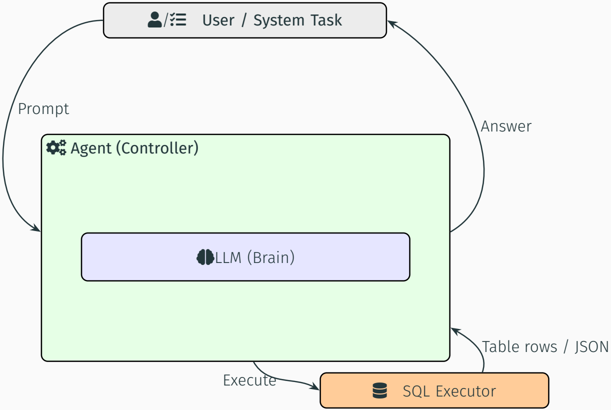
Tool Calling



Tool Calling Instance: RAG



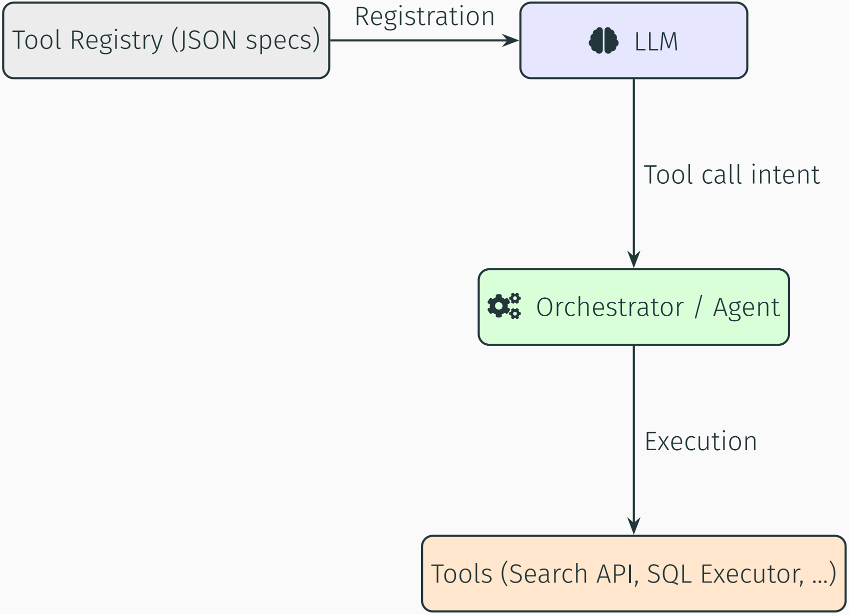
Tool Calling Instance: Text-to-SQL



Tool Registration?

- LLMs do not “magically” know what tools exist.
- Each tool must be **registered** via a structured description (usually JSON) that includes:
  - **Name** and **purpose** of the tool,
  - **Input parameters** (types, constraints, defaults),
  - **Expected outputs**.
- This information is added to the LLM context, so the model can decide when and how to use the tool.
- Tool registration is the gateway for RAG, Text-to-SQL, and many other applications.

Tool Registration?



Tool Registration: SQL Executor

```json
{
  "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"
    ]
  }
}
```

- Registers a **SQL tool** with schema info.
- Enables Text-to-SQL: natural language → SQL query → DB results.

Tool Registration: Search API

```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 }
  }
}
```

- Registers a **retrieval tool** for semantic search.
- Enables RAG: query → retrieve docs → ground generation.

Structured Message: Search API Result


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


This message shows a possible return of the **Search API tool** as a structured message.

(Fictitious company names for illustration)

Structured Message: SQL Executor Result

```json
{
  "type": "tool_result",
  "tool_name": "SQL Executor",
  "parameters": {
    "sql_query": "SELECT company, AVG(close_price) ..."
  },
  "output": [
    { "company": "ACME", "avg_price": 78.52 },
    { "company": "Gray Matter", "avg_price": 12.37 }
  ]
}
```

Here the **SQL Executor** tool returns aggregated values for each company, also as a structured message.

Structured Message: Final Answer


```json
{
  "type": "final_answer",
  "summary": "Average stock prices",
  "data": [
    {  "company": "ACME",
       "avg_price": 78.52,
       "currency": "USD" },
    {  "company": "Gray Matter",
       "avg_price": 12.37,
       "currency": "USD" }
  ],
  "sources": [
    "https://example.com/nextera",
    "https://example.com/Gray Matter"
  ]
}
```


The LLM synthesizes tool outputs into a clear, user-facing **Final Answer**.

