ChatGPT

Technical Design Document: Multi-Agent AI Platform Backend

Architecture Overview

The backend is designed as a **monolithic FastAPI application** that serves as the central hub of a multi-agent AI platform. It acts as a **Backend-for-Frontend (BFF)** for the Next.js client and orchestrates communication among AI agents. All interactions use a **RESTful API over HTTP with JSON** data exchange. The Next.js frontend (web client) communicates with this FastAPI backend exclusively via HTTP endpoints – for example, fetching data or posting tasks returns JSON responses to the frontend [1] [2]. The backend does not render UI; it purely provides APIs (ensuring an **API-first design**).

In the overall platform architecture, the flow is as follows [3]:

- **Browser (User)** → interacts with **Next.js Frontend** (React-based web app). The frontend handles UI and may perform server-side rendering or client-side data fetching.
- **Next.js Frontend** → communicates with the **FastAPI Backend** via REST API calls (e.g., `fetch('/api/endpoint')` from Next.js). The backend serves as the BFF, meaning it is tailored to the frontend's needs and consolidates data from various internal sources for the UI [4]. The BFF hides backend complexities and presents a simplified API to the frontend.
- **FastAPI Backend** → contains all server-side logic: it handles client requests, enforces authentication/authorization, and coordinates actions (including delegating work to AI agents, database operations, scheduling, etc.). The backend communicates with:
- The **PostgreSQL database** for persistent data (user info, projects, agent configurations, etc.).
- The **Redis message bus** for agent messaging and task queueing.
- **In-memory or local agents** running within the process (or background threads) that process messages/tasks.
- The **Vector database** for storing and querying embeddings (agent long-term memory).
- **Local ML models** (on GPU/CPU) via internal endpoints or function calls for AI inference tasks (LLM queries, embeddings).

Internally, the backend follows an **event-driven, asynchronous architecture** for agent communication. Agents do not call each other directly via function calls; instead, they send messages over a lightweight message bus. This decouples agents and allows flexible multi-agent interactions. For example, one agent can publish an event or task that multiple other agents subscribe to, enabling parallel tool execution or inter-agent collaboration [5] [6]. On a single-machine monolith, this is implemented with either an in-process asynchronous queue or Redis Pub/Sub channels. The FastAPI app can spawn background tasks that listen for messages addressed to each agent and route them appropriately.

All API endpoints are designed around **JSON input/output**, making the system frontend-agnostic and easily testable. The backend's role within the larger platform is to provide a unified interface for the UI and a coordination layer for the agents: - It receives user requests (e.g., a query or command to an agent team)

from the Next.js client and creates corresponding tasks/messages for the appropriate agent(s). - It ensures agents' outputs (answers, actions) are collected and sent back to the frontend or stored as needed. - It manages shared resources like the database, vector store, and scheduling of recurring tasks.

By concentrating these responsibilities in one service (monolithic design), we simplify development and avoid the overhead of deploying multiple microservices. FastAPI is capable of high performance and supports asynchronous operation, which suits this approach. (Notably, FastAPI is often used in microservice architectures [7], but here we leverage it for a single all-in-one service for simplicity and tight integration of components.) The monolithic backend can still be scaled vertically (or replicated behind a load balancer if needed), and the use of internal queues means that even within one process, long-running agent tasks won't block API responsiveness.

In summary, the architecture ensures that the Next.js frontend has a single point-of-contact (the FastAPI BFF) and that the AI agents have a structured, decoupled way to communicate and perform tasks. All communication is via well-defined JSON APIs or message schemas, ensuring clarity and testability of each part of the system.

Technology Stack

The backend is built with a stack of proven technologies, chosen for their performance, scalability, and ecosystem support:

- **FastAPI (Python)** – The core web framework. FastAPI is a high-performance, async-ready framework ideal for building RESTful APIs quickly. It provides automatic data validation (via Pydantic models) and interactive documentation (OpenAPI) out of the box. FastAPI's design makes it easy to integrate with AI/ML libraries and to write asynchronous code (concurrent requests) [8]. Its "lightning-fast performance and ease of use" combined with our database make it a great choice for this backend [9]. The entire API layer (endpoints for authentication, agent management, etc.) is implemented with FastAPI.

- **Uvicorn** – An ASGI server to run the FastAPI app. In development we use Uvicorn for its speed and simplicity. For production, we can run Uvicorn workers (possibly managed by Gunicorn) to handle multiple concurrent requests efficiently [10].

- **PostgreSQL** – The relational database for persistent storage. PostgreSQL is a reliable, ACID-compliant database suited for storing structured data (users, projects, agent configs, etc.). We choose Postgres for its robustness, scalability, and rich feature set (and to allow complex SQL if needed for reports or audits). As one author put it, FastAPI's performance paired with PostgreSQL's *"proven reliability and scalability"* makes them a perfect duo for demanding API projects [9]. We will use an ORM (such as **SQLAlchemy/SQLModel**) or query builder for interactions, making it easier to define our schema as Python models and perform migrations.

- **Redis** – In-memory data store used as a **message broker and cache**. Redis will serve as the **message bus** for agent communication and task queuing. When an agent or scheduler emits a task, it can be published to a Redis channel or pushed to a Redis list, and subscribing workers (or background tasks) will pick it up. Redis is extremely fast and well-suited for transient message

passing. It allows us to **decouple task submission from execution**, improving scalability and reliability by not tying up web request threads [11]. In addition, Redis can be used for caching frequently accessed data or rate-limiting if needed. (If we keep everything in-process, Redis might be optional, but using it prepares the system for multi-process or distributed scenarios and persistent queues.)

- **APScheduler** (Advanced Python Scheduler) – for in-process scheduling of tasks. APScheduler allows us to schedule functions to run at specific times or intervals within the FastAPI app process [12]. It's lightweight and doesn't require additional services, which fits a monolithic single-machine setup. We will use APScheduler to handle features like daily or weekly agent tasks (cron jobs) directly inside the app. Alternatively, for heavier scheduling needs or to offload work, we consider **Celery**:

- **Celery** (with Redis broker) – for background task processing and scheduling. Celery is a distributed task queue that can manage asynchronous jobs outside the web request cycle. In our design, Celery could be used to execute agent tasks in parallel and to schedule periodic tasks via **celery beat**. Celery beat is a scheduler process that **"kicks off tasks at regular intervals, executed by worker nodes"** [13]. We may use Celery if we anticipate long-running tasks or want to scale workers separately from the API. (For now, APScheduler covers scheduling within one process; Celery is an option if we later distribute the load or need more robust job management.)

- **Vector Database** – integration for agent memory embeddings. We plan to use a vector database such as **ChromaDB**, **Qdrant**, or **Weaviate** to store high-dimensional embeddings of text. A vector DB enables efficient similarity search over embeddings, which is crucial for an AI agent's long-term memory retrieval. For example, ChromaDB is an open-source vector store **designed for LLM applications**, offering a simple API and good performance [14]. By using a vector DB, the backend can store chunks of text or conversation history as vectors and quickly find relevant pieces later via similarity search. This augments the agents' context beyond what fits in prompt tokens. (The implementation can be switched out; we'll ensure an abstraction so that any of the popular vector DBs could be used with minimal changes.)

- **PyTorch and Hugging Face Transformers** – for local GPU-accelerated inference. The platform will include AI capabilities (like running language model prompts or computing text embeddings) on local hardware. We will load pretrained models (possibly from Hugging Face) using PyTorch. FastAPI endpoints (or internal services) will wrap these models for inference tasks. For example, we might host a local LLM for text generation and an embedding model for vectorizing text. The code will detect if a **GPU (CUDA)** is available and move models to GPU for faster processing [15] [16]. We take care to load models at application startup and keep them in memory, to avoid reloading on each request. To optimize performance, we can also pre-warm the models (run a dummy input once) so that GPU kernels are initialized before real requests [17]. Using local models ensures data privacy (no external API calls needed for AI tasks) and reduces latency, at the cost of using the server's GPU resources. If the server lacks a GPU, the models will run on CPU (slower but functional).

- **APScheduler / Celery (continued)** – Just to note, we will pick either APScheduler or Celery (or a combination: APScheduler for lightweight scheduling, Celery for offloaded tasks) based on complexity. On a single machine with limited concurrent tasks, APScheduler may suffice. Celery would shine if we expect to spawn many parallel agent tasks or potentially scale out workers. The

design is flexible to accommodate either: the scheduling interface in our code will be abstract (e.g., a service layer that either schedules via APScheduler or enqueues a Celery task).

- **Other Python libraries**: We will utilize **Pydantic** (for data models and settings management), **Passlib/Bcrypt** for password hashing if needed, **PyJWT or Python-JOSE** for JWT token encoding/decoding, and possibly **SQLAlchemy ORM** for database interactions. We will also use FastAPI's dependency injection system to manage things like DB sessions, authentication, and config.

All components are container-friendly and cross-platform. During development, we can run PostgreSQL and Redis in Docker containers or local installations. The FastAPI app can also be containerized for deployment. The chosen stack ensures that the backend is **high-performance, modular, and ready for AI integrations**. Each piece of the stack plays a role: FastAPI for the API layer, Postgres for durable data, Redis for fast messaging, scheduling for automation, vector DB for semantic memory, and PyTorch for the AI brain of the agents.

Database Design

We define a relational schema in PostgreSQL to persist the core entities of the platform. Below are the main tables (models) and their intended structure:

Users

This table stores user accounts and authentication info, with support for role-based access control.

- **Fields:** `id` (PK), `username` (or email, unique), `password_hash`, `role`, and audit timestamps (e.g., `created_at`, `last_login`).
- **Description:** Each row is a user. The `role` field can be an enum or text indicating the user's role (e.g., "Superadmin", "ProjectAdmin", "ProjectUser"). Passwords are stored securely as hashed values. We may include additional fields like `full_name` or `is_active`.
- **Usage:** Users authenticate via username/email and password. On successful login, a JWT is issued encoding their user ID and role. The role determines authorization levels in the backend. **Superadmin** can manage all projects and settings; **Project Admin** can manage a specific project (or projects they own); **Project User** has limited access (e.g., can view and run agents in the project but not alter global settings).
- **Relationships:** Could reference projects (e.g., an ownership relation) but that might be via a separate join table if users can belong to multiple projects. We may introduce a ProjectMembers table to assign users to projects with specific roles (to handle per-project roles in more detail).

Projects

Projects represent workspaces or teams in which agents operate. It groups agents and memories under a logical entity.

- **Fields:** `id` (PK), `name` (unique or per user unique), `description`, `owner_user_id` (FK to Users, indicating the creator or lead), maybe `settings` (JSON field for project-specific configurations), and timestamps.

