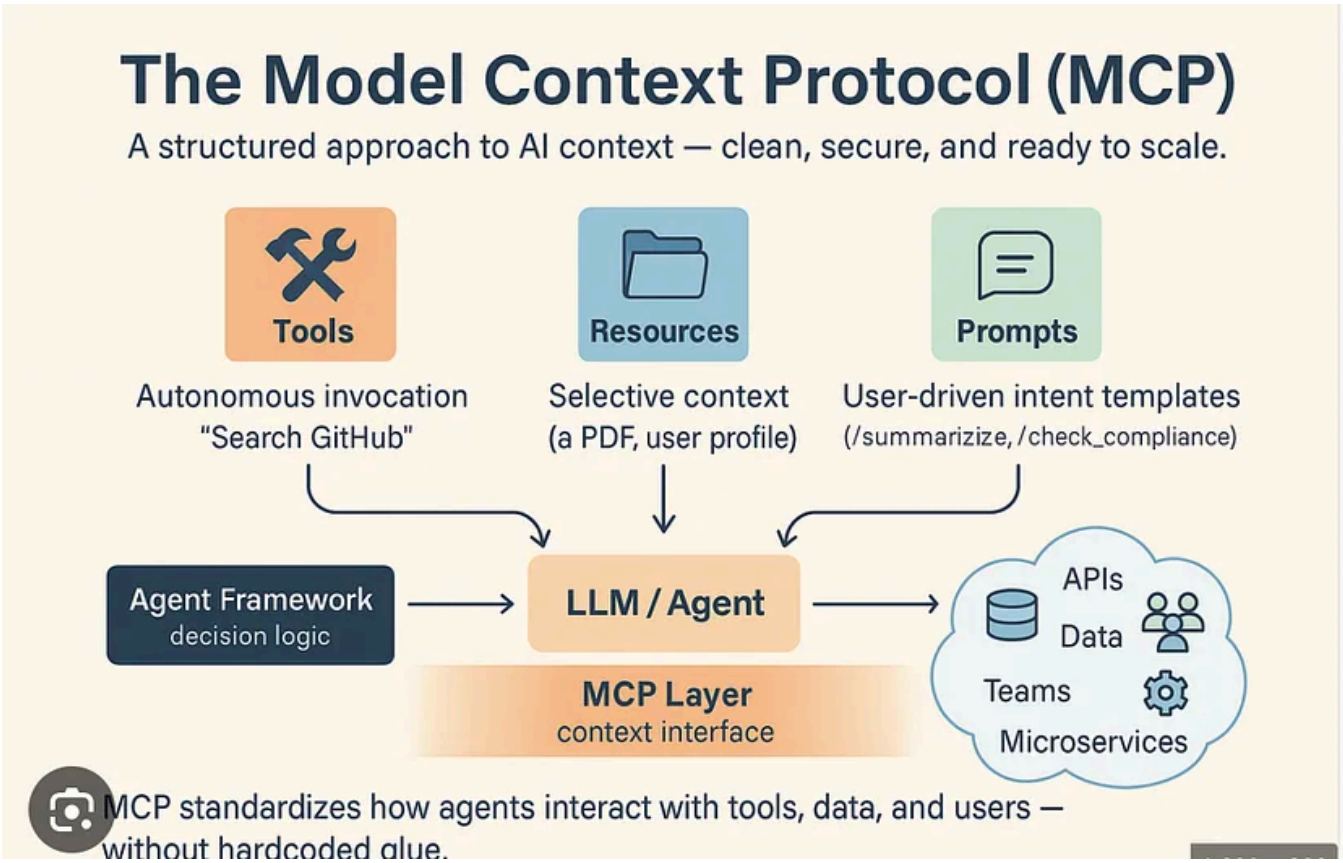
Model Context Protocol (MCP) in AI Agent Development :Text To Sql

6 min read · Apr 17, 2025

Sanjeeb Panda Follow



Introduction

The Model Context Protocol (MCP) is an open standard developed by Anthropic to enable seamless integration between Large Language Models (LLMs) and external data sources, tools, and services. By acting as a “universal connector” for AI applications, MCP allows AI agents to dynamically access real-time data, execute actions, and interact with external systems like file systems, APIs, or databases. This

blog explores how MCP is implemented in the lama_dev project (**code repo**) to build intelligent AI agents and highlights the benefits of using MCP in such applications.

