Here’s a crisp-but-deep “interview‑ready” note you can keep for revision. No code—just the how/why in detail.

**The three architectures for JD ↔ Resume matching**

**Option 1 — Cross‑encoder (joint encoding; best accuracy)**

**Idea:** Concatenate JD and Resume into one sequence, run a single transformer over both, and read a shared representation (usually the [CLS] token) to predict your scores.

**Input layout:**  
[CLS] JD tokens … [SEP] Resume tokens … [SEP]

**What happens inside:**

* **Self‑attention is global.** Every JD token can attend to every Resume token (and vice‑versa). Token‑level alignments (e.g., “Python” ↔ “Python”, “SEO” ↔ “search engine optimization”) emerge directly inside the layers.
* **[CLS] becomes a joint summary.** Through pretraining and fine‑tuning, the model learns to pack sequence‑level information (content + relationships) into [CLS].
* **Multi‑output head.** A small regression/classification head with **N outputs** sits on top of [CLS] to produce:
  + Overall score
  + Macro scores (category‑level: experience, education, domain relevance, etc.)
  + Micro scores (skill‑level: Python, SEO, CRM, etc.)  
    One forward pass → all scores.

**Why it’s strong:**  
It **explicitly** models cross‑text token interactions. You’re not asking the model to guess alignment from two separate summaries; you let it compare words/phrases directly across JD and Resume.

**Trade‑offs:**

* **Accuracy:** Highest (especially with small datasets).
* **Speed:** Slow at inference—must process *each pair* together.
* **Memory/length:** Limited by max tokens; you need smart truncation of resumes (prioritize experience bullets over fluff).

**When it shines:**  
Small/medium datasets, need high fidelity, need multiple outputs (overall + macro + micro) with good calibration and explainability.

**Option 2 — Bi‑encoder / Siamese (separate encoding; fastest retrieval)**

**Idea:** Encode JD and Resume **separately** into embeddings with a shared transformer; compare embeddings to score alignment.

**Pipeline:**

* JD → encoder → vector
* Resume → encoder → vector
* Compare vectors (cosine/dot or a small MLP) → single score  
  *(for multiple scores, you can concatenate the two vectors and add a multi‑output head, but that’s an extra design step).*

**What happens inside:**

* **Self‑attention is local to each text.** There is **no** token‑level cross‑attention between JD tokens and Resume tokens during encoding.
* The model is forced to compress each text into a single embedding that “works for everything.”

**Why it’s fast:**  
You can **precompute all resume embeddings** and only encode the JD at query time. Ranking thousands of resumes becomes a few dot products.

**Trade‑offs:**

* **Accuracy:** Lower on nuanced matching (no explicit cross‑token alignment).
* **Speed:** Excellent—ideal for large‑scale retrieval.
* **Multi‑output:** Possible, but you must bolt on heads/losses that learn those outputs from the two embeddings; needs more data.

**When it shines:**  
Huge candidate/job pools, retrieval scenarios, latency‑critical pipelines. Often paired with a cross‑encoder re‑ranker.

**Option 3 — Poly‑encoder / Hybrid (late interaction; middle ground)**

**Idea:** Keep separate encodings like a bi‑encoder, but add a **lightweight cross‑attention step** between the two sides **after** encoding. Think: a small “meeting” between a few summary vectors and the other text’s token embeddings.

**Pipeline (conceptually):**

1. JD → encoder → JD token embeddings (or a few learned “context” vectors).
2. Resume → encoder → Resume token embeddings.
3. **Late interaction:** A **small number (k)** of JD context vectors attend over **all** Resume embeddings (or vice‑versa).
4. Aggregate the attended vectors → scores.

**Why it’s “lightweight”:**

* Full cross‑encoder compares **all tokens to all tokens** (O(N×M)).
* Poly‑encoder compares **k summaries to all tokens** (O(k×M), k ≪ N).
* You get some cross‑signal without the full cost.

**Trade‑offs:**

* **Accuracy:** Better than pure bi‑encoder; below full cross‑encoder.
* **Speed:** Slower than pure bi‑encoder; faster than cross‑encoder.
* **Complexity:** More moving parts (choosing k, designing the late interaction).

**When it shines:**  
You need better alignment than a bi‑encoder but can’t afford cross‑encoder cost for every pair.

**Why choose Option 1 for your MVP**

* **Data size:** You have ~1k labeled pairs. Cross‑encoders excel in **low‑data** settings because they **directly attend** across the two texts and don’t need tons of supervision to learn what matters.
* **Multiple outputs:** You want overall + macro + micro. Cross‑encoder gives you **all scores in one pass** from one shared representation with a simple multi‑output head.
* **Explainability & calibration:** The joint representation aligns closely with human judgment. With a small validation set you can calibrate the outputs so “80 = good fit” actually feels right.
* **Product fit:** You’re not ranking millions live on day one. Latency is fine, quality matters more.

