Conversational Finance Assistant with FastMCP Backend and Langchain Agent UI

Tao Jin
https://www.linkedin.com/in/taojin19/

1. Abstract

The primary goal of this project is to develop a conversational finance assistant that provides users with real-time stock quotes, market news, and insights on market movers through natural language interactions. The system is designed as a Minimum Viable Product (MVP) to demonstrate the integration of the Model Context Protocol (MCP) for secure and standardized backend API interactions, and Langchain for frontend agent orchestration. The architecture employs a decoupled design: a secure backend server built with the **FastMCP framework** acts as a standardized gateway to external financial APIs (Finnhub, Alpha Vantage), while a **Streamlit frontend UI** integrates a **Langchain OpenAI Tools agent**. This agent interprets user requests, selects appropriate capabilities exposed by the MCP server (via defined Langchain tools), orchestrates data retrieval through **FastMCP client** calls, and synthesizes conversational responses. The report highlights the strategic benefits derived from using the Model Context Protocol (MCP) architecture, explains key implementation details of the agent-MCP integration, proposes a reusable design pattern, and discusses challenges encountered.

2. Introduction

2.1. Problem Statement:

Accessing diverse, real-time financial data often requires integrating multiple, disparate APIs with varying authentication and data formats. Creating a user-friendly, conversational interface for this data necessitates robust natural language understanding and secure, standardized access to backend resources.

2.2. Project Goal:

To develop a system allowing conversational querying of real-time stock prices, news, and market movers, leveraging the strengths of LLMs for understanding and FastMCP for secure, abstracted data access.

2.3. Challenges Addressed:

- Secure API Management: Safeguarding sensitive API keys and credentials.
- Standardized Integration: Reducing the complexity of integrating multiple AI applications with various tools and data sources.
- Modular Architecture: Facilitating easy maintenance and scalability.
- Enhanced User Experience: Providing intuitive, conversational access to financial data.

2.4. Overview of the Model Context Protocol (MCP)

- **What is MCP?**

The Model Context Protocol (MCP) is an open standard developed by Anthropic to standardize how AI applications connect with external tools, data sources, and systems. It transforms the traditional M×N integration problem—where M AI applications need to integrate with N tools, requiring M×N custom integrations—into a more manageable M+N problem. This is achieved by establishing a common protocol where:

- Hosts: AI applications (e.g., chatbots, IDE assistants) that users interact with.
- Clients: Components within the host applications that manage connections to MCP servers.
- Servers: External programs exposing tools, resources, and prompts via a standardized API. This architecture allows for modular, scalable, and secure integration between AI applications and external systems.
- **Core Components of MCP**
    - Tools (Model-controlled): Functions that AI models can invoke to perform specific actions, such as fetching stock prices.
    - Resources (Application-controlled): Data sources that AI models can access, akin to RESTful GET endpoints.
    - Prompts (User-controlled): Predefined templates guiding the AI's interactions with tools and resources.

These components enable a structured and efficient interaction between AI models and external systems.

- **Why MCP? (Benefits & Rationale)**

The Model Context Protocol (MCP) was chosen for the backend interface, aligning with principles often highlighted in MCP introductions: - **Standardization:** MCP provides a common "language" for the frontend agent to interact with backend capabilities, regardless of whether the data comes from Finnhub, Alpha Vantage, or future sources. The agent interacts with consistent MCP tools/resources (`get_price`, `news://...`), not specific vendor APIs. - **Security:** A primary driver. Sensitive API keys (`FINNHUB_API_KEY`, `ALPHA_VANTAGE_API_KEY`) reside solely on the secure MCP server, never exposed to the UI, the LLM, or the end-user's browser. This is crucial for handling potentially paid API credentials. - **Abstraction & Decoupling:** The MCP server hides the complexity of calling different financial APIs. The UI/Agent only needs to know the MCP interface. This decoupling allows the backend data sources or fetching logic to change without requiring modifications to the frontend agent, promoting maintainability. - **Discoverability (Potential):** While not explicitly used by the Langchain agent in this implementation (as tools were predefined), MCP allows clients to dynamically query a server's capabilities (`listTools`, `listResources`), enabling more flexible agent interactions in other scenarios. - **Control & Ownership:** The owner of the MCP server maintains full control over the exposed tools, data transformations, and API interactions.