The lama_dev project is part of a broader repository focused on AI learning, specifically leveraging the LLaMA model and integrating MCP to enhance agentic capabilities. This documentation provides an overview of the project, explains how MCP is used, and discusses the advantages of adopting MCP for AI development.

What is the Model Context Protocol (MCP)?

MCP is a standardized protocol that enables AI models to interact with external systems through lightweight servers (MCP servers). It provides a structured way for AI agents to:

- Discover tools: Dynamically identify available tools and their capabilities.
- Execute actions: Call specific tools with appropriate parameters to fetch data or perform tasks.
- Maintain context: Integrate real-time data into the AI's reasoning process for more accurate responses.

MCP operates using JSON-RPC 2.0 for communication and supports two transport mechanisms:

- STDIO: For local integrations where the server runs as a subprocess.
- HTTP+SSE (or Streamable HTTP): For remote connections, allowing streaming of data.

Think of MCP as a "USB-C for AI" — a universal interface that simplifies connecting AI models to diverse tools, from file systems to cloud APIs, without requiring custom integrations for each.

How MCP is used in the code base

In our text-to-SQL application, the Multi-Context Processing (MCP) framework serves as the backbone for orchestrating an agentic process that transforms natural language questions into executable SQL queries with high accuracy.

This client centralizes three critical tools:

1. retrieve_schema

2. list_tables,

3. execute_sql_query

The LangChain ReAct agent leverages to interact with a SQLite database (sample.db). The MCPClient class acts as a dispatcher, routing tool calls with structured inputs (e.g., {“query”: “SELECT COUNT(DISTINCT Producer) FROM airplanes;”, “db”: db}) to their respective functions.

For example, when answering “How many unique airplane producers are there?”, the agent uses the

retrieve_schema tool to fetch the airplanes table schema (CREATE TABLE airplanes (Airplane_id INT(10) PRIMARY KEY, Producer VARCHAR(20), Type VARCHAR(10));)

execute_sql_query tool to run SELECT COUNT(DISTINCT Producer) FROM airplanes;, returning [(3)]. This MCP-driven approach ensures the agent operates within a rich contextual environment, seamlessly integrating schema awareness, database metadata, and query execution.

This MCP-driven approach ensures the agent operates within a rich contextual environment, seamlessly integrating schema awareness, database metadata, and query execution.

```
"""
src/tools/mcp_tools.py: Defines tools for text-to-SQL operations.
- Uses direct function calls instead of fastmcp for simplicity.
- Includes a LangChain-compatible SchemaRetrievalTool with Pydantic fields.
"""

from langchain_core.tools import BaseTool
from pydantic import Field
from src.config.config import CONFIG
from src.core import schema_retrieval
from src.utils.logging import setup_logging

logger = setup_logging(__name__)

def execute_sql_query(query: str, db) -> str:
    """Execute a SQL query and return results or error message."""
    try:
        result = db.run(query)
        return str(result)
```

```
        except Exception as e:
            return f"Error: {str(e)}"

    def list_tables(db) -> str:
        """List available tables in the database."""
        return str(db.get_usable_table_names())

    def retrieve_schema_tool(query: str, vector_store) -> str:
        """Retrieve relevant schema from ChromaDB."""
        if vector_store is None:
            # Fallback: Use schema from config if vector_store is not provided
            logger.info("Using fallback schema from CONFIG")
            return CONFIG["schema"]
        return schema_retrieval.retrieve_schema(vector_store, query)

    class MCPClient:
        def call_tool(self, tool_name: str, args: dict):
            """Mock MCP client to call tools directly."""
            if tool_name == "execute_sql_query":
                return execute_sql_query(args["query"], args["db"])
            elif tool_name == "list_tables":
                return list_tables(args["db"])
            elif tool_name == "retrieve_schema":
                return retrieve_schema_tool(args["query"], args.get("vector_store")
            raise ValueError(f"Unknown tool: {tool_name}")

    def initialize_mcp_tools(db, vector_store):
        """Initialize mock MCP client."""
        return MCPClient()

    class SchemaRetrievalTool(BaseTool):
        name: str = "schema_retrieval"
        description: str = "Retrieve relevant database schema for a given query."
        mcp_client: object = Field(description="MCP client for schema retrieval")

        def __init__(self, mcp_client):
            super().__init__(mcp_client=mcp_client)

        def _run(self, query: str) -> str:
            """Synchronous execution of schema retrieval."""
            return self.mcp_client.call_tool("retrieve_schema", {"query": query, "v

        async def _arun(self, query: str) -> str:
            """Asynchronous execution (not used, but required by BaseTool)."""
            return self._run(query)
```

The agentic process is powered by the ReAct (Reasoning and Acting) framework, where MCP's tools enable iterative reasoning and action cycles.

