**1.1. Project Vision**

The Agentic MCP (Multi-Capability Platform) UI is an advanced AI orchestration system designed to serve as a single, intelligent interface to a suite of specialized AI agents. The project's vision is to abstract the complexity of diverse AI tasks behind a simple, conversational UI, allowing users to leverage powerful AI capabilities without needing to understand the underlying models or workflows.

**1.2. Core Capabilities**

The system currently orchestrates three distinct AI services:

1. **Poster Generation**: An end-to-end pipeline that transforms a simple user idea into a complete, visually appealing poster with generated text and imagery.
2. **Lead/Sales Intent Analysis**: A sophisticated tool that analyzes sales conversations, predicts buyer intent using a fine-tuned BERT model, and suggests strategic next steps.
3. **Content Cluster Analysis**: A service designed to break down broad topics into structured content clusters for marketing and SEO strategies.

**1.3. Technological Stack**

* **Backend**: Python 3.x, FastAPI, Uvicorn
* **Frontend**: HTML5, CSS3, JavaScript (ES6+)
* **AI Models & APIs**:
  + OpenAI API Client (for Groq, Kimi)
  + LLaMA 3 & Kimi K2 (via API) for text generation
  + Qwen-Image (via API) for image synthesis
  + Custom Fine-Tuned BERT for intent classification
* **Core Python Libraries**: transformers, torch, pydantic, python-dotenv

**2. System Architecture & Design**

**2.1. High-Level Architectural Diagram**

Frontend Layer:

┌─────────────────────────────────────┐

│ User Interface │

│ (index.html) │

└─────────────────────────────────────┘

│

▼ JSON Request (prompt)

Backend Layer - FastAPI:

┌─────────────────────────────────────┐

│ API Endpoint: /generate │

│ ├─ Router Logic: detect\_intent │

│ ├─ App Registry: app\_registry.json │

│ └─ Claude Agent: Intent/Payload │

└─────────────────────────────────────┘

│

▼ Routes payload to

Specialized Agents:

┌─────────────────────────────────────┐

│ ┌─────────────┐ ┌─────────────┐ │

│ │ Poster │ │ Sales │ │

│ │ Agent │ │ Agent │ │

│ └─────────────┘ └─────────────┘ │

│ ┌─────────────┐ │

│ │ Cluster │ │

│ │ Agent │ │

│ └─────────────┘ │

└─────────────────────────────────────┘

│

▼ Calls

AI/ML Services:

┌─────────────────────────────────────┐

│ ┌─────────────┐ ┌─────────────┐ │

│ │ LLM Service │ │ Image Gen │ │

│ │ Groq/Kimi │ │ Service │ │

│ └─────────────┘ │ Qwen │ │

│ ┌─────────────┐ └─────────────┘ │

│ │ BERT Model │ │

│ │ (Local) │ │

│ └─────────────┘ │

└─────────────────────────────────────┘

**2.2. Design Philosophy**

The architecture is founded on the **separation of concerns** and a **pluggable, agent-based design**.

* **Router as an Orchestrator**: The central router (router\_logic.py) does not perform business logic itself. Its sole responsibility is to understand the user's request and delegate it to the correct specialized component.
* **Configuration over Code**: The app\_registry.json file allows developers to register new agents and define their capabilities without modifying the core routing code, making the system highly extensible.
* **Layered Intelligence**: The system uses a tiered approach to AI. A general-purpose LLM (claude\_agent) acts as a high-level "understanding" layer, while specialized models (BERT, LLaMA for specific tasks) act as "execution" layers.

**3. Data Flow & Request Lifecycle**

This section details the journey of a user request from submission to response:

1. **Initiation**: The user types a prompt (e.g., "make a poster for a python bootcamp") into the frontend and clicks "Send". A JSON object {"prompt": "..."} is sent to the backend.
2. **Ingestion**: The FastAPI app receives the POST request at /generate. The PromptRequest Pydantic model validates the incoming data.
3. **Intent Detection (Step 1)**: router\_logic.detect\_intent is called. It passes the prompt to claude\_agent.process\_user\_prompt. An LLM call is made to analyze the prompt and return a JSON object containing the intent, recommended\_app, and raw\_prompt.
4. **Registry Lookup**: The router uses the recommended\_app name (e.g., "Poster Generator") to look up its configuration in app\_registry.json, retrieving its required fields and entrypoint.
5. **Payload Generation (Step 2)**: The router calls the appropriate payload generation function in claude\_agent.py (e.g., generate\_input\_payload\_for\_app). This function makes another LLM call to transform the user's simple prompt into a detailed, structured dictionary that matches the agent's required input schema.
6. **Pydantic Model Instantiation**: The generated dictionary is used to create a strict Pydantic model instance (e.g., PosterRequest(\*\*payload)). This ensures the data passed to the agent is valid.
7. **Agent Execution (Step 3)**: The router dynamically calls the run method of the specified agent (e.g., poster\_agent.run(poster\_payload)).
8. **Task Fulfillment**: The agent executes its internal pipeline (e.g., generating content, building an image prompt, calling the image API).
9. **Response Aggregation**: The agent returns a final JSON object containing the results (e.g., {"status": "success", "image\_base64": "...", "poster\_fields": {...}}).
10. **Final Delivery**: The router passes this result back through the FastAPI endpoint to the frontend.
11. **Rendering**: The frontend JavaScript parses the JSON and dynamically renders the appropriate UI components (e.g., an <img> tag for the poster or a formatted <div> for the sales analysis).

**4. Backend Implementation (FastAPI)**

The backend is clean, modern, and efficient.

* **main.py**: The main entry point of the application. It defines the FastAPI app instance, includes CORS middleware, and declares the single /generate route.
* **@app.post("/generate")**: This asynchronous function handles all incoming traffic. It performs critical checks to ensure the final intent\_data is a valid dictionary before returning it, preventing server errors from malformed agent responses.
* **Pydantic Models (data/models.py)**:
  + PromptRequest: Defines the initial input from the user.
  + PosterRequest, SalesRequest: These are crucial for data integrity. They define the exact structure and data types that the specialized agents expect, including optional fields and boolean toggles. This prevents invalid data from ever reaching the core agent logic.

**5. Core Routing & Orchestration (service/router\_logic.py)**

This is the brain of the operation.

* **detect\_intent(prompt: str)**: This function orchestrates the entire workflow. It's a pipeline that chains together the intent detection, payload generation, and agent execution steps.
* **Dynamic Routing**: The if/elif block that checks app\_name is the core of the routing mechanism. It acts as a switchboard, directing the validated payload to the correct agent based on the initial AI-driven analysis.
* **app\_registry.json**: This file is key to the system's modularity. By externalizing the agent configuration, you can:
  + Add a new agent (e.g., "Email Writer") by simply adding a new entry to the JSON file.
  + Update an agent's required fields without touching the Python code.
  + Change an agent's entrypoint (e.g., email\_agent.run\_v2).

**6. Primary Intelligence Layer (agents/claude\_agent.py)**

This module is responsible for interpreting the user's request and preparing it for the specialized agents. It acts as a "translation" layer between human language and structured machine input.

* **process\_user\_prompt**: This function uses a carefully crafted prompt to instruct an LLM to act as a "receptionist," figuring out what the user wants and which "department" (agent) to send them to.
* **generate\_input\_payload\_for\_app**: This is a brilliant example of prompt chaining. It takes the output of the first LLM call and uses it to fuel a second, more specific LLM call. This function's prompt instructs the AI to act as a "creative director," fleshing out a simple idea into a detailed brief for the poster agent.
* **generate\_sales\_payload\_for\_app**: This function is more focused on data cleaning. Its prompt instructs the AI to parse a messy block of text and extract only the relevant conversational data, formatting it cleanly for the sales agent.

**7. Specialized Agent Deep Dive**

**7.1. Poster Agent**

This is a multi-step pipeline for creative generation.

**7.1.1. poster\_agent.py - The Controller**

**poster\_agent.run**: The main controller. It orchestrates the three sub-steps: generating fields, building the prompt, and generating the image. It includes robust try/except blocks to handle potential failures at each stage, especially JSON decoding errors from the LLM.

**7.1.2. utils/llama\_generate\_fields.py - Content Generation**

**call\_llama\_generate\_fields**: This function calls a LLaMA model with a highly detailed system prompt. This prompt is a masterpiece of prompt engineering, providing the model with a task flow, rules, field constraints, and a perfect format example. This significantly increases the reliability and quality of the structured JSON output.

**7.1.3. utils/prompt\_builder.py - Advanced Prompt Engineering**

**build\_image\_generation\_prompt**: This utility is more than a simple string format. It dynamically constructs a layout based on which fields were actually generated by the LLM. It also appends critical instructions for the image model regarding typography and composition, which is essential for generating images with legible text.