Demo Time

Retrieval-Augmented Generation

Definition

Retrieval-Augmented Generation (RAG) is a framework that improves language model outputs by retrieving relevant external documents and injecting them into the prompt, so the model can generate grounded and evidence-based responses.

Motivation

- LLMs may **hallucinate** when training data is insufficient, outdated, or domain-specific.
- RAG grounds model outputs in **external, authoritative sources**.
- Aims at (1) reducing unsupported generated content and (2) enabling handling of queries beyond the pretraining corpus.

A typical RAG pipeline

- Break available documents into pieces and index them.
- Retrieve relevant pieces or passages at query time.
- Inject retrieved chunks into the LLM's context window before generation.

A typical RAG pipeline

- Break available documents into pieces and index them.
- Retrieve relevant pieces or passages at query time.
- Inject retrieved chunks into the LLM's context window before generation.

Goal: generate output that combines fluency of the LLM with evidence-based retrieval.

Phase 1: Indexing (Offline)

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

Chunking Step: Strategies

- **Fixed-size chunks**
  Split text into uniform blocks (e.g., 500 tokens), regardless of structure.
- **Recursive chunking**
  Split hierarchically: paragraphs → sentences → smaller units if needed.
- **Semantic chunking**
  Use embeddings or similarity to group text into meaning-preserving segments.
- **Structure-based chunking**
  Exploit document structure (e.g., sections, headings, tables, code blocks).

Chunking Step: Visual Comparison of Strategies

Original document: *”The cat sits outside. It is sunny today. The dog barks loudly. Dogs often bark when they see strangers.”*

Fixed-size: [”The cat sits outside. It is sunny”, ”today. The dog barks loudly. Dogs often bark when they see strangers.”]

Recursive: [”The cat sits outside.”, ”It is sunny today.”, ”The dog barks loudly.”, ”Dogs often bark when they see strangers.”]

Semantic: [”The cat sits outside. It is sunny today.”, ”The dog barks loudly. Dogs often bark when they see strangers.”]

Structure-based: [”**Section:** — The cat sits outside. It is sunny today.”, ”**Section:** — The dog barks loudly. Dogs often bark when they see strangers.”]

Embedding Step

```python
from langchain.embeddings import OpenAIEmbeddings

# Example texts (chunks from a document)
texts = [
    "The cat sits outside.",
    "It is sunny today.",
    "The dog barks loudly."
]

# Create embedding model
embedding_model = OpenAIEmbeddings()

# Generate vector representations
vectors = embedding_model.embed_documents(texts)

print(len(vectors), "embeddings generated.")
print("Dimension of each embedding:", len(vectors[0]))
```

Each text chunk is mapped to a high-dimensional vector capturing semantic meaning.

Vector Storage Step

- After generating embeddings, store them in a **vector database**.
- Each entry typically contains:
  - The **embedding vector** (high-dimensional representation).
  - The **original text chunk**.
  - Optional **metadata** (source, page number, section, etc.).
- Vector DBs (e.g., `Chroma`, `FAISS`, `Weaviate`, `Pinecone`) enable:
  - Fast similarity search (cosine, dot product).
  - Efficient retrieval of relevant chunks for grounding.

Vector Storage Step

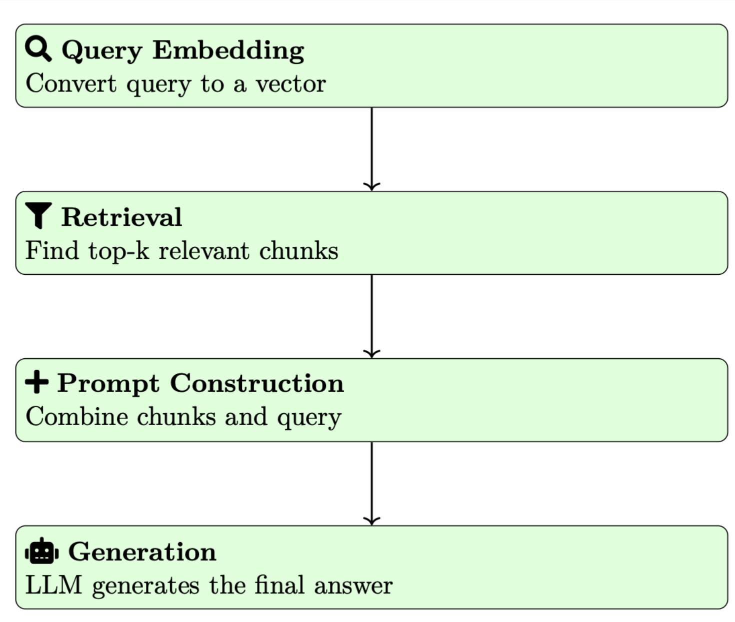
- After generating embeddings, store them in a **vector database**.
- Each entry typically contains:
  - The **embedding vector** (high-dimensional representation).
  - The **original text chunk**.
  - Optional **metadata** (source, page number, section, etc.).
- Vector DBs (e.g., `Chroma`, `FAISS`, `Weaviate`, `Pinecone`) enable:
  - Fast similarity search (cosine, dot product).
  - Efficient retrieval of relevant chunks for grounding.