In src/agents/agent.py,

1. the ReAct agent is initialized with a prompt that instructs it to reason about the question, invoke MCP tools, and verify results.
2. The SchemaRetrievalTool, a LangChain-compatible tool inheriting from BaseTool, uses the MCP client to fetch the schema when vector_store is unavailable, falling back to the schema defined in src/config/config.py.
3. The ListSQLDatabaseTool and QuerySQLDataBaseTool from LangChain, integrated via MCP, allow the agent to confirm the presence of the airplanes table and execute queries, respectively. For instance, the agent's reasoning trace shows it first reasoning about the need to count distinct Producer values, then invoking sql_db_query with the generated query, observing the result ([(3)]), and verifying it aligns with the schema.
4. If errors occur (e.g., a syntax issue), the agent retries up to five iterations, using MCP's contextual tools to refine the query, such as checking table names or schema details. This iterative process, facilitated by MCP's ability to maintain and process multiple contexts, ensures robust query generation and execution.

```
"""
src/agents/agent.py: Initializes and runs the LangChain ReAct agent for text-to
- Uses tools for SQL execution, table listing, and schema retrieval.
- Executes generated queries and returns accurate results.
- Supports self-correction via error analysis and retries.
"""

from langchain.agents import create_react_agent, AgentExecutor
from langchain_community.tools.sql_database.tool import (
    QuerySQLDataBaseTool,
    ListSQLDatabaseTool,
)
from langchain_core.prompts import PromptTemplate
from src.tools.mcp_tools import SchemaRetrievalTool
from src.utils.logging import setup_logging
from src.config.config import CONFIG
from langchain_community.utilities import SQLDatabase

logger = setup_logging(__name__)

def initialize_agent(llm, db, mcp_client):
    """Initialize LangChain ReAct agent."""
    # LangChain tools
    query_tool = QuerySQLDataBaseTool(db=db)
```

```
    list_tables_tool = ListSQLDatabaseTool(db=db)
    schema_tool = SchemaRetrievalTool(mcp_client)

    # ReAct-compatible prompt template
    SQL_PROMPT = PromptTemplate.from_template(
        """
        You are an expert SQL assistant tasked with answering user questions by
        Your goal is to:
        1. Generate a syntactically correct SQL query to answer the question.
        2. Execute the query using the sql_db_query tool to retrieve the result
        3. Verify the result makes sense given the schema and question.
        If the query fails or the result seems incorrect, analyze the error, cc

        Question: {question}
        Schema: {schema}

        Available tools:
        {tools}

        Tool names: {tool_names}

        Follow this process:
        Thought: [Explain your reasoning for the query]
        Action: sql_db_query
        Input: [The SQL query to execute]

        After execution:
        Observation: [The result returned by the tool]
        Thought: [Verify if the result answers the question correctly]
        Final Answer: [The query result, e.g., [(3)]]

        If the question explicitly asks for only the query, return:
        Final Answer: [SQL query]

        Scratchpad for intermediate steps:
        {agent_scratchpad}
        """
    )

    # Create agent
    agent = create_react_agent(
        llm=llm,
        tools=[query_tool, list_tables_tool, schema_tool],
        prompt=SQL_PROMPT,
    )

    # Create executor
    executor = AgentExecutor(
        agent=agent,
        tools=[query_tool, list_tables_tool, schema_tool],
        verbose=True,
        max_iterations=5,
        handle_parsing_errors=True,
```

```
        )

        logger.info("Initialized LangChain agent")
        return executor

    def process_text_to_sql(executor, mcp_client, question: str) -> dict:
        """Process a user query and return SQL query/result."""
        try:
            schema = mcp_client.call_tool("retrieve_schema", {"query": question, "v
            logger.info(f"Retrieved schema: {schema}")

            result = executor.invoke({"question": question, "schema": schema})
            logger.info(f"Agent result: {result}")

            output = result.get("output", "No result returned.")
            # Extract query and result from output
            query = None
            query_result = output

            # If output contains a query, extract it
            if "SELECT" in output:
                # Try to find the query in the agent's reasoning steps
                for step in result.get("intermediate_steps", []):
                    if step[0].tool == "sql_db_query":
                        query = step[0].tool_input
                        query_result = step[1] if step[1] else output
                        break

            # Verify result by re-executing the query
            if query and query_result and query_result != "No result returned.":
                db = SQLDatabase.from_uri(CONFIG["database_uri"])
                try:
                    verified_result = db.run(query)
                    if verified_result != query_result:
                        logger.warning(f"Agent result {query_result} differs from v
                        query_result = verified_result
                except Exception as e:
                    logger.error(f"Verification failed: {str(e)}")
                    query_result = f"Error verifying result: {str(e)}"

            return {
                "question": question,
                "sql_query": query,
                "result": query_result
            }
        except Exception as e:
            logger.error(f"Error processing query: {str(e)}")
            return {"question": question, "error": str(e)}
```

