Recruitment AI Platform

An **AI-powered recruitment automation platform** built for WOGOM. It streamlines the entire hiring pipeline — from JD creation to candidate evaluation — using LLM-powered agents (Groq/Llama 3.3), a FastAPI backend, PostgreSQL database, and a React frontend.

Table of Contents

Features

For Team Leads

- **AI-powered JD Creation** — Generate professional job descriptions via a structured form or natural language chat
- **Clarifying Questions** — AI generates MCQs from the Head-of-Department perspective to refine role requirements
- **Candidate Profile Building** — Auto-generate ideal candidate profiles from JD data
- **Job Request Workflow** — Create, edit, submit, and cancel job requests with real-time status tracking

For HR Managers

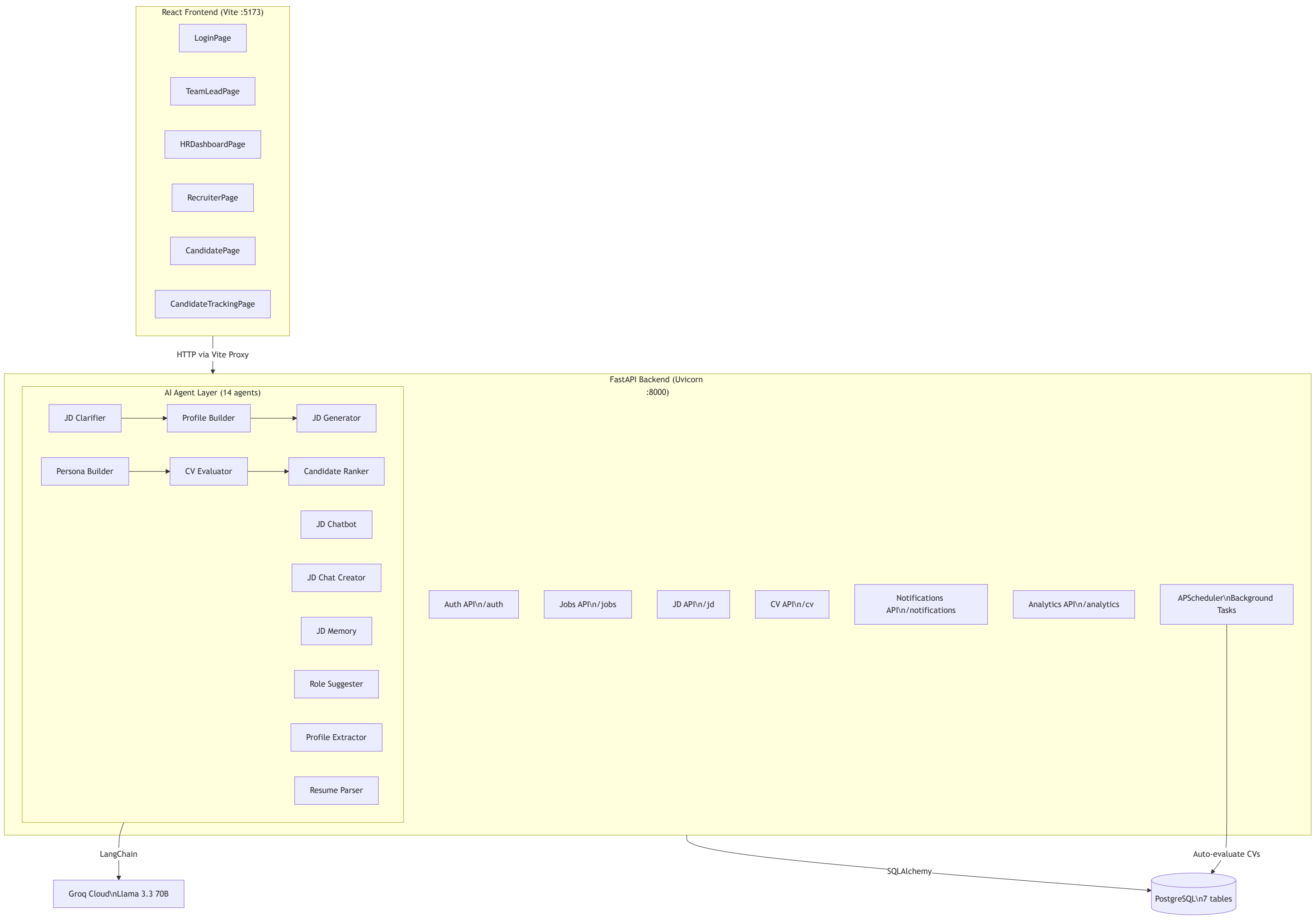
- **Job Review Dashboard** — Review, edit, and activate pending job requests from team leads
- **Candidate Tracking** — Monitor candidates across all active jobs
- **CV Analysis Pipeline** — Upload resumes, generate evaluation personas, score and rank candidates
- **Analytics** — View hiring pipeline metrics across all jobs

- **Notifications** — Real-time alerts for job submissions, activations, evaluations, and deadlines

AI Capabilities

- **JD Memory System** — Learns each user's writing preferences over time
- **Role Suggestions** — AI suggests alternative job titles based on the profile
- **Multi-persona CV Evaluation** — Evaluates candidates through multiple AI-generated expert personas
- **Chat-based JD Refinement** — Iteratively improve JDs through natural conversation
- **DOCX Export** — Export formatted JDs as Word documents

