ATS.ai

AI-Powered Candidate Scoring Engine

Complete Project Report

Developed by **Krishna & Arya**

February 2026

🌐 Live: **https://ats-ai-app.vercel.app**

📁 Source: **https://github.com/AryaJagtap/ATS.ai**

1. Problem Statement

Recruiters and HR teams manually screening resumes is slow, inconsistent, and doesn't scale. A company receiving 100+ resumes per job opening spends hours reading each one, often missing good candidates or applying inconsistent criteria.

Goal: Build an AI-powered ATS that:

- Scores resumes against job descriptions using LLM analysis
- Handles 100+ resumes in minutes, not hours
- Provides actionable insights (best-fit role, missing skills, recommendations)
- Exports professional reports for HR teams

2. Starting Point — The Initial Codebase

The project began with a pre-existing codebase that had a FastAPI backend with basic resume scoring, a Next.js frontend with a basic UI, OpenAI GPT-4o as the primary LLM, pdfplumber for PDF extraction, CSV/XLSX file upload for candidate lists, and basic sequential processing (one resume at a time).

Initial Problems Identified

Problem	Impact
Import errors (genai module)	Backend wouldn't start
Sequential processing	~5-6 seconds per resume, 100 resumes = 10+ minutes
No fallback when OpenAI fails	Rate limits (429 errors) caused complete failures
No single resume upload	Users had to create a spreadsheet even for 1 resume
PDF filename used as candidate name	"resume (1).pdf" shown instead of actual name
No multi-JD support	Only one job description at a time
Noisy terminal warnings	pdfminer font warnings cluttered logs

3. Development Phases — Step by Step

Phase 1: Fixing Critical Errors

Problem: The backend had import errors related to the google-genai module. The import genai syntax was outdated.

Solution: Updated the import to use the correct module path: from google import genai. Fixed the initialization pattern to match the latest google-genai SDK.

What we gained: A working backend that could start without errors. What we lost: Nothing — purely a bug fix.

Phase 2: Performance — Reducing Per-Resume Analysis Time

Each resume took 5-6 seconds to analyze. For 100 resumes, this meant 8-10 minutes of sequential processing.

Time Breakdown Analysis

Step	Time	Bottleneck?
PDF download	~0.5s	No
Text extraction (pdfplumber)	~0.1-0.2s	No
Keyword scoring (TF-IDF)	~0.1s	No
LLM API call	~3-5s	YES — the bottleneck
Total	~5-6s	

Optimizations Applied

- Switched LLM model (gpt-4o → gpt-4o-mini): Slightly less "creative" responses, but for structured JSON scoring output, quality is identical. 2-3x faster response times, significantly cheaper API costs. Clear win — no quality loss for our use case.
- Parallel keyword + LLM scoring: Used ThreadPoolExecutor to run keyword_score and llm_score simultaneously instead of sequentially. Saved ~0.1-0.2s per resume.
- Reduced LLM token overhead: Added max_tokens=500 to limit response length. Set temperature=0 for deterministic results. Compacted the prompt to reduce input token count.
- Reusable LLM client singletons: Created single OpenAI/Gemini client instances instead of re-creating them per request. Eliminated connection setup overhead.
- Pre-imported heavy modules: Moved sklearn and regex imports to module level instead of inside functions. Saved ~0.1s on first call.

Result: Per-resume time dropped from 5-6s → 2-3s ✓

Phase 3: Single Resume Upload

Problem: Users had to create a CSV/XLSX file even to analyze a single resume. Tedious for small tasks.

Solution: Added a toggle in the Upload Candidates section: "Single Resume" (drag & drop a single PDF/DOCX) and "Multiple Resumes" (upload CSV/XLSX or select multiple PDFs).

Files modified: FileUpload.js (new toggle UI), page.js (state management), main.py (new /api/analyze-direct endpoint).

What we gained: Much better UX for quick single-resume checks.

Phase 4: Candidate Name Extraction from Resume

Problem: When a PDF was named generically (e.g., "resume (1).pdf"), the system showed the filename as the candidate name.

Solution: The LLM was already extracting the candidate name from resume content. Updated the logic to use the LLM-extracted name when the filename didn't appear to be a real name.

What we gained: Accurate candidate names from resume content.

Phase 5: Multi-JD (Job Description) Matching

Problem: Companies often have multiple open positions. Users had to run separate analyses for each job description.

Solution: Enabled uploading multiple JD files. The system scores each candidate against ALL job descriptions and selects the best match.

Implementation

- JobDescription.js — Updated to accept multiple file uploads
- main.py — Modified /api/analyze to iterate over multiple JDs per candidate
- ResultsTable.js — Added "Matched Role" column showing which JD was the best fit
- scorer.py — Score function runs against each JD, returns the highest score

Architecture decision: Score against ALL JDs and pick the best match automatically. More useful for HR teams, less manual work.

Phase 6: Project Cleanup

The codebase had accumulated legacy files, unused dependencies, and outdated configs. Removed 14 legacy files and folders, updated requirements.txt with pinned versions, cleaned up .gitignore.