2.5 Technology Stack:

- **Backend Server:** Python 3.10+, FastMCP (`fin_server_v2.py`).
- **Financial Data APIs:** Finnhub, Alpha Vantage.
- **Frontend UI:** Python 3.10+, Streamlit (`fin_langchain_v2.py`)
- **Agent Framework:** Langchain (`langchain`, `langchain-openai`, `langchainhub`)
- **LLM:** OpenAI API (`gpt-4-turbo-preview`) via `langchain-openai`
- **MCP Client:** `fastmcp.Client`.

- **Supporting:** `httpx`, `python-dotenv`, `pydantic-settings`, `pydantic`, `asyncio`, `openai`.

3. System Architecture

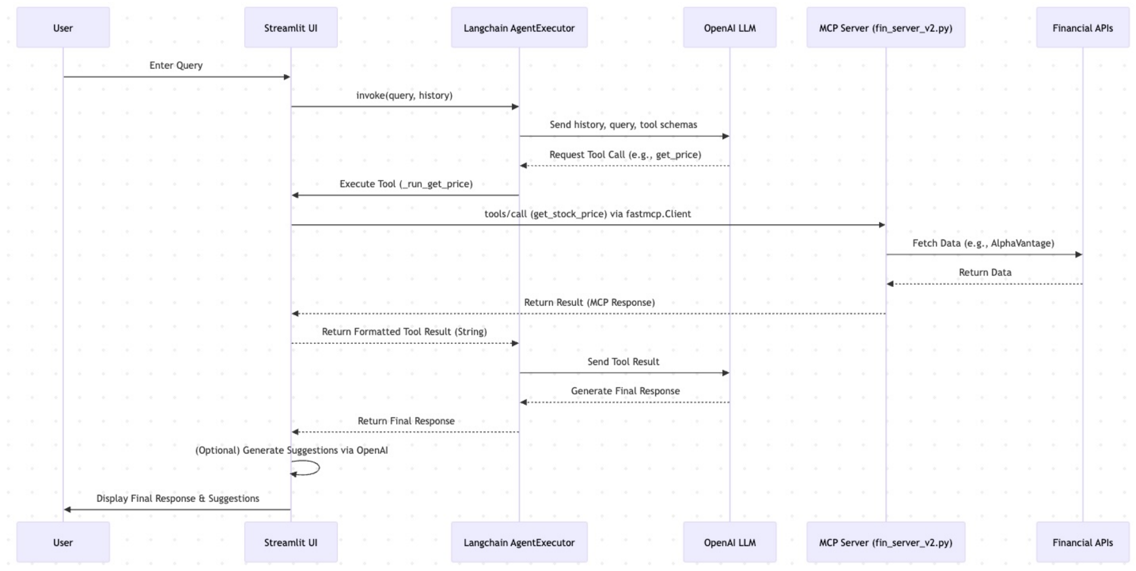
3.1 High-Level Architecture The system is divided into two main components:

- Frontend: A Streamlit-based UI employing Langchain to manage user interactions and orchestrate tool usage.
- Backend: A FastMCP server handling secure API interactions with external financial data providers like Finnhub and Alpha Vantage.

This separation ensures modularity, security, and ease of maintenance.

The system utilizes a decoupled client-server architecture orchestrated by the Langchain agent within the Streamlit UI.