Key idea: Store once (offline), query many times (online).

Vector Storage Step

```
from langchain.vectorstores import Chroma
from langchain.embeddings import OpenAIEmbeddings
from langchain.text_splitter import CharacterTextSplitter

# Example document
text = "The cat sits outside. It is sunny today. The dog barks
↪  loudly."
chunks = CharacterTextSplitter(chunk_size=40,
↪  chunk_overlap=0).split_text(text)

# Embedding model
embedding_model = OpenAIEmbeddings()

# Store chunks + embeddings in Chroma vector DB
vectorstore = Chroma.from_texts(chunks, embedding_model)

# Example query
query = "What is the weather like?"
docs = vectorstore.similarity_search(query, k=2)

for d in docs:
    print(d.page_content)
```

Phase 2: Retrieval & Generation (Online)



Query Embedding Step

Query $\rightarrow$ Embedding $\rightarrow$ Vector DB similarity search.

- When a user submits a **query**, it is also converted into an embedding vector.
- This embedding captures the **semantic meaning** of the query.
- The query vector is then compared (via similarity search) to the stored document vectors.
- The most similar chunks are retrieved and provided to the LLM as context.
- **Key idea:** Questions and documents live in the **same vector space**.

Retrieval Step

Query vector $\rightarrow$ similarity search $\rightarrow$ top-k chunks.

- The query embedding is compared with all stored document embeddings.
- A similarity function (e.g., cosine similarity) measures closeness in the vector space.
- The system retrieves the **top-k most relevant chunks**.
- Retrieved chunks are injected into the LLM prompt as additional context.
- **Key idea:** Retrieval bridges the user query with the most useful knowledge.

Prompt Construction Step

Instruction + Context + Question $\rightarrow$ LLM input

- Retrieved chunks are concatenated with the user query.
- The combined text forms the **augmented prompt** sent to the LLM.
- Ensures that generation is grounded in **relevant external knowledge**.
- Prompt typically includes:
  - **Instruction:** what the model should do.
  - **Context:** retrieved chunks from the vector DB.
  - **Question:** the user's original query.
- **Key idea:** Retrieval + Query $\rightarrow$ Prompt for grounded generation.

Generation Step

LLM + augmented prompt $\Rightarrow$ grounded response.

- The augmented prompt (instruction + retrieved chunks + user query) is sent to the LLM.
- The model generates a **grounded answer**, combining fluency with retrieved evidence.
- Output may include:
  - **Direct answer** to the user's query.
  - **Citations or references** from the retrieved chunks.
  - **Structured formats** (tables, JSON, summaries), depending on the task.
- **Key idea:** The LLM no longer relies only on pretraining — it reasons over the retrieved knowledge.

Example Code (LangChain)

```python
from langchain.vectorstores import Chroma
from langchain.embeddings import OpenAIEmbeddings
from langchain.chains import RetrievalQA
from utils import get_llm  # helper for model selection

# Build index
vectorstore = Chroma.from_documents(docs, embedding=OpenAIEmbeddings())

# Create retriever
retriever = vectorstore.as_retriever(search_kwargs={"k": 3})

# RAG pipeline
qa = RetrievalQA.from_chain_type(llm=get_llm(), retriever=retriever)
qa.invoke({"query": "Summarize the main differences between RAG and fine-tuning"})
```