Architecture



Tech Stack

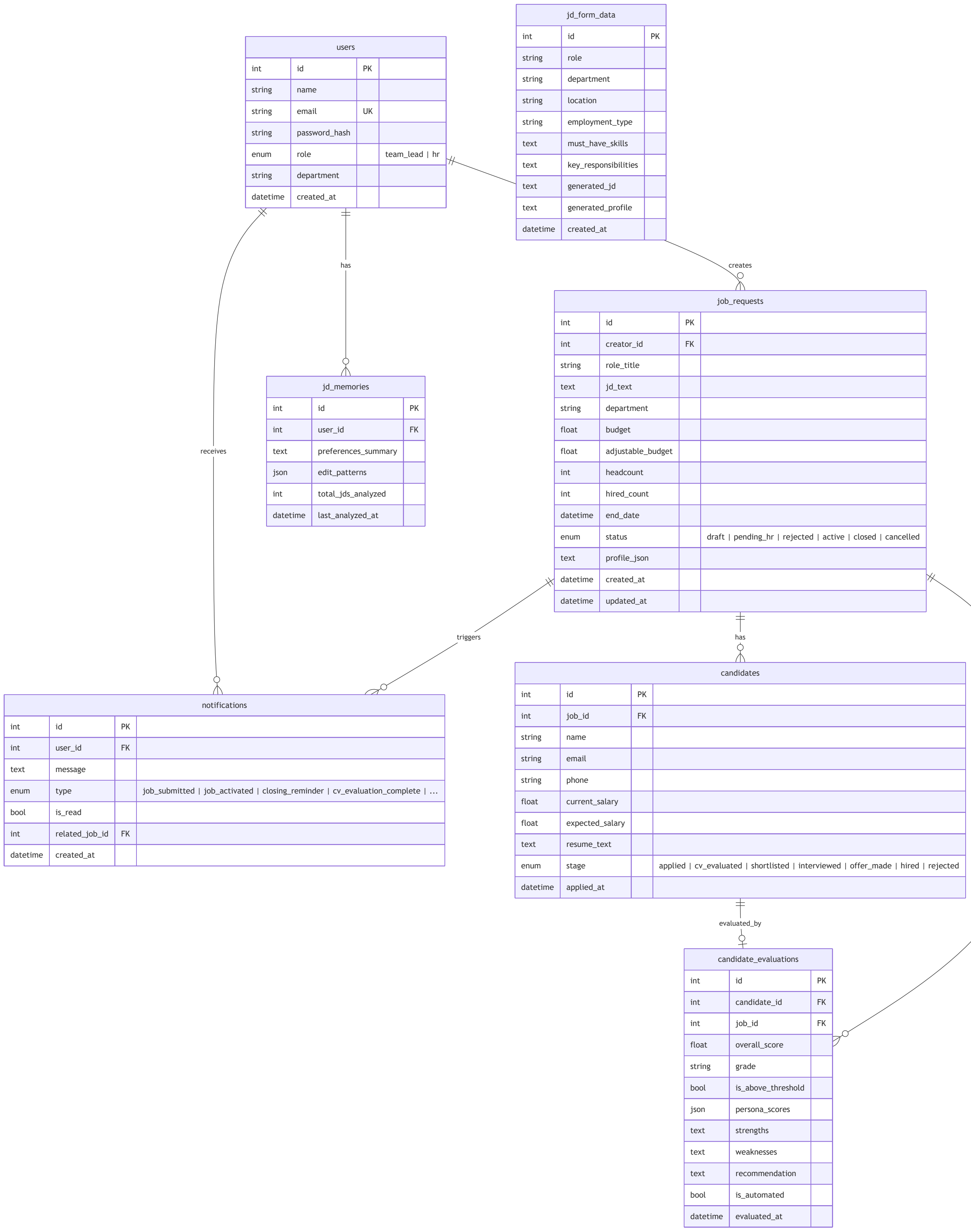
Layer	Technology
Frontend	React 18, Vite 5, React Router 6, Lucide Icons, react-markdown
Backend	Python 3.11+, FastAPI, Uvicorn, SQLAlchemy, Pydantic v2
Database	PostgreSQL 15+
AI/LLM	Groq Cloud (Llama 3.3 70B), LangChain, LangGraph
Embeddings	Sentence Transformers, FAISS
Auth	JWT (python-jose), bcrypt
Scheduling	APScheduler (background tasks)
File Processing	pdfplumber, python-docx, pypdf, ReportLab

Project Structure

```
recruitment_ai/
│
├── app/                        # FastAPI backend
│   ├── __init__.py
│   ├── main.py                 # App entrypoint, router registration, lifespan
│   │
│   ├── api/                    # API route handlers (6 routers)
│   │   ├── auth.py             # Register, login, JWT, role-based access
│   │   ├── jd.py               # JD creation pipeline (forms, clarify, generate, refine, export)
│   │   ├── job_requests.py     # Job request CRUD, submit/activate workflow, notifications
│   │   ├── cv_analysis.py      # Persona generation, CV evaluation, ranking
│   │   ├── notifications.py    # Notification list, read, unread count
│   │   └── analytics.py        # Hiring pipeline analytics
│   │
│   ├── agents/                 # AI agent modules (LLM-powered)
│   │   ├── jd_clarifier.py     # Agent 1: Generate clarifying MCQs
│   │   ├── profile_builder.py  # Agent 2: Build ideal candidate profile
│   │   ├── jd_generator.py     # Agent 3: Generate full JD from profile
│   │   ├── jd_chatbot.py       # Agent 4: Refine JD via chat
│   │   ├── jd_chat_creator.py  # Agent 5: Create JD from natural language
│   │   ├── jd_memory.py        # Agent 6: Learn user preferences
│   │   ├── persona_builder.py  # Agent 7: Generate evaluation personas
│   │   ├── cv_evaluator.py     # Agent 8: Evaluate CVs against personas
│   │   ├── candidate_ranker.py # Agent 9: Rank and shortlist candidates
│   │   ├── role_suggester.py   # Suggest alternative job titles
│   │   ├── profile_extractor.py # Extract profile from raw JD text
│   │   ├── resume_parser.py    # Parse resumes from PDF/DOCX/ZIP
│   │   ├── jd_parser.py        # Parse JD text into structured sections
│   │   └── candidate_intel.py  # Candidate intelligence analysis
│   │
│   ├── db/                     # Database layer
│   │   ├── database.py         # SQLAlchemy engine, session, Base
│   │   └── models.py           # 7 ORM models + enums
│   │
│   └── utils/                  # Shared utilities
│       ├── llm.py              # Groq LLM client factory
│       ├── constants.py        # WOGOM brand constants
│       ├── scheduler.py        # APScheduler background tasks
│       ├── file_export.py      # DOCX/PDF export generators
│       ├── text_cleanup.py     # Text extraction from files
│       └── resume_skills.py    # Skill extraction from resumes
│
├── frontend/                   # React/Vite frontend
│   ├── src/
│   │   ├── App.jsx               # Router + AppShell layout
│   │   ├── main.jsx              # React entry point
│   │   ├── index.css             # Global styles
│   │   ├── services/
│   │   │   └── api.js            # All API calls + auth token management
│   │   ├── components/
│   │   │   ├── Sidebar.jsx/css   # Navigation sidebar
│   │   │   ├── TopBar.jsx/css    # Header with notifications
│   │   │   ├── ProtectedRoute.jsx# Role-based route guard
│   │   │   ├── JdChatCreator.*   # Chat-based JD creation widget
│   │   │   ├── FileUpload.*      # File upload component
│   │   │   ├── StepProgress.*    # Multi-step progress indicator
│   │   │   ├── Navbar.*          # Top navigation bar
│   │   │   └── JdPreview.*       # JD preview component
│   │   └── pages/
│   │       ├── LoginPage.*       # Auth login/register page
│   │       ├── TeamLeadPage.*    # Team Lead dashboard
│   │       ├── HRDashboardPage.* # HR dashboard with job management
│   │       ├── RecruiterPage.*   # JD creation tool (form-based flow)
│   │       ├── CandidatePage.*   # CV analysis pipeline UI
│   │       └── CandidateTrackingPage.* # Candidate tracking across jobs
│   ├── vite.config.js            # Vite config with API proxy
│   └── package.json
│
├── scripts/
│   └── seed_db.py                # Database seeding script
│
├── tests/                        # Test directory
├── .env                          # Environment variables
├── .gitignore
├── requirements.txt              # Python dependencies
└── README.md                     # This file
```