3.2 Data Flow:


User
Streamlit UI
Langchain AgentExecutor
OpenAI LLM
MCP Server (fin_server_v2.py)
Financial APIs
Enter Query
invoke(query, history)
Send history, query, tool schemas
Request Tool Call (e.g., get_price)
Execute Tool (_run_get_price)
tools/call (get_stock_price) via fastmcp.Client
Fetch Data (e.g., AlphaVantage)
Return Data
Return Result (MCP Response)
Return Formatted Tool Result (String)
Send Tool Result
Generate Final Response
Return Final Response
(Optional) Generate Suggestions via OpenAI
Display Final Response & Suggestions
User
Streamlit UI
Langchain AgentExecutor
OpenAI LLM
MCP Server (fin_server_v2.py)
Financial APIs

1. User query enters Streamlit UI.
2. UI adds query to history and invokes the **Langchain Agent Executor**.
3. Executor combines input, memory (`ConversationBufferWindowMemory`), and agent prompt (from Langchain Hub, customized with system instructions).
4. Executor sends this to the LLM (`ChatOpenAI`).
5. LLM plans: either generates a direct response or selects a tool (`get_price`, `get_news`, `get_market_movers`) based on its description and the prompt.
6. If a tool is selected, Executor invokes the corresponding **Langchain StructuredTool** (`_run_get_price`, `_run_get_news`, `_run_get_market_movers`).
7. The tool's coroutine (`_run_...`) calls the core logic function (`_get_price_logic`, etc.).
8. The core logic function calls the appropriate **MCP Client Helper** (`call_mcp_tool` or `read_mcp_resource`).
9. The MCP Client Helper uses `fastmcp.Client` to send the request (e.g., `tools/call` for `get_stock_price`) to the **FastMCP Server**.
10. The FastMCP Server executes the relevant `@mcp.tool` or `@mcp.resource` function, calling external APIs via `httpx`.

11. The FastMCP Server returns the result (data dict or error dict) via MCP.
12. The MCP Client Helper receives the result.
13. The core logic function receives the result dictionary.
14. The Langchain Tool wrapper (`_run_...`) **formats** this dictionary into a user-friendly markdown string using `format_price_response` , `format_news_response` , or `format_movers_response` .
15. The `AgentExecutor` receives this formatted string as the tool's output.
16. The executor sends the formatted output back to the LLM (as a `ToolMessage`).
17. The LLM synthesizes the final natural language response using the formatted tool output.
18. The executor returns the final response text.
19. The UI displays the final response and triggers suggestion generation.
20. A separate OpenAI call generates follow-up suggestions.
21. Suggestions are displayed as buttons.

4. MCP Server Design

4.1 Implementation with FastMCP

The MCP server is implemented using the FastMCP framework in Python. It defines tools and resources using decorators:

- `@mcp.tool` : Defines functions that perform actions, such as retrieving stock prices.
- `@mcp.resource` : Defines functions that provide data resources, like news articles.

Example:

```python
from fastmcp import MCP

mcp = MCP()

@mcp.tool()
def get_stock_price(symbol: str) -> dict:
    """Fetches the current stock price for a given symbol."""
    # Logic to interact with external API
    return {"symbol": symbol, "price": 100.0}
```

This setup abstracts the complexity of external APIs, providing a standardized interface for the frontend to interact with.

4.2 Security and Abstraction

By centralizing API interactions within the MCP server:

- **API Keys Protection:** Sensitive credentials are kept secure on the server side.
- **Simplified Frontend:** The frontend does not need to handle API-specific logic.
- **Ease of Maintenance:** Updates to API interactions are confined to the server, without affecting the frontend.

5. Langchain Agent Design

5.1 Agent Configuration

The Langchain agent is configured using the `create_openai_tools_agent` function, integrating various tools defined as StructuredTool instances. Each tool corresponds to a specific function that interacts with the MCP server.

Example:

```python
from langchain.agents import create_openai_tools_agent
from langchain.tools import StructuredTool

def fetch_stock_price(symbol: str) -> str:
    # Logic to call MCP client and retrieve stock price
    return f"The current price of {symbol} is $100.0."

get_price_tool = StructuredTool.from_function(
    func=fetch_stock_price,
    name="get_price",
    description="Fetches the current stock price for a given symbol."
)

agent = create_openai_tools_agent(
    tools=[get_price_tool],
    llm=llm_instance,
    verbose=True
)
```

This configuration allows the agent to decide when to invoke tools based on user input.