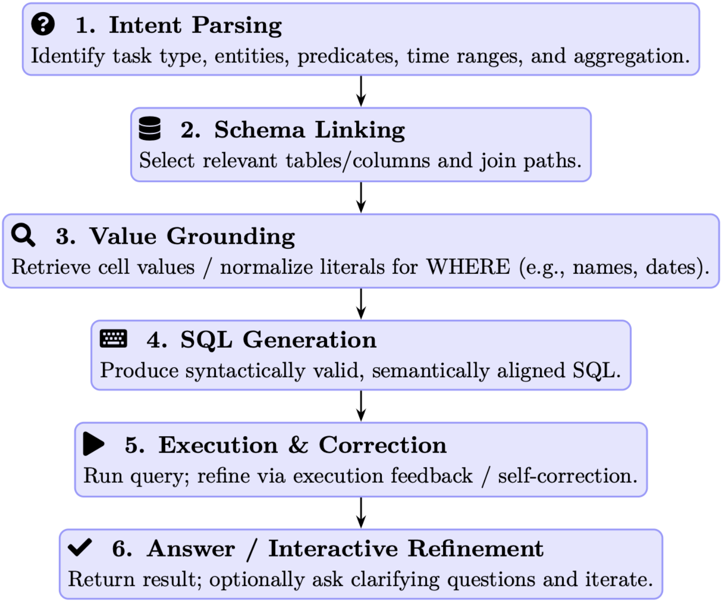
Demo Time

Text-to-SQL

Motivation

- Goal: Convert natural language queries into executable SQL statements.
- Enables non-technical users to query relational databases.
- Applications:
  - Business intelligence dashboards.
  - Conversational assistants and chatbots.
  - Self-service data exploration.
- In LLM-based agents, Text-to-SQL is treated as a **tool call**.

Pipeline Overview



Intent Parsing Step

Natural language query $\rightarrow$ task type + entities + constraints

- The model interprets the **user query** in natural language.
- Identifies:
  - **Task type:** aggregation, filtering, joining, etc.
  - **Entities:** tables or attributes referenced.
  - **Constraints:** conditions, time ranges, limits.
- **Key idea:** Translate free text into a structured representation of the intent.

Schema Linking Step

Query intent $\rightarrow$ relevant schema elements

- Maps terms from the query to the **database schema**.
- Identifies relevant:
  - **Tables** mentioned explicitly or implicitly.
  - **Columns** needed for filtering, joining, or selection.
- Example: "customers by sales" $\rightarrow$ tables `customers`, `orders`.
- **Key idea:** Ground natural language entities in the actual schema.

Schema Linking Step

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

Schema Linking Examples

Prompt (Figure ref.)	Relevant schema elements
Figure 1.8a	`customers: customer_id, name` `orders: customer_id, total_amount`
Figure 1.8b	`tickets: ticket_id, closed_date`

Value Grounding Step

Ambiguous references $\rightarrow$ precise values

- Resolves vague or relative references in the query.
- Examples:
  - "last month" $\rightarrow$ specific date range.
  - "Apple" $\rightarrow$ entity = Apple Inc.
- Ensures SQL conditions use the correct values and formats.
- **Key idea:** Map natural language mentions into concrete database values.

Value Grounding Step

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.

SQL Generation Step

Structured intent + schema + values $\rightarrow$ SQL query

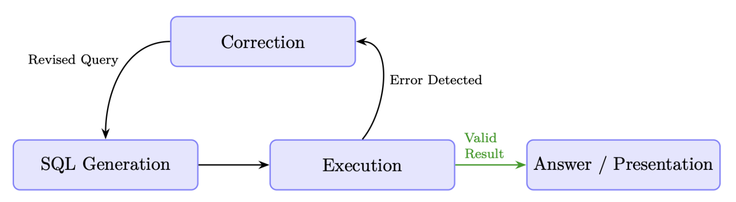
- The model generates an executable **SQL statement**.
- SQL includes:
  - **SELECT / WHERE / GROUP BY / ORDER BY**.
  - **JOINs** between relevant tables.
- Example: "Top 5 customers by sales in Q2" $\rightarrow$ SQL with aggregation and LIMIT.
- **Key idea:** Convert structured understanding into a valid SQL program.

Execution and Correction Step

Generated SQL $\rightarrow$ execution + iterative repair

- Execute the SQL query on the database.
- If an error occurs:
  - The error message is returned to the model.
  - The model attempts to **repair and re-execute**.
- Loop continues until a valid result is obtained.
- **Key idea:** Feedback loop ensures correctness and robustness.

Execution and Correction Step



Iterative process: generate → execute → correct until valid result.

