BinBot: Dockerized Basement Inventory System (ChromaDB Hybrid)

1. Project Goal

The objective is to create a local-first inventory management system that uses AI for a natural, voice- or text-based interface. The system must run entirely within a single Docker container, which will encapsulate the backend, front-end, and the ChromaDB service. This ensures the application is portable, easy to deploy, and keeps all data private on the user's machine.

2. Technical Stack and Components

The application will be built using the following technologies:

- **Containerization**: Docker (single container)
- **Backend**: Python 3.11 with FastAPI
- **Vector Database**: ChromaDB (embedded mode, running within the Python process)
- **External LLM**: OpenAI API or Google Gemini API (cloud-based)
- **Front-end**: A single index.html file with vanilla JavaScript, HTML, and Tailwind CSS
- **Persistent Storage**: Docker volumes for ChromaDB and image files

3. Project File Structure

The AI agent will create the following file structure:

```
/project
├── Dockerfile
├── requirements.txt
├── app.py
├── config.yaml
├── api_schemas.py
├── frontend/
│   └── index.html
└── start.sh
```

4. Implementation Details for the AI Agent

4.1. The Dockerfile

The Dockerfile will create a single container that installs all dependencies, copies the application code, and sets up the entry point to run the backend server and its associated processes.

- Use `python:3.11` as the base image
- Install `fastapi`, `uvicorn`, `chromadb`, `pydantic`, and other Python libraries from `requirements.txt`

- Copy `app.py`, `config.yaml`, `api_schemas.py`, and the `frontend` directory into the container
- Define a volume for persistent ChromaDB data and image storage
- The entry point (CMD) will execute a shell script (`start.sh`) that ensures all services are ready

4.2. The start.sh Script

This script is crucial for managing the local services:

- Start the FastAPI server
- Validate external LLM API connectivity (OpenAI or Gemini)
- Launch the Uvicorn server to host the FastAPI application

4.3. The app.py Backend

The Python backend will implement the core logic:

- **Configuration Management**: Load settings from `config.yaml` including external LLM API configuration, model parameters, embedding model selection, and fallback configurations
- **Error Handling**: Implement comprehensive error handling with graceful degradation:
  - If external LLM API is unavailable, provide clear error messages and retry logic
  - Handle API rate limits with exponential backoff
  - If ChromaDB fails, attempt recovery or provide data backup options
  - If embedding generation fails, retry with exponential backoff
- **Embedding Model Consistency**: Maintain a single embedding model per collection to ensure search consistency. Store the embedding model version in collection metadata and prevent model switching without data migration
- Initialize ChromaDB in persistent mode, specifying a path within the container's volume for data storage
- Create a ChromaDB collection with appropriate metadata fields (`bin_id`, `name`, `description`, `image_path`, `embedding_model_version`)
- Implement API endpoints for the front-end (see API Schema section below):
  - `/add`: Handles new items with transaction support for bulk operations
  - `/remove`: Handles item deletion with disambiguation when multiple matches found
  - `/search`: Handles user queries with ranked results
  - `/move`: Handles moving items with disambiguation support
  - `/undo`: Leverages audit log to reverse the last operation
  - `/health`: System health check endpoint
- Implement the multimodal functionality: for image uploads, the backend will use the configured multimodal model to generate embeddings

4.4. API Schemas and Response Formats

All API endpoints will use standardized request/response schemas defined in `api_schemas.py`:

Standard Response Format:

```
{
  "success": boolean,
```

```
  "data": object | array | null,
  "error": {
    "code": string,
    "message": string,
    "details": object
  } | null,
  "disambiguation": {
    "required": boolean,
    "options": array,
    "query_id": string
  } | null
}
```

Key Endpoints:

- `POST /add`: Request includes `items[]`, `bin_id`, `image?`, `bulk_transaction_id?`
- `POST /remove`: Request includes `query`, `bin_id?`, `confirmation_id?`
- `POST /move`: Request includes `query`, `source_bin_id?`, `target_bin_id`, `confirmation_id?`
- `GET /search`: Query params include `q`, `bin_id?`, `limit?`
- `POST /undo`: Request includes `operation_id` from audit log
- `GET /health`: Returns system status and LLM connectivity