*(Later, if you add bulk search or job‑tracker workflows, you can distill to a bi‑encoder for retrieval and re‑rank top‑K with the cross‑encoder—best of both.)*

**Option 1 in your project: the mechanics, end‑to‑end**

**Input construction (per pair)**

* Build a single sequence: [CLS] JD … [SEP] Resume … [SEP].
* **Token budget:** keep JD intact (usually shorter). For resumes, **prioritize**:
  1. Experience bullets most semantically similar to JD themes
  2. Key projects
  3. Skills/certs (brief)  
     Avoid headers/PII fluff.

**What the transformer actually does**

* **Layer by layer,** every token sees all others (global self‑attention).
* JD tokens and Resume tokens **cross‑attend**, so evidence lines “match” internally (e.g., JD:“lead PPC strategy” ↔ Resume:“owned Google Ads with 15% CPA drop”).
* The final **[CLS] embedding** becomes the **joint summary** of “JD compared to Resume,” influenced by all token‑to‑token relationships.

**Multi‑output prediction head**

* You attach a small feed‑forward head with **N outputs**:
  + 1 for **overall match**
  + M for **macro** categories (experience, domain fit, education, etc.)
  + K for **micro** skills (Python, SEO, CRM, etc.)
* **Single forward pass → N continuous scores.**  
  Training objective is **multi‑output regression** (e.g., mean squared error averaged over outputs).

**Targets you learn from**

* From your dataset, you already have:
  + **Aggregated macro score** (0–10)
  + **Aggregated micro score** (0–10)
  + Optional per‑criterion macro/micro items
* You map them to your product scale:
  + e.g., **Overall = 0.6×Macro + 0.4×Micro**, then ×10 → **0–100**
  + You can also **predict macro/micro aggregates directly** as two outputs, and optionally a few top micro skills the dataset labels consistently.

**How the model “knows” which micro to predict**

* The **output head has a fixed set of neurons** aligned to the fixed micro/macro criteria you choose to learn (e.g., “experience”, “education”, or a chosen subset of stable skills).
* Each neuron learns to read the relevant **subspace** of the [CLS] vector that correlates with that criterion across the dataset.
* You’re not hand‑coding locations; the mapping is **learned** during fine‑tuning.

**Calibration & what “80” means**

* Raw regression outputs aren’t perfectly “human‑calibrated.”  
  Use your validation split to apply **isotonic regression or Platt scaling** so that:
  + 80 ≈ resumes interviewers actually think are a good fit
  + 60 ≈ borderline
  + 40 ≈ weak match  
    This converts a technically correct score into a **human‑meaningful** one.

**Handling long resumes (practical detail)**

* Token limit is real. To avoid chopping useful evidence:
  + Keep JD fully.
  + Rank resume bullets by **semantic similarity to JD** (quick embedding pass) and include highest‑scoring bullets first.
  + Add a tiny tail of skills/certs at the end.  
    This preserves the most relevant evidence for the cross‑encoder to compare.

**What the UI can surface (from the same pass)**

* **Overall score** (0–100), plus
* **Macro score(s)** and **Micro score(s)** you trained to predict,
* **Justifications** (from your dataset or from a light rule layer): show top JD lines and matched resume bullets (found via simple embedding similarity outside the model—cheap and effective).

**Interview‑style comparisons you can quote**

**Cross‑encoder vs Bi‑encoder**

* *“Cross‑encoders jointly encode both texts, enabling token‑level cross‑attention. They deliver the best semantic alignment on small data, but inference is slower because each pair is processed together. Bi‑encoders encode texts separately; they scale and support retrieval, but lack precise token‑to‑token comparison unless you add late‑interaction or a re‑ranker.”*

**Why [CLS] works**

* *“In BERT‑style models, [CLS] is trained during pretraining to summarize the input for sequence‑level tasks. In fine‑tuning, we put a multi‑output head on [CLS] so one forward pass yields overall + macro + micro scores.”*

**Why multi‑output regression**

* *“We’re predicting several continuous scores simultaneously: overall, macro aggregates, and selected micro skills. A single head with N outputs shares representation and reduces overfitting versus training many separate models.”*

**Why Option 1 for MVP**

* *“With ~1k labeled pairs, cross‑encoder accuracy and calibration matter more than speed. We can distill to a bi‑encoder later for retrieval use‑cases and re‑rank top‑K with the cross‑encoder.”*

**When you’ll switch or add other options**

* **Add bi‑encoder (Option 2)** when you need to **search thousands** of resumes per JD quickly. Use it as a **first‑stage retriever**, then **re‑rank top‑K with the cross‑encoder** for final scores.
* **Consider poly‑encoder (Option 3)** if cross‑encoder cost is too high but pure bi‑encoder quality is insufficient—late interaction may bridge the gap.