This process is a **"Filter-then-Rank"** model. It is designed to first discard all non-viable candidates using hard business rules (the Filter) and then calculate an intelligent, weighted score for the few remaining candidates (the Rank).

**Step A: Data Preparation**

Before any comparison, the script prepares all necessary data:

1. **Load Policy:** The New Policy JSON is loaded and "flattened" into a single-level dictionary of key-value pairs (e.g., rule.payer.name: 'Aetna').
2. **Vectorize Policy:** Each "soft" text attribute from the flattened policy (like rule.summary) is sent to the embedding model (the runtime ML call) to get its corresponding vector. This creates a matching set of vectors for the new policy.
3. **Build Rules DB:** The Rules CSV is loaded and transformed from a long list of attributes into an efficient, in-memory database grouped by rule\_id. Each rule entry contains all its attributes and their pre-computed vectors.
4. **Load Weights:** The Weights.json file is loaded. This file defines the business importance of each attribute (e.g., payor is worth 15 points, summary is 10 points, documentDate is 0.1 points).

**Step B: Filter-then-Rank Loop**

The script now iterates through every single rule in the in-memory Rules DB to find the best match.

1. **Loop Start:** The script checks, "Are there more rules in the Rules DB to process?"
   * If **NO** (all rules are processed): It exits the loop and proceeds to **Step C**.
   * If **YES**: It takes the next rule and proceeds to Filter 1.
2. **Filter 1: Payor Match (Hard Rule)**
   * The script checks: Does the payor in the policy *exactly* match the payor for the current rule?
   * If **NO**: This rule is a bad match. It is **discarded**, and the script returns to **Step 1 (Loop Start)** to get the next rule.
   * If **YES**: The rule passes. It proceeds to Filter 2.
3. **Filter 2: Code Set Match (Hard Rule)**
   * The script checks: Do the procedureCodes (or other key code sets) have a high enough similarity? (e.g., a Jaccard similarity score > 50%).
   * If **NO**: This rule is a bad match. It is **discarded**, and the script returns to **Step 1 (Loop Start)**.
   * If **YES**: This rule has passed all "hard" filters and is now considered a **"Candidate"**. It proceeds to the Ranking step.
4. **Rank: Calculate Weighted Score**
   * A weighted score is calculated for this "Candidate" rule. The final score is TotalScore / TotalPossibleWeight.
   * The TotalPossibleWeight is calculated by summing the weights (from Weights.json) for every attribute in the *new policy*.
   * The TotalScore is calculated by:
     + **For "Hard" Attributes** (like payor): It gets 1.0 \* weight (full credit).
     + **For "Soft" Attributes** (like summary): It uses **Cosine Similarity** on the vectors (the policy vector vs. the rule's vector). This gives a score (e.g., 0.95), which is then multiplied by its weight (0.95 \* weight).
   * All these scores are summed up to get the TotalScore.
5. **Store Score:** The final score (e.g., 87.5%) is stored in a list, associated with its rule\_id.
6. **Continue Loop:** The script returns to **Step 1 (Loop Start)** to process the next rule in the database.

**Step C: Final Classification**

This stage begins *after* the loop in Step B has finished.

1. **Find Best Match:** The script looks at all the stored scores for the "Candidate" rules and finds the rule\_id with the single **MAXIMUM** score.
2. **Final Decision:** The script checks this top score against the threshold:
   * If **Best Score > 80%**: The policy is a clear match to an existing rule. It is classified as **UPDATE**.
   * If **Best Score <= 80%**: Even the best-scoring candidate was not good enough. The policy is classified as **NEW**.