Please go through **code base** and share your updates. Happy Learning.

Follow

Written by Sanjeeb Panda

65 followers · 14 following

Building scalable Data and ML application at Amazon

Responses (1)

Write a response

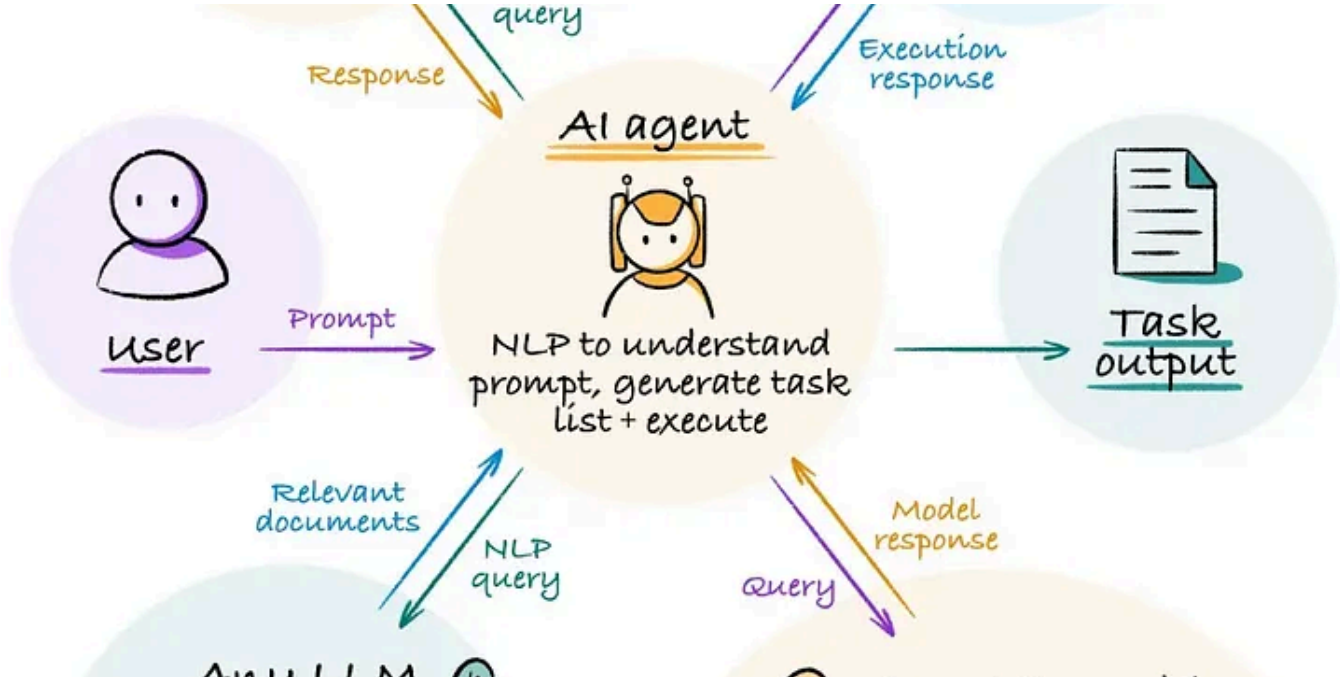
What are your thoughts?

