1. Overall Architecture

The system will be built on a standard Django MVT (Model-View-Template) architecture, extended to support real-time agentic workflows. The architecture is designed to be modular, separating the web interface from the agentic backend.

- **Django Components:**
  - **Models:** Define the core data structures: BotSession (to track individual conversations), MemoryEntry (for long-term knowledge storage, akin to a vector store), AgentLog (for recording agent actions, tool use, and LLM responses for transparency and debugging), and User (using Django's built-in user model).
  - **Views:** Handle HTTP requests from the user's browser. There will be two types of views:
    1. **Standard Views:** Render the dashboard UI using Django templates (e.g., login page, session history, memory management interface).
    2. **API Views (using Django Rest Framework):** Provide endpoints for the frontend to interact with the agent (e.g., /api/prompt, /api/get_history). These views will receive user prompts and trigger the agentic workflow.
  - **Templates:** Create the HTML for the user-facing dashboard. This will include pages for interacting with the agent, viewing session history, and managing the bot's memory. We will use a modern CSS framework and JavaScript to create a dynamic, single-page application feel for the chat interface.
  - **URLs:** Map incoming web requests to the appropriate views.
- **System Interaction Flow:**
  1. The **User** interacts with the dashboard UI (Django Templates).
  2. A user prompt is sent via a JavaScript fetch call to a **Django API View**.
  3. The View authenticates the user and hands the request to the **Orchestration Layer (MCP Controller)**.
  4. The Orchestrator queries the **Memory/Storage Layer** (PostgreSQL database with pgvector for RAG) to retrieve relevant context.
  5. The Orchestrator combines the user prompt and retrieved context, then calls the appropriate **LLM** via a local API endpoint.
  6. The LLM's response (which could be a plan, a code snippet, or a request to use a tool) is processed by the Orchestrator.
  7. The response is logged, stored in memory if necessary, and sent back to the Django API View.
  8. The View returns the response to the frontend, which updates the UI.

2. Core Protocols & Models

The agent's "brain" will be a central controller that orchestrates the various components, following patterns observed in systems like GitHub Copilot's agent mode.

- **Master Control Protocol (MCP):** We will implement a central Python class, AgentOrchestrator, that acts as the "brain" or MCP. This orchestrator will manage the agent's cognitive cycle (Reason-Act loop).
    - **Responsibility:** The AgentOrchestrator will receive a user request, manage the state of the task, and coordinate between memory, tools, and the LLM. It will implement the core logic of planning, tool use, and reflection as described in the research documents.
- **Data Flow & Cognitive Cycle (ReAct Pattern):**
    1. **User Query:** The AgentOrchestrator receives a prompt (e.g., "Analyze the users app and suggest improvements.").
    2. **Memory Lookup (Retrieval):** The orchestrator first queries the MemoryEntry model using a vector similarity search on the prompt's content. This retrieves past interactions, successful code patterns, or relevant documentation (RAG).
    3. **Prompt Augmentation:** The original prompt is augmented with the retrieved context, tool definitions, and conversation history to create a rich system prompt.
    4. **LLM Inference (Reasoning):** The augmented prompt is sent to the local LLM. The LLM's task is to generate a plan or a direct response. For example, it might respond with a JSON object: {"action": "tool_use", "tool": "read_file", "path": "users/models.py"}.
    5. **Tool Execution (Action):** The AgentOrchestrator parses the LLM's response. If a tool is specified, it executes the corresponding function (e.g., a tool to read files from the local file system).
    6. **Observation & Reflection:** The output of the tool (e.g., the file's content or an error message) is captured.
    7. **Iteration:** The observation is fed back into the context, and the loop returns to step 4. This cycle continues until the LLM determines the task is complete and generates a final response for the user.
    8. **Response:** The final answer is sent back to the user and saved to the AgentLog and MemoryEntry tables.

3. Key Tools & Libraries

- **Web Framework: Django** (with Django Rest Framework for APIs).
- **Agent Logic & LLM Integration: LangChain** or **LangGraph**. LangChain is excellent for composing the ReAct pattern and managing prompt templates. LangGraph would be ideal for defining the agent's stateful, cyclical reasoning

process.