Database Schema

The application uses **7 tables** managed by SQLAlchemy ORM.



users

int	id	PK	
string	name		
string	email	UK	
string	password_hash		
enum	role		team_lead | hr
string	department		
datetime	created_at		

jd_form_data

int	id	PK
string	role	
string	department	
string	location	
string	employment_type	
text	must_have_skills	
text	key_responsibilities	
text	generated_jd	
text	generated_profile	
datetime	created_at	

jd_memories

int	id	PK
int	user_id	FK
text	preferences_summary	
json	edit_patterns	
int	total_jds_analyzed	
datetime	last_analyzed_at	

job_requests

int	id	PK	
int	creator_id	FK	
string	role_title		
text	jd_text		
string	department		
float	budget		
float	adjustable_budget		
int	headcount		
int	hired_count		
datetime	end_date		
enum	status		draft | pending_hr | rejected | active | closed | cancelled
text	profile_json		
datetime	created_at		
datetime	updated_at		

notifications

int	id	PK	
int	user_id	FK	
text	message		
enum	type		job_submitted | job_activated | closing_reminder | cv_evaluation_complete | …
bool	is_read		
int	related_job_id	FK	
datetime	created_at		

candidates

int	id	PK	
int	job_id	FK	
string	name		
string	email		
string	phone		
float	current_salary		
float	expected_salary		
text	resume_text		
enum	stage		applied | cv_evaluated | shortlisted | interviewed | offer_made | hired | rejected
datetime	applied_at		

candidate_evaluations

int	id	PK
int	candidate_id	FK
int	job_id	FK
float	overall_score	
string	grade	
bool	is_above_threshold	
json	persona_scores	
text	strengths	
text	weaknesses	
text	recommendation	
bool	is_automated	
datetime	evaluated_at	

Key Enums

Enum	Values
`UserRole`	`team_lead` , `hr`
`JobStatus`	`draft` , `pending_hr` , `rejected` , `active` , `closed` , `cancelled`
`CandidateStage`	`applied` → `cv_evaluated` → `shortlisted` → `interviewed` → `offer_made` → `hired` / `rejected`
`NotificationType`	`job_submitted` , `job_cancelled` , `job_activated` , `job_rejected` , `closing_reminder` , `cv_evaluation_complete` , `general`

Agent Prompts

All LLM prompts used across the 14 AI agents. Each subsection shows the **variable name**, **file location**, **input variables**, and the **full prompt text**.

1. CLARIFY_PROMPT — JD Clarifier

File: `app/agents/jd_clarifier.py`

Inputs: `{department}` , `{title}` , `{location}` , `{experience_level}` , `{work_mode}` , `{key_skills}` , `{key_responsibilities}` , `{reporting_to}` , `{additional_info}`

```
You are a senior recruitment strategist.

SCENARIO:
The **Head of {department}** has requested to hire a **{title}**.
The Head may not be fully aware of every detail this role needs.
Your job is to generate clarifying questions FROM the Head's perspective —
questions that help the Head think more deeply about what they truly need.

CONTEXT FROM GOOGLE FORM:
- Job Title: {title}
- Department: {department}
- Location: {location}
- Experience Level: {experience_level}
- Work Mode: {work_mode}
- Must-Have Skills: {key_skills}
- Key Responsibilities: {key_responsibilities}
- Reporting To: {reporting_to}
- Additional Info: {additional_info}

TASK:
Generate exactly 5 multiple-choice questions that:
1. Are phrased as if asking the Department Head directly
2. Help clarify responsibilities, success metrics, team dynamics, authority level, and ownership
3. Each question MUST have exactly 4 options
4. Options should be meaningful, specific to the {title} role, and allow MULTI-SELECT
5. Focus on gaps in the form data — things the Head might not have thought about

OUTPUT FORMAT (STRICT JSON ONLY — EXACTLY 5 QUESTIONS):
[
  {
    "id": "q1",
    "question": "As the Head of {department}, ...",
    "options": ["Option A", "Option B", "Option C", "Option D"],
    "target_section": "responsibilities|authority|ownership|success_metrics|team_dynamics"
  },
  ...
]

RULES:
- Output ONLY valid JSON array with exactly 5 questions
- NEVER include "Not Applicable" as an option
- Each question MUST have exactly 4 options
- Questions must feel like they come from a Head-of-Department conversation
- No explanations, no markdown, no extra text
- Do NOT ask about salary, CTC, compensation, work mode, remote/hybrid, travel, shift timing, or urgency
```