4.5. The frontend/index.html File

The front-end is a simple interface for user interaction:

- A minimalist UI with a text input field for queries and a microphone button for voice commands
- **Voice Transcription**: The browser's native Web Speech API will be used to convert audio input to text directly on the client side. This eliminates latency and ensures the raw audio data never leaves the user's device
- A section to display search results, including item descriptions and photos
- **Disambiguation Interface**: When the backend returns multiple matches, display options for user selection
- A simple form for adding new items, with fields for the bin number and a file input for an image
- **Undo Button**: Prominently displayed button to reverse the last operation

4.6. Conversational Context Management

To enable a natural, multi-turn conversation, the backend will maintain a simple, in-memory context store for each user session. This context will be used to enhance subsequent queries.

- **Context State**: The backend will store the user's `bin_id` and the most recently discussed item in a session dictionary
- **Contextual Query Augmentation**: When a new command is received, the backend will check the context. If a `bin_id` is present in the session, it will automatically apply it as a metadata filter to the ChromaDB query. For example, a command like "add this plastic tubing" would be automatically treated as "add this plastic tubing to bin 12" if the `bin_id` is stored in the context

- **Handling Missing Context**: If a required piece of information like `bin_id` is not present in either the command or the session context, the backend will send a conversational response to the user asking for clarification. For example, it might respond with, "I can't tell which bin you mean. Which bin would you like to add the motor to?"
- **Context Reset**: The context will be cleared after a period of inactivity or when a new command explicitly changes the location (e.g., "let's go to bin 15")

4.7. Bulk Add Functionality with Transaction Support

To handle commands that mention multiple items at once, the backend will implement a bulk add feature with proper transaction handling:

- **Intent Recognition**: The LLM will be prompted to identify a list of individual items from a single sentence. For example, the sentence "bin 5 has washers, bolts, a motor and velcro straps" will be parsed into `["washers", "bolts", "a motor", "velcro straps"]` and `bin_id: 5`
- **Transaction Management**: Each bulk operation receives a unique `bulk_transaction_id`. If any item in the batch fails to add, the entire transaction is rolled back to maintain data consistency
- **Atomic Processing**: The backend will prepare all embeddings first, then commit all items to ChromaDB in a single atomic operation
- **Error Recovery**: If the operation fails partway through, the system will automatically clean up any partially added items and provide detailed error information
- **Confirmation**: The system will provide a single, consolidated confirmation message with the transaction ID for potential undo operations

4.8. Image Handling for Multiple Items

This is a key part of the multimodal functionality:

- **Image Attachment**: Images will be attached to individual items, not to the bin as a whole. A single image file can be associated with multiple items
- **Multimodal Analysis and User Confirmation**: When a user uploads a photo of a bin's contents, the backend sends the image to the multimodal LLM. If the LLM identifies multiple distinct items, the system will not automatically add them. Instead, it will use its conversational logic to ask the user for confirmation (e.g., "I've identified a motor and some washers in this picture. Which item would you like to add?")
- **Unique Image Reference**: The `image_path` metadata field in the ChromaDB document for each item will point to the same original image file on the local file system. This prevents data duplication and keeps the database lean

4.9. Audit Log with Undo Functionality

To keep a detailed history of changes to the inventory and enable undo operations, the backend will implement a persistent audit log:

- **Dedicated Collection**: A new ChromaDB collection, `audit_log`, will be created for storing a record of all significant events
- **Event-Driven Logging**: For each successful add, remove, or move operation, the backend will create a new log entry

- **Enhanced Log Entry Structure**: Each log entry will be a ChromaDB document with the following metadata:
  - `operation_id`: Unique identifier for the operation (used for undo)
  - `timestamp`: The exact date and time of the event
  - `action`: A string describing the event (e.g., "item_added", "item_removed", "item_moved", "bulk_add")
  - `bin_id`: The bin affected by the change
  - `item_id`: The unique ID of the item(s)
  - `previous_state`: JSON snapshot of the item's state before the operation (for undo)
  - `new_state`: JSON snapshot of the item's state after the operation
  - `bulk_transaction_id`: For bulk operations, links related changes
  - `description`: A human-readable summary of the change
  - `reversible`: Boolean indicating if the operation can be undone
- **Undo Implementation**: The `/undo` endpoint will:
  - Look up the most recent reversible operation
  - Restore items to their `previous_state`
  - Create a new audit log entry for the undo operation
  - Handle bulk transaction reversals atomically
- **Queryable History**: This design allows for natural-language queries on the audit log itself, such as "what did I add to bin 5 yesterday?" or "show me all the changes I've made to my motors."