5.2 Memory and Prompting

The Langchain agent employs sophisticated memory management and prompting strategies to facilitate effective, context-aware interactions with users:

- **ConversationBufferWindowMemory**:
  - Retains a sliding window of recent interactions (e.g., the last 5 to 10 exchanges).
  - Ensures relevant context is provided to the LLM, improving response accuracy while preventing memory overload.

```python
from langchain.memory import ConversationBufferWindowMemory

memory = ConversationBufferWindowMemory(
    k=5,  # Retain the last 5 interactions
    memory_key="chat_history"
)
```

- **Custom System Prompting**:
  - Overrides the default Langchain prompts with tailored instructions that clearly define:
    - The capabilities and constraints of available tools.
    - The expected format of responses.
    - How to gracefully handle out-of-scope or ambiguous requests.

```
from langchain.prompts import ChatPromptTemplate, SystemMessagePromptTemplate

system_message = SystemMessagePromptTemplate.from_template(
    "You are a finance assistant capable of fetching real-time stock prices, news, and
market movers. "
    "If a query falls outside these capabilities, politely decline to respond."
)

prompt = ChatPromptTemplate.from_messages([system_message])
```

5.3 Error Handling and Robustness

Robustness in agent-tool interactions is critical, and Langchain provides built-in mechanisms for handling errors gracefully:

- **StructuredTool Configuration**:
  - `handle_tool_error=True`: Ensures errors from MCP interactions or external APIs are captured, logged, and returned to the agent for intelligent handling.
  - Error messages are formatted into user-friendly responses, improving transparency and user experience.

```
stock_price_tool = StructuredTool.from_function(
    func=fetch_stock_price,
    args_schema=StockPriceSchema,
    description="Fetches current stock prices.",
    handle_tool_error=True
)
```

5.4 Agent Executor

The `AgentExecutor` class orchestrates the agent's workflow, encapsulating interactions between user inputs, LLM decisions, tool execution, and response synthesis:

- **Verbose Logging**:
  - Enabled for comprehensive debugging, tracking each decision made by the LLM and tool invocations.
- **Async Invocation**:
  - Uses `ainvoke()` for asynchronous, non-blocking interactions with the MCP server and external APIs, ensuring responsiveness.

```
from langchain.agents import AgentExecutor

agent_executor = AgentExecutor(
    agent=agent,
    tools=[stock_price_tool, news_tool, market_movers_tool],
    memory=memory,
    verbose=True,
    handle_parsing_errors=True
)

response = await agent_executor.ainvoke(user_input)
```

6. Integration Between Langchain Agent and MCP Server

6.1 Interaction Flow

The integration between the Langchain agent and the MCP server follows a clear interaction flow:

- **Intent Recognition**:
  - Langchain's LLM identifies user intent from natural language input.
- **Structured Tool Invocation**:
  - Based on intent, the appropriate `StructuredTool` coroutine function is triggered.
- **MCP Client Communication**:
  - The structured tool communicates via the MCP client (`fastmcp.Client`), encapsulating all MCP interactions.

```python
from fastmcp import Client

async def fetch_stock_price(symbol: str):
    async with Client(base_url="http://mcp_server") as client:
        response = await client.call("tools/get_stock_price", {"symbol": symbol})
    return response
```

- **MCP Server Request Handling**:
  - The MCP server receives the structured request, invokes the corresponding `@mcp.tool` or `@mcp.resource` endpoint, fetches external API data, validates responses, and returns results.

```python
@mcp.tool()
def get_stock_price(symbol: str):
    response = httpx.get(f"https://api.finnhub.io/.../quote?symbol={symbol}", headers=
{"X-API-Key": FINNHUB_API_KEY})
    validated_data = StockQuote(**response.json())
    return validated_data.dict()
```

- **Data Formatting**:
  - The MCP client retrieves the MCP server's response, formats it into structured markdown or JSON, and provides it to the Langchain agent.
- **Response Synthesis**:
  - The Langchain agent receives the formatted data, synthesizes a natural language response via the LLM, and returns it to the user via the UI.

