Technical Documentation: FinAdvisor AI - Flask Server (`server.py`)

Automated Documentation based on Provided Code

May 10, 2025

Note: This document describes the provided `server.py` Python script.

Contents

1 Introduction

1.1 Purpose

This document provides a technical overview of the Python Flask server script, `server.py`. The server acts as the backend for the FinAdvisor AI agent, handling user interactions, interfacing with an OpenAI Assistant, performing data lookups via SQL queries against a PostgreSQL database, generating audio responses, and managing PDF report generation and delivery.

1.2 Scope

The scope of this documentation covers the structure, key functionalities, components, and operational flow of the `server.py` script as provided. It includes details on API endpoints, database interaction, LLM integration, and scheduled tasks.

2 System Architecture Overview

The server facilitates a multi-component architecture:

- **User Interface (External):** Interacts with the Flask server via HTTP requests (e.g., a web frontend).
- **Flask Server (`server.py`):** The central component that:
  - Receives user questions and manages conversation threads.
  - Interacts with the OpenAI Assistants API for natural language processing and function/tool execution.
  - Connects to a PostgreSQL database to execute SQL queries (generated or invoked via tools).
  - Generates audio responses using OpenAI's Text-to-Speech (TTS) API.
  - Generates PDF reports from messages using ReportLab.
  - Handles scheduled daily tasks for PDF generation and delivery.
- **OpenAI Platform:**
  - Assistants API: For managing conversation state (threads), running assistants, and handling tool calls.
  - Text-to-Speech (TTS) API: For generating audio from text.
  - File API: For uploading files to be used by the assistant (e.g., for file search/retrieval).
- **PostgreSQL Database:** Stores transaction data loaded from a CSV file, queried by the server.
- **External PDF Delivery API (Hypothetical):** An endpoint (`http://localhost:8080/send-pdf-email` or `/send-pdf`) is configured for delivering generated PDFs.

(A diagram illustrating these interactions would be beneficial here.)

3 Core Components and Functionalities

3.1 Configuration

The server utilizes several configuration parameters, ideally managed via environment variables in a production setting:

- **OpenAI API Key & Assistant ID:** Hardcoded in the provided script for testing purposes. **Security Note: These should always be loaded from environment variables in production.**
- `DELIVER_API_URL`: URL for the downstream PDF delivery service. Defaults to `http://localhost:8080`
- `DAILY_TIME`: Time for the scheduled daily PDF generation job (e.g., "02:00").
- `OUTPUT_DIR`: Local directory for saving generated PDFs (e.g., `"./pdf_output"`).
- `DB_CONFIG`: Connection parameters for the PostgreSQL database (host, database name, user, password). **Security Note: Database credentials should be securely managed, e.g., via environment variables or a secrets manager.**
- `CSV_FILE`: Path to the source CSV file for initial data loading into PostgreSQL.

The output directory is created if it doesn't exist.

3.2 Database Setup and Data Loading (`load_csv_to_postgres`)

- Establishes a connection to the PostgreSQL database using `DB_CONFIG`.
- Checks if the `jordan_transactions` table already contains data. If so, it skips the CSV import.
- If the table is empty, it reads data from the `CSV_FILE`.
- Dynamically creates the `jordan_transactions` table if it doesn't exist, inferring column names from the CSV header and setting all column types to `TEXT`.
- Inserts each row from the CSV into the PostgreSQL table.
- This function is called once at server startup.

Note: For production systems, a more robust schema definition with appropriate data types (e.g., INTEGER, REAL, TIMESTAMP, VARCHAR) for PostgreSQL columns would be preferable over using TEXT for all fields.

3.3 Flask Application Endpoints

3.3.1 `POST /ask`

- **Purpose:** Main endpoint for users to submit questions to the AI assistant.
- **Request JSON Body:**
  - `question` (string): The user's question.
  - `blind_mode` (boolean, optional, default: `false`): If true, an audio response is generated.
  - `thread_id` (string, optional): The ID of an existing conversation thread. If not provided, a new thread is created.
- **Processing:**
  1. Retrieves or creates an OpenAI Assistant thread.
  2. Adds the user's message to the thread.
  3. Creates and polls a "run" on the thread with the specified `ASSISTANT_ID`.
  4. **Tool Handling:** If the run status becomes `"requires_action"`, it iterates through `tool_calls`:

- Extracts the function name and arguments.
- Currently, it's hardcoded to expect a tool that provides a "query" argument (presumably for SQL).
- Calls `run_sql_query(query_string)` with the extracted query.
- Submits the tool output (result of the SQL query, converted to a string) back to the run.