- **Description:** A project is like a container for a multi-agent team or a knowledge base. It has metadata and possibly configuration (e.g., default vector index name, or project-wide parameters).
- **Usage:** Projects are the primary unit of organization. An authenticated user with proper role can create a project, invite others (if multi-user scenarios), and within a project define agents and data. When agents retrieve memory or communicate, they typically do so within the scope of a project (to avoid cross-project data leakage).
- **Relationships:** One-to-many: One project has many Agents, many MemoryEntries, and possibly many Tasks/Messages (if we log them). The `owner_user_id` indicates who can administer the project (usually a Project Admin).

Agents

Each agent is a distinct AI entity with a configuration. Agents belong to a project.

- **Fields:** `id` (PK), `project_id` (FK to Projects), `name`, `purpose` (a brief description or the agent's role), `system_prompt` (text for the agent's persona or initial prompt), `model_config` (JSON for model settings, e.g., model name or parameters), `tools_allowed` (could be an array of tool identifiers or a reference to AgentTools mapping), and `status` (active/inactive). Possibly an `agent_type` (e.g., "react-agent", "conversation-bot", etc.) if we classify agents.
- **Description:** Stores how the agent should behave and any static configuration. The **system_prompt** is particularly important: it sets the context for the agent (for example, an agent that summarizes documents might have a system prompt instructing it how to act). `model_config` might include which language model the agent uses (if we allow selecting from multiple local models or settings like temperature for generation).
- **Usage:** When the backend spawns or initializes an agent instance (in memory), it will load the config from this table. Agents may also have dynamic state (not stored here, but e.g. stored in memory tables or in the vector DB). If an agent is "paused" or removed, we could mark `status` accordingly.
- **Relationships:** Many agents per project. Could have a one-to-many to AgentTools (if each agent has multiple tool entries). Could also link to a **Schedule** (if an agent has scheduled tasks).

Tools

A registry of all possible tools that agents can use. This acts as a reference list of external or internal tools (e.g., web search, calculator, database query, etc).

- **Fields:** `id` (PK), `name` (unique key, e.g., "web_search", "send_email"), `description` (what the tool does), `endpoint` or `function_ref` (some reference on how to execute the tool – this might be a string that maps to a Python function or API endpoint), and maybe `restricted` (boolean if only certain agents can use it by default).
- **Description:** It enumerates the capabilities that agents can be endowed with. Tools can be thought of as plugins that extend what an agent can do beyond basic LLM responses. For example, a "Calculator" tool might allow an agent to perform arithmetic; a "WebSearch" tool might hit an external API.
- **Usage:** Tools in this table are not yet assigned to agents – they're the *catalog* of all tools in the system. Developers or admins can add new tools here (with proper implementation on the backend to perform the action).

- **Relationships:** Many-to-many with Agents (realized via the AgentTools mapping table described next).

(If the platform doesn't require a global tool registry, and tools are only defined per agent, we could simplify by storing allowed tool names in a JSON field on Agents. However, a dedicated Tools table provides a clear separation and the ability to globally enable/disable certain tools or add new ones in one place.)

AgentTools (Agent-Tool Mapping)

This table maps which agents have access to which tools, effectively linking Agents to Tools.

- **Fields:** `agent_id` (FK to Agents), `tool_id` (FK to Tools). Together these two are the composite primary key (an agent-tool pair is unique). Optionally, could have a field like `config` (JSON for any tool-specific configuration per agent) or `last_used` timestamp for analytics.
- **Description:** It's essentially a join table indicating that a given agent can use a given tool. By populating this table, we configure each agent's toolset.
- **Usage:** When an agent is executing and decides to use a tool, the system will verify that the agent is allowed to use that tool (by checking this table or the agent's allowed list). This allows for fine-grained control, e.g., one agent might have internet access tools, while another is sandboxed without them.
- **Relationships:** N-to-N between Agents and Tools. We may manage this via application logic or an admin UI (where you can toggle tools for each agent).

(In some designs, the AgentTools table might also double as the tool registry by including tool details, but here we've separated them for clarity. AgentTools primarily focuses on assignments, whereas Tools focuses on definition.)

Tasks / Messages

This table acts as a log or queue of messages exchanged between senders and receivers (which can be agents or users or system). It represents the message bus in a durable form (for auditing or persistence across restarts), although real-time delivery is handled via in-memory/Redis.

- **Fields:** `id` (PK), `project_id` (FK to Projects, indicating context), `sender_type` and `sender_id` (e.g., sender could be "User" with an ID referencing Users table, or "Agent" with ID referencing Agents table, or "System" for scheduler), `receiver_type` and `receiver_id` (similarly referencing Agents or perhaps a special type for broadcast/system), `content` (text payload or JSON data of the message), `status` (e.g., "queued", "delivered", "complete", "error"), `timestamp` (when message was created), and possibly `response` or `result` (if this message corresponds to a task result).
- **Description:** Each entry is a discrete message or task instruction. For example, if a user asks a question to an agent, the backend might create a message with sender_type=User, receiver_type=Agent, content containing the question. The agent, when processing it, may update the status or create a new message in response. Likewise, if Agent A delegates work to Agent B, it creates a message record addressed to B.
- **Usage:** In-memory, these messages are passed via Redis or Python queues for immediate handling. The database table allows persistence: we can reconstruct conversations or tasks for history and debugging. It also provides an abstraction if we needed to use a different queue backend; our code

can treat this table as the source of truth for what tasks exist. (However, for performance, the real queueing is done outside the DB in Redis; the DB is secondary for logging).
- **Relationships:** Linked to Project for scoping (so agents only react to messages in their project). Sender and receiver are polymorphic relationships (we distinguish in fields or we could use nullable FKs: e.g., an agent message has `sender_agent_id`, a user message has `sender_user_id`; a design choice).

MemoryEntries

This table stores pieces of text (knowledge, conversation history, etc.) with their embeddings for semantic search. It works in conjunction with the vector database.

- **Fields:** `id` (PK), `project_id` (FK to Projects), `agent_id` (optional FK if memory is tied to a specific agent; could be null for project-wide memory), `content` (text of the entry), `embedding` (vector, possibly stored as an array type or separately in the vector DB), `metadata` (JSON for additional info, e.g., source, date, tags), and `timestamp`.
- **Description:** Think of MemoryEntries as the long-term memory store for agents. These could be chunks of documents, transcripts of prior conversations, or facts learned. When an agent needs context, the system will take the query or relevant text, compute its embedding, and query the vector database for similar embeddings within the same project (and possibly agent).
- **Usage:** When we add a MemoryEntry, we will also add the embedding to the vector database (like Chroma or Qdrant) under a collection corresponding to the project (or agent). The `embedding` field in Postgres could store a small identifier or nothing at all if we rely solely on the vector DB for retrieval. However, we might store the raw vector or at least a reference so that backup of data is complete. Queries: to retrieve memory, the agent calls a function that queries the vector DB (filtering by project_id = current project) to get the top-N similar entries. Those entries can then be fetched (by IDs) from this table to get the full text. The table also allows manual inspection of what is in memory and could help implement forgetting or updates.
- **Relationships:** Many MemoryEntries belong to a Project. Possibly relate to an Agent if we consider personal agent memory, but likely project scope is enough (all agents in a project share the knowledge base).

Schedule

This table defines scheduled tasks (recurring or one-off in future). It's used by the scheduling system to persist schedule definitions and possibly track last run times.

- **Fields:** `id` (PK), `project_id` (FK to Projects, if the schedule is project-specific), `agent_id` (FK to Agents, if the schedule triggers an agent), `cron` (text cron expression) or `interval` (interval specification), `task` (description of what to do – e.g., a reference to an agent function or a message payload to send), `next_run` (timestamp of next execution, managed by the scheduler), `enabled` (bool).
- **Description:** Each row represents a periodic job. For example, an entry might state: project=X, agent=ReportAgent, cron="0 9 * * *", task="generate_daily_report". This means every day at 9:00, the system should trigger the ReportAgent to generate a daily report.
- **Usage:** On application startup, the scheduler (APScheduler or Celery beat) can load all enabled schedules from this table and schedule them. Alternatively, we can rely on the scheduler's own persistence, but having them in the DB is convenient for dynamic control (so that an admin could

create or modify schedules via an API call which updates the DB). When a schedule triggers, the scheduler will typically create a Task/Message entry (for an agent or broadcast) and push it to the queue for processing.

- **Relationships:** Could link to an Agent (if the scheduled task is specifically to invoke an agent's action). Or it might not link directly, instead the `task` field might encode something like "send message to agent X with payload Y". We might parse that in the scheduling logic.

Additional Notes on Database Design

All foreign keys will be properly indexed for performance. We will likely use **SQLAlchemy** or **SQLModel** to define these models in code and use Alembic for migrations. This ensures that our DB schema stays in sync with the code definitions.

The schema is designed to support **multi-tenancy by project** – data is largely scoped by project to prevent cross-project interference. A Superadmin may have access to all projects' data, but regular project users only see their project's agents, memory, etc.

We also ensure **referential integrity**: e.g., if a Project is deleted, we might cascade delete its Agents, MemoryEntries, etc., or mark them for cleanup. Similarly, deleting a user or agent might have cascading effects which we handle carefully (possibly soft-deleting if preserving history is important).

Finally, while the above covers relational data, we will use the **vector database** for similarity search. The vector DB (Chroma/Qdrant) will store the actual embedding vectors and allow queries. Typically, the vector DB will have a **collection per project** (or per project per agent, depending on isolation needs). The `MemoryEntries` table's primary key could be used as the vector ID in the vector DB. The metadata in the vector DB can include project and other fields as well [18], enabling filtering queries like "find similar content where project_id = X". This setup combines the strength of relational DB for structured info and vector DB for unstructured semantic search.

Authentication and RBAC

Authentication is handled via JWT (JSON Web Tokens), implementing a stateless auth model suitable for SPA (Single Page Application) frontends. We will likely use OAuth2 Password flow (username & password exchange for token) as provided by FastAPI's security utilities [19], or a similar login endpoint that returns a JWT on success.

Key points of the auth system:

- **JWT Tokens:** When a user logs in (by providing valid credentials to a `/auth/login` endpoint), the backend issues a signed JWT (using a secret key) that encodes the user's identity and role. The token is a Bearer token that the client must include in the Authorization header of subsequent requests. The JWT will have an expiration (e.g., 1 hour or 24 hours) for security. We might also implement refresh tokens if needed, but initially a simple token with reasonable expiry is used.

- **Roles and Claims:** The JWT payload includes a claim for the user's role(s). For example, we could include `"role": "ProjectAdmin"` or for more complex cases, an array `"roles": ["ProjectUser"]` [20]. If needed, the token can also carry the user's project memberships or

scopes, but since we can derive that from the DB, it may not be necessary in the token (keeping it small). On each request, a FastAPI dependency will verify the JWT (signature and expiry) and extract the claims. We will then enforce RBAC logic in our endpoints based on these claims.

- **RBAC (Role-Based Access Control):** We define three primary roles:

- **Superadmin** – Top-level administrators. They have full permissions across the system: managing all projects, all users, agents, etc. Endpoints that alter global settings or view all projects would require Superadmin role.
- **Project Admin** – Users who can manage a specific project. They can create or delete agents in their project, adjust project settings, view all data in their project, and invite/remove project members. They cannot affect other projects that they are not an admin of.
- **Project User** – Regular users within a project. They can query agents and view results, maybe add memory entries, but typically cannot change agent configurations or delete things. Their access is mostly read/execute but not modify settings.