2. PROFILE_BUILDER_PROMPT — Profile Builder

File: `app/agents/profile_builder.py`

Inputs: `{role}` , `{department}` , `{form_data}` , `{clarification_answers}`

```
You are a senior HR strategist building an "Ideal Candidate Profile" for the role of **{role}** in the *

You are given:
1. Raw job data collected from a Google Form (ground truth).
2. Clarifying answers given by the Department Head.

YOUR TASK:
Build a structured candidate profile that will later be used to write a Job Description.

INPUTS:
─────────────────────────────
GOOGLE FORM DATA (Ground Truth):
{form_data}

CLARIFICATION ANSWERS (from Dept. Head):
{clarification_answers}
─────────────────────────────


OUTPUT FORMAT (STRICT JSON ONLY):
{
  "role": "{role}",
  "department": "{department}",
  "profile_summary": "2-3 sentence paragraph...",
  "core_competencies": ["Competency 1: explanation", ...],
  "behavioral_traits": ["Trait 1: why it's relevant", ...],
  "success_metrics": ["30 days?", "90 days?", "6 months?"],
  "team_context": "1-2 sentences...",
  "key_responsibilities_refined": ["Responsibility 1", ...],
  "must_have_skills_refined": ["Skill 1", ...],
  "nice_to_have_skills": ["Skill 1", ...]
}

RULES:
- Output ONLY valid JSON. No markdown, no explanations.
- Use the Google Form data as the primary source.
- Use the clarification answers to add depth and specificity.
- Do NOT invent information not supported by the inputs.
```

3. JD_GENERATOR_PROMPT — JD Generator

File: `app/agents/jd_generator.py`

Inputs: `{profile_json}` , `{role}` , `{department}` , `{location}` , `{experience_phrase}` , `{facts}` , `{mission}` , `{vision}` , `{tone}` , `{culture}` , `{language_rules}` , `{about_wogom}`

```
You are a senior HR and talent acquisition expert at WOGOM.

Your task is to generate a clean, professional, hiring-ready Job Description (JD).

──────────────────────────────

🔴 CRITICAL SOURCE OF TRUTH 🔴
──────────────────────────────

The **IDEAL CANDIDATE PROFILE** below is your **SINGLE SOURCE OF TRUTH**.
- You MUST use the exact content from this profile (responsibilities, skills, traits).
- The Google Form data is ONLY for metadata (Location, CTC, etc.).
- If the Form data contradicts the Profile, **IGNORE THE FORM DATA**.
- Do NOT hallucinate or invent new requirements.

──────────────────────────────

IDEAL CANDIDATE PROFILE (PRIMARY)
──────────────────────────────

{profile_json}

──────────────────────────────

METADATA & CONTEXT
──────────────────────────────

Role: {role}
Department: {department}
Location: {location}
Experience: {experience_phrase}

GOOGLE FORM DATA (SECONDARY - Metadata Only):
{facts}

──────────────────────────────

COMPANY BRAND GUIDELINES
──────────────────────────────

Mission: {mission}
Vision: {vision}
Tone: {tone}
Culture: {culture}
Language Rules: {language_rules}

──────────────────────────────

OUTPUT FORMAT (STRICTLY FOLLOW)
──────────────────────────────


# {role}

**Location:** {location}

## About Us
```

```
{about_wogom}

## Role Overview
2–3 concise sentences from `profile_summary`.

## Key Responsibilities
5–7 bullet points from `key_responsibilities_refined`.
STRICT: ONE LINE ONLY per bullet, MAXIMUM 30 words, start with "• ".

## Requirements
### Must-Have Skills
4–6 bullet points from `must_have_skills_refined` and `core_competencies`.

### Nice-to-Have Skills
2–3 bullet points from `nice_to_have_skills`.

## Who Will Succeed in This Role
2–3 sentences from `behavioral_traits` and `success_metrics`.

─────────────────────────────
FINAL CHECKLIST
─────────────────────────────
- Did you use the Profile as the source of truth?
- Are all bullets SINGLE LINE?
- Is the tone professional?
- Output ONLY the formatted JD.
```

4. REFINE_PROMPT — JD Chatbot (Refinement)

File: `app/agents/jd_chatbot.py`

Inputs: `{current_jd}` , `{instruction}`

```
You are an expert Job Description editor.

You are given:
1. The current Job Description
2. A user instruction to modify it

TASK:
Apply the user's instruction to the Job Description and return the UPDATED version.

RULES:
- Apply ONLY the requested change. Do NOT rewrite unrelated sections.
- If the user says "remove X", remove it.
- If the user says "add X", add it in the most appropriate section.
- If the user says "make it more concise", shorten sentences without losing meaning.
- If the user says "change tone to X", adjust the language tone.
- Preserve the overall structure (headings, sections, bullets).
- Output ONLY the updated Job Description. No explanations, no notes.

CURRENT JD:
{current_jd}

USER INSTRUCTION:
{instruction}

Return ONLY the updated Job Description:
```

5. CHAT_CREATE_PROMPT — JD Chat Creator

File: `app/agents/jd_chat_creator.py`

Inputs: `{about_wogom}` , `{mission}` , `{vision}` , `{tone}` , `{culture}` , `{language_rules}` , `{memory_context}` , `{user_prompt}`

```
You are an expert HR and talent acquisition specialist at WOGOM.

Your task: Generate a COMPLETE, professional, hiring-ready Job Description (JD) from the user's natural

─────────────────────────────
ABOUT THE COMPANY
─────────────────────────────
{about_wogom}

─────────────────────────────
COMPANY BRAND GUIDELINES
─────────────────────────────
Mission: {mission}
Vision: {vision}
Tone: {tone}
Culture: {culture}
Language Rules: {language_rules}

{memory_context}

─────────────────────────────
USER PROMPT
─────────────────────────────
{user_prompt}

─────────────────────────────
INSTRUCTIONS
─────────────────────────────
From the user's prompt, extract:
- Role/Title, Department, Location, Experience level
- Key responsibilities, Skills, Employment type
- Any specific tone or style preferences

Then generate a complete JD with sections:
# [Role Title], ## About Us, ## Role Overview, ## Key Responsibilities,
## Requirements (### Must-Have / ### Nice-to-Have), ## Who Will Succeed

─────────────────────────────
OUTPUT FORMAT
─────────────────────────────
Return a JSON object:
{
    "jd": "<the complete JD in markdown>",
    "role": "<extracted role title>",
    "department": "<department or 'General'>",
    "location": "<location or 'India'>",
    "experience": "<experience or ''>",
    "employment_type": "<type or 'Full-time'>"
}

RULES:
- Output ONLY the JSON object. No explanations, no markdown code fences.
- If the user prompt is vague, make reasonable assumptions.
- Incorporate any user preferences from the memory context section.
```

