Create a REST API for a Task Management System using Node.js, Express, and MongoDB. The system will manage tasks with categories and priority levels.

Requirements

Database Design (3 points)

Design a MongoDB collection for tasks using Mongoose schema.

Each task should include title (required, max 100 characters), description (optional text), status (enum: 'pending', 'in-progress', 'completed' with default 'pending'), priority (enum: 'low', 'medium', 'high' with default 'medium'), dueDate (date field, required), category (string with options: 'Work', 'Personal', 'Shopping', 'Health', 'Other'), and automatic timestamps for creation and updates.

The schema must include proper validation rules such as required fields, string length limits, and enum constraints.

Backend Implementation (5 points)

Implement a REST API using Node.js and Express with complete CRUD operations.

Create endpoints for adding new tasks (POST /api/tasks), retrieving all tasks (GET /api/tasks), getting a single task by ID (GET /api/tasks/:id), updating task details (PUT /api/tasks/:id), and deleting tasks (DELETE /api/tasks/:id).

Include proper error handling with appropriate HTTP status codes for scenarios like invalid data, resource not found, and server errors.

Use middleware for CORS and JSON parsing for request bodies. Structure your code with separate files for server configuration, Mongoose models, and route handlers.

Data Validation (2 points)

Implement validation using Mongoose validators to enforce data integrity including required fields, string lengths, enum values, and proper date formats.

Ensure that validation errors return clear error messages with appropriate HTTP status codes.

Test validation by attempting to create tasks with invalid data such as missing required fields, invalid enum values, and incorrect data types.

Deliverables

Submit screenshots showing successful CRUD operations. Include screenshots of MongoDB showing the tasks collection with your sample data visible in MongoDB Compass or the mongo shell.

Part 2 — AI Memory FastAPI + MongoDB + Ollama local LLM) (10 points)

Recreate the "short-term, long-term (summary), and episodic memory" behavior using FastAPI and a local model via Ollama (e.g., phi3:mini, or another small model of your choice). Use the same MongoDB database as Part 1 (reuse the instance; create new collections for memory).

MongoDB collections (reuse the same DB)

Create new collections: **(3 points)**

- messages
  Fields: user_id, session_id, role (user/assistant), content, created_at.

- summaries
  Fields: user_id, session_id (nullable for lifetime), scope ('session' | 'user'), text, created_at.

- episodes
  Fields: user_id, session_id, fact (short text), importance (0..1), embedding (array of floats), created_at.

NOTE: Keep your tasks collection from Part 1 untouched. You are only reusing the same database/connection.

FastAPI endpoints **(5 point)**

1. POST /api/chat

- Request: { user_id, session_id (optional), message }
- Server steps:
  - Save the user message to messages.
  - Short-term memory: fetch the last N messages for the session (N = SHORT_TERM_N).
  - Long-term memory: fetch the latest session summary and the latest lifetime user summary from summaries.
  - Compose the prompt for the chat model:
    - A short system primer (what the assistant is and how to answer),
    - The long-term summaries (lifetime + session),
    - The short-term window (last N messages),
    - The current user message,
    - A compact line with the top-k episodic facts found.
  - Call the Ollama chat API (model = CHAT_MODEL), get the assistant reply.
  - Save the assistant message to messages.
  - Long-term summarization: every SUMMARIZE_EVERY_USER_MSGS user messages, generate a session summary from recent messages and upsert it to summaries. Occasionally recompute/update the lifetime summary by condensing session summaries.
  - Response body: return the assistant reply + an object showing what was used:
    - the short-term messages count or the short-term list,
    - the text of the long-term summary used (if any),
    - the episodic facts that were retrieved for this turn.

2. GET /api/memory/{user_id}
   - Return a JSON view of:
     - last ~16 messages in the (default) session for that user,
     - latest session summary (if any),
     - latest lifetime user summary (if any),

        - last ~20 episodic facts (text only is fine).

3. GET /api/aggregate/{user_id}
    - Return daily message counts (group by date of created_at) and a small list of the most recent summaries (e.g., lifetime + latest session).

Memory logic (what to implement) **(2 points)**

- Short-term memory: sliding window of the last N messages for the session (read from messages; optional in-memory cache is allowed but not required).
- Long-term summaries:
    - Trigger every M user message (configurable).
    - Summarize recent conversation into concise bullets; store under scope='session'.
    - Build/refresh a lifetime summary by condensing multiple session summaries into a single profile; store under scope='user' with session_id = null.
- Episodic memory:
    - From each user message, extract up to 3 short facts likely to be useful later, with an importance score (0..1).
    - Embed each fact using the embed model; store arrays in MongoDB.
    - For each turn, retrieve the top-k relevant episodes by cosine similarity between the current message embedding and stored episodes.

How to run and test

1. Use Postman/cURL to:
    - Send at least 8–10 messages in one session to trigger short-term dynamics and 1–2 summaries.
    - Confirm that episodes are being extracted and retrieved.
2. Optionally build a simple HTML page with one input box calling POST /api/chat to simulate a mini chat interface (not required, but recommended for screenshots).

What to submit for Part 2

- Screenshots:
  - A conversation showing the assistant’s replies.
  - The JSON result of GET /api/memory/{user_id} (with short-term summaries, episodic).
  - The JSON result of GET /api/aggregate/{user_id}.
  - MongoDB Compass showing the messages, summaries, and episodes collections with documents.
- Your code files for:
  - The endpoints (/api/chat, /api/memory, /api/aggregate).
  - The memory logic functions you wrote (episode extraction, summarization trigger, embedding + retrieval, prompt assembly).