The backend will enforce these roles. For example: - The endpoint `POST /projects` (to create a new project) might require at least ProjectAdmin (or Superadmin) rights. - `DELETE /projects/{id}` might require Superadmin (to delete any project) or require that the caller is ProjectAdmin of that project (and possibly Superadmin for certain projects). - `POST /projects/{id}/agents` to create an agent in project X would require the user be an admin of project X. - `GET /projects/{id}/agents` might allow any project member to list agents, but `PUT /projects/{id}/agents/{agent_id}` (to update config) would be limited to admin. - And so on for other resources (Memory entries might be viewable by all in project, but deletable only by admins, etc.)

FastAPI makes it straightforward to implement RBAC checks. We can create dependency functions like `get_current_user(role: str)` which checks the JWT and ensures the user's role matches the required role (or is higher). We might also check the project membership: for endpoints with `{project_id}` path, verify that the JWT user has access to that project (we may include a `project_id` claim if the user is limited to one project, or we query a ProjectMembers table).

- **Backend-for-Frontend (BFF) Pattern:** As mentioned, our FastAPI service is acting as a BFF for the Next.js app. In practice, this means the FastAPI backend is **tailored to the needs of that frontend** and will aggregate data or customize responses to simplify the frontend's job [21]. For example, instead of the frontend calling three different endpoints to gather user info, project info, and some agent status and then merging them, we might have a single endpoint that the BFF provides (e.g., `/dashboard`) that returns a consolidated JSON containing all needed info for that view. This reduces round trips and offloads data massaging to the server side, which is typically more powerful and secure. The BFF is effectively a facade: the Next.js app trusts it as the sole backend it needs to talk to, even if under the hood the FastAPI server might be interacting with multiple subsystems (database, vector store, etc.). By keeping this pattern, the frontend remains decoupled from any microservice complexity or external APIs – it just talks to the BFF service, which in turn can use whatever protocols or internal calls needed to fulfill the request [4].

In terms of security, the BFF pattern also means the FastAPI can act as a gatekeeper. For instance, the Next.js app will not directly call the vector database or an LLM service; it will call FastAPI, which will authenticate the request and then internally query the vector DB or model and return results. This ensures

that all critical resources are behind one authorization layer (the FastAPI JWT auth). It also simplifies CORS and networking – only the FastAPI domain needs to be exposed to the frontend.

- **JWT Implementation Details:** We will use a library (such as PyJWT or jose) to encode and decode tokens. We'll configure a strong secret key (and possibly algorithms like HS256 or RS256 if using asymmetric keys). Passwords will be verified using a hashing algorithm (BCrypt or Argon2) – FastAPI's OAuth2PasswordBearer and OAuth2PasswordRequestForm can streamline this. During login, we check the password, then create a JWT with claims including user ID, role, and perhaps a list of project IDs the user has access to (depending on needs). The token might look like:

```
{
  "sub": "<user_id>",
  "role": "ProjectAdmin",
  "exp": 1690000000  (expiry timestamp)
}
```

  This token is returned to the client to store (usually in memory or cookie, in our case likely stored in Next.js app memory since it's an SPA).

- **Authorization Enforcement:** We will protect routes using dependencies. For example:

```
from fastapi import Depends, HTTPException
from .auth import get_current_user, require_role

@app.get("/projects/{project_id}/agents")
def list_agents(project_id: int, current_user=Depends(get_current_user)):
    # require that user has access to this project
    if not current_user.can_access(project_id):
        raise HTTPException(403, "Not allowed")
    ...
    return [...]
```

  And for role-specific:

```
@app.post("/admin/something")
def do_admin_action(current_user=Depends(require_role("Superadmin"))):
    ...
```

  The `require_role` dependency will check the JWT's role claim and raise 401/403 if not sufficient. In some cases, we might allow multiple roles: e.g., require_role("ProjectAdmin","Superadmin").

- **Backend-for-Frontend Explanation (continued):** To explicitly address the BFF pattern: it was introduced to solve issues where frontends had to call multiple microservices or deal with data in shapes not ideal for the UI [22] [23]. In our case, while we have a monolith (so not multiple

microservices), we still apply the principle by making sure the FastAPI endpoints align closely with the UI needs. This could mean the FastAPI performs certain aggregations. For instance, an endpoint for the project dashboard might fetch the latest status of all agents, recent activity logs, and summary of memory usage in one go, whereas internally that might involve several DB queries. The goal is to keep the frontend code simple and **"avoid doing complex computations in the UI layer"** [24]. The BFF (FastAPI) can evolve in parallel with the frontend – if the UI needs a new combined data structure, we implement a new endpoint or adjust an existing one, without exposing unnecessary data or requiring the UI to stitch responses. This pattern also helps with versioning and A/B testing (the BFF can serve different data shapes to different app versions as needed, without the frontend calling different services) [25].

In summary, authentication is via JWT bearer tokens, and authorization is role-based, enforced on every request. The backend's BFF approach ensures the frontend and backend work in tandem, providing a secure and optimized data exchange. We have a clear separation of concerns: the frontend handles presentation and user interaction, the backend (BFF) handles authentication, authorization, and data/service integration, and the backend's internal components (DB, agents, etc.) handle the heavy lifting behind the scenes.

Agent Communication

One of the core features of the platform is the multi-agent system – multiple AI agents that can communicate and collaborate on tasks. The backend facilitates this with an **in-process message queue and routing logic** tailored for single-machine operation (though extensible to distributed setups if needed). The design takes inspiration from actor model and event-driven systems, ensuring agents are decoupled and communicate via messages rather than direct calls.

Message Bus Design: We implement a lightweight message bus to pass messages between agents (and between users and agents, and system and agents). There are a few implementation options: - For pure in-memory operation (within one Uvicorn worker), we can use Python `asyncio` queues or simple `queue.Queue` (for threads) to pass messages. - To support multi-process (or just to have durability), we use **Redis Pub/Sub or Redis Streams** as the backbone. Each agent could have its own Redis channel (e.g., channel name "agent_{id}") for incoming messages. Publishers (other agents or the system) publish to that channel; the agent's listener subscribes to it and processes messages. - We could also use a Redis-backed task queue (like Celery tasks targeted by agent id), but that might be overkill for our needs. Simpler pub/sub suffices for real-time message passing.

We favor using Redis Pub/Sub for flexibility: it allows easy fan-out (multiple subscribers) and can work across processes if we ever run multiple worker processes for agents. But we will abstract this such that the agent communication code doesn't deeply care whether it's an asyncio Queue or Redis – we can have an interface like `MessageBroker.send_message(message)` and `MessageBroker.subscribe(agent_id)`.

Message Schema: Each message will have a uniform schema (as reflected in the Tasks/Messages table): - `sender`: Who is sending the message. This could be represented as a tuple of (type, id) or a single string identifier. For example, `"user:42"` or `"agent:7"` or `"system:scheduler"`. This identifies the origin. - `receiver`: The target of the message, similarly identified. Often this will be an agent ID (like `"agent:5"`), but it could also be a group or broadcast (we might define `"all_agents"` in a project, though

implementing broadcast means multiple deliveries). - `project_id`: Context to ensure the message is routed to the correct project environment. Agents will likely ignore messages not matching their project (for safety). - `payload`: The content of the message. This could be plain text (like a user question: "What is the status?") or a structured JSON (e.g., a command for another agent: `{"action": "fetch_data", "parameters": {...}}`). We may also include a message type or performative, such as "QUERY", "RESPONSE", "COMMAND", etc., to help agents interpret the message. - `message_id` and metadata: Each message has a unique ID and timestamp. If the messaging system is reliable, we can track acknowledgments or statuses, but within a single machine we might keep it simple (fire-and-forget with in-memory queue, while logging to DB for persistence).

This schema allows for a flexible communication pattern. For example: - A user question comes in via API: the backend creates a message `sender=user:123, receiver=agent:45, payload="Hello, can you summarize the report?"` and enqueues it. - Agent 45's listener picks it up, processes it (maybe uses tools, etc.), and then produces a reply message `sender=agent:45, receiver=user:123 (or some proxy for user session), payload="Sure, here is the summary: ..."` which the backend can then forward back to the frontend (via WebSocket or polling or next API call). - Or Agent 45 might spawn a sub-task: it could send `sender=agent:45, receiver=agent:46, payload={"action": "get_latest_data"}` if it delegates to another agent. Agent 46 will respond with data, etc.

Agent Loop / Listener: Each agent in the system runs an asynchronous loop waiting for messages addressed to it. In a monolithic deployment, we can implement this in a few ways: - **Background Tasks:** When the app starts (or when a project/agent is initialized), we launch a background task (via `asyncio.create_task` or FastAPI's `BackgroundTasks`) that subscribes to the agent's channel and enters a loop:

```
async def agent_loop(agent_id):
    async for msg in broker.subscribe(channel=f"agent_{agent_id}"):
        await handle_message(agent_id, msg)
```

`handle_message` would contain the logic to interpret the message and produce a response or perform an action. - **Thread per Agent:** Alternatively, use threads if some agents do blocking work (but since we can do async and our tasks may involve I/O or model calls that release GIL, async is fine). - We need to be cautious with too many agents and loops – if we have many agents, a loop per agent is fine as long as it's mostly idle (waiting for messages). This is like having multiple consumers on a message queue.

Agent Routing Logic: When a message is sent, the message bus needs to route it to the correct agent's handler. If using Redis, the routing is handled by publishing to the agent's channel. If using an internal queue, we could have a dictionary mapping agent_id to a Queue, and `send_message` just does `queues[agent_id].put(message)`.

For example, an event-driven approach described in a discussion suggests **topics and subscribers**: one can publish an event on a topic and multiple tools or agents subscribed to that topic will get it [5] [6]. We can leverage this concept for certain scenarios: - We might define topics like `"context_gathering"` and have multiple agents or tool handlers subscribe to it (as in the example, one agent does keyword search, another

does vector search in parallel) [26]. This would allow **parallel processing** and then the results could be collected. - In general, our system could allow an agent to specify interest in a class of messages (topic-based pub/sub) in addition to direct addressing. However, initially we will implement direct addressing (point-to-point messaging).

The **benefit of using a message bus** is flexibility: - Agents can run concurrently and communicate asynchronously without tight coupling. If Agent A is waiting on Agent B's reply, it doesn't block the entire app, only that coroutine. - We can more easily introduce new agents or change communication patterns by subscribing/publishing to different channels without rewriting core logic [27]. - It's also easier to monitor and log all messages for debugging or playback.

