CyFortis Training Assistant – Design & Architecture

1. Problem & goals

Security awareness training is often tracked in a backend system that only admins can understand.
The assignment asks for an **AI assistant** that lets:

- **Employees** check whether they've finished their required cybersecurity training and which videos are missing.
- **The CISO** inspect a single employee's status and training summary, or query all employees by status and see statistics (min/max/avg time to complete, fastest, slowest).

The assistant must:

- Enforce **authentication** with `employee_id` + `employee_name` before exposing any training data.
- Infer intent from **natural language** and call the right backend queries.
- Stay **strictly on-topic** (security training only).
- Operate in a **read-only** way and avoid mutating global state.

The solution should be implemented as:

- **Frontend** – React (TypeScript) chat UI.
- **Backend** – Python + FastAPI.
- **Model** – OpenAI-compatible LLM.
- **Data** – Provided SQLite database (`employees.db`).

---

2. Requirements → implementation mapping

This section maps the assignment's functional requirements to concrete pieces in the implementation.

- **Employee: check completion & missing videos**
    - Implemented via LLM tools that:
        - Fetch an employee's training status.
        - List completed vs missing videos.
    - Backed by read-only helper functions in `backend/app/db/` .
- **CISO: single employee view**
    - Tools for:
        - Getting an employee's overall training status (`NOT_STARTED` , `IN_PROGRESS` , `FINISHED`).
        - Returning a concise summary of their progress.
- **CISO: aggregate view & statistics**
    - Tools that:
        - List employees by status.
        - Compute min/max/avg time to complete, and identify fastest/slowest employees.
    - These tools are CISO-only and use cached queries for performance.
- **Authentication (ID + name)**
    - Backend checks `(employee_id, employee_name)` via `employee_exists_in_database` in `queries.py` .
    - If the record doesn't exist, the agent responds as "unknown user" and does not expose training data.
- **Guardrails & read-only behavior**
    - System prompt instructs the LLM to:
        - Stay on cybersecurity training only.
        - Politely refuse out-of-scope questions.
    - All tools are read-only wrappers over SQLite; there are no write/update operations.

---

3. System architecture

3.1 Overview

The system is split into:

- **Frontend**: React + Vite TypeScript app providing a chat UI.
- **Backend**: FastAPI service exposing a single `POST /chat` endpoint.
- **LLM layer**: OpenAI-compatible client configured with system prompts and tool definitions.
- **Data**: SQLite database `backend/data/employees.db` .

All conversations flow through:

1. The frontend sends chat history (+ optional `employee_id` , `employee_name`) to `/chat` .
2. The backend:
    - Authenticates the user.
    - Builds the LLM call with system prompt, messages, and tool definitions.

    - Lets the model select tools.
    - Executes the corresponding tool handlers (read-only DB queries).
3. The backend returns a structured `ChatResponse` to the frontend, which renders it in the chat UI.

3.2 Single `/chat` endpoint

All roles and flows (employee or CISO) are unified behind `POST /chat`:

- The frontend never talks directly to the database.
- Auth logic, tool invocation, and LLM prompts all live in one place.
- This makes it simple to add:
    - New tools.
    - Additional guardrails.
    - New models (by changing LLM configuration only).

The backend is stateless: each `/chat` request carries everything it needs (history, auth context), which allows easy horizontal scaling.

4. Database & data access

4.1 SQLite

The assignment provides `employees.db` in `backend/data/`. For the purposes of the home assignment:

- **SQLite** is sufficient and easy to package with Docker.
- The same abstraction layer can later be backed by Postgres/MySQL without impacting the rest of the system.

4.2 Data access layer

All database reads are implemented in `backend/app/db/queries.py` using Python's `sqlite3`:

- There is deliberately **no ORM**, to keep:
    - The code transparent.
    - The behavior easy to inspect.
    - The focus on the LLM + tool integration rather than ORM boilerplate.

Tool handlers call these helper functions instead of embedding SQL directly in the LLM or the UI.

5. Authentication & authorization

5.1 Authentication flow

The assignment requires **ID + employee name** before any meaningful answer. The flow:

1. **Unauthenticated user**
    - The LLM is instructed to ask for name and ID first.
    - Until both are provided and verified, only generic prompts like
      "Please provide your name and ID to continue." are allowed.
2. **Verification**
    - The backend checks `(employee_id, employee_name)` via `employee_exists_in_database` in `db/verifiers.py`.
    - If the record doesn't exist, the LLM is instructed to respond as an **unknown user** and not reveal any training data.
3. **Session handling**
    - The frontend stores `employee_id` and `employee_name` in `useChatHistory` per session.
    - Every `/chat` request includes these values so **stateless** backend instances can authenticate each call.

5.2 Role detection (Employee vs CISO)

- The database tracks whether a user is a CISO via an `is_ciso` flag.
- After authentication, the backend:
    - Computes `is_ciso`.
    - Injects this flag into the context passed to the LLM.
- Tool availability is then filtered based on this flag.