6. MEMORY_ANALYSIS_PROMPT — JD Memory (Session Analysis)

File: `app/agents/jd_memory.py`

Inputs: `{initial_prompt}` , `{edit_history}` , `{final_jd}` , `{existing_preferences}`

```
You are a user preference analyst for a Job Description creation tool.

Your task: Analyze this user's JD creation session to understand their preferences, style, and patterns.

─────────────────────────────
SESSION DATA
─────────────────────────────
Initial Prompt: {initial_prompt}

Edit History (chronological):
{edit_history}

Final JD (the version the user accepted):
{final_jd}

{existing_preferences}

─────────────────────────────
ANALYSIS INSTRUCTIONS
─────────────────────────────
Study the user's behavior:
1. What did they add/remove/change in their edits?
2. What tone do they prefer? (formal, casual, professional, creative)
3. What sections do they emphasize or de-emphasize?
4. How detailed do they like bullet points?
5. Do they prefer concise or comprehensive JDs?
6. Any recurring patterns in their edit instructions?
7. What structure preferences do they have?

Return a JSON object:
{
    "preferences_summary": "<3-5 sentence natural language summary>",
    "patterns": {
        "tone": "<preferred tone>",
        "detail_level": "<concise|moderate|comprehensive>",
        "bullet_count_preference": "<few|moderate|many>",
        "emphasis_sections": ["<sections they expand>"],
        "removed_sections": ["<sections they remove>"],
        "style_notes": ["<specific style observations>"]
    }
}

RULES:
- Output ONLY the JSON object. No explanations.
- Be specific — vague preferences like "good JDs" are useless.
```

7. MERGE_PROMPT — JD Memory (Preference Merging)

File: `app/agents/jd_memory.py`

Inputs: `{existing}` , `{new_observations}` , `{total_jds}`

```
You are merging JD creation preferences for a user.

EXISTING PREFERENCES:
{existing}

NEW SESSION OBSERVATIONS:
{new_observations}

TOTAL JDs ANALYZED: {total_jds}

Create an UPDATED preferences summary that intelligently combines the existing preferences
with the new observations. Give more weight to consistent patterns seen across multiple sessions.

Return a JSON object:
{
    "preferences_summary": "<Updated 3-5 sentence summary reflecting all sessions>",
    "patterns": {
        "tone": "<preferred tone>",
        "detail_level": "<concise|moderate|comprehensive>",
        "bullet_count_preference": "<few|moderate|many>",
        "emphasis_sections": ["<sections they expand or care about>"],
        "removed_sections": ["<sections they remove or minimize>"],
        "style_notes": ["<specific style observations>"]
    }
}

Output ONLY the JSON. No explanations.
```

8. PERSONA_BUILDER_PROMPT — Persona Builder

File: `app/agents/persona_builder.py`

Inputs: `{profile}`

```
You are a senior hiring strategist.

You are given an Ideal Candidate Profile for a role.
Your task is to create 3–5 DISTINCT ideal candidate personas — each representing a DIFFERENT type of
person who could succeed in this role.

These personas will later be used to evaluate real candidate CVs, so they must be specific and actionabl

INPUT:
─────────────────────────────
IDEAL CANDIDATE PROFILE:
{profile}
─────────────────────────────

OUTPUT FORMAT (STRICT JSON ARRAY):
[
  {
    "persona_id": "P1",
    "name": "Short Persona Title (e.g. 'The Scalable Systems Expert')",
    "summary": "2–3 sentence description of who this persona is",
    "experience_range": "X–Y years",
    "core_strengths": ["Strength 1: why it matters", ...],
    "required_skills": ["Skill 1", "Skill 2", ...],
    "nice_to_have_skills": ["Skill 1", ...],
    "behavioral_traits": ["Trait 1: why it's relevant", ...],
    "red_flags": ["Warning sign 1", ...],
    "success_definition": "What does success look like in 6 months?"
  }
]

RULES:
- Create 3–5 personas. Each must represent a DIFFERENT hiring path.
- Use ONLY information from the given profile. Do NOT hallucinate.
- Each persona should have different experience ranges and strengths.
- Output ONLY valid JSON array. No markdown, no explanations.
```

9. CV_VS_PERSONA_PROMPT — CV Evaluator

File: `app/agents/cv_evaluator.py`

Inputs: `{cv}` , `{persona}` , `{persona_id}`

```
You are an expert technical interviewer and hiring evaluator.

You are given:
1. A candidate's CV (parsed into structured sections).
2. An ideal candidate persona describing what a successful hire looks like.

YOUR TASK:
Evaluate how well the candidate matches this specific persona.

SCORING DIMENSIONS (evaluate each):
- Skill Match: Do their skills align with required skills?
- Experience Fit: Years and depth of experience vs. persona expectation
- Responsibility Match: Have they done similar work?
- Behavioral Signals: Ownership, leadership, initiative
- Domain Fit: Industry or domain relevance
- Risk Flags: Job hopping, shallow roles, gaps

Give a score from 0 to 100.

INPUTS:
─────────────────────────────
CANDIDATE CV:
{cv}

PERSONA:
{persona}
─────────────────────────────

OUTPUT FORMAT (STRICT JSON ONLY):
{
  "persona_id": "{persona_id}",
  "score": <integer 0–100>,
  "grade": "<A+ / A / A- / B+ / B / B- / C+ / C / C- / D / F>",
  "strengths": ["Strength 1", "Strength 2"],
  "gaps": ["Gap 1", "Gap 2"],
  "explanation": "2–3 sentence summary of the fit analysis"
}

RULES:
- Be strict but fair. Do not inflate scores.
- Cite specific evidence from the CV.
- Output ONLY valid JSON. No markdown, no extra text.
```