5. Once the run is `"completed"`, it retrieves the latest assistant message from the thread.
6. **Post-processing of Message (Conditional):** If a tool was used (`toolused == 1`), it attempts to run the assistant's text response *itself* as an SQL query via `run_sql_query`. **This step might be problematic if the assistant's final text message is not intended to be a valid SQL query after tool use. It should typically be the interpreted result from the tool call.**
7. Stores the textual answer and generates an audio file path if `blind_mode` is true using `process_audio`.

- **Response JSON Body:**
  - `answer` (string): The assistant's textual response.
  - `response_id` (string): A unique ID for this response instance.
  - `audio_file` (string, optional): Path to the generated audio file if `blind_mode` was true.
  - `thread_id` (string): The ID of the conversation thread (to be used for follow-up messages).

3.3.2 `POST /upload`

- **Purpose:** Allows uploading files to be associated with the OpenAI Assistant, likely for use with the File Search tool (Retrieval).
- **Request:** Expects multipart/form-data with a list of files under the key `files[]`.
- **Processing:**
  1. Retrieves the current assistant's configuration to get existing `vector_store_ids`.
  2. Uploads the new files to OpenAI using `client.files.create`.
  3. Retrieves the list of file IDs already in the current vector store.
  4. Creates a *new* vector store containing both the existing file IDs and the newly uploaded file IDs.
  5. Updates the assistant to use this new vector store ID for its `file_search` tool resource.
- **Response JSON Body:** Information about uploaded files, previous and new vector store IDs, and assistant update status.
- **Note:** This implementation replaces the entire list of vector stores with a new one containing all old and new files. For frequent updates or very large numbers of files, managing individual files within a vector store might be more efficient if the API supports it granularly.

3.3.3 `GET /get_response/<response_id>`

- **Purpose:** Allows polling for a response, particularly useful if audio generation is asynchronous.

- **Response JSON Body:** The answer and audio file path, or a status update if audio is still processing.

3.3.4 `GET /audio/<filename>`

- **Purpose:** Serves the generated audio (MP3) file for playback.
- **Response:** The audio file with MIME type `audio/mpeg`.

3.4 OpenAI Assistant Integration

- Uses a pre-defined `ASSISTANT_ID`. This assistant is assumed to be configured in the OpenAI platform with necessary instructions and potentially tools (like function calling definitions for SQL execution, or File Search).
- Manages conversation state using Threads.
- Handles the "run" lifecycle, including polling for status and submitting tool outputs.

3.5 SQL Query Execution (`run_sql_query`)

- **Input:** An SQL query string.
- **Security:** Includes a basic check to only allow queries starting with "SELECT" (case-insensitive). **Caution: This is a very basic safeguard and not sufficient for protecting against all forms of malicious SQL or prompt injection if the SQL is generated by an LLM without further validation.**
- **Database:** Connects to the PostgreSQL database defined in `DB_CONFIG`.
- **Output:** Returns a dictionary `{"columns": [...], "rows": [...] }` on success, or a string error message.

3.6 Audio Generation (`process_audio`)

- Takes a `response_id` to fetch the corresponding text answer.
- Uses OpenAI's `tts-1` model (voice "onyx") to convert the text to speech.
- Saves the audio content as an MP3 file locally (e.g., `output_<response_id>.mp3`).
- Stores the file path for later retrieval via the `/audio/` endpoint.

3.7 PDF Generation and Delivery (Scheduled Job)

- **`normalize_messages`**: Ensures input messages are a list of strings.
- **`messages_to_pdf_bytes`**: Renders a list of messages into a PDF in memory using ReportLab. Handles basic text wrapping.
- **`save_pdf_locally`**: Saves the generated PDF buffer to a local file in `OUTPUT_DIR` with a timestamped filename.
- **`deliver_pdf`**: Sends the PDF buffer to the `DELIVER_API_URL` (currently `http://localhost:8080/se`
- **`daily_job`**:
  - This function is intended to be run daily by the scheduler.