Processing a Message: The `handle_message(agent_id, msg)` function (conceptually) will: 1. Parse the message payload. 2. Depending on agent's design, either treat it as a user query or an instruction: - If it's a user query or something needing an LLM response, it will invoke the local LLM (via the GPU inference service) with the appropriate prompt (likely consisting of the system prompt, conversation history, and the new query). - If it's a command (like an agent instructing another to use a tool), the agent's logic will execute that tool. For example, if msg says `{"action": "search_web", "query": "Python news"}`, the agent might call a `WebSearchTool` function and gather results, then send a follow-up message back. 3. The agent may produce one or multiple outgoing messages as a result. For user queries, the outgoing message is usually a reply to the user (or to a coordinator agent). For tool results, it might be a message back to the requesting agent with data. 4. The agent could also update its internal state or memory: e.g., storing the conversation so far or adding a summary to MemoryEntries via the vector store.

Tool Execution and System Interface: Notably, some messages might not be intended for an agent's cognitive processing but rather as an instruction for the system to do something (like scheduling or a tool invocation). In the ReAct pattern, an agent might output a *tool use command* that the system intercepts to actually call the tool and return the result [28] [29]. We will implement a mechanism for this: - If an agent decides to use a tool, it can send a message (perhaps to a special system receiver like `receiver=system:tool_runner` or simply as a self-addressed message with structured content). Our infrastructure will recognize it as a tool request, call the appropriate function (outside of the LLM), and then send the result back to the agent as an observation message. - This aligns with how ReAct agents are described: the agent outputs a structured tool call, the system executes it externally, then the result is given back to the agent [30] [28].

Alternatively, we can incorporate tool use in the agent's code directly, but doing it via messages decouples the LLM reasoning from actual execution, which can be safer and more manageable.

Concurrency and Order: The system should handle messages asynchronously. If multiple messages arrive for an agent in quick succession, we can queue them. Agents ideally process one message at a time (depending on logic), unless they spawn sub-tasks. We might need locking or tracking to prevent, say, an agent from answering two user queries concurrently in a confused way. A simple approach is each agent's loop fetches one message at a time and processes it fully before taking the next. If an agent's processing involves awaiting tool responses (which come as messages), it could either be structured as sub-tasks in the same loop or use an internal state machine.

Handling user interactions: When a user triggers an action via the frontend, the backend may either: - Immediately call an agent and wait for the result (if it's quick) – essentially treating the agent as a function.

But if the agent might do multi-step reasoning or long tasks, it's better to handle asynchronously. - Likely, the frontend will initiate an agent task and then poll or listen for the result. We could use WebSockets or Server-Sent Events to push agent responses to the UI for a live conversation feel. FastAPI supports WebSocket routes easily. In that case, the agent when ready would send the reply message, and the backend, seeing the receiver is a user (or UI session), would push it out over the WebSocket connection.

For initial simplicity, possibly a polling mechanism or synchronous wait with a timeout might be used (depending on UI expectations). However, for a multi-agent conversation, asynchronous is the way to go.

Example Flow (Daily Report Scenario): A scheduled task triggers a "Manager Agent" to compile a daily report. The scheduler (system) sends a message to ManagerAgent: `{"command": "create_daily_report", "date": "...", ...}`. ManagerAgent receives it and breaks the task: - It sends queries to two other agents: DataAgent (to gather data) and AnalysisAgent (to analyze trends) in parallel. It does this by publishing on, say, topic `"daily_report_{date}"` which both DataAgent and AnalysisAgent subscribe to [5] [31]. Both agents do their job (e.g., DataAgent queries a database or API, AnalysisAgent runs some stats). - When each finishes, they send a message back to ManagerAgent (or just publish results on another channel `"daily_report_results_{date}"` that ManagerAgent listens to). ManagerAgent collects results, then formulates the final report. - ManagerAgent then sends the report as a message to maybe a LoggingAgent or directly to a user channel (if it should email someone, maybe it uses an EmailTool, or if just stored, it could put it in memory). - Throughout, the message bus allowed these communications without hard-wiring the agents to each other. New agents can be added to the process if needed for other tasks, and they just subscribe/publish appropriately. This event-driven approach yields **better parallelism and flexibility** [32] [27].

Error Handling: If an agent fails to process a message (e.g., exception or timeout), we should handle that gracefully. Possibly mark the message status as "error" and send an error message back to the sender. Monitoring components can pick up these errors. In a single-process environment, an unhandled exception in an agent task could crash that task – we need to catch exceptions in agent loops to keep them running. Logging the error (with stack trace) will be important for debugging agent behaviors.

Security: Agents are powerful (especially if they have tool access), so the system must ensure agents only do what they're allowed. The message bus should not allow an agent to arbitrarily call internal admin endpoints, for instance. But since agents are code we wrote, it's more about avoiding bugs than malicious intent. We will still isolate scopes: e.g., an agent is always tied to a project context, so it won't see messages from another project's user.

To summarize this section: The agent communication subsystem uses an asynchronous message-driven pattern. Each agent has a **mailbox** and processes incoming messages sequentially. Messages contain sender, receiver, project, and content, enabling a traceable conversation. By using a message bus approach, we achieve: - **Loose coupling:** Agents need not be aware of each other's internal implementation, only the message formats. - **Parallelism:** Multiple agents can work simultaneously on different tasks or even sub-tasks of the same overall job, which can reduce latency for complex workflows. - **Scalability:** In the future, agent loops could run in separate processes or even separate machines, subscribing to a central Redis or message broker. The current design on one machine can evolve to a distributed one if needed, simply by having agents subscribe over the network (Redis, etc.) and ensuring id uniqueness across deployment. - **Maintainability:** All messages can be logged (in the Tasks/Messages table) for audit trails. We can replay sequences in dev to understand agent interactions.

Our implementation will encapsulate this in an AgentManager or similar component that starts/stops agent listeners and provides a method to send messages to agents. The AgentManager will also coordinate with the **GPU inference service** for any LLM calls the agents need (see next section).

Scheduling System

The platform includes a scheduling component to handle recurring tasks or delayed executions (e.g., run an agent routine daily, or periodically ingest new data). We have two primary strategies available for scheduling: **APScheduler (in-app)** and **Celery Beat (external)**. We will detail both and choose what fits the monolithic design:

Using APScheduler (AsyncIOScheduler): APScheduler can run inside the FastAPI app process, scheduling jobs on the event loop. It's simple to set up and works well for a single-instance deployment. According to its docs, APScheduler is a "light but powerful in-process task scheduler" suitable for scheduling functions at specified times [12].

Implementation steps for APScheduler: - Install `apscheduler` and import AsyncIOScheduler. - During FastAPI startup, initialize the scheduler and start it (FastAPI provides a startup event or lifespan context to do this) [33] [34]. - Define the scheduled jobs either in code (with `@scheduler.scheduled_job` decorators) or by reading from the Schedule table. - APScheduler supports cron-like schedules, interval schedules, or one-time schedules. For example, to run a job every day at 13:05 UTC:

```
@scheduler.scheduled_job('cron', hour=13, minute=5)
async def daily_job():
    ...
```

This matches the example from documentation [35]. - The job function could be an async function (which APScheduler can handle using AsyncIOScheduler). Inside that job, we would invoke the desired action, e.g., sending a message to an agent or calling a function. - We make sure to shut down the scheduler on application shutdown to avoid orphaned threads (APScheduler handles this via scheduler.shutdown) [34].

Using APScheduler keeps everything in one process. However, be mindful: if we run multiple Uvicorn workers (processes), each would start its own scheduler, potentially causing duplicate job executions. For a monolith, we might run a single instance or designate one as the scheduler. Alternatively, we set a flag or use a database lock to ensure only one scheduler runs (noting APScheduler itself doesn't coordinate across processes). In many cases, it's simplest to run the app as a single process if using APScheduler for cron jobs.

Using Celery Beat and Workers: Celery offers a more robust, distributed task solution. **Celery Beat** is a dedicated scheduler process that sends tasks to the Celery workers at the specified intervals [13]. If we use Celery: - We configure Celery with Redis as the broker (and possibly Redis or the database as result backend if needed). - Define Celery tasks corresponding to agent actions or maintenance tasks. E.g., a Celery task `run_agent_task(project_id, agent_id, command)` that when called will send the appropriate message to the agent or even directly execute the agent's logic (if it's short). - Use Celery Beat to schedule periodic tasks. This can be done by either a configuration (celery beat schedule config) or by adding tasks in code via `add_periodic_task` [36] [37]. For example:

```
from celery.schedules import crontab
app.conf.beat_schedule = {
  'daily-report-job': {
    'task': 'tasks.run_agent_task',
    'schedule': crontab(hour=9, minute=0),
    'args': (project_id, agent_id, {"command": "create_daily_report"})
  }
}
app.conf.timezone = 'UTC'
```

Celery beat will read this and ensure the task is sent on schedule. - A Celery worker (most likely within the same Docker container or VM as the FastAPI app, to keep it "monolithic" in deployment, though it's a separate process) will receive the task and execute it. The execution might involve generating messages to agents as described. - Celery ensures reliability: tasks are stored in Redis until processed, and if a worker is down, tasks wait. We must ensure only one beat scheduler runs (Celery's docs highlight that you should not run multiple beat instances without coordination to avoid duplicate triggers [38]).

Celery's advantage is if an agent task is CPU-bound or long-running, it won't block the FastAPI server at all; it happens in the Celery worker. In our monolith scenario, we can still include Celery as a background worker thread (Celery can actually be run in the same process via gevent/eventlet, but that's complex – better to just run a separate worker process alongside).

Given the scope, we likely start with **APScheduler** for simplicity, and keep Celery as a future enhancement for heavy lifting or scaling. The TDD covers both to be thorough.

Defining Scheduled Tasks: Regardless of implementation, we identify what tasks need scheduling: - **Agent Invocations:** e.g., "Daily Report Agent" at 9am every day (as described), or a "Weekly Summary Agent" every Friday, or an "Email notification agent" every hour to check something. - **Data Fetching/Sync:** If agents rely on external data that should be updated periodically, a scheduled job could fetch data (via an agent or a direct function) and update memory or databases. - **Maintenance tasks:** e.g., clearing old logs, refreshing ML models at midnight, or health checks.

We will populate the **Schedule** table with such tasks. For example:

```
id | project_id | agent_id | cron          | task                    | enabled
1  | 7          | 45       | 0 9 * * *     | {"action": "daily_report"} | true
2  | NULL       | NULL     | */5 * * * *   | {"action": "clear_cache"}  | true
```

The first is project-specific, telling agent 45 in project 7 to do a daily_report at 9:00. The second is a system task (no specific project/agent) every 5 minutes to clear cache (just a hypothetical example).

At application startup, we can query for all enabled schedules and register them with APScheduler. If using Celery, the beat schedule could be dynamically built from the DB as well (though usually, beat schedule is configured at start and not dynamic, but we could update schedules in the DB and restart beat or implement a custom scheduler).