6.2 Key Integration Considerations

- **Consistency in Data Formats**:
  - Ensure all structured tools follow consistent input/output schemas (defined via Pydantic models).

    - Facilitates easier validation and error handling.
- **Error Transparency**:
    - Transparently pass errors from the MCP server through the Langchain agent, enabling meaningful user communication and debugging.
- **Scalability and Extensibility**:
    - Maintain clear modularity and separation between frontend (agent) logic and backend (MCP) API handling.
    - Simplifies future expansions or adjustments (new APIs, additional tools/resources).

7. Component Implementation Details

Finance MCP Server (`fin_server_v2.py`)

- **MCP Tool Implementation (`@mcp.tool()`):**

```
@mcp.tool()
async def get_market_movers(ctx: Context, limit_per_category: int = 5) ->
Dict[str, Any]:
    """
    Gets the lists of top gaining, top losing, and most actively traded
stocks...
    """
    # 1. Check API Key (using settings)
    if not settings.alpha_vantage_api_key:
        await ctx.error("Alpha Vantage API key is not configured.")
        return {"error": "Alpha Vantage API key not configured"}
    # 2. Log start (using ctx for MCP client visibility)
    await ctx.info(f"Tool Call: get_market_movers (limit:
{limit_per_category})")
    try:
        # 3. Make external API call (using httpx)
        params = {"function": "TOP_GAINERS_LOSERS", "apikey":
settings.alpha_vantage_api_key}
        response = await http_client.get("https://www.alphavantage.co/query",
params=params)
        response.raise_for_status()
        data = response.json()
        # 4. Handle API-specific responses/errors
        if not data or ("Note" in data or "Information" in data):
            # ... handle rate limits / empty data ...
            return {"error": "..."}
        # 5. Validate/Process data (using Pydantic)
        movers = MarketMoversData.model_validate(data)
        output_data = {
            "top_gainers": [g.model_dump() for g in
movers.top_gainers[:limit_per_category]],
            # ... process other categories ...
            "last_updated": movers.last_updated
        }
        await ctx.info(f"Success fetching market movers...")
        # 6. Return structured data (or error dict)
        return output_data
    except httpx.HTTPStatusError as e:
        await ctx.error(...)
        return {"error": ...}
    except Exception as e:
        await ctx.error(...)
        return {"error": ...}
```

Explanation: The `@mcp.tool()` decorator registers the function. FastMCP uses the function signature (`limit_per_category: int = 5`) and docstring to create the tool definition for MCP clients. The function performs the core logic (API call, validation) and returns a dictionary, which FastMCP serializes and sends back. `ctx` allows logging *within the MCP context*.

- **MCP Resource Implementation (`@mcp.resource()`):**

```
@mcp.resource(uri="news://ticker/{ticker}", description="...")
async def get_ticker_news(ticker: str) -> List[Dict[str, Any]]: # No ctx
    """Fetches news headlines for a specific ticker..."""
    # 1. Check API Key (using settings)
    if not settings.finnhub_api_key:
        log.error(...) # Use standard logging
        return [{"error": "..."}]
    log.info(f"Resource Call: Fetching news for ticker {ticker}...") # Standard
logging
    try:
        # 2. Make external API call (using httpx)
        # ... params setup ...
        response = await http_client.get("...", params=params)
        response.raise_for_status()
        news_list = response.json()
        # 3. Validate/Process data
        # ... validation and formatting ...
        return validated_news
    except Exception as e:
        log.exception(...) # Standard logging
        return [{"error": "..."}]
```

Explanation: `@mcp.resource()` registers this as a template. FastMCP extracts `ticker` from the requested URI (`news://ticker/AAPL`) and passes it to the function. Since `ctx` was problematic, standard logging is used. It returns a list of dictionaries, serialized by FastMCP.