  - It currently uses a hardcoded sample string (`listMessages`) as the source of messages for the PDF. **In a real application, this would fetch relevant messages from a persistent store or based on specific criteria.**
  - Generates a PDF from these messages, saves it locally, and attempts to deliver it via the `deliver_pdf` function.
- **Scheduling:** The `schedule` library is used to run `daily_job` at the time specified by `DAILY_TIME` (default "02:00"). The main `if __name__ == '__main__':` block starts this scheduler loop after starting the Flask app.

3.8 Text Cleaning (`clean_text`)

A utility function to remove specific patterns from text:

- Text within brackets.
- Asterisks (*).
- Hash symbols (#).

This function is defined but not explicitly called on the final assistant message in the `/ask` route in the provided snippet, though it was commented out.

4 Data Flow Example (User Query to SQL to Response)

1. User sends a question (e.g., "How many failed transactions at Z Mall?") to the `/ask` endpoint.
2. Flask server creates/retrieves an OpenAI Assistant thread and adds the message.
3. A "run" is created.
4. The LLM (OpenAI Assistant), based on its instructions (which should include the database schema and Text-to-SQL guidance), determines a database query is needed.
5. The run status becomes `"requires_action"`. The `tool_calls` attribute contains the function name (e.g., "execute_sql_query") and arguments (e.g., `{"query": "SELECT COUNT(*) FROM jordan_transactions WHERE mall_name='Z Mall' AND transaction_status='Fa }`).
6. The Flask server extracts the SQL query string from `tool_args`.
7. `run_sql_query` is called with the SQL string.
8. The SQL query is executed against the PostgreSQL database.
9. The database results (e.g., `{"columns": ["count"], "rows": [[5]] }`) are returned to the Flask server.
10. The Flask server submits this result (as a string) back to the OpenAI run via `tool_outputs`.
11. The run continues and completes.
12. The Flask server retrieves the final message from the Assistant. This message should be the LLM's natural language interpretation of the tool's output (e.g., "There were 5 failed transactions at Z Mall.").
13. The server then sends this natural language response (and optionally an audio version) back to the client.

Improvement Note: The current code in `/ask` has a line `if(toolused == 1): new_message = run_sql_query(new_message)`. This implies that if a tool was used, the final text message from the assistant is *itself* treated as an SQL query. This is likely not the intended behavior. Typically, the assistant's final message after a tool call is the natural language interpretation of the tool's output, not another SQL query. This part of the logic might need revision.

5 Error Handling and Logging

- Basic error handling is present for API calls (e.g., `requests.post(...).raise_for_status()`) and database operations.
- The Python `logging` module is configured for INFO level, logging messages with timestamps.
- The `daily_job` function has a general `except Exception` block to log errors without crashing the scheduler.

6 Potential Areas for Improvement and Future Development

- **Enhanced Security:**
  - Move all API keys, database credentials, and sensitive configurations to environment variables or a secure secrets management system.
  - Implement more robust SQL validation/sanitization if LLM-generated SQL is directly executed, beyond the basic "SELECT" check, or adopt a stricter function-calling approach where the LLM only provides parameters to pre-defined, safe SQL queries.
- **Database Schema:** Define a more specific schema for the PostgreSQL table with appropriate data types (TIMESTAMPZ for dates, INTEGER/NUMERIC for amounts, VARCHAR for text) rather than TEXT for all columns, to leverage PostgreSQL's type system and improve data integrity and query performance.
- **SQL Generation Robustness:** The system prompt for the OpenAI Assistant needs to be very detailed and carefully crafted to ensure reliable SQL generation, especially for complex queries involving date manipulations or specific SQLite/PostgreSQL functions.
- **Tool Call Logic in `/ask`:** Review and potentially revise the logic: `if(toolused == 1): new_message = run_sql_query(new_message)`. The assistant's final message after a tool call should be the interpreted result, not raw SQL to be re-executed.
- **PDF Report Content:** The `daily_job` currently uses a hardcoded string for PDF content. This should be replaced with logic to fetch meaningful data (e.g., summaries, reports from the database, or conversation logs).
- **File Upload Strategy:** The `/upload` endpoint creates a new vector store each time by combining old and new files. For large numbers of files or frequent updates, explore options for adding/removing files from an existing vector store if the API allows, to avoid redundant processing and storage.
- **Asynchronous Operations:** For long-running tasks like complex LLM responses or extensive data processing, consider implementing asynchronous task queues (e.g., Celery, RQ) to prevent blocking Flask worker threads.
- **Scalability:** For higher loads, consider deploying the Flask application using a production-grade WSGI server (e.g., Gunicorn, uWSGI) behind a reverse proxy (e.g., Nginx).
- **Input Validation:** Add more robust validation for all incoming request data in the API endpoints.