Answer & Refinement Step

SQL result $\rightarrow$ natural language answer

- The raw SQL result is transformed into a **user-friendly response**.
- Model may:
  - Summarize numerical results.
  - Explain trends or comparisons.
  - Support follow-up queries in context.
- Example: SQL result = 5 rows $\rightarrow$ Answer = "The top 5 customers are A, B, C, D, E."
- **Key idea:** Bridge database outputs and natural language interaction.

Step 6: Answer and Refinement

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.

SQL results reformulated into natural language; supports follow-up queries.

Demo Time

Final Remarks

The year is 1998!

> 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)*

27 years later, LLMs gave new life to this vision.

Current Challenges

Hallucination and factuality
Models may produce fluent but factually incorrect information.

Privacy and security
Sensitive data can be leaked or misused if not properly controlled.

Transparency (reasoning trace)
Understanding how answers are derived remains difficult.

Bias and ethical alignment
Outputs may reinforce social biases and require value alignment.

Future Trends

Collaborative multi-agents
Multiple agents coordinating to solve complex tasks together.

Multimodality (text, image, voice, video)
Seamless integration across diverse input and output modalities.

Persistent memories
Long-term memory enabling continuity across sessions.

Domain specialization
Tailored models optimized for specific industries or fields.

References

Rylan Schaeffer, Brando Miranda, and Sanmi Koyejo. Are emergent abilities of large language models a mirage? *arXiv preprint arXiv:2304.15004*, 2023. URL `https://arxiv.org/abs/2304.15004`.

Jason Wei, Yi Tay, Rishi Bommasani, Colin Raffel, Barret Zoph, Sebastian Borgeaud, Dani Yogatama, Maarten Bosma, Denny Zhou, Donald Metzler, Ed H. Chi, Tatsunori B. Hashimoto, Oriol Vinyals, Percy Liang, Jeff Dean, and William Fedus. Emergent abilities of large language models. *arXiv preprint arXiv:2206.07682*, 2022. URL `https://arxiv.org/abs/2206.07682`.

Shunyu Yao, Jeffrey Zhao, Dian Yu, Nan Du, Wen-tau Yu, Izhak Shafran, Thomas L. Griffiths, Graham Neubig, Claire Cao, and Karthik Narasimhan. React: Synergizing reasoning and acting in language models. *arXiv preprint arXiv:2210.03629*, 2022. URL `https://arxiv.org/abs/2210.03629`.

Backup slides

In-Context Learning (ICL)

- **Definition:** LLMs can learn a task from examples given in the prompt, without updating model weights.
- Acts like a “mini training session” at inference time.

Prompt (examples inside context):

```
``I love this movie!'' → Positive
``This was a terrible day.'' → Negative
``The food was amazing!'' → ?
```

Model output:

```
Positive
```

Key idea: The model generalizes from the given examples in the same prompt.

FAQ: Tool Calling & Orchestration

If LLMs are stateless, how do they “know” tools?

Tool specs are injected in the **prompt context** by the orchestrator at every step.

Who ensures the tool call format is correct?

The orchestrator validates JSON and can re-prompt the LLM if malformed.

Do we need to re-register tools every call?

No — only once per session or when available tools change.

FAQ: Retrieval-Augmented Generation (RAG)

Why agents if RAG already works?

Agents integrate RAG with other tools and reasoning loops.

What is the difference between “plain RAG” and agent RAG?

Plain RAG = query + retrieval + generation.

Agent RAG = retrieval as one **step in a multi-tool process**.

How to choose chunk size and embeddings?

Balance context size vs. relevance. Evaluate with retrieval benchmarks.

FAQ: Text-to-SQL

Does the agent “understand” SQL?

No — it maps natural language into SQL patterns using context.

What if the query fails?

The orchestrator passes back the error; the LLM attempts **self-correction**.

How do we prevent dangerous queries (DROP, DELETE)?

Apply schema filters, allowlist queries, or sandbox execution.

FAQ: Challenges & Risks

❓ What about hallucinated tools?

Orchestrator ignores unregistered calls; validates all outputs.

❓ How to protect sensitive data?

Enforce access control, redact inputs, keep logs.

❓ Who is responsible if the agent makes a mistake?

Responsibility lies in the **system design**, not the LLM alone.

FAQ: Practical Use

Which framework should I use?

LangChain, LangGraph, Semantic Kernel — depends on ecosystem.

Can this run locally?

Yes — with open-source LLMs (e.g., LLaMA, Mistral, Ollama).

What about cost?

Depends on model size and call volume. Local models reduce API costs.

How to evaluate quality?

Use **task success rate**, hallucination tests, and domain metrics.