4.10. Configuration Management

The system will use a `config.yaml` file for centralized configuration management:

```yaml
# LLM Configuration
llm:
  provider: "openai"  # or "gemini"
  openai:
    api_key: "${OPENAI_API_KEY}"
    model: "gpt-3.5-turbo"
    embedding_model: "text-embedding-ada-002"
    timeout: 30
  gemini:
    api_key: "${GEMINI_API_KEY}"
    model: "gemini-pro"
    timeout: 30

# ChromaDB Configuration
database:
  persist_directory: "/app/data/chromadb"
  collection_name: "inventory"

# Error Handling
error_handling:
  max_retries: 3
  retry_delay: 1.0
  fallback_to_cloud: true
```

```
# Transaction Settings
transactions:
  bulk_timeout: 300
  max_bulk_size: 100
```

4.11. Item Disambiguation Strategy

For `/remove` and `/move` operations that return multiple matches:

- **Similarity Threshold**: Items with similarity scores above 0.8 are considered potential matches
- **Disambiguation Response**: When multiple items match, return a `disambiguation` object with:
  - `required: true`
  - `options`: Array of matching items with their details and confidence scores
  - `query_id`: Unique identifier for the disambiguation session
- **User Selection**: Frontend displays options with item details, images, and bin locations
- **Confirmation**: User selects the intended item, frontend sends confirmation with `query_id` and selected `item_id`
- **Fallback**: If no clear matches found, ask user to provide more specific details

4.12. Key Functionality: Example Multi-Turn Workflow

Here is a specific example for the agent to implement, which demonstrates the new context management system:

User query (Command 1): "What's in bin 12"

- **Action**: The backend performs a search with a metadata filter for `bin_id=12`
- **Context Saved**: The backend stores `bin_id: 12` in the session context
- **Response**: The UI displays the contents of bin 12

User query (Command 2): "add this plastic tubing"

- **Action**: The backend checks the context and finds `bin_id: 12`. It augments the command to "add this plastic tubing to bin 12"
- **Action**: The backend generates an embedding for "plastic tubing," and adds the item to the ChromaDB collection with `bin_id: 12`. It also adds a new entry to the `audit_log` collection
- **Response**: The UI is updated, and the system confirms the item was added to the correct bin

User query (Command 3): A picture is attached with the command "here's a picture"

- **Action**: The backend checks the context and finds `bin_id: 12`
- **Action**: The backend sends the image along with the text "plastic tubing" to a multimodal LLM to generate a visual embedding
- **Action**: The backend adds this new item (with the image path and embedding) to the ChromaDB collection. It also adds a new entry to the `audit_log` collection
- **Response**: The UI is updated, and the system confirms the image was added

User query (Command 4): "move the plastic tubing to bin 15"

- **Action**: The backend checks the context and finds `bin_id: 12`. It searches for "plastic tubing" in bin 12 and identifies the correct item
- **Action**: The backend updates the document for the plastic tubing, changing its bin_id from 12 to 15. It also creates a new audit_log entry with the action "item_moved", storing both previous and new states for undo capability
- **Response**: The UI is updated, and the system confirms the item was moved

User query (Command 5): "undo that"

- **Action**: The backend looks up the most recent reversible operation (the move operation)
- **Action**: The backend restores the plastic tubing to bin 12 using the `previous_state` from the audit log
- **Action**: A new audit log entry is created with action "operation_undone"
- **Response**: The UI confirms the move was undone and the plastic tubing is back in bin 12

Note: The agent should still consider a multi-container docker-compose setup as a potential alternative for a more robust separation of the backend and vector database services, though a single container is the preferred starting point for this project.