What we gained: Cleaner codebase, smaller deployment size.

Phase 7: Batch Processing Optimization

Problem: 120 resumes took 10+ minutes even with per-resume optimizations.

Evolution of Batch Size

Stage	Batch Size	120 Resumes Time	Issues
Initial	1 (sequential)	~15+ min	Way too slow
v1	5	~12 min	Still slow
v2	8	~10 min	Some rate limits
v3	12	~8 min	Occasional 429s
v4 (Final)	15	~6 min ✓	Retry logic handles 429s

Key decision: How high can we push batch concurrency? Constraint: OpenAI rate limits (RPM = Requests Per Minute). Tier 1 accounts: ~500 RPM for gpt-4o-mini. Solution: Batch size 15 + retry logic with exponential backoff. If a 429 hits, the system waits and retries (up to 2 times) instead of failing.

Retry Logic Flow

- Attempt 1: Call OpenAI → If 429/500 error: wait 1 second
- Attempt 2: Retry OpenAI → If still fails: wait 2 seconds
- Attempt 3: Retry OpenAI → If still fails: fall back to Gemini (same retry pattern)
- If Gemini also fails: fall back to keyword-only scoring

Result: 120 resumes in ~5 min 51 sec ✓ with 0 failures (down from 6 failures).

Phase 8: Error Handling & Terminal Cleanup

Problem: Terminal was flooded with pdfminer font warnings. Also, 6 out of 120 resumes were failing due to rate limits.

Solutions Applied

- pdfminer warnings: Added logging suppression to hide non-critical font warnings
- Rate limit failures: Implemented retry logic with exponential backoff (Phase 7)
- TXT file support: Added .txt as a valid file type for job descriptions

Phase 9: Time Elapsed Counter

Problem: Users had no visibility into how long the analysis was taking.

Solution: Implemented a dual-display timer:

- During processing: Live ticking "⏱ Time elapsed: 2m 15s" below the progress bar
- After completion: Static "⏱ Completed in 5m 51s" badge above the metrics cards

Bug found and fixed: Both timers showed simultaneously during processing. Fixed by gating the "Completed in" badge with a processing state check.

Phase 10: PDF Parser Upgrade — PyMuPDF

Parser Comparison

Parser	Speed	Quality	Memory	Our Choice
pdfplumber (was primary)	Moderate	Excellent	High	❌ Replaced
PyPDF2 (was fallback)	Slow	Good	Low	✓ Kept as fallback
PyMuPDF / fitz	~10x faster	Excellent	Moderate	✓ New primary
langextract (considered)	Unknown	Unknown	Unknown	❌ Not well-known

The user asked about langextract as an alternative. After analysis: (1) PDF extraction is NOT the bottleneck (0.1-0.2s vs 2-3s for LLM), (2) langextract is not a widely-used, well-documented package, (3) PyMuPDF is the industry-standard fastest PDF parser.

Result: Switched to PyMuPDF as primary, PyPDF2 as fallback. Saved ~0.1s per resume. Removed pdfplumber dependency.

Phase 11: Photo Link Extraction from Spreadsheet

Problem: The input spreadsheet had a "Photograph" column with Google Drive photo URLs, but the output showed "Not Found" for all photos.

Root cause: The system was asking the LLM to find photo links inside the resume text. Resumes don't contain photo URL links — those links were in the spreadsheet itself.

Solution: Auto-detect columns named "Photo", "Photograph", "Image", or "Picture" in the input spreadsheet. Pass the URL through to the result, overriding the LLM's "Not Found".

Performance impact: Zero — just reading an existing column value.

Phase 12: Favicon & Branding

Problem: The browser tab showed a generic emoji (🎯) as the favicon, and the project name was too long.

Solutions

- Generated a custom AI/recruitment-themed logo (purple-to-magenta gradient)
- Placed as src/app/icon.png (Next.js App Router auto-serves this)
- Deleted old favicon.ico that was overriding custom icon

Name Selection Process

Category	Names Considered
Short & Punchy	HireIQ, TalentLens, ScoreHire, ResumeAI, HireFlow
Techy / Modern	ATS.ai, Recruitr, MatchIQ, HireNex, TalentQ
Professional	ProHire, HireMetrics, SmartATS, CandidateIQ

Selected: ATS.ai — clean, memorable, domain-style, instantly communicates purpose.

Phase 13: Full Rebranding to ATS.ai

File	Old Value	New Value
layout.js (tab)	AI Recruitment & ATS Platform	ATS.ai
Header.js (title)	AI Recruitment & ATS Platform	ATS.ai
Header.js (subtitle)	Fast, Rate-Limit Safe...	AI-Powered Candidate Scoring Engine
page.js (footer)	AI Recruitment & ATS Platform v2.0	ATS.ai
main.py (FastAPI)	AI Recruitment & ATS Platform	ATS.ai
README.md	AI Recruitment & ATS Platform v2.0	ATS.ai — AI-Powered Recruitment Platform

Phase 14: Production Deployment

Component	Platform	Plan	URL
Frontend	Vercel	Free (Hobby)	ats-ai-app.vercel.app
Backend	Render	Free (Docker)	ats-ai-backend-kx9z.onrender.com