Asish mohanty
Apr 25

good work

Reply

More from Sanjeeb Panda



Sanjeeb Panda

How to Build AI Agents with Amazon Bedrock and Claude: A Beginner's Guide for AI Agents...

Overview

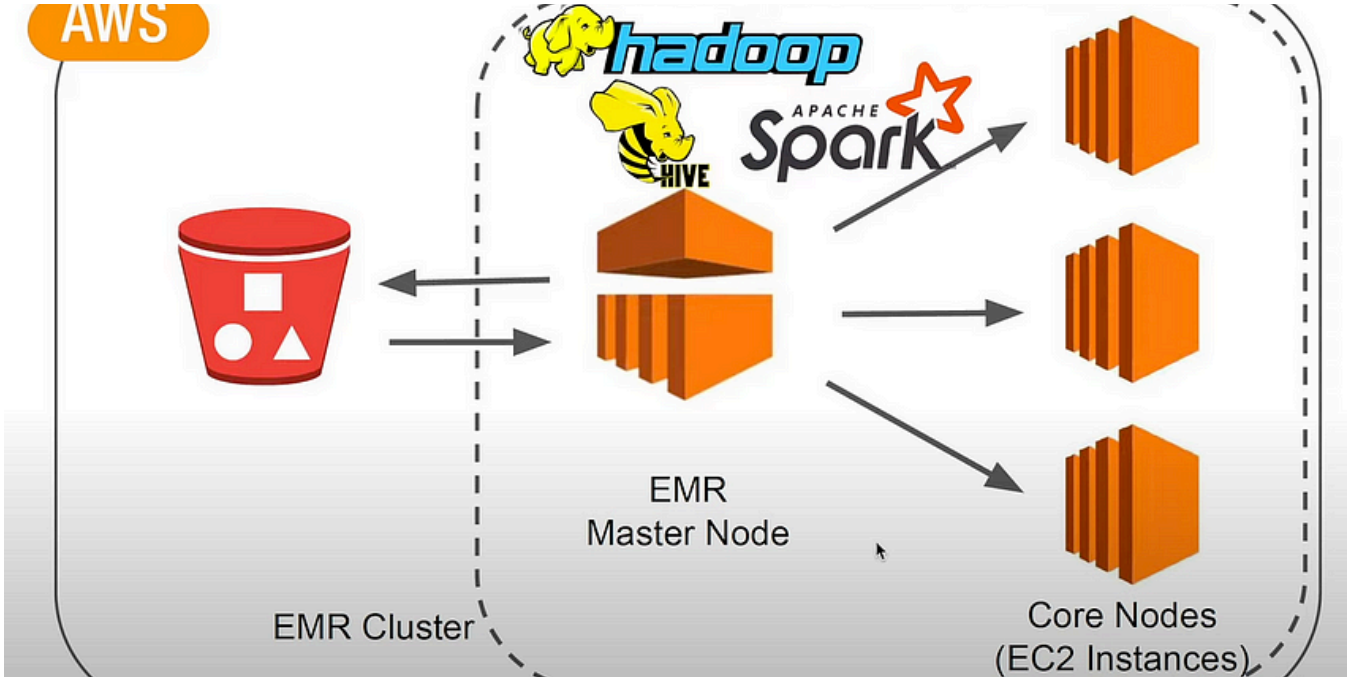
Apr 24 1 1

Sanjeeb Panda

Invoking LLM models using Bedrock from AWS.

Why Amazon Bedrock?