Streamlit UI (`fin_langchain_v2.py`)

- **Langchain Tool Definition (`StructuredTool`):**

```
tools_list = [
    StructuredTool.from_function(
        func=None, # No sync version
        coroutine=_get_market_movers_logic, # Points to the core logic function
        name="get_market_movers",
        description="Gets lists of the current top gaining...", # Used by LLM
        args_schema=GetMarketMoversSchema, # Pydantic schema for LLM arg
generation
        return_direct=False, # Agent receives the dict result
        handle_tool_error=True # Agent handles exceptions raised by the logic
function
    ),
    # ... other tools defined similarly ...
]
```

Explanation: This defines the tool for the *Langchain Agent*. `StructuredTool` uses the `args_schema` (Pydantic model) to tell the LLM what arguments are expected. When the LLM decides to call this tool, the `AgentExecutor` runs the specified `coroutine` (`_get_market_movers_logic`). `handle_tool_error=True` ensures that if `_get_market_movers_logic` raises an exception (e.g., because the MCP call failed), the agent can process it.

- **Tool Core Logic & MCP Call:**

```python
async def _get_market_movers_logic(limit_per_category: int = 5) -> dict:
    """Core logic: Gets market movers dict from MCP server."""
    log_ui.info(f"Executing tool logic 'get_market_movers' with limit:
{limit_per_category}")
    # Calls the MCP helper function, passing args
    movers_data = await call_mcp_tool("get_market_movers",
{"limit_per_category": limit_per_category})
    log_ui.info(f"MCP tool 'get_market_movers' returned: {movers_data}")
    # Raise exception if MCP tool returned an error dictionary
    if isinstance(movers_data, dict) and "error" in movers_data:
        raise Exception(movers_data["error"])
    return movers_data # Return raw dict to the Langchain Tool wrapper
```

Explanation: This is the async function executed by the Langchain `StructuredTool`. Its sole purpose is to call the appropriate MCP helper (`call_mcp_tool`) with the correct arguments received from the LLM (via Langchain). It then returns the dictionary result from the MCP server or raises an exception if the helper indicated an error.

- **Agent Executor Setup:**

```python
llm = ChatOpenAI(model=OPENAI_MODEL, temperature=0.1)
prompt_template = hub.pull(AGENT_PROMPT_HUB_REPO)
# ... (prompt customization and placeholder checks) ...
agent = create_openai_tools_agent(llm, tools_list, prompt_template)
st.session_state.agent_executor = AgentExecutor(
    agent=agent,
    tools=tools_list, # Provide the list of StructuredTool objects
    memory=st.session_state.memory,
    verbose=True,
    handle_parsing_errors=True,
    max_iterations=5
)
```

Explanation: This sets up the standard Langchain agent: loading the LLM, getting a base prompt, ensuring necessary placeholders exist, creating the agent logic (`create_openai_tools_agent`), and wrapping it all in an `AgentExecutor` which manages the conversation flow, memory, and tool interactions.

8. Demonstration Outcomes and Analysis

8.1 Successful Demonstrations

The MVP effectively demonstrates:

- **Secure and Abstracted API Management**:
  - API keys and complex interactions are secured within the MCP server, completely abstracted from frontend tools.
- **Conversational AI Integration**:
  - Seamless, intuitive interaction via the Langchain agent interface, enhancing user accessibility and engagement.
- **Robust Error Handling and User Experience**:
  - User interactions gracefully handle errors, providing clear, actionable feedback.

8.2 Analysis and Limitations

- **Performance and Latency**:
  - Minor latency observed in async interactions, typically due to external API responsiveness.
- **Data Source Reliability**:
  - Dependency on third-party financial APIs (Finnhub, Alpha Vantage) introduces potential points of failure, requiring robust error handling.
- **Scalability Evaluation**:
  - Architecture supports horizontal scaling (multiple MCP servers) and vertical extension (additional tools/resources).