Component	Platform	Plan	URL
Source Code	GitHub	Public	github.com/AryaJagtap/ATS.ai

Deployment Steps Completed

- Added MIT LICENSE file
- Initialized Git repository
- Pushed to GitHub (with Personal Access Token authentication)
- Created Render web service (Docker runtime, Singapore region)
- Set ALLOWED_ORIGINS environment variable for CORS
- Created Vercel project (Root Directory: frontend)
- Set NEXT_PUBLIC_API_URL pointing to Render backend
- Updated CORS to allow the real Vercel URL
- Verified end-to-end functionality

4. Technology Stack — Final

Layer	Technology	Version
Frontend Framework	Next.js	15.x
UI Library	React	19.x
Styling	CSS Variables (custom design system)	—
Backend Framework	FastAPI	0.131.0
Language	Python	3.10+
Primary LLM	OpenAI GPT-4o-mini	—
Fallback LLM	Google Gemini 2.5 Flash	—
Offline Scoring	scikit-learn TF-IDF + cosine similarity	1.8.0
PDF Parser (primary)	PyMuPDF (fitz)	1.27.1
PDF Parser (fallback)	PyPDF2	3.0.1
DOCX Parser	python-docx	1.2.0
Excel Export	openpyxl	3.1.5
Data Processing	pandas	3.0.1
File Download	gdown + requests	5.2.1 / 2.32.5
Frontend Hosting	Vercel	Free

Layer	Technology	Version
Backend Hosting	Render (Docker)	Free

5. Performance Journey

Metric	Start	Phase 2	Phase 7	Final
Per-resume time	5-6s	2-3s	2-3s	2-3s
120 resumes total	15+ min	~12 min	~8 min	5m 51s
Batch size	1	5	12	15
Failure rate	High	Moderate	6/120	0/120
LLM model	gpt-4o	gpt-4o-mini	gpt-4o-mini	gpt-4o-mini
PDF parser	pdfplumber	pdfplumber	pdfplumber	PyMuPDF

6. Key Lessons Learned

What TO Use

Decision	Why
gpt-4o-mini over gpt-4o	2-3x faster, 10x cheaper, identical quality for structured JSON
PyMuPDF over pdfplumber	~10x faster for text extraction, smaller dependency
Batch concurrency (15)	Maximizes throughput without overwhelming API rate limits
Retry with exponential backoff	Handles transient API failures — zero resume failures
SSE (Server-Sent Events)	Real-time progress without WebSocket complexity
ThreadPoolExecutor	Simple, effective parallelism for I/O-bound LLM calls
CSS Variables	Clean dark/light theme toggle without library overhead

What NOT to Use

Decision	Why
gpt-4o for scoring	Overkill — slower and expensive, no quality gain for JSON output

Decision	Why
pdfplumber as primary	Slower than PyMuPDF, heavier dependency (pulls in pdfminer)
langextract	Not well-known, no clear advantage, PDF parsing isn't the bottleneck
Sequential processing	Unacceptable for 100+ resumes — must use batch concurrency
WebSockets for progress	SSE is simpler, sufficient for one-directional progress updates
Hardcoded API keys	Security risk — use env vars or user-provided keys

Key Tradeoffs Made

Tradeoff	Chose	Over	Reason
Speed vs accuracy	gpt-4o-mini (faster)	gpt-4o (slower)	No measurable accuracy loss for scoring
Batch size vs rate limits	15 + retries	Conservative batch of 5	Retries handle occasional 429s
API key model	User-provided keys	Developer pays for all	Sustainable — no cost to developer
Multi-JD scoring	Auto score all JDs	User selects per candidate	More automation, better UX
Photo extraction	Read from spreadsheet	Extract from resume via LLM	Resumes don't contain photo URLs

7. Architecture Flow

User uploads CSV/XLSX + Job Description(s)

Frontend (Next.js / Vercel)

- File upload (drag & drop)
- JD text paste / file upload (multi-file)
- API key input via Settings
- SSE listener for real-time progress
- Timer (elapsed / completed)

↓ **POST /api/analyze (multipart form data)**

Backend (FastAPI / Render)

- Parse CSV/XLSX → extract candidates (auto-detect Name, URL, Photo columns)
- For each batch of 15 candidates (ThreadPoolExecutor):
- a. Download resume (gdown / requests)
- b. Extract text (PyMuPDF primary → PyPDF2 fallback)
- c. Score in parallel: Keyword + TF-IDF scoring (weight: 0.3) and LLM cascade: GPT-4o-mini → Gemini Flash → keyword-only (weight: 0.7)
- d. If multi-JD: select best match across all JDs
- e. Inject photo URL from spreadsheet
- Stream results via SSE (Server-Sent Events)
- Export to styled Excel report (openpyxl)

8. Final Feature List

#	Feature	Status
1	Multi-LLM cascade (GPT → Gemini → Keyword)	✓
2	ATS scoring (0-100, weighted blend)	✓
3	Real-time SSE progress streaming	✓
4	Dark / Light theme