TDA RAG System: Closed-Loop Improvement & Maintenance

1. Introduction: The Self-Improving Agent

The application's Retrieval-Augmented Generation (RAG) system is a closed-loop feedback mechanism. Its primary goal is to improve the Planner's [cite: src/trusted_data_agent/agent/planner.py] decision-making over time by allowing it to learn from its own past successes.

The system is designed to automatically:

1. **Capture** every successful agent turn.
2. **Analyze** its efficiency (based on token cost).
3. **Identify** the single "best-in-class" strategy for any given user query.
4. **Feed** this "best-in-class" example back to the Planner on future, similar queries.

This document details the complete data lifecycle, from real-time processing to batch maintenance.

2. Key Components & Data Flow

The RAG system relies on three main storage locations:

1. **tda_sessions/** (The Raw Log):
   - This is the "black box recorder" of the application.
   - It contains the raw JSON logs for every user session, storing the complete workflow_history for every turn, including failures, errors, and conversational chats.
   - It is the **source material** for the RAG miner.
2. **rag/tda_rag_cases/** (The Case Study Archive):
   - This directory is defined by RAG_CASES_DIR in config.py [cite: src/trusted_data_agent/core/config.py].
   - It is the "filing cabinet" of *processed* case studies.
   - When a turn is processed by the RAG system, it is extracted, cleaned, and saved here as a single case_[uuid].json file. This archive contains *all* processed successful turns, not just the most efficient ones.
3. **.chromadb_rag_cache/** (The Search Index):
   - This is the persistent vector database (ChromaDB), defined by RAG_PERSIST_DIR in config.py [cite: src/trusted_data_agent/core/config.py].
   - It does **not** store the full JSON. It stores a *vector embedding* of the user's query and a flat metadata object.
   - Crucially, this metadata includes the case_id (which links back to the file in RAG_CASES_DIR) and the is_most_efficient flag, which is the key to the entire

system.

3. The RAG Approach: Real-Time Closed-Loop

The primary RAG pipeline is a real-time, asynchronous "Producer-Consumer" system. This ensures that agent improvements are captured immediately without impacting user-facing performance.

Part 1: The "Producer" (in executor.py)

1. A user's query is successfully completed by the PlanExecutor.
2. In the finally block of the PlanExecutor.run method, the agent finalizes the turn_summary object, which contains the query, the plan, all execution steps, and the final token counts [cite: src/trusted_data_agent/agent/executor.py].
3. The PlanExecutor adds the session_id to this turn_summary and places it into the global APP_STATE['rag_processing_queue'] [cite: src/trusted_data_agent/agent/executor.py, src/trusted_data_agent/core/config.py].
4. This action is instantaneous. The user's final_answer has already been sent, so they experience no delay. This entire step is gated by the RAG_ENABLED flag.

Part 2: The "Consumer" (in main.py)

1. When the application starts, it launches a single, persistent background task: rag_processing_worker() [cite: src/trusted_data_agent/main.py].
2. This worker is the **only** consumer of the rag_processing_queue. It runs in an infinite loop, pulling one turn_summary at a time.
3. This singleton worker design **guarantees atomicity** and prevents the database race conditions we previously discussed.
4. The worker calls the RAGRetriever's central processing method.

Part 3: The "Processor" (in rag_retriever.py)

This is the core of the RAG logic, performed by the RAGRetriever instance [cite: src/trusted_data_agent/agent/rag_retriever.py].

1. **Extract & Filter:** The worker calls await self.retriever.process_turn_for_rag(turn_summary). This method first uses _extract_case_from_turn_summary to parse the turn. If the turn was not a successful, tool-using plan (e.g., it was a failure or a TDA_ContextReport), the process stops, and the turn is ignored.
2. **Archive Case File:** The valid "case study" JSON is saved to the rag/tda_rag_cases/ directory.
3. **Query ChromaDB:** It queries the vector database to find the *current* champion for this exact user query (i.e., where is_most_efficient: True).
4. **Compare Efficiency:** It compares the output_tokens of the new case against the output_tokens of the current champion (if one exists).

5. **Perform Atomic Transaction:**
   - **Case A (New case wins):** The new case is more efficient. It is upsert-ed to ChromaDB with is_most_efficient: True. The retriever then issues an update command to **demote** the old champion, setting its is_most_efficient flag to False.
   - **Case B (Old case wins):** The new case is less efficient. It is upsert-ed to ChromaDB with is_most_efficient: False. The old champion remains the winner.

Part 4: How the Agent Uses the Data

1. **Retrieval:** When a *new* query comes in, the Planner calls self.rag_retriever.retrieve_examples() [cite: src/trusted_data_agent/agent/planner.py].
2. **Filtering:** This retrieve_examples method *only* searches ChromaDB for cases matching the query where is_most_efficient: True [cite: src/trusted_data_agent/agent/rag_retriever.py].
3. **Augmentation:** The "few-shot examples" from these champion cases are formatted and injected directly into the Planner's prompt, guiding it to generate a high-quality, efficient plan based on proven strategies.

4. Maintenance Script: rag_miner.py

This script is a command-line "catch-up" utility to process historical data from tda_sessions that the real-time worker may have missed (e.g., turns from before the RAG system was active).

Purpose

The rag_miner.py script [cite: src/trusted_data_agent/rag_miner.py] scans all session files in the tda_sessions directory. For every turn it finds, it feeds it into the **exact same** RAGRetriever.process_turn_for_rag method used by the real-time worker.

This guarantees that all historical data is processed using the **identical** filtering, efficiency comparison, and atomic update logic as the real-time loop.

How to Use rag_miner.py

Critical Warning: Concurrency Error

You **must stop the main web server** (python -m src.trusted_data_agent.main) before running the rag_miner.py script.

Both processes connect to the same .chromadb_rag_cache/ database. If both are running, the server will hold a lock on the database file, and the miner will fail with a sqlite3.OperationalError: attempt to write a readonly database error.

Workflow:

1. Ctrl+C to stop the main.py server.

2. Run the rag_miner.py script (see commands below).
3. Restart the main.py server.

Basic Command

From the trusted-data-agent root directory:

```
# Ensure your virtual environment is active
# (e.g., source .venv/bin/activate)

# Run the miner
python src/trusted_data_agent/rag_miner.py
```

(Note: If your CWD is the rag directory, you can use python rag_miner.py as you have been)

Command-Line Arguments

- --force:
  - This is the "fresh start" or "rebuild" flag.
  - It will **DELETE all case files** from rag/tda_rag_cases/.
  - It will **DELETE the entire .chromadb_rag_cache/ directory**, wiping the vector database.
  - Use this if you suspect the RAG store is corrupted or you want to rebuild it from scratch using only the data in tda_sessions.
- --sessions_dir <path>:
  - Tells the miner to look in a different directory for session logs.
  - Default: tda_sessions/ [cite: src/trusted_data_agent/rag_miner.py]
- --output_dir <path>:
  - Tells the miner to save the case_*.json files to a different directory.
  - Default: rag/tda_rag_cases/ [cite: src/trusted_data_agent/rag_miner.py]