**AI-Powered Semantic Ranking System for 1M x 1M Job-Candidate Matching**

Build a production-ready system capable of accurately and efficiently ranking 1 million job descriptions against 1 million candidate profiles using deep semantic understanding.

**System Architecture Overview (ColBERT + FAISS + MaxSim)**

This section explains each step of the diagram in a simple, beginner-friendly manner so anyone—regardless of technical background—can follow how the system processes, compares, and ranks candidates.

**Stage 1: Dense Embedding Indexing (Understanding Job & Candidate Content)**

* **What happens here?** Think of this as the stage where both job descriptions and candidate profiles are "translated" into a language the computer can understand: vectors (lists of numbers that capture meaning).
* **Tool Used**: ColBERT (a model based on transformers like MPNet)
* **Data**:
  + Candidate profiles are split into parts like experience, education, and projects.
  + These parts are broken into short text chunks and encoded.
* **FAISS Index**:
  + These encoded vectors are stored in a fast lookup table (FAISS) to make it easy for the system to find and compare similar entries.
* **Important**: This stage is **not filtering out any candidates**—it prepares all 1 million of them for scoring.

**Stage 2: Fine-Grained Scoring with MaxSim (Deep Comparison Between Job & Candidates)**

* **What happens here?** This is where the system compares every job with every candidate in detail—word by word—using a method called MaxSim.
* **Inputs**:
  + The job description, now encoded as tokens (small meaningful pieces of text)
  + Candidate vectors pulled from FAISS
  + The original text chunks (from Parquet files)
* **What’s special?** Instead of just comparing overall resumes, the system zooms in: "Does this candidate’s project match this job task? How well do their skills align with this responsibility?"
* **Why it matters**: This token-level comparison is what gives the system its high semantic accuracy.

**Stage 3: Semantic Aggregation (Combining Multiple Scores into One Meaningful Result)**

* **What happens here?** The system now combines all the small scores from the previous step to calculate an overall match score.
* **Example**: If a candidate has 3 project matches and 2 good experience matches, those scores are combined.
* **How is it done?** Through weighted logic: e.g., 60% importance to experience, 20% to projects, 20% to certifications.
* **Why this matters**: It allows recruiters to customize scoring logic based on job needs.

**Stage 4: Feature-Based Re-Ranker (Optional Advanced Tiebreaker)**

* **What happens here?** This optional layer uses a machine learning model (LightGBM/XGBoost) to fine-tune the ranking.
* **Why optional?** It’s used when two or more candidates have similar scores, and deeper features like skill overlap, education level, or number of years of experience can help sort them.
* **Visual Note**: This block is shown with a dashed orange border to signal it’s not required in every deployment.

**Data Pipeline & Storage (What’s Stored and Where)**

**Input:**

* Recruiters upload job descriptions and candidate profiles in a structured JSON format.
* These are parsed and stored in an efficient file format (Parquet).

**Tokenizing & Chunking:**

* Each section (like experience or education) is split into parts no longer than 512 tokens to fit model input limits.
* **What if a section exceeds 512 tokens?**
  + The system includes a built-in mechanism to split long sections into multiple overlapping or punctuated chunks.
  + Example: A 900-token description would be split into two chunks, e.g.:
    - Chunk 1: tokens 0–511
    - Chunk 2: tokens 450–900 (with 60-token overlap to preserve context)
  + Splitting happens at **logical boundaries** like sentence ends, bullet points, commas, or semicolons to preserve meaning.
  + These chunks are scored independently and later combined in aggregation.

**Embeddings:**

* Candidate chunks are encoded and saved into the FAISS database.
* Jobs are tokenized and encoded in real-time or from cache.
* FAISS is **not used to filter** but to supply all possible candidate vectors for scoring.

**Scores:**

* Every job-candidate score is calculated and stored.
* Results are saved as matrices (tables) and aggregated.
* Final results are saved in Redis and PostgreSQL for fast access.

**Tech Stack (What Tools Are Used)**

|  |  |
| --- | --- |
| **Component** | **Technology** |
| Embedding | ColBERT (MPNet) |
| Vector DB | FAISS (FlatIP, IVF) |
| Fine-Grained Scoring | MaxSim + PyTorch |
| Reranker | LightGBM / XGBoost (Optional) |
| Storage | Parquet + PostgreSQL + Redis |
| Serving | FastAPI + ONNX (optional for CPU) |
| Batch Mgmt | Ray / Dask |

**Deployment Strategy (How It Runs in Production)**

* ColBERT encodes all job and candidate data in batches.
* FAISS provides vector batches for fast comparison, **not filtering**.
* MaxSim scores are computed on GPU using PyTorch.
* Results are aggregated and optionally re-ranked.
* Final scores are served via a FastAPI endpoint.

**Role of ONNX in the Architecture**

* **When does ONNX come into play?**
  + ONNX (Open Neural Network Exchange) is used when you want to **deploy transformer-based models like ColBERT in a fast, platform-independent format**.
  + Instead of running the PyTorch model directly, it’s exported as an ONNX graph which can run with lightweight inference engines like ONNX Runtime.
* **Why use ONNX?**
  + Faster inference on CPU
  + Lower memory usage
  + Easier deployment in production environments (containerized or edge computing)
  + Hardware portability: ONNX models can be executed across GPU, CPU, or even specialized hardware like Intel OpenVINO or NVIDIA TensorRT.
* **Impact on architecture**:
  + When ONNX is used, the ColBERT encoder and/or MaxSim computation layer runs inside ONNX Runtime rather than PyTorch.
  + This makes the serving layer (via FastAPI) much **faster**, especially when GPUs are not available.
  + It enables **scaling the backend inference service to handle 1M × 1M comparisons** more cost-effectively.

**Output (What Recruiters See)**

* REST API:
  + /rank\_all\_candidates?job\_id=... returns all ranked results
  + /get\_candidate\_ranking\_breakdown?job\_id=... explains section scores
* Frontend:
  + A full leaderboard view of all candidates ranked by job-match relevance

**Visual Architecture Validation (Diagram Review)**

**1. Arrow from Job Descriptions → MaxSim Token-to-Token Scoring**

* Present: Clear connection showing job tokens directly feeding into MaxSim
* This reflects token-level scoring between job and candidate chunks (ColBERT's key strength)

**2. LightGBM / XGBoost marked as Optional**

* Displayed with dashed border and orange stroke
* Visually clarifies that it's a post-aggregation optimization, not mandatory

**3. FAISS represented as a batch provider (not scorer)**

* FAISS connects to MaxSim, not to output
* No Top-K filtering box — supports the “score all” mode
* FAISS is correctly used for exhaustive candidate vector access (IndexFlatIP/IVF)

This architecture balances FAISS speed with ColBERT's token-level semantic granularity, ensuring accuracy across all candidate-job pairs at production scale.