Triggering Agent on Schedule: When a schedule triggers, what happens? Two approaches: 1. **Via Message Bus:** The scheduled job simply sends a message to the target agent (through the same message mechanism as any other message). This is clean because it treats the scheduled event like any other event. For example, at 9:00, APScheduler calls:

```
broker.send_message({
    "sender": "system:scheduler",
    "receiver": f"agent:{agent_id}",
    "project_id": project_id,
    "payload": {"command": "create_daily_report", "date": "2025-07-28"}
})
```

The agent then picks it up and handles it. The advantage is the scheduler doesn't need to know agent internals; it just nudges the agent. 2. **Direct Invocation:** Alternatively, the scheduled job could directly call an agent function or Celery task representing the agent's work. For example, Celery could have a task that loads the agent and runs it. But that could bypass the agent's normal messaging logic and might complicate state tracking. Using the message bus keeps it consistent.

We will lean towards **sending messages for agent-related schedules**. For non-agent tasks, the job can directly do the work (like cleanup tasks).

Example - Recurring Agent Task: Daily report was discussed in Agent Communication; let's formalize: - APScheduler job at 9:00 calls `schedule_daily_report()` which does:

```
message = {
    "sender": "system:scheduler",
    "receiver": f"agent:{report_agent_id}",
    "project_id": proj_id,
    "payload": {"action": "create_report", "date": today_date}
}
broker.send_message(message)
```

- The ReportAgent (id = report_agent_id) receives the message and proceeds to generate the report, possibly interacting with other agents or tools. - When the report is ready, it might send its own message to a user or to a storage mechanism. For instance, it could send: `sender=agent:ReportAgent, receiver=system:email_service, payload={"to": admin_email, "subject": "Daily Report", "body": report_text}` if emailing results is required. Or directly to user if they're expecting it in the UI. - If the agent needs to run for a while (say gathering data), it might send intermediate status updates as messages as well (which could be logged or shown in UI if needed).

Integration with Message Queue vs. Celery Task: If using Celery for scheduling and work, the Celery task for daily report could *either* follow the above message approach (i.e., the Celery worker just delegates to message bus and returns immediately), or the Celery task could contain the agent logic itself. However, since our agents are part of the core system, it's cleaner to keep their logic in the agents rather than split between Celery tasks and agent code. So even with Celery, the likely scenario: Celery beat triggers a task,

Celery worker simply sends a message to the agent's queue and maybe waits for a response (or not, if it's fire-and-forget).

Scheduling interface to users: We may provide an API for scheduling. For instance, an admin could call `POST /projects/{id}/schedule` with details like cron timing and target agent/action to create a new schedule. The backend would: - Validate the user has rights (ProjectAdmin). - Insert a row in Schedule table. - Add the job to APScheduler (if running) or if using Celery, we might require a restart or have a way to dynamically update schedule (APScheduler can add jobs on the fly easily). - Respond with the created schedule info.

Timezones: By default we will store and use UTC for all scheduling (and note that to users). APScheduler can be configured with timezone support [39] ; Celery beat uses UTC by default. We might allow specifying a timezone per schedule if needed by converting to UTC internally.

One-off Scheduling: The system might also allow scheduling a one-time task (like "run this agent in 30 minutes"). This could be implemented by scheduling a job with `run_date` (APScheduler supports `date` trigger for one-time [40]). Celery allows scheduling tasks in the future via countdown or ETA as well. So if needed, we can provide an endpoint to schedule a one-time agent run.

Monitoring Scheduled Jobs: We should log when jobs run and if they succeed. APScheduler by default will log if a job throws an exception. We might integrate this with our logging system (structured logs). Also, if a job is missed (server down at that moment), APScheduler can run missed jobs on restart or not depending on settings. For Celery, if the scheduler was down, those runs are missed unless using an external schedule store.

Decision: Given the monolithic and likely initial moderate scale, we will implement **APScheduler** for simplicity. Celery can be introduced if we find APScheduler limiting (for example, if we want multi-process execution of heavy tasks). We'll document both so the development team (or AI assistant building this) understands the design.

In conclusion, the scheduling system ensures the platform can perform **automated, recurring tasks without human initiation**. It works either by directly invoking functionality or by sending messages to agents to do the work. This enables capabilities like: - **Regular data updates** (e.g., an agent scrapes or fetches new info at intervals), - **Timed notifications** (agents summarizing work at end of day), - **Maintenance** (clearing outdated memory entries periodically, etc.).

We'll implement scheduling in a modular way so that switching from APScheduler to Celery or vice versa is not too difficult (perhaps by isolating schedule configuration in one module).

Memory and Vector Database

Agents in this platform are designed to have long-term memory beyond the immediate conversation. This is achieved by storing textual information (documents, past dialogues, notes, etc.) in a vector database for semantic similarity search. The combination of a memory store and an embedding model allows agents to **retrieve contextually relevant information** when responding to user queries or collaborating with other agents.

Vector Database Choice: We will use an open-source vector database such as **ChromaDB**, **Qdrant**, or **Weaviate**. These databases specialize in handling high-dimensional vectors and performing nearest-neighbor searches efficiently. For instance, ChromaDB is specifically geared towards LLM applications and provides a simple local setup with persistent or in-memory storage [14], while Qdrant offers a high-performance vector engine with filtering support and can run as a separate service [41] [42]. For our monolithic setup, ChromaDB is attractive because it can run embedded within our Python process (no separate service needed) and persist data to disk. However, we'll abstract the usage so we could swap in Qdrant (which would run as a separate process on the same machine, accessed via API) if needed for scale.

MemoryEntries and Vector Store Integration: When a new MemoryEntry is added in PostgreSQL (e.g., an agent learns a new fact or ingests a document), the backend will also: - Compute the embedding vector for the content. This uses an embedding model (likely a smaller transformer model or a call to an LLM embedding API if needed, but we prefer local model for privacy – e.g., use SentenceTransformers or Hugging Face embedding model). This could be done synchronously on the API call or as a background task. - Store that vector in the vector database, tagging it with metadata like the MemoryEntry ID, project_id, perhaps the agent_id or any labels [18]. For example, in Chroma:

```
chroma_collection.add(
    documents=[text],
    embeddings=[vector],
    ids=[str(memory_entry_id)],
    metadata={"project_id": project_id, "agent": agent_name, "tags": [...]}
)
```

- The combination of storing in Postgres and vector DB gives us reliability (Postgres) and fast similarity search (vector DB).

Retrieving Context: When an agent needs to recall or find information: - The agent (or a memory utility component) will form a query vector from the question or context it currently has. For example, if the user asks a question, we embed that question using the same embedding model. - We query the vector database for the most similar vectors, filtering by project_id (so we only search that project's knowledge base). Vector DBs allow filtering with metadata; e.g., Qdrant can filter by structured payload conditions [43] [44] and ChromaDB similarly can filter by metadata keys. - The vector DB returns, say, the top 5 results with similarity scores. We then retrieve the corresponding MemoryEntry records (if not already loaded through the vector DB's returned metadata). - Those pieces of text are then provided to the agent (likely by injecting into its prompt or using them to answer). Essentially, we are implementing a **Retrieval-Augmented Generation (RAG)** approach: the agent's LLM response is augmented with knowledge fetched from the vector store relevant to the query [45]. For example, if the user asks, "What did we decide about budget in last meeting?", the agent will vectorize that question, find the nearest memory entries (maybe a chunk of last meeting notes about budget) and include that content when formulating the answer.

Vector DB Schema and Collections: - If using ChromaDB, we might create one **Collection per Project**. Each collection stores vectors for that project's data. This naturally partitions data by project. The collection name could be `project_<id>_memory`. This way, we don't even need to filter by project at query time if we query the specific collection. Chroma can persist data to a directory (which we'll configure). - If using Qdrant (which is server-based), we can either create one index (collection) with a metadata field for project,

or separate collections per project. Qdrant supports filtering by metadata very well [46] [47], so a single collection could suffice up to certain scale. Separate collections might simplify some things (like deletion of a project's data). - Either way, each vector entry in the store should include an ID that ties it to our Postgres MemoryEntry ID, so we can keep them in sync.

Embeddings: The quality of the system depends on the embedding model. We will likely use a pre-trained model like `all-MiniLM-L6-v2` (a popular 384-dimensional sentence embedding model) or any similar model from HuggingFace for generating text embeddings. This can be run locally on CPU or GPU. The embedding model can be loaded at startup as part of our **GPU Inference Services**. For instance, we might have an endpoint or internal function `embed_text(text: str) -> list[float]` that uses a Transformer model (like SentenceTransformer or a HuggingFace pipeline) to produce the embedding.

The vector DB stores these embeddings and can compare them using cosine similarity or dot product. We will choose a distance metric appropriate to the embedding model (commonly cosine similarity). Many vector DBs default to cosine or allow choosing – Qdrant, for example, allows specifying metric (Cosine, Dot, Euclidean) at collection creation [48] [49]. We'll ensure to configure that consistently with how we use the embeddings (cosine similarity is common for text embeddings).

Memory Management: Over time, memory entries might grow large. We should consider some strategy for relevance or expiration: - Possibly limit how many memory entries an agent or project keeps (to avoid performance issues). - Implement a retrieval cutoff or a way to archive older ones (maybe move to long-term storage if rarely needed). - These are future concerns; initially, we will assume the data volume is manageable.

Using Memory in Agents: The agent's logic (likely within `handle_message`) when faced with a query will: 1. Maybe parse the query for keywords to decide if it should do a knowledge lookup (some agents might always do a lookup for user questions). 2. Call the memory retrieval function (embedding and vector search). 3. Get back top relevant pieces of content. 4. Construct its LLM prompt to include those pieces. For example, if using an OpenAI GPT-like model, the system or assistant prompt can say: "Here is some context: \n[content]\nUser's question: ..." 5. The agent LLM then uses this to answer more accurately about specific details not in its base knowledge.

This mechanism addresses the known limitation that LLMs have a fixed context window and cannot remember arbitrary long histories or large knowledge bases by themselves. By using a vector DB, we **augment the agent's knowledge on-the-fly** [50] [51] – the agent can access information outside of what was in the prompt by explicitly querying its memory.

Example: Suppose we have a Project that stores documentation. A DocumentationAgent can answer questions by looking up relevant docs. - The docs have been chunked and stored in MemoryEntries with embeddings. - User asks "How do I reset my password?". - The agent's memory retrieval finds a chunk of text from the documentation that mentions "reset password" steps. - The agent then responds, citing those steps. If the documentation updates, we would update the memory store with new embedding entries (and possibly remove outdated ones).

Project-Specific Memory: Each project's memory is separate. So an agent in project A will not retrieve knowledge from project B's memory. This is ensured by collection partitioning or metadata filter. This provides data isolation (important for multi-tenant scenarios or just organizational clarity).

Data Ingestion Tools: We might provide tools or endpoints to populate MemoryEntries. E.g., a “File Import” where a PDF or text can be uploaded, chunked, embedded, and saved. Or an agent could read and summarize a webpage and store key points. This is beyond core design, but the infrastructure supports it.