10. ROLE_SUGGESTER_PROMPT — Role Suggester

File: `app/agents/role_suggester.py`
Inputs: `{profile_json}`

```
You are a senior HR naming specialist.

Given the Ideal Candidate Profile below, suggest **5 to 7 professional job role titles** that
accurately describe the position.

RULES:
1. The FIRST title in your list MUST be the original role name from the profile.
2. Each alternative title must be realistic, industry-standard, and commonly used on job boards.
3. Titles should vary in seniority language (e.g., "Lead", "Senior", "Specialist", "Analyst").
4. Do NOT invent fantasy titles. Stick to real-world job titles.
5. Keep titles concise (2–5 words).
6. Output ONLY a valid JSON array of strings. No markdown, no explanations.

PROFILE:
{profile_json}

OUTPUT FORMAT (STRICT):
["Original Role Title", "Alternative 1", "Alternative 2", "Alternative 3", ...]
```

11. PROFILE_FROM_JD_PROMPT — Profile Extractor

File: `app/agents/profile_extractor.py`
Inputs: `{jd_text}`

```
You are a senior HR strategist.

You are given a finalized Job Description (JD). Your task is to extract
a structured "Ideal Candidate Profile" from it.

This profile will later be used to:
1. Generate ideal candidate personas.
2. Evaluate candidate CVs against those personas.

So make it specific, detailed, and grounded in what the JD says.

_____________________________
JOB DESCRIPTION:
{jd_text}
_____________________________


OUTPUT FORMAT (STRICT JSON ONLY):
{
  "role": "<job title>",
  "department": "<department or 'General'>",
  "profile_summary": "2–3 sentence paragraph...",
  "core_competencies": ["Competency: explanation", ...],
  "behavioral_traits": ["Trait: why relevant", ...],
  "success_metrics": ["30 days?", "90 days?", "6 months?"],
  "team_context": "1–2 sentences...",
  "key_responsibilities_refined": [...],
  "must_have_skills_refined": [...],
  "nice_to_have_skills": [...]
}

RULES:
- Output ONLY valid JSON. No markdown, no explanations.
- Extract ALL information from the JD. Do NOT hallucinate.
- If a section is not mentioned, use an empty list [] or a reasonable inference.
```

12. JD_PARSER_PROMPT — JD Parser

File: `app/agents/jd_parser.py`
Inputs: `{jd_text}`

```
You are an information extraction system.
Extract structured data from the Job Description below.

Return ONLY valid JSON in this schema:
{
  "role": string,
  "must_have_skills": list of strings,
  "nice_to_have_skills": list of strings,
  "experience_years": number,
  "responsibilities": list of strings
}

Job Description:
{jd_text}
```

> **Note:** This prompt is used as a **fallback only** — the parser first attempts regex-based extraction and calls the LLM only when critical fields are missing.

13. Candidate Intel (No LLM — Serper API)

File: `app/agents/candidate_intel.py`

This agent does **not** use an LLM prompt. Instead, it:

1. Extracts the job role from the JD title
2. Builds 5 HR-centric search queries (e.g., "companies known for strong {role} talent")
3. Calls the **Serper API** (Google Search) to find companies where experienced professionals in this role work
4. Returns structured company intelligence data

14. Candidate Ranker (No LLM — Pure Python)

File: `app/agents/candidate_ranker.py`

This agent does **not** use an LLM. It applies a **deterministic ranking algorithm**:

1. Sorts all evaluated candidates by `overall_score` (descending)
2. Applies **persona diversity capping** (`max_per_persona = max(4, top_n / 2)`) so no single persona type dominates
3. Returns a balanced top-N shortlist with rank, score, grade, best-fit persona, and explanation

AI Agents

The platform uses a multi-agent architecture where each agent handles a specific step:

JD Creation Pipeline

#	Agent	File	Purpose
1	**JD Clarifier**	`jd_clarifier.py`	Generates 5 MCQs from the department head's perspective
2	**Profile Builder**	`profile_builder.py`	Builds ideal candidate profile from form data + clarification answers
3	**JD Generator**	`jd_generator.py`	Generates a complete JD from profile, with WOGOM brand voice
4	**JD Chatbot**	`jd_chatbot.py`	Interactive JD refinement via natural language instructions
5	**JD Chat Creator**	`jd_chat_creator.py`	Creates JDs from a single free-text prompt
6	**JD Memory**	`jd_memory.py`	Analyzes edit patterns to learn user preferences over time

CV Analysis Pipeline

#	Agent	File	Purpose
7	**Persona Builder**	`persona_builder.py`	Generates 5 expert evaluation personas from a role profile
8	**CV Evaluator**	`cv_evaluator.py`	Scores each CV against all personas with detailed feedback
9	**Candidate Ranker**	`candidate_ranker.py`	Ranks candidates and produces a top-N shortlist

Support Agents

Agent	File	Purpose
Role Suggester	`role_suggester.py`	Suggests alternative job titles
Profile Extractor	`profile_extractor.py`	Extracts structured profile from raw JD text
Resume Parser	`resume_parser.py`	Parses resumes from PDF/DOCX/ZIP files

Agent	File	Purpose
JD Parser	`jd_parser.py`	Parses JD markdown into structured sections
Candidate Intel	`candidate_intel.py`	Generates candidate intelligence insights

All agents use **Groq's Llama 3.3 70B** model via LangChain for fast, free inference.

API Reference

Authentication (`/auth`)

Method	Endpoint	Description
`POST`	`/auth/register`	Register a new user (team_lead or hr)
`POST`	`/auth/login`	Login with email/password → JWT token
`GET`	`/auth/me`	Get current user info (requires auth)

Job Requests (`/jobs`)

Method	Endpoint	Description	Auth
`POST`	`/jobs/`	Create a draft job	Team Lead
`GET`	`/jobs/`	List user's jobs (HR sees all)	Any
`GET`	`/jobs/{id}`	Get job details	Any
`PUT`	`/jobs/{id}`	Update a draft job	Team Lead
`POST`	`/jobs/{id}/submit`	Submit to HR for review	Team Lead
`POST`	`/jobs/{id}/cancel`	Cancel a job	Team Lead
`GET`	`/jobs/incoming/pending`	List pending jobs	HR
`POST`	`/jobs/{id}/activate`	Activate a pending job	HR
`PUT`	`/jobs/{id}/hr-edit`	HR edits before activation	HR
`GET`	`/jobs/all-candidates`	All candidates across jobs	HR
`POST`	`/jobs/parse-content`	Parse DOCX/PDF to text	Any

JD Pipeline (`/jd`)