9. Proposed Design Pattern: "Agent-Mediated Service Abstraction"

Based on this project, a reusable pattern emerges for connecting conversational agents to potentially complex or sensitive backend services:

- **Intent:** Securely expose diverse backend functionalities (APIs, databases, calculations) to an LLM-based conversational agent through a standardized, abstracted interface, without exposing backend implementation details or credentials to the agent or UI.
- **Components:**
  1. **Conversational UI:** (e.g., Streamlit, Gradio, Custom Web App) Handles user interaction, displays conversation, manages session state.

2. **Agent Orchestrator:** (e.g., Langchain `AgentExecutor`, OpenAI direct tool loop logic) Resides within the UI backend. Receives user input, manages conversation history/memory, interacts with the LLM.
3. **LLM:** (e.g., GPT-4, Claude 3,Gemini) The core reasoning engine. Interprets user intent, selects tools, synthesizes responses based on system prompts and tool results.
4. **Agent Tool Definitions:** Schemas provided to the LLM describing available actions (names, descriptions, parameters). In Langchain, these are often derived from `BaseTool` or `StructuredTool`.
5. **Tool Implementation Wrappers (UI Layer):** Python functions (often async) corresponding to the Agent Tool Definitions. These functions are called by the Agent Orchestrator. **Crucially, their primary role is to interact with the MCP Client.** They may also handle basic formatting of results *before* returning them to the orchestrator/LLM if desired (though the final formatting is often best left to the LLM guided by the prompt).
6. **MCP Client:** (`fastmcp.Client`) Used by the Tool Implementation Wrappers to communicate with the MCP Server.
7. **MCP Server:** (e.g., FastMCP application) The backend gateway.
   - Exposes backend capabilities as standardized MCP Tools and Resources/Templates.
   - Handles authentication and interaction with the actual Backend Services/APIs.
   - Contains the core business logic for data fetching and processing.
   - Manages API keys and credentials securely.
8. **Backend Services/APIs:** The actual external data sources or functionalities (e.g., Finnhub, Alpha Vantage, internal databases).

- **Interactions:** User -> UI -> Agent Orchestrator -> LLM -> Agent Orchestrator -> Tool Wrapper -> MCP Client -> MCP Server -> Backend Service -> (Return Path).
- **Benefits:** Security (keys in backend), Abstraction (agent uses MCP tools, not specific APIs), Modularity (UI, Agent, MCP Server, Backend Services are separate), Maintainability (changes to backend APIs only affect MCP server), Standardization (MCP interface).
- **Applicability:** Ideal when building conversational interfaces that need to securely access diverse, potentially sensitive, or complex backend systems/APIs. Suitable when you want to decouple the agent's reasoning logic from the data-fetching implementation.

10. Conclusion

This project successfully delivered a minimum viable product (MVP) for a conversational financial assistant, showcasing the powerful synergy between the Model Context Protocol (MCP) and the Langchain framework. The implemented architecture, integrating FastMCP for secure backend abstraction with a Streamlit-powered Langchain agent (utilizing AgentExecutor and StructuredTool for advanced natural language processing and orchestration), proved robust, adaptable, and efficient. Despite early challenges in dependency management and adapting to evolving agent framework APIs—resolved by leveraging Langchain's core agent components—the resulting system highlights the strengths of combining specialized, secure backend services via MCP with the flexibility of modern large language model frameworks.

The MVP establishes a solid foundation for future development, enabling enhancements such as expanded financial capabilities through additional data sources, predictive analytics, and market sentiment analysis. Furthermore, improvements in user experience are planned, including advanced conversational features, personalized financial insights, and richer interaction models. To ensure scalability and reliability, future efforts will focus on robust deployment strategies using Docker and container orchestration platforms like Kubernetes for high-availability production environments.

Ultimately, this MVP validates the feasibility and transformative potential of MCP-Langchain integrations for building sophisticated, real-world AI applications. It sets a promising trajectory for continued innovation in conversational AI, paving the way for scalable, secure, and intelligent financial assistant solutions.