Search capability: We could also have endpoints for searching the memory (for users with access). For example, a project admin might want to manually query the vector DB (“knowledge base search”). We can expose an API that takes a query and returns the top relevant MemoryEntries.

To sum up, the **Memory System** is built around the combination of: - **Embedding model** (transform text to vector). - **Vector database** (store and search vectors). - **MemoryEntries table** (tie vectors back to original text and context).

By using a vector DB, our agents can **“store encoded unstructured objects, like text, as lists of numbers and compare them to find relevant documents for a given query”** [51] . This greatly enhances their abilities, letting them recall information not in the prompt or even not originally known to the base model. It effectively gives each project a custom knowledge base the agents can learn from and refer to.

We ensure that this component is well-integrated but also modular: if in the future we switch to a different vector DB or if we incorporate an external knowledge store, we should only need to adjust the memory service implementation, not the agent logic or other parts of the system.

GPU Inference Services

The backend includes local AI model inference capabilities to support agent reasoning and embedding generation. Rather than relying exclusively on external AI APIs, we host models on our server (especially since we want a multi-agent setup possibly running offline or on-prem). We utilize **PyTorch** and **Hugging Face Transformers** libraries to load and run these models, leveraging GPU acceleration when available.

Local LLM Endpoints: We will expose internal API endpoints (or just internal functions) for AI tasks such as: - Generating text (given a prompt, return a completion or response). - Computing embeddings for a piece of text. - Perhaps other utilities like classification or summarization if needed, though those can also be done via prompting a generative model.

For example, we may have an endpoint: `POST /llm/generate` which accepts a JSON body like `{"prompt": "...", "max_tokens": 200}` and returns `{"completion": "..."}`. Agents can call this endpoint (or even better, an internal Python function if in same process) to get LLM outputs.

Another endpoint: `POST /llm/embed` could accept `{"text": "..."}` (or a list of texts) and return the embedding vector(s).

However, since these calls are internal (agents could just call a function to do it), we might not need to expose them to external clients. For development and possibly debugging, having them as endpoints is useful. Also, if we ever separate the inference to another process (for example, run heavy models in a separate service), having a REST interface would allow that. But in the monolith, they can be part of the same FastAPI app.

Model Loading: On application startup, we will load the necessary models into memory. For instance: - Load a transformer language model (like GPT-2, GPT-J, LLaMA 2, or any appropriate model depending on hardware capacity). We might use `AutoModelForCausalLM` or `AutoModelForSeq2SeqLM` depending on the type (for chat, a causal LM is typical). - Load a tokenizer for it. - If a large model, ensure device placement on GPU: e.g.,

```
device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
model = AutoModelForCausalLM.from_pretrained(model_name, device_map="auto" or
to=device)
model.eval()
```

Similarly for embedding model:

```
embed_model = AutoModel.from_pretrained(embedding_model_name)
embed_model.to(device).eval()
```

(Or use SentenceTransformer which abstracts this.) - Possibly allocate multiple models: one for generation (LLM) and one for embeddings (which might be smaller). Or we use one model for both if applicable (some LLMs can do embeddings by taking an embedding of a special token, but often it's better to use a dedicated model or at least a different invocation mode). - We also consider model size and GPU memory. If multiple agents share one large model, that's fine – they will serialize access (or we handle concurrent calls by batch processing or queue). - We should **warm up** the models after loading: run a dummy inference. For example, run a single token through the LLM to initialize CUDA kernels. As noted, warming up can reduce first query latency [17] .

Serving Inference Requests: With models loaded, our endpoints (or internal functions) will handle requests: - For generation, we might use the model's `generate` method. Because this can be slow for large outputs, we may want to offload it to a background task if the API call should return immediately. However, since agents likely work asynchronously, it might be fine to `await` the generation in the agent's logic (not blocking other tasks due to async nature). - We can also incorporate features like streaming output (if using fastapi's StreamingResponse or websockets) for real-time token streaming to the UI, but that's a stretch goal. Initially, returning the full completion is okay. - For embeddings, since they're usually quick (a single forward pass), a normal request/response is fine.

Concurrency & Batching: If multiple agents ask the LLM at once, we have a concurrency challenge. Running multiple large model forwards concurrently might overload the GPU or cause context switching that slows overall throughput. Strategies: - **Queueing:** We could funnel all LLM generate requests through a single worker (or a limited pool) that processes them one by one or in micro-batches. This ensures the GPU isn't thrashing. We can implement this by having an `LLMManager` that agents call, rather than calling the model directly. The LLMManager could accumulate requests that come at nearly the same time and do a batch generate if the model and library support it (many transformer models can generate in batch if prompts are of equal length or padded). - **Parallel if possible:** If the model is smaller or if using a multi-GPU setup, we could allow some parallelism. But in a monolithic single GPU scenario, likely serializing is safest for stability.

Initially, a simple approach is fine: allow async tasks to call the model. PyTorch releases the GIL during compute, but if two tasks try to use the same model concurrently, PyTorch will likely execute them sequentially anyway on the GPU (unless using multithreading). We may wrap model calls with an asyncio.Lock to ensure one at a time for now. This can be optimized later.

Model Selection: We might enable different agents to use different models (some lightweight, some heavy). The Agents table's `model_config` might have a model name. The GPU service can manage multiple models loaded (if VRAM allows) or load on demand. However, loading on demand is slow (many seconds), so for now we assume one primary model for all agent conversations. This is a design decision to keep things simpler and fit in memory. If needed, we can run separate processes or threads for different models.

Example Implementation of an LLM Endpoint:

```
from pydantic import BaseModel

class GenerateRequest(BaseModel):
    prompt: str
    max_tokens: int = 200
    temperature: float = 0.7

@app.post("/llm/generate")
async def generate_text(req: GenerateRequest):
    async with llm_lock:  # ensure single generation at a time (optional)
        inputs = tokenizer(req.prompt, return_tensors="pt").to(device)
        outputs = model.generate(**inputs, max_new_tokens=req.max_tokens,
do_sample=True, temperature=req.temperature)
        result_text = tokenizer.decode(outputs[0], skip_special_tokens=True)
        return {"completion": result_text}
```

This is a simplified example. In practice, for chat models, we might need to format prompts with roles (system/user/assistant roles) and possibly use a chat-specific model or a prompt template.

Performance Considerations: Running LLMs is resource-intensive. We should monitor GPU memory and utilization. If the system is expected to handle multiple queries per second, a very large model might not suffice. We could opt for medium-sized models (like 6B or 7B parameter models) or even use quantization to reduce memory. Since this is a design doc, the exact model isn't fixed, but we'll note these possibilities in implementation.

We also ensure that the **GPU inference endpoints are secured** – probably we won't expose them to the outside world except through the agent logic. If they were exposed, one could misuse them to generate lots of text (like an open API). But since only authenticated/authorized parts of our app (like the agent or maybe an admin UI) call them, it's controlled.

Example of Embedding Use:

```
@app.post("/llm/embed")
async def embed_text(req: BaseModel):
    # req could have a 'text' field or 'texts' list
    text = req.text
    with torch.no_grad():
        inputs = embed_tokenizer(text, return_tensors="pt").to(device)
        vector = embed_model(**inputs).last_hidden_state.mean(dim=1)  # e.g.
mean pooling for sentence embedding
        vec_list = vector.cpu().numpy().tolist()[0]
    return {"embedding": vec_list}
```

(This depends on how the specific model returns embeddings; some models provide sentence embeddings directly.)

Tool Inference Services: In addition to the core LLM and embedding, if we plan to use any ML models as agent tools (e.g., a sentiment analysis model or image recognition), we could similarly wrap them in endpoints or functions. For example, an image processing agent might send an image to an endpoint `/vision/detect_objects` which runs a PyTorch vision model. The design supports adding such endpoints as needed.

Testing and Usage: We will test that the inference pipeline works by calling these endpoints with sample input. For instance, hitting `/llm/generate` with a simple prompt should yield a coherent completion. Also verify that `torch.cuda.is_available()` is True on the target deployment (with proper drivers installed) so that models do use the GPU. If running on CPU (like on a Mac or a dev laptop without GPU), it will still work but slower.

Integration with Agents: Agents will typically not call the HTTP endpoints but rather use an internal interface to the models. We can abstract it such that:

```
class LLMService:
    def __init__(self, model, tokenizer):
        ...
    async def generate(prompt:str, **kwargs) -> str:
        # similar to above, but no HTTP overhead
```

Then agents can call `await llm_service.generate(prompt)`. Underneath, it might acquire the lock and do the model.generate as shown. This is more efficient than the agent making an HTTP call to itself. However, having the endpoints is still useful for testing or if we allow some admin to query the model directly.

In addition, if we wanted to split the model serving to a separate process (for example, keep the main API reactive and offload heavy model computation to a worker), these endpoints could be hosted on a separate FastAPI (or other) service. But as a monolith, we'll run them in-process.

Resource Management: We will add a **healthcheck** for the models perhaps. E.g., an endpoint `/llm/health` that returns the models loaded and maybe memory usage. Also, if a model is too large to keep always loaded, we might consider lazy loading (not load until first request), but that adds latency. We prefer upfront load so that user interactions are fast after startup.

Example of Use in agent flow: Agent receives a message that requires an answer. Pseudocode:

```
async def handle_message(agent_id, message):
    if message.type == "query":
        query = message.content
        # incorporate system prompt and context
        prompt = agent.system_prompt + "\nUser: " + query + "\nAssistant:"
        # possibly retrieve memory entries and append to prompt as context
        related_info = memory.retrieve(agent.project_id, query)
        if related_info:
            prompt = agent.system_prompt + "\n[Context]\n" + related_info +
"\n[EndContext]\nUser: " + query + "\nAssistant:"
        response = await llm_service.generate(prompt, max_tokens=200)
        # send response back
        send_message({
            "sender": f"agent:{agent_id}",
            "receiver": message.sender,  # user or agent who asked
            "project_id": agent.project_id,
            "payload": response
        })
```

This shows how the GPU service (llm_service) is invoked as part of the agent's message handling. Similarly, for embedding:

```
embedding = await llm_service.embed(text)
memory_index.add(text, embedding, metadata={...})
```

This architecture is essentially adding a local "AI brain" accessible via the llm_service.

To conclude this section: The backend's **GPU Inference Services** enable the AI agents to actually "think" and "remember." By hosting models locally, we maintain data privacy and potentially reduce cost (no per-query API fees). We have to manage the complexity of running these models, but frameworks like FastAPI and PyTorch are up to the task. We'll put emphasis on: - Proper initialization (device selection, model loading), - Efficient usage (locking, maybe batching), - Error handling (if the model fails to generate, e.g., out-of-memory, we catch and possibly recover by freeing some memory or returning an error message). - And extensibility (we can add new models or replace them as needed, since the rest of the system just calls an interface).

With these services in place, the multi-agent platform is essentially self-contained: it can process language (LLM), recall knowledge (vector DB), and even perform specialized tasks via additional models or tools, all on the local machine.

