**Explanation & implementation plan — Intelligent AI-Powered HR Recruitment Assistant**

Nice brief — this is a great practical student project. Below I’ll explain **what** the system does, **why** each piece is needed, and give a concrete roadmap (models, preprocessing, scoring, UI, evaluation, limitations, and future improvements) so you — or a student team — can build a working prototype.

**1) High-level idea**

When a new job post is created, the assistant automatically finds the best candidates from the existing CV/Resume store by converting resumes and job descriptions into meaningful vectors, comparing them semantically, and returning a ranked, explainable shortlist. The system reduces manual screening time, improves consistency, and provides clear reasons for every match.

**2) Core components (architecture)**

1. **Ingestion & Storage**
   * Input: PDFs, DOCX, plain text, images (scanned resumes).
   * Tools: PDF parsers (pdfminer / Apache Tika), python-docx, OCR (Tesseract) for scanned images.
   * Store raw text, parsed structured fields, and a canonical CV document in the HRMS or a separate database (Postgres / MongoDB).
2. **Preprocessing & Parsing**
   * Extract sections: contact, summary, experience, education, skills, certifications, dates.
   * Normalize entities: skill synonyms, degree names, company names.
   * Tools: spaCy for NER, rule-based regex for emails/phones/dates, custom skill dictionary.
3. **Feature Extraction**
   * Two tracks:
     + **Structured features**: years of experience (computed from dates), degree level, universities, certifications, location.
     + **Unstructured semantic features**: sentence / paragraph embeddings for job description and CV (full-text and per-section).
   * Models: Sentence Transformers (SBERT) family for embeddings; optionally LaBSE for multilingual support.
4. **Indexing & Similarity Search**
   * Vector index for fast retrieval: FAISS, Annoy, or Milvus.
   * Optionally bulk compute embeddings for all CVs and index them so a job description returns nearest neighbors quickly.
5. **Matching & Scoring**
   * Compute semantic similarity (cosine) between job description embedding and CV embeddings.
   * Combine semantic score with structured feature matches (skills overlap, education, recency, experience) using a weighted scoring function.
   * Produce explainable breakdown (why the CV scored well).
6. **Explainability / Insights**
   * Extract matched skills, years in relevant technologies, education match, notable achievements.
   * Produce human-readable line: e.g., “5 years Java, Master’s in CS, experience with AWS and REST APIs, last updated CV: 2024-06-10.”
7. **Dashboard / UI**
   * Show ranked candidates, score breakdown, highlighted matched sentences, filters (location, last update, university, certs).
   * Allow manual override / save to job pipeline.
8. **Monitoring & Feedback**
   * Collect HR feedback (accept/reject) to improve ranking (optional: simple supervised re-ranking or fine-tune threshold).

**3) Suggested open-source models & libraries**

* **Embeddings / Semantic**: Sentence Transformers (e.g., all-MiniLM-L6-v2 for speed; all-mpnet-base-v2 for better accuracy). For multilingual CVs: LaBSE or distiluse-base-multilingual-cased.
* **NLP parsing**: spaCy (NER, tokenization), Hugging Face Transformers for any extra tasks.
* **Resume parsing**: pyresparser, resume-parser, or custom pipeline using spaCy + rules.
* **OCR**: Tesseract (for scanned images).
* **Vector DB / Index**: FAISS (local), Annoy, Milvus (scalable).
* **Backend / API**: Python (FastAPI / Flask).
* **Frontend**: React + Tailwind (or simple Flask templates) for prototype.
* **Storage**: PostgreSQL (structured), object store for files (S3 or local).

**4) Preprocessing steps (detailed)**

1. **File handling**
   * Convert DOCX/PDF → plain text. If scanned, run OCR.
2. **Section detection**
   * Use heuristics (keywords: “experience”, “education”, “skills”) and NLP chunking.
3. **Normalization**
   * Lowercase, remove stopwords for some tasks, unify synonyms in a skill dictionary (e.g., ‘js’ → ‘JavaScript’).
4. **Entity extraction**
   * Extract dates (to compute years of experience), degrees, universities, companies.
5. **Skill extraction**
   * Use dictionary + fuzzy matching + embeddings to detect skills even if phrased differently.
6. **Embedding computation**
   * Compute sentence / paragraph embeddings and a full-document embedding. Store both.

**5) Similarity scoring — practical formula & example**

Combine semantic similarity with structured matches. One simple linear scoring scheme:

Score = 0.50 × SemanticSim  
   + 0.20 × SkillMatchScore  
   + 0.15 × ExperienceScore  
   + 0.10 × EducationScore  
   + 0.05 × RecencyScore

Where each subscore is normalized to [0,1].