Method	Endpoint	Description
`GET`	`/jd/forms`	List saved JD intake forms
`POST`	`/jd/forms`	Save a new form
`PUT`	`/jd/forms/{id}/jd`	Update generated JD on form
`PUT`	`/jd/forms/{id}/profile`	Save generated profile on form
`DELETE`	`/jd/forms/{id}`	Delete a saved form
`POST`	`/jd/clarify`	Generate clarifying questions
`POST`	`/jd/profile`	Build candidate profile
`POST`	`/jd/suggest-roles`	Suggest alternative role titles
`POST`	`/jd/generate`	Generate full JD
`POST`	`/jd/refine`	Refine JD with instructions
`POST`	`/jd/chat-create`	Create JD from prompt
`POST`	`/jd/chat-refine`	Refine JD via chat
`GET`	`/jd/memory`	Get user's JD preferences
`POST`	`/jd/memory/analyze`	Analyze session & update memory
`POST`	`/jd/export-docx`	Export JD as DOCX

CV Analysis (`/cv`)

Method	Endpoint	Description
`POST`	`/cv/personas`	Generate evaluation personas
`POST`	`/cv/evaluate`	Evaluate CVs against personas
`POST`	`/cv/rank`	Rank evaluated candidates
`POST`	`/cv/full`	Full pipeline (personas → evaluate → rank)

Notifications (`/notifications`)

Method	Endpoint	Description
`GET`	`/notifications/`	List user's notifications
`GET`	`/notifications/unread-count`	Get unread count
`POST`	`/notifications/{id}/read`	Mark one as read
`POST`	`/notifications/read-all`	Mark all as read

Analytics (`/analytics`)

Method	Endpoint	Description
`GET`	`/analytics/pipeline`	Hiring pipeline metrics (HR only)

Getting Started

Prerequisites

- **Python 3.11+**
- **Node.js 18+** and npm
- **PostgreSQL 15+**
- **Groq API Key** (free at console.groq.com)

1. Clone & Setup Backend

```
git clone <repo-url>
cd recruitment_ai

# Create virtual environment
python -m venv venv
venv\Scripts\activate        # Windows
# source venv/bin/activate   # Linux/Mac

# Install dependencies
pip install -r requirements.txt
```

2. Configure Environment

Create a `.env` file in the project root:

```
DATABASE_URL=postgresql://postgres:yourpassword@localhost:5432/recruitment_ai
GROQ_API_KEY=gsk_your_groq_api_key_here
JWT_SECRET=your-secret-key-change-in-production
```

3. Setup Database

```
# Create the PostgreSQL database
psql -U postgres -c "CREATE DATABASE recruitment_ai;"

# Seed initial users (optional)
python scripts/seed_db.py
```

Default seeded users:

Email	Password	Role
`lead@example.com`	`password123`	Team Lead
`hr@example.com`	`password123`	HR Manager

4. Run Backend

```
uvicorn app.main:app --reload
# → Backend running at http://localhost:8000
```

5. Setup & Run Frontend

```
cd frontend
npm install
npm run dev
# → Frontend running at http://localhost:5173
```

Open **http://localhost:5173** and log in with the seeded credentials.

Environment Variables

Variable	Required	Default	Description
`DATABASE_URL`	✅	`postgresql://postgres:wogom@localhost:5432/recruitment_ai`	PostgreSQL connection string
`GROQ_API_KEY`	✅	—	Groq API key for LLM inference
`JWT_SECRET`	⚠️	`super-secret-change-me`	JWT signing secret (change in production!)

AWS Deployment Guide

This section covers deploying the Recruitment AI platform to AWS with a production-ready setup.

Recommended AWS Architecture

```
┌─────────────────────────────────────────────────────┐
│                   Route 53 (DNS)                    │
│               recruit.yourdomain.com                 │
└─────────────────────────────────────────────────────┘
                      │
                      │
┌─────────────────────▼───────────────────────────────┐
│             CloudFront (CDN) — optional             │
│         Static frontend assets + caching            │
└─────────────────────────────────────────────────────┘
                      │
                      │
┌────────────┬────────▼──────────┬────────────────────┐
│  S3 Bucket │  Application Load │  Amazon RDS        │
│  (Frontend)│  Balancer (ALB)   │  (PostgreSQL 15)   │
│  Static Site│        │         │  db.t3.micro       │
└────────────┘        │         └────────────────────┘
                      │
              ┌───────▼────────┐
              │   EC2 / ECS    │
              │   (Backend)    │
              │  FastAPI +     │
              │  Uvicorn       │
              │  + Gunicorn    │
              └────────────────┘
```

AWS Services Required

Service	Purpose	Estimated Cost
EC2 (t3.small or t3.medium)	Backend server (FastAPI + Uvicorn)	~$15-30/month
RDS (db.t3.micro PostgreSQL)	Managed PostgreSQL database	~$15-25/month
S3	Host static React frontend build	~$1-3/month
CloudFront	CDN for frontend (optional)	~$1-5/month
Route 53	DNS management	~$0.50/month
ALB	Load balancer for backend (optional)	~$16/month
ACM	Free SSL/TLS certificates	Free
Secrets Manager	Store API keys and secrets	~$1/month
Total (minimal)	EC2 + RDS + S3	**~$30-60/month**

Step-by-Step Deployment

1. Setup RDS (PostgreSQL)

```
# Create an RDS instance via AWS Console or CLI:
aws rds create-db-instance \
  --db-instance-identifier recruitment-ai-db \
  --db-instance-class db.t3.micro \
  --engine postgres \
  --engine-version 15 \
  --master-username postgres \
  --master-user-password YOUR_SECURE_PASSWORD \
  --allocated-storage 20 \
  --publicly-accessible \
  --vpc-security-group-ids sg-xxxxxxxx
```

Note the **RDS endpoint** (e.g., `recruitment-ai-db.xxxxx.us-east-1.rds.amazonaws.com`).