Logging and Monitoring

A robust logging and monitoring strategy is vital for operating and debugging the platform, especially with AI agents whose reasoning we may need to trace. We will implement **structured logging**, health checks, and lay groundwork for advanced monitoring (potentially even using an agent to monitor others in the future).

Structured Logging: Instead of unstructured text logs, we use a structured format (like JSON) for logs. This makes it easier to filter and analyze log data using tools (e.g., ELK stack, CloudWatch, etc.). We will configure Python's `logging` module to output JSON lines with key details. For example: - Include timestamp, log level, and message. - For HTTP access logs, include request method, URL, status code, and response time. - For errors, include stack trace and error message. - For agent events, include agent id and message ids involved.

We can achieve this by setting a custom `logging.Formatter` that formats records as JSON. A simple implementation might just put the message and extra fields into a dict and serialize [52] [53]. In code:

```
import logging, json
class JsonFormatter(logging.Formatter):
    def format(self, record):
        log_record = {
            "level": record.levelname,
            "message": record.getMessage(),
            "time": self.formatTime(record, datefmt="%Y-%m-%dT%H:%M:%S")
        }
        # Include extra context if available
        for key in ["req", "res", "err", "agent"]:
            if key in record.__dict__:
                log_record[key] = record.__dict__[key]
        return json.dumps(log_record)
```

We then attach this formatter to our log handlers (stdout handler typically) [54] [55]. We may disable Uvicorn's default access logger and use our own middleware to log requests in our format [56] [57].

Example of a structured log entry:

```
{
  "level": "INFO",
  "message": "Incoming request",
  "time": "2025-07-28T09:53:22",
  "req": {"method": "GET", "url": "/projects/7/agents"},
```

```
  "res": {"status_code": 200}
}
```

This was achieved by logging in a middleware after processing a request [58] [53]. Similarly, for agent messages we could log:

```
{
  "level": "INFO",
  "message": "Agent message processed",
  "time": "...",
  "agent": 45,
  "req": {"sender": "user:123", "receiver": "agent:45", "msg_id": "abc123"},
  "res": {"status": "completed", "response_summary": "Daily report generated."}
}
```

Here we might treat the agent message handling as a "request" for logging purposes: `req` being the message and `res` the outcome.

Agent Debug Logs: We anticipate that debugging agent behavior might require logging the agent's chain-of-thought or tool usage. For instance, when an agent decides to use a tool, we can log something like:

```
{"level": "DEBUG", "message": "Agent 45 invoking tool SearchWeb with query X"}
```

And when the result returns:

```
{"level": "DEBUG", "message": "Agent 45 received tool result with summary: ..."}
```

We have to be careful with log volume (especially if agents produce a lot of intermediate text). Perhaps keep detailed reasoning logs at DEBUG level so they can be turned on when needed.

Error Logging: Any exceptions (in request handlers, agent loops, scheduled tasks) should be logged with stack traces. In structured logs, we could include the exception info in an `err` field [59]. Python's logging can capture exception info with `exc_info=True`.

For example:

```
{
  "level": "ERROR",
  "message": "Exception in agent loop",
  "agent": 46,
  "err": "Traceback (most recent call last): ... ZeroDivisionError: division by
```

```
zero"
}
```

This helps in monitoring via centralized log systems.

Healthcheck Endpoints: We'll create a simple healthcheck endpoint, e.g., `GET /healthz` or `/health`. This will return a 200 OK with some basic info like `{"status": "ok", "uptime": "...", "db": "ok", "redis": "ok"}`. Internally it can perform quick checks: - Ping the database (SELECT 1). - Ping Redis (PING command). - Possibly check that the vector DB is reachable (if separate, or if using Chroma in-process, maybe not needed). - If critical, check GPU availability or model loaded (could be part of a separate `/healthz/models` check). - If any check fails (exception or slow response), report unhealthy (500 or include details).

This endpoint is useful for Kubernetes liveness/readiness probes or general monitoring.

We might also add a `/metrics` endpoint in the future (to expose Prometheus metrics for requests per second, etc.) but that can be done later. Right now, logs and health are primary.

Monitoring (Future – Agent-based): The prompt specifically asks for notes on *future agent-based monitoring*. This suggests an idea: using an AI agent to monitor the system itself. For instance, an "MonitoringAgent" could read logs or metrics and act if something's wrong. This is speculative but we can outline a possibility: - A special agent that subscribes to system events (like errors, performance metrics). - If it detects anomalies (e.g., many errors in a short time, or an agent stuck in a loop), it could alert administrators or even attempt corrective actions (like restarting an agent or freeing some memory). - This agent could use an LLM to analyze log text for patterns. For example, integrating something like described by Vishal Mysore: *"integrating Agentic AI into log management allows systems to autonomously process logs and trigger actions based on log content"* [60]. The logs become not just static records but triggers for the monitoring agent to reason about them. - For now, we will not implement such an automated AI Ops agent, but the architecture (with the message bus and all logs accessible) could allow it. We note it as a future enhancement: a monitoring agent that could, for example, summarize the system health daily, or notice if an agent is producing repetitive errors and temporarily disable it or alert a human.

Standard Monitoring Tools: In addition to our custom logging: - We can integrate with **Prometheus** for metrics: e.g., track number of requests, their latency, number of messages processed by each agent, GPU utilization, etc. This might involve instrumenting the code or using existing middleware (there are FastAPI instrumentations available). - Use **Sentry or similar** for error tracking. Sentry SDK can capture exceptions and tracebacks, which can be very helpful for debugging in production. - For resource monitoring: we might output logs on memory usage periodically, or ensure the host OS metrics are accessible.

Log Persistence: In development, console logs are fine. In production, we should persist or ship logs to a file or logging service. We might configure the logging to write to a file (with rotation) and to stdout. Many deployments will capture stdout anyway.

Security in Logs: We must avoid logging sensitive info like passwords or full JWTs. We'll be mindful to log IDs or hashed values rather than raw secrets. Structured logging helps because we control exactly what fields get output.

Example: Monitoring specific events – If an agent fails to respond, we might emit a warning log. If the scheduler job runs longer than expected, log a warning. If memory DB query returns nothing and it should have, maybe log that (or at info level).

Using Logs for Auditing: Because multi-agent systems can be unpredictable, having a comprehensive log of agent decisions and interactions is crucial for auditing why an agent did X. Our message logs plus debug info provide a trace. Potentially, we could even implement a "replay" feature (for debugging) where the sequence of messages is fed back into agents to reproduce an outcome (this would need deterministic agents or fixed random seeds). For now, just keeping the data (messages, timings) is the first step.

Summary: The logging system will produce machine-readable logs that can be ingested by observability tools. It will log requests, agent interactions, and errors with appropriate context. The monitoring aspect includes: - Health endpoints for external systems to check. - Option to integrate alerting (e.g., if health check fails or error rate is high). - A notion that down the line, an AI agent might be tasked with observing these logs to automatically diagnose issues (aligning with the concept of AI operations). For instance, such an agent could detect that *"Agent A has failed 5 times today with memory errors"* and alert accordingly, demonstrating how logs can become "triggers for intelligent actions" [60] .

Implementing these logging and monitoring features will ensure that as we develop and run the platform, we have visibility into what's happening and can maintain reliability and trust in the system's operations.

Setup Instructions

Setting up the development and runtime environment for this FastAPI monolithic backend involves preparing the Python environment, databases, and supporting services (PostgreSQL, Redis, etc.), and then running the server. Below are step-by-step instructions for Ubuntu/Linux and macOS (the process is similar).

1. System Requirements and Preparation

- Ensure you have **Python 3.10+** installed (for FastAPI and Pydantic v2 support). Python 3.11 is recommended if available for better performance.
- For GPU usage: You need an NVIDIA GPU with the appropriate CUDA drivers if you plan to use GPU inference. Install CUDA toolkit and cuDNN as required. Alternatively, use CPU-only if GPU is not available (just be aware of slower AI performance).
- Install system dependencies:
- On Ubuntu, you might need development tools and libraries: `sudo apt update && sudo apt install -y build-essential python3-dev`.
- For PostgreSQL, either install the server: `sudo apt install -y postgresql` (and start the service), or use Docker (see below).
- For Redis, `sudo apt install -y redis-server` (or use Docker).
- (Optional) If using a vector DB like Qdrant and not embedding it, you may need to run it via Docker or install it.

2. Python Environment Setup

It's best to use a virtual environment to isolate dependencies:

```
python3 -m venv venv           # create virtual environment
source venv/bin/activate       # activate it (on Windows, use venv\\Scripts\
\activate)
```

Now, install the required Python packages. In the project repository, there should be a `requirements.txt` or `pyproject.toml`. If `requirements.txt` is provided, run:

```
pip install -r requirements.txt
```

This will install FastAPI, Uvicorn, databases libraries, etc. For example, the requirements likely include: - `fastapi` - `uvicorn[standard]` - `sqlalchemy` (or `sqlmodel`) - `psycopg2-binary` (PostgreSQL driver) [61] [62] - `redis` (Python client for Redis) - `apscheduler` - `pydantic` - `python-jose[cryptography]` or `PyJWT` (for JWT auth) - `passlib[bcrypt]` (for password hashing) - `chromadb` or `qdrant-client` - `torch` (PyTorch) and `transformers` - possibly `uvicorn[standard]` brings in uvloop and other performance improvements.

If any of these are missing, use `pip install <package>` to get them.

Note: On M1/M2 Macs, installing PyTorch via pip might need a specific command (or use Conda). The transformers package also might require `pip install transformers`.

3. Database Setup (PostgreSQL)

If you have PostgreSQL installed locally: - Start the PostgreSQL service (`sudo service postgresql start` on Ubuntu, or if on macOS using Homebrew, `brew services start postgresql`). - Create a database and user for the app:

```
sudo -u postgres psql    # open postgres shell on Ubuntu
# Within psql:
CREATE DATABASE ai_platform;
CREATE USER ai_user WITH PASSWORD 'ai_password';
GRANT ALL PRIVILEGES ON DATABASE ai_platform TO ai_user;
\q
```

Adjust user/password as desired (and update the app's config accordingly).

If using Docker for the DB: - Ensure Docker is installed and running. - Run: `docker run -d --name postgres -p 5432:5432 -e POSTGRES_PASSWORD=ai_password -e POSTGRES_USER=ai_user -e POSTGRES_DB=ai_platform postgres:14` - This will start a PostgreSQL container accessible on port 5432.

Apply migrations or schema: If the project uses Alembic for migrations, run `alembic upgrade head` to create tables. Otherwise, if a SQL script is provided for schema, run that. For a fresh project, we might use SQLAlchemy's `create_all()` during first run, but explicit migrations are better.

Make sure the database connection URL is set as an environment variable or config (like `DATABASE_URL=postgresql://ai_user:ai_password@localhost:5432/ai_platform`). Our FastAPI app likely reads this in a config class (using Pydantic BaseSettings) [63].