7 Conclusion

The `server.py` script provides a functional backend for an AI-powered financial transaction analysis agent. It successfully integrates Flask for API endpoints, OpenAI's Assistants and TTS APIs for AI capabilities, PostgreSQL for data storage, and ReportLab for PDF generation. Key areas for attention in a production environment would be enhancing security, refining the SQL generation and tool call logic, and ensuring robust data management for the PDF reporting feature.

A Key Python Libraries Used

- `flask`: Web framework for creating API endpoints.
- `flask_cors`: Handles Cross-Origin Resource Sharing for the Flask app.
- `openai`: Official OpenAI Python client library.
- `psycopg2`: PostgreSQL adapter for Python.
- `requests`: For making HTTP requests (e.g., to the PDF delivery API).
- `schedule`: For scheduling the daily job.
- `reportlab`: For PDF generation.
- `csv`: For reading the CSV file.
- `sqlite3`: (Imported but `run_sql_query` uses psycopg2, sqlite3.Error catch might be legacy or for a different part not shown).
- Standard libraries: `os`, `time`, `io`, `logging`, `textwrap`, `json`, `datetime`, `re`.

B Responses for the benchmark questions

what is the total numbar of failed transactions at Z mall

the total number of failed transactions at z mall is 38.

Which branch has the highest transactions failure rate

the branch with the highest transaction failure rate is "y mall tla'a al-ali," with a failure rate of approximately 10.34%. there were 21 failed transactions out of a total of 203 transactions.

Compare tax collection between branches in the last month

here is the comparison of tax collection between branches in the last month:

1. c mall amman: 0.152
2. y mall tla'a al-ali: 0.152
3. z mall al bayader: 0.152
4. z mall gardens: 0.152
5. c mall irbid: 0.07
6. z mall al jubeiha: 0.0035

the branches c mall amman, y mall tla'a al-ali, z mall al bayader, and z mall gardens all collected the highest amount of tax at 0.152.

Ask a question

highest amount of tax at 0.152.

Are there any ususual transaction patterns in the data?

Could you please specify what type of unusual transaction patterns you are interested in? For example, are you looking for patterns related to specific branches, transaction types, transaction statuses, or perhaps anomalies in transaction amounts?

anomalies in transaction amount

here are some anomalies in transaction amounts based on the data:

1. transaction id: jo-2504-2052-45224, branch: z mall al bayader, amount: 85.52
2. transaction id: jo-2504-7263-64116, branch: z mall gardens, amount: 64.55
3. transaction id: jo-2504-7599-47299, branch: z mall al jubeiha, amount: 50.76
4. transaction id: jo-2504-2329-84766, branch: c mall amman, amount: 46.06
5. transaction id: jo-2504-8497-60155, branch: y mall tla'a al-ali, amount: 43.73
6. transaction id: jo-2504-4241-47576, branch: c mall amman, amount: 43.66
7. transaction id: jo-2504-5341-50460, branch: c mall aqaba, amount: 43.08
8. transaction id: jo-2504-4079-85858, branch: z mall al jubeiha, amount: 41.11
9. transaction id: jo-2504-6145-78878, branch: z mall al jubeiha, amount: 40.53
10. transaction id: jo-2504-7347-15015, branch: c mall irbid, amount: 38.33

these transactions are significantly higher than the average transaction amounts, indicating potential anomalies.

okay thank you

You're welcome! If you have any more questions or need further assistance, feel free to ask. Have a great day!

Ask a question