5.3 Authorization rules

- **Unauthorized**
    - Can only provide name and ID to "log in".
- **Employees**
    - Can only query their **own** training status and videos.
    - They have no access to organization-wide stats or other employees.

- **CISO**
  - Can query any employee by name and ID.
  - Can list all employees by status.
  - Can request global training statistics.

All of these checks are enforced in backend tool handlers, not in the frontend, to prevent bypassing via custom clients or direct HTTP calls.

---

6. LLM & tools design

6.1 LLM configuration

The backend uses an OpenAI-compatible LLM (default: `gpt-4o-mini`) chosen for:

- **Speed** – responsive chat experience.
- **Cost** – appropriate for a home assignment and scalable in real deployments.

Configuration is split into:

- `llm_config.py` – model name, system prompts, and instruction strings.
- `llm_client_setup.py` – client instantiation using environment variables.

This separation makes it easy to switch models or tune prompts without touching business logic.

6.2 Tool definitions

Tools are:

- **Declared** in `services/agent_tools/tools.py`:
  - Name, description, parameter schema.
- **Implemented** in `llm_tool_handlers.py`:
  - Input validation (e.g., employee must exist).
  - Read-only access via `db/queries.py`.
  - Returning structured Python dicts for the LLM to verbalize.

Each tool follows this pattern:

1. Validate inputs (IDs, names, roles).
2. Query SQLite via the helper functions.
3. Return a structured result (e.g., `{ "status": "IN_PROGRESS", "missing_videos": [...] }`).

New capabilities can be added by:

- Declaring a tool schema.
- Implementing the handler.
- Registering it in `TOOL_HANDLERS`.

No LLM prompt changes are required to expose new functionality to the UI, aside from describing the new tool.

---

7. Guardrails & safety

The system combines prompt-level and architecture-level guardrails:

- **On-topic only**
  - System prompts instruct the LLM to only discuss **cybersecurity training data** derived from the provided database.
  - Non-training questions (e.g., general tech support, personal queries) are answered with a polite refusal or redirection.
- **Read-only tools**
  - All tools only perform **SELECT**-style queries on SQLite.
  - There are no update/insert/delete operations wired into the agent.
- **Role-based tools**
  - Tools exposing organization-wide training data or aggregate statistics are only available when `is_ciso` is `true`.
  - Employees never see other employees' data.
- **Unknown users**
  - If `(employee_id, employee_name)` doesn't exist in the database, the user is treated as unknown and receives only generic guidance.

This setup reduces the risk of data leakage, accidental state changes, and off-topic behavior.

---

8. Scaling, performance & caching

8.1 Stateless backend

- Each `/chat` request includes all necessary context:
  - Conversation history.
  - Auth data (ID + name).
- Any FastAPI instance can handle any request:
  - No sticky sessions.
  - Easy to run multiple replicas behind a load balancer.

8.2 Database considerations

- **SQLite** fits the assignment constraints:
  - Zero-configuration.
  - Single file packaged in Docker.
- For higher scale, the same query abstractions could be backed by:
  - Postgres
  - MySQL without changing the LLM or frontend.

8.3 Caching

To reduce latency and costs:

- **CISO statistics queries** (which can be expensive) are cached.
- **LLM responses** can also be cached for repeated queries.

This is handled in `backend/app/services/cache.py` and the `cache/` module.

8.4 Future performance optimizations

Potential improvements:

- **Streaming partial LLM responses** to the frontend for better UX.
- Switching to a more scalable RDBMS if write-heavy features are added.
- Adding background jobs for precomputing expensive statistics.

9. Limitations & future work

Current limitations:

- No persistent sessions across browser tabs or devices (history is in-memory on the frontend).
- No admin UI for browsing training stats – the CISO interacts only via chat.
- No automated tests checked in yet; a test strategy (e.g., for DB queries, tool handlers, and prompt behavior) is still to be implemented.
- Error handling is basic; the frontend could benefit from more robust error boundaries and user-friendly messages.

Future work:

- Add **end-to-end tests** (e.g., Playwright or Cypress) and backend unit tests.
- Persist chat history per user in a proper data store.
- Extend the system with a **CISO dashboard** that reuses the same backend tools for a graphical view.
- Add observability (structured logging, metrics around tool usage, and LLM latency).

10. Summary

This design implements the CyFortis Training Assistant as specified:

- Natural-language interface for **employees** and **the CISO** over cybersecurity training data.
- **Authentication** via `employee_id + employee_name` enforced on every request.
- **Clear separation of concerns**:
  - React UI.
  - FastAPI backend.
  - LLM tools.
  - SQLite database and query layer.
- **Read-only**, guarded LLM usage focused strictly on the cybersecurity training task.
- A single `/chat` endpoint that centralizes:
  - Conversation logic.
  - Tool orchestration.
  - Auth and role-based access control.

The overall design emphasizes safety, extensibility, and developer ergonomics, while remaining faithful to the assignment constraints.