- **Multi-Agent Orchestration: Crew AI** or **AutoGen**. While we can start with a single agent using LangChain, Crew AI would allow us to define specialized agents (e.g., a "Planning Agent," a "Coding Agent," and a "Debugging Agent") that collaborate, mimicking the advanced architecture of Copilot Workspace.
- **Database: PostgreSQL** with the **pgvector** extension to enable efficient similarity searches for the RAG implementation.
- **Local LLM Serving: Ollama** or a direct **Hugging Face Transformers** implementation with a simple Flask/FastAPI wrapper. Ollama is simpler to set up and provides an instant API endpoint for various open-source models.
- **Asynchronous Tasks: Celery** with **Redis** as the message broker. This is crucial for offloading long-running LLM inference tasks, preventing the web server from timing out.

4. Hardware & Environment Requirements

The specified hardware is well-suited for this project.

- **CPU:** AMD Ryzen 5 2600 (12 threads) is sufficient for running the Django application, database, and Celery workers.
- **RAM:** 16 GiB is adequate. The LLM will be the most memory-intensive component. A 7B parameter model (like Llama 3 8B Instruct) will fit comfortably within this memory budget when loaded with 4-bit quantization.
- **GPU:** The RTX 4060 (8GB VRAM) is excellent for accelerating local LLM inference. We will use libraries that support CUDA to offload model layers to the GPU, dramatically improving response times.
- **OS & Python:** Ubuntu 24.04 LTS and Python 3.10+ are standard and fully supported by all the required libraries.

5. Implementation Steps

1. **Scaffold Django Project:**
   - Create a new Django project and a dashboard app.
   - Set up Django Rest Framework and configure the PostgreSQL database connection.
   - Install and configure pgvector.
2. **Define Models:**
   - In dashboard/models.py, define the BotSession, MemoryEntry, and AgentLog models.
   - Run makemigrations and migrate to create the database schema.
3. **Local LLM Setup:**
   - Install and run Ollama. Pull a suitable model like llama3 or codegemma.

    - Verify you can interact with the model via its REST API (http://localhost:11434).
4. **Integrate LangChain & Orchestrator:**
    - Create a services directory within the dashboard app.
    - Implement the AgentOrchestrator class.
    - Use LangChain to create prompt templates and to connect to the local Ollama LLM.
    - Define a basic set of "tools" the agent can use (e.g., read_file, write_file, list_directory).
5. **Wire up API Views:**
    - Create a PromptAPIView in dashboard/views.py.
    - This view will receive a prompt, instantiate the AgentOrchestrator, and execute the main agentic loop.
    - Integrate Celery to run the orchestrator's main method as a background task. The view will immediately return a task ID, and the frontend will poll another endpoint to get the result.
6. **Build the Dashboard UI:**
    - Develop the Django templates for the main interface.
    - Write the JavaScript code to handle form submissions, send prompts to the API, poll for results, and display the conversation history dynamically.
7. **Test End-to-End Flow:**
    - Start the Django development server, Celery worker, and Redis server.
    - Perform a full test: send a prompt from the UI, verify the Celery task is created, check the AgentLog to see the agent's reasoning steps, and ensure the final response is displayed correctly in the UI.

6. Optional Enhancements

- **Advanced RAG:** Implement a more sophisticated RAG pipeline by automatically ingesting project documentation or even the entire codebase into the MemoryEntry vector store.
- **Multi-Agent Collaboration:** Evolve the single AgentOrchestrator into a multi-agent system using Crew AI. For example, a "Planner Agent" could decompose the task, and a "Code Generation Agent" could execute each step.
- **Caching:** Use Django's caching framework (with Redis) to cache responses for frequent or identical queries to reduce LLM calls.
- **Security & Sandboxing:** For agents that execute code, integrate a sandboxing tool like Docker. The agent's execute_code tool would run the code inside a temporary, isolated container to prevent security risks.