Spintly Smart Assistant — Technical Project Documentation

---

1. User Interface (UI)

The frontend or UI component was not the initial focus of the project due to time constraints and its relatively straightforward implementation. However, the final version of the project will include a clean, user-friendly interface built with React. This UI will enable end users (like team leads or HR managers) to interact with the assistant via natural language queries such as:

- "What was my attendance last month?"
- "List all employees on leave today."

Since the backend and LLM logic are modular, the UI will consume outputs directly from the server via a structured API once integrated. The trial UI was not developed to save time for more critical architectural decisions. This decision enabled faster iteration over LLM tuning and MCP design.

---

2. LLM-Based Intent Extraction (Colab-based)

Given hardware limitations on the local system, we chose Google Colab for running large language models (LLMs) such as **Phi-3** and **Mistral 7B**. Colab provided a GPU-backed environment that allowed us to:

- Load models from Hugging Face directly.
- Run inference with pipelines.
- Extract intents and parameters from user queries.
- Achieve faster execution and avoid memory constraints faced locally.

Prompt Design Strategy

We employed a robust formatting prompt to help the model reliably identify the type and structure of each query. The model was instructed to respond in:

```
type: chat
output: <response text>

OR

type: task
output: intent=<intent_name>, params=<key1=value1, key2=value2>
```

This minimized hallucinations and ensured predictable parsing.

Output Sample (Summarized)

Query 1: "Capital of India"

```
{
  "type": "chat",
  "output": "The capital city of India is New Delhi. Is there anything else
you'd like to talk about?"
}
```

Query 2: "What is My and Bob's attendance of last month"

```
{
  "type": "task",
  "raw_output": "intent=attendance, params=employee1=My, month=last_month,
employee2=Bob"
}
```

Query 3: Contextual Limitation

```
{
  "type": "chat",
  "output":
"The model is not designed to understand user-specific pronouns like 'my' or 'I'
in context unless explicitly named."
}
```

Conclusion from LLM Phase

- Mistral-7B performed well within constraints.
- Phi-3 showed issues due to transformer version mismatches.
- Outputs were sufficient for structured classification, though context linking (e.g., "my" → name) requires a more advanced model or memory injection.
- Google Colab proved to be a highly viable testing ground for inference.

---

3. MCP Server & Database Integration

The **Module Context Protocol (MCP)** server was built using the `mcp` package from Spintly. It acts as the backend reasoning engine that:

- Accepts structured input (intent + parameters).
- Communicates with a PostgreSQL database.

- Returns structured results or summaries to be rendered on the UI.

MCP Input and Design

- **Tool Functions:** Defined to handle specific intents such as `get_all_users`, `get_attendance`, etc.
- **Input Format:**

```
{
  "tool": "get_attendance",
  "params": {
    "employee": "Bob",
    "month": "May"
  }
}
```

- **Output Format:**

```
{
  "result": [ {"employee": "Bob", "days_present": 18} ]
}
```

- **Execution Flow:** LLM → MCP tool → DB → result → UI.

DB Connectivity

We used the `psycopg2` library to connect the MCP to a PostgreSQL instance. All sensitive information was secured via a `.env` file. Data access was verified via manual test scripts before MCP tool wrapping.

Testing Approach and Workaround

Due to MCP's streaming nature using STDIO (which doesn't play well with Windows/MINGW), we had to:

- Test tool logic independently via custom Python test scripts.
- Use internal `print()`-based debugging.
- Simulate LLM-like input by feeding manually crafted JSON requests.

This helped validate tool behavior in the absence of full LLM-MCP integration.

---

4. Output Formatting and UI Display

Once the LLM determines a task and MCP returns the result, the assistant converts the structured output into a human-readable string. This was done by either:

- Using simple templating (e.g., f"Bob was present for 18 days in May.").

- Or invoking a smaller LLM on-device (planned).

This text response will be sent to the UI for final display.

---

5. Toward Unified Deployment: Constraints & Needs

To bring the full system into a deployable, production-grade state, the following points must be addressed:

Hardware Constraints

- **Minimum Specs:** 16 GB RAM, i7 CPU, and preferably 20–24 GB VRAM (GPU) to run Mistral or LLaMA efficiently.
- **Alternative:** Use cloud (AWS/GCP) instances or APIs (e.g., Ollama, HuggingFace Inference).

Tools & Environment Requirements

- Python libraries: `transformers`, `psycopg2`, `mcp`, `dotenv`, `fastapi`, `uvicorn`.
- PostgreSQL setup and proper `.env` credential management.
- MCP toolset configured and tested for all core intents.

Integration Steps for Deployment

- Run LLM, MCP server, and database in the same environment (Docker ideal).
- Connect UI frontend with FastAPI or GraphQL layer to interact with backend.
- Establish logging, authentication, and validation pipelines.
- Add session state management for better personalization.

Development Gaps & Final Remarks

- Database access was mocked due to delayed access.
- MCP-to-LLM live streaming wasn't tested due to hardware constraints.
- Contextual learning and memory were not implemented.
- The project architecture is modular and can be extended easily with better models, richer databases, and scalable UI.

This document serves as the core technical breakdown and execution log for the Spintly Assistant. Future sessions will include frontend build, authentication, and full containerization for deployment.