2. Launch EC2 Instance

```
# Recommended: Amazon Linux 2023 or Ubuntu 22.04
# Instance type: t3.small (2 vCPU, 2 GB RAM) minimum

# SSH into your instance
ssh -i your-key.pem ec2-user@your-ec2-ip
```

3. Install Dependencies on EC2

```
# Update system
sudo yum update -y                    # Amazon Linux
# sudo apt update && sudo apt upgrade -y  # Ubuntu

# Install Python 3.11+
sudo yum install python3.11 python3.11-pip -y

# Install PostgreSQL client (for psql)
sudo yum install postgresql15 -y

# Install Node.js 18 (for building frontend)
curl -fsSL https://rpm.nodesource.com/setup_18.x | sudo bash -
sudo yum install nodejs -y

# Install Nginx (reverse proxy)
sudo yum install nginx -y
```

4. Deploy Backend

```
# Clone your repo
git clone <your-repo-url> /home/ec2-user/recruitment_ai
cd /home/ec2-user/recruitment_ai

# Create virtual environment
python3.11 -m venv venv
source venv/bin/activate

# Install Python dependencies
pip install -r requirements.txt
pip install gunicorn

# Create .env file
cat > .env << EOF
DATABASE_URL=postgresql://postgres:YOUR_PASSWORD@your-rds-endpoint:5432/recruitment_ai
GROQ_API_KEY=gsk_your_key_here
JWT_SECRET=$(openssl rand -hex 32)
EOF

# Create the database
psql -h your-rds-endpoint -U postgres -c "CREATE DATABASE recruitment_ai;"

# Seed initial data (optional)
python scripts/seed_db.py

# Test startup
uvicorn app.main:app --host 0.0.0.0 --port 8000
```

5. Setup Gunicorn + Systemd Service

Create `/etc/systemd/system/recruitment-ai.service` :

```
[Unit]
Description=Recruitment AI Backend
After=network.target

[Service]
Type=simple
User=ec2-user
WorkingDirectory=/home/ec2-user/recruitment_ai
Environment="PATH=/home/ec2-user/recruitment_ai/venv/bin"
EnvironmentFile=/home/ec2-user/recruitment_ai/.env
ExecStart=/home/ec2-user/recruitment_ai/venv/bin/gunicorn \
    app.main:app \
    --workers 4 \
    --worker-class uvicorn.workers.UvicornWorker \
    --bind 0.0.0.0:8000 \
    --timeout 120 \
    --access-logfile /var/log/recruitment-ai/access.log \
    --error-logfile /var/log/recruitment-ai/error.log
Restart=always
RestartSec=3

[Install]
WantedBy=multi-user.target
```

```
# Create log directory
sudo mkdir -p /var/log/recruitment-ai
sudo chown ec2-user:ec2-user /var/log/recruitment-ai

# Enable and start the service
sudo systemctl daemon-reload
sudo systemctl enable recruitment-ai
sudo systemctl start recruitment-ai

# Check status
sudo systemctl status recruitment-ai
```

6. Build & Deploy Frontend to S3

```
# Build the frontend
cd /home/ec2-user/recruitment_ai/frontend

# Update API base URL for production
# In src/services/api.js, change:
#   const API_BASE = '';
# to:
#   const API_BASE = 'https://api.yourdomain.com';

npm install
npm run build

# Upload to S3
aws s3 sync dist/ s3://your-bucket-name --delete

# Enable static website hosting on the S3 bucket
aws s3 website s3://your-bucket-name \
  --index-document index.html \
  --error-document index.html
```

7. Configure Nginx as Reverse Proxy

Create `/etc/nginx/conf.d/recruitment-ai.conf`:

```
server {
    listen 80;
    server_name api.yourdomain.com;

    location / {
        proxy_pass http://127.0.0.1:8000;
        proxy_set_header Host $host;
        proxy_set_header X-Real-IP $remote_addr;
        proxy_set_header X-Forwarded-For $proxy_add_x_forwarded_for;
        proxy_set_header X-Forwarded-Proto $scheme;

        # Increase timeout for LLM calls
        proxy_read_timeout 120s;
        proxy_connect_timeout 120s;
        proxy_send_timeout 120s;

        # Max upload size for resumes
        client_max_body_size 50M;
    }
}
```

```
sudo nginx -t
sudo systemctl enable nginx
sudo systemctl start nginx
```

8. SSL with Certbot (Free)

```
# Install Certbot
sudo yum install certbot python3-certbot-nginx -y

# Get SSL certificate
sudo certbot --nginx -d api.yourdomain.com

# Auto-renewal is configured automatically
```

9. CloudFront for Frontend (Optional)

1. Create a CloudFront distribution pointing to your S3 bucket
2. Set the **default root object** to `index.html`
3. Add a custom error response: 403 → `/index.html` (200) for SPA routing
4. Attach an ACM certificate for `recruit.yourdomain.com`
5. Point Route 53 to the CloudFront distribution

Security Group Configuration

Port	Source	Purpose
22	Your IP	SSH access
80	0.0.0.0/0	HTTP (redirects to HTTPS)
443	0.0.0.0/0	HTTPS
8000	Security Group (self)	Backend (internal only)
5432	EC2 Security Group	RDS PostgreSQL (internal only)

Production Checklist

- [ ] Change `JWT_SECRET` to a strong random value
- [ ] Restrict CORS origins in `main.py` (replace `allow_origins=["*"]` with your domain)
- [ ] Enable RDS automated backups
- [ ] Set up CloudWatch alarms for EC2/RDS
- [ ] Configure S3 bucket policy for public read (frontend only)
- [ ] Store secrets in AWS Secrets Manager instead of `.env`

- [ ] Enable HTTPS on all endpoints
- [ ] Set up a CI/CD pipeline (GitHub Actions → EC2 deploy)
- [ ] Configure log rotation for Gunicorn logs

Alternative: Docker Deployment on ECS

If you prefer containers, create this `Dockerfile` in the project root:

```
FROM python:3.11-slim

WORKDIR /app
COPY requirements.txt .
RUN pip install --no-cache-dir -r requirements.txt gunicorn

COPY app/ ./app/
COPY scripts/ ./scripts/
COPY .env .

EXPOSE 8000

CMD ["gunicorn", "app.main:app", \
     "--workers", "4", \
     "--worker-class", "uvicorn.workers.UvicornWorker", \
     "--bind", "0.0.0.0:8000", \
     "--timeout", "120"]
```

Then deploy to **ECS Fargate** for a fully managed, serverless container setup (~$30-50/month for minimal usage).

License