Oct 21, 2023 5

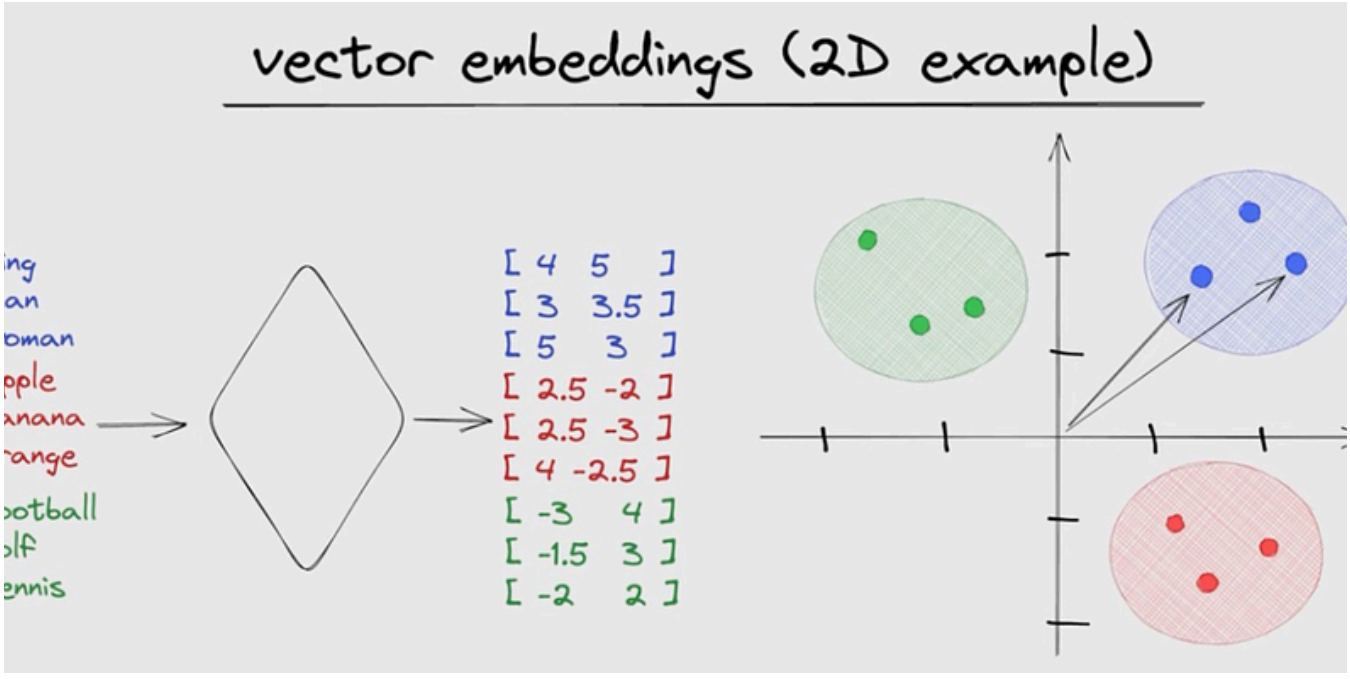


Sanjeeb Panda

How did I reduce the execution time of EMR job from 24 hrs to 4 hr and cost of processing by 50%.-Pa

Overview

Apr 22 1



Sanjeeb Panda

Implement Vector Database Using AWS Bedrock

Overview:

Oct 23, 2023 61

See all from Sanjeeb Panda

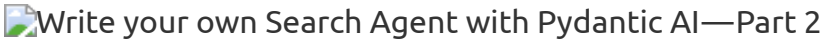
Recommended from Medium

Stefentaime

Optimizing SQL Queries with Semantic Caching and Text-to-SQL Generation

Understanding the Challenge

Apr 14



Bao Nguyen

Write your own Search Agent with Pydantic AI—Part 2

Hello everyone, this is a follow-up of my recent blog about Writing Your Own Search Agent with Pydantic AI. In the earlier post, I...

Jan 12 17 2

Naveen Pandey

Building a Web Browsing AI Agent with Pydantic + MCP

In today's fast-paced digital world, efficiently extracting and summarizing information from websites can be a game-changer. Large language...

May 3 10

In GoPenAI by Sai Abhinav Parvathaneni

LangChain Agents—Giving Your LLMs a Brain, Hands, and a To-Do List

"If a chatbot is like a smart assistant, an agent is like that assistant with internet access, a calculator, and a todo list—and it...

May 1 1

In Data Science Collective by Raphael Schols

Let Users Talk to Your Databases: Build a RAG-Powered SQL Assistant with Streamlit

Create a database-agnostic chatbot that connects to SQLite, BigQuery, and Redshift.

Apr 30 651 4

DevOpsDynamo

Build Your Own MCP Server: A File Summarizer Powered by AI

👉 if you're not a Medium member, read this story for free, here.

May 2 171

See more recommendations