Example (digit-by-digit style):

* Semantic cosine similarity = 0.82 → contributes 0.50 × 0.82 = 0.41
* SkillMatchScore: matched 8 of 10 required skills = 0.8 → contributes 0.20 × 0.8 = 0.16
* ExperienceScore: required 5 years; candidate has 6 years → normalized min(1,6/5)=1.0 → 0.15 × 1.0 = 0.15
* EducationScore: job needs Master’s; candidate has Master’s → 1.0 → 0.10 × 1.0 = 0.10
* RecencyScore: last CV update 6 months ago → map to 1/(1+months/12)= ~0.6667 → 0.05 × 0.6667 ≈ 0.0333

Total score = 0.41 + 0.16 + 0.15 + 0.10 + 0.0333 = **0.8533** → display as **85% match**.

(You can tune weights per job family or let HR configure them.)

**6) Explainability — what to show**

For each shortlisted CV show:

* Match percentage and color-coded badge (e.g., 85% — Strong Match).
* Breakdown bars for each component (semantic, skills, experience, education, recency).
* Highlight snippets from CV that led to matches (e.g., “Worked on AWS Lambda and RESTful APIs — matched to ‘cloud computing’ requirement”).
* Short auto-generated sentence: “5 years Java development; Master’s in Computer Science; AWS & REST APIs experience; last updated June 2024.”

Why this matters: HR needs transparency to trust the assistant and be able to justify choices.

**7) Dashboard & UX suggestions**

* Main view: job post at top, ranked candidate list with filters (location, years exp, degree, last updated).
* Candidate detail modal: full CV text, highlighted matches, score breakdown, contact buttons (email, schedule interview).
* Bulk actions: mark as shortlisted, reject, send message.
* Feedback buttons: “Good match / Bad match” — feed into logging for later improvement.

Tech stack for prototype:

* Backend: FastAPI + Uvicorn
* Frontend: React (Vite) + Tailwind
* DB: Postgres + FAISS on server for vectors
* Auth: simple JWT or mock auth for prototype

**8) Data, training and evaluation**

* **Datasets**: Use synthetic or anonymized CV collections. Public datasets exist (resume datasets) — ensure privacy.
* **Evaluation**:
  + Precision@k (how many of top-k are actually relevant).
  + NDCG (ranking quality).
  + Human evaluation by HR graders for relevance and explainability.
* **A/B testing**: If integrating into an HR team, compare time-to-hire and shortlisting accuracy vs manual.

**9) Student task breakdown (milestones)**

1. **Week 1–2**: Research & design; pick models and tech stack. Prepare sample CV dataset.
2. **Week 3–4**: Implement ingestion and parsing pipeline (PDF, DOCX, OCR path).
3. **Week 5**: Build embedding pipeline (use SBERT), index CV embeddings into FAISS.
4. **Week 6**: Implement similarity scoring + structured feature extraction (experience, education).
5. **Week 7**: Build basic UI to show ranked candidates and breakdown.
6. **Week 8**: Add filters, explainability highlights, and collect HR feedback mockups; finalize demo + slides + docs.

Deliverables: working prototype, presentation (architecture, models, limitations), documentation including future work.

**10) Limitations, ethics & future improvements**

* **Bias**: Model may pick up biased language (gender, age, nationality). Include bias detection (e.g., audit scores by protected attributes) and allow blind screening (hide names, photos).
* **Privacy**: Resumes contain PII — enforce encryption at rest, access control, and data retention policies.
* **Multilingual CVs**: incorporate LaBSE or translate+embed pipelines.
* **Domain adaptation**: for niche roles (e.g., embedded C) you may need domain-specific embeddings or fine-tuning.
* **Dynamic learning**: use HR feedback to train a re-ranker (offline), but be careful to avoid reinforcing biases.
* **Integration**: LinkedIn or external sources — consider consent and legal constraints (GDPR).

**11) Quick starter code pointers (conceptual)**

* Use sentence-transformers to compute embeddings:
  + model.encode(job\_description) and model.encode(resume\_text)
* Build FAISS index: faiss.IndexFlatIP (for cosine/dot product with normalized vectors).
* Structured extraction: date parsing → compute years of experience.

**12) Evaluation metrics / success criteria**

* Time saved in shortlisting (target: 70% reduction in manual screening time).
* Precision@10 > 0.7 for relevant roles (tunable).
* HR acceptance rate of top 5 candidates > baseline.

**13) Short demo script for presentation**

1. Show CSV of resumes raw.
2. Upload a job description.
3. System returns top 10 candidates with match %, highlights, and filters applied.
4. HR selects candidate and confirms — show feedback logging.

If you want, I can:

* Provide a **minimal code prototype** (backend script to parse resumes, compute embeddings, build FAISS index, and query with a job description).
* Or produce a **React dashboard mockup** and example JSON outputs for a ranked candidate.

Which would you prefer next — the code prototype (Python) or a frontend prototype (React)?