**7.1.4. utils/image\_generator.py - Image Synthesis**

**generate\_poster\_image**: This function handles the final step of calling the image generation API. It correctly handles the API key, constructs the request payload, downloads the resulting image bytes, and encodes them into a base64 string suitable for direct embedding in the frontend HTML.

**7.2. Sales Agent**

This agent demonstrates a powerful fusion of a fine-tuned model and a general LLM.

**7.2.1. sales\_agent.py - The Controller**

**sales\_agent.run**: The controller for the sales analysis task. It receives the cleaned conversation and the predicted intent.

**7.2.2. utils/bert\_intent.py - Fine-Tuned Intent Prediction**

**predict\_sales\_intent**: This is the core of the agent's unique value. It uses a locally-hosted, fine-tuned BERT model that has been specifically trained to recognize sales-related intents. This approach is far more accurate and reliable for this specific task than a general-purpose LLM would be. The function handles tokenization, model inference, and mapping the output index to a human-readable label.

**LLM for "Next Best Action"**: After the BERT model provides the "what" (the intent), the sales\_agent uses a LLaMA 3 model to determine the "so what" (the next action). The prompt cleverly provides the LLM with both the conversation and the predicted intent as context, allowing it to generate a highly relevant and strategic suggestion.

**8. Frontend Implementation (index.html)**

The frontend is a single-page application that is both aesthetically pleasing and highly functional.

* **UI/UX**: The dark theme, gradient accents, and animated particle background create a professional and modern feel. The chat bubble interface is intuitive and familiar to users.
* **handleSend()**: This asynchronous JavaScript function is the heart of the frontend logic. It manages the entire request-response cycle, including disabling the UI during processing, showing a typing indicator, and re-enabling the UI afterward.
* **handleAgentResponse()**: This is the central dispatcher on the frontend. Instead of assuming a single response format, it inspects the keys of the incoming JSON data (image\_base64, next\_best\_action, etc.) to determine which rendering function to call. This makes the frontend just as modular as the backend.
* **Dynamic Rendering**: Functions like handlePosterResponse and handleSalesResponse dynamically create HTML elements to display the results. This is far more flexible than having static templates. The code correctly handles base64 image data by prepending the necessary data:image/png;base64, prefix.
* **Error Handling**: The try/catch block in handleSend is robust. It not only catches errors but provides helpful messages to the user, suggesting common problems like the backend server not running. The inclusion of a addDebugInfo function is an excellent practice for diagnosing tricky frontend-backend communication issues.

**9. Dependencies & Environment**

**9.1. Python Dependencies (requirements.txt)**

# Backend Framework fastapi uvicorn # AI & ML transformers torch openai # Data Handling pydantic python-dotenv requests # Additional utilities json os base64

The provided list of dependencies is well-structured, separating libraries by their purpose (backend, AI, data handling). This is crucial for creating a reproducible environment.

**9.2. Environment Variables (.env)**

The use of python-dotenv and os.getenv is a critical security best practice. It ensures that sensitive API keys are not hardcoded in the source code, allowing them to be managed safely in different environments (development, production).

GROQ\_API\_KEY=your\_groq\_api\_key KIMI\_API\_KEY=your\_kimi\_api\_key QWEN\_API\_KEY=your\_qwen\_api\_key

**10. Conclusion & Future Work**

**10.1. Key Strengths**

* **Sophisticated Architecture**: This is not a simple script; it's a well-architected, multi-agent system.
* **Hybrid AI Strategy**: The project masterfully combines the strengths of general-purpose LLMs (for creativity and flexibility) and fine-tuned models (for accuracy and reliability on specific tasks).
* **Extensibility**: The configuration-driven design means new agents can be added with minimal code changes, making the system future-proof.
* **Excellent User Experience**: The polished and responsive frontend makes the powerful backend capabilities accessible and enjoyable to use.

**10.2. Potential Enhancements**

* **Streaming Responses**: For a more interactive feel, the backend could stream LLM responses token-by-token to the frontend using FastAPI's StreamingResponse.
* **Stateful Conversations**: Implement session management to allow for multi-turn conversations where the agent remembers previous interactions.
* **Agent Inter-communication**: Develop a protocol for agents to call each other. For example, the Content Cluster agent could call the Poster Agent to create a visual for each content pillar.
* **Deployment & Scaling**: Containerize the application using Docker and plan for deployment on a cloud service like AWS or Google Cloud for scalability.