4. Redis Setup

Start Redis: - If installed locally, simply run the service (`redis-server` or it might already be running as a service on port 6379). - With Docker: `docker run -d --name redis -p 6379:6379 redis:7`.

No further setup needed for Redis (no schema). Just ensure it's reachable. The app might also use an environment variable like `REDIS_URL=redis://localhost:6379/0`.

5. (Optional) Vector Database Setup

If using ChromaDB in-process, no external service needed. It will create files for persistence (by default in `./chroma_data` or a configured folder). Ensure the app has write permission to whatever directory is configured for Chroma.

If using Qdrant: - Run Qdrant via Docker: `docker run -d --name qdrant -p 6333:6333 qdrant/qdrant` - This exposes Qdrant on port 6333. Set `QDRANT_URL=http://localhost:6333` in the app config. - You may want to volume-mount for persistent storage.

If using Weaviate, similar approach (Weaviate has a Docker image too). But let's assume Chroma for dev simplicity.

6. Environment Configuration

Typically, there will be a `.env` file or environment variables expected by the app. Common ones: - `DATABASE_URL` (as mentioned) - `REDIS_URL` - `SECRET_KEY` for JWT signing (generate a random string for this). - `ENV` or `DEBUG` flag (to toggle debug mode). - Possibly `VECTOR_DB` choice or connection info. - If using Celery: `CELERY_BROKER_URL` (likely same as REDIS_URL) and `CELERY_RESULT_BACKEND`. - If using external LLM APIs (not in our case, but if OpenAI API was an option, there'd be an API key env).

Create a `.env` file in the project root (and ensure it's listed in .gitignore). An example `.env`:

```
DATABASE_URL=postgresql://ai_user:ai_password@localhost:5432/ai_platform
REDIS_URL=redis://localhost:6379/0
SECRET_KEY=supersecretkeychangeit
```

FastAPI's settings (via Pydantic BaseSettings) can auto load from .env or environment.

7. Running the FastAPI Server

Activate the venv (if not already) and run:

```
uvicorn app.main:app --reload
```

Replace `app.main:app` with the actual import path to the FastAPI application instance (for example, in our project structure, maybe the main file is `main.py` with `app = FastAPI()` in it, so `uvicorn main:app --reload` from within that directory). The `--reload` flag is for development (auto-restarts on code changes).

You should see Uvicorn startup logs:

```
INFO:     Uvicorn running on http://0.0.0.0:8000 (Press CTRL+C to quit)
```

By default, it's on port 8000. If you want to change, add `--port 8080` etc.

Test quickly: - Open a browser or use curl: `curl http://localhost:8000/healthz` (or the appropriate health endpoint). Expect an "ok" response. - If there's a docs interface, visit `http://localhost:8000/docs` to see the Swagger UI for the API.

8. Running the Additional Services (Celery workers, etc.)

If you configured Celery for background tasks: - Start a Celery worker process: `celery -A app.celery_app worker --loglevel=info` (the `-A` should point to where the Celery app is defined). - Start Celery beat scheduler: `celery -A app.celery_app beat --loglevel=info`. These should connect to the Redis broker you set. Ensure they're running concurrently with the main FastAPI (you might use separate terminal windows or a process manager like supervisord).

If using APScheduler, no extra process; it runs within FastAPI.

For development, you might not run Celery at all and rely on APScheduler within the app.

9. Verifying GPU Inference Setup

If you plan to use the local models: - Ensure PyTorch sees the GPU: run a quick Python snippet:

```
import torch; print(torch.cuda.is_available())
```

This should print `True` if GPU is set up. If `False`, double-check driver installation or that you're running on a machine with a GPU. - The first time you use a HuggingFace model, it will download weights. E.g., if our code loads `facebook/opt-1.3b`, it will download that model (~2.7GB) to `~/.cache/huggingface`. Ensure you have internet for that initial download or have the model files locally. You can also use smaller

models for quick tests. - Test the model endpoint if exposed: e.g. `curl -X POST http://localhost:8000/llm/generate -H "Content-Type: application/json" -d '{"prompt": "Hello AI, how are you?"}'`. Expect a JSON with a completion. (In dev mode, the first request might take some time due to model initialization.)

10. Next.js Frontend Integration (brief mention)

Although this doc is about the backend, when running the frontend (Next.js app), you'll typically: - Set NEXT_PUBLIC_API_URL to `http://localhost:8000` (or whatever address FastAPI is on). - Ensure CORS is configured on FastAPI to allow the Next.js domain (in dev, http://localhost:3000). We might add a middleware in FastAPI:

```
from fastapi.middleware.cors import CORSMiddleware
app.add_middleware(CORSMiddleware, allow_origins=["http://localhost:3000"],
allow_credentials=True, allow_methods=["*"], allow_headers=["*"])
```

This allows the dev frontend to call the backend. In production, adjust allowed origins accordingly (e.g., your domain).

11. Running in Docker (optional)

For a consistent environment, we might provide a Docker setup: - A Dockerfile for the FastAPI app (based on python:3.10, installing requirements, copying code, exposing 8000, and running uvicorn). - A `docker-compose.yml` to run the app along with Postgres, Redis (and possibly Qdrant if using). - If GPU is needed in Docker, use NVIDIA Docker base image and runtime arguments.

Given the complexity of GPU in Docker, you might develop locally first, then containerize once things work.

12. First-Time Initialization

After everything is running: - Create a superadmin user. There might be an API to register or you might need to insert into the database manually for the first user. If a CLI script is provided, use that. Otherwise, one approach is to run a one-off script or open a DB console:

```
psql -U ai_user -d ai_platform
INSERT INTO users (username, password_hash, role) VALUES ('admin', '<hash>',
'Superadmin');
```

To get a password hash, you can use a Python snippet with Passlib. But perhaps the backend has an `/auth/register` endpoint (if open registration is allowed, or a specific script for admin creation). - Verify you can obtain a JWT by hitting the login endpoint (maybe `/auth/login` with admin credentials). - Test creating a project, an agent, etc., through the API or via the frontend if connected.

13. Development Workflow

- Use `uvicorn --reload` for auto-reload on code changes. (Do not use `--reload` in production.)
- You can run tests (if provided) perhaps with `pytest`.
- Check logs in the console for any errors during startup or usage. Our structured logs will appear as JSON; if that's hard to read, you might adjust log format to something simpler during dev, or use a tool to pretty-print JSON logs.

14. Production Considerations

When moving to production: - Use a process manager or container orchestrator. For example, in Docker, you'd run `uvicorn app.main:app --host 0.0.0.0 --port 8000 --workers 1` (or more workers if the machine is powerful, but careful with APScheduler duplication). - Set appropriate environment variables (especially SECRET_KEY to a secure value, and configure DB/Redis endpoints). - Possibly use Gunicorn with Uvicorn workers for multiple processes (e.g., `gunicorn -w 4 -k uvicorn.workers.UvicornWorker app.main:app`). - Ensure the GPU drivers are present on the host if using GPU. - Enable persistence for Postgres (volume or managed DB) and for vector DB if using one with persistent storage. - Set up a logging solution (the JSON logs can be captured by services like Datadog, ELK, etc., and we might integrate that). - Configure domain and TLS if exposing to the web (e.g., behind an Nginx reverse proxy or using a cloud service).

Finally, with everything up and running, the multi-agent backend should be fully functional. The Next.js frontend can communicate with it to authenticate users, create projects, initiate agent queries, and receive responses. Agents will be running within the backend, ready to process messages and utilize the integrated memory and inference systems.

By following this setup guide, developers or devops engineers can get the platform running locally for development or in a server environment for production, ensuring all moving parts (FastAPI app, Postgres, Redis, vector DB, GPU) are properly configured and connected.

---

1 2 3 7 8 10 Creating a Scalable Full-Stack Web App with Next.js and FastAPI | by Vijay Potta (pottavijay) | Medium
https://medium.com/@pottavijay/creating-a-scalable-full-stack-web-app-with-next-js-and-fastapi-eb4db44f4f4e

4 21 22 23 24 25 BFF - Backend for Frontend Design Pattern with Next.js - DEV Community
https://dev.to/adelhamad/bff-backend-for-frontend-design-pattern-with-nextjs-3od0

5 6 26 27 31 32 Event-Driven Patterns for AI Agents : r/LangChain
https://www.reddit.com/r/LangChain/comments/1ha8mrc/eventdriven_patterns_for_ai_agents/

9 61 62 Fast API & PostgreSQL: A Match Made in Heaven ⚡ | by Bryan Antoine | Medium
https://medium.com/@b.antoine.se/fast-api-postgresql-a-match-made-in-heaven-5309aa11221c

11 Redis as a Message Broker: Deep Dive - DEV Community
https://dev.to/nileshprasad137/redis-as-a-message-broker-deep-dive-3oek

12 33 34 35 39 40 Running Scheduled Jobs in FastAPI
https://www.nashruddinamin.com/blog/running-scheduled-jobs-in-fastapi

13 36 37 38 Periodic Tasks — Celery 5.5.3 documentation
https://docs.celeryq.dev/en/stable/userguide/periodic-tasks.html

14 18 45 50 51 Embeddings and Vector Databases With ChromaDB – Real Python
https://realpython.com/chromadb-vector-database/

15 16 17 How to Deploy FastAPI Applications with GPU Access in the Cloud
https://www.runpod.io/articles/guides/deploy-fastapi-applications-gpu-cloud

19 FastAPI Role Base Access Control With JWT - Stackademic
https://stackademic.com/blog/fastapi-role-base-access-control-with-jwt-9fa2922a088c

20 FastAPI Security: How I Nailed JWT, Role-Based Access & More for ...
https://python.plainenglish.io/fastapi-security-how-i-nailed-jwt-role-based-access-more-for-bulletproof-apis-fa4ba473860f

28 29 30 Multi-agent System Design Patterns From Scratch In Python | ReAct Agents | by Prince Krampah | . | Medium
https://medium.com/aimonks/multi-agent-system-design-patterns-from-scratch-in-python-react-agents-e4480d099f38

41 42 43 44 46 47 48 49 Building a Semantic Search System with Qdrant and FastAPI: A Practical Guide to AI-Powered Customer Support Service | by Ibrahim Halil Koyuncu | Medium
https://ibrahimhkoyuncu.medium.com/building-a-semantic-search-system-with-qdrant-and-fastapi-a-practical-guide-to-ai-powered-customer-84cd1abee3a8

52 53 54 55 56 57 58 59 Structured JSON Logging using FastAPI
https://www.sheshbabu.com/posts/fastapi-structured-json-logging/

60 Leveraging Agentic AI for Automated Log Management | by Vishal Mysore | Medium
https://medium.com/@visrow/leveraging-agentic-ai-for-automated-log-management-75866b2f78d6

63 Adding a production-grade database to your FastAPI project — Local Setup | by David Egbert | Python in Plain English
https://python.plainenglish.io/adding-a-production-grade-database-to-your-fastapi-project-local-setup-50107b10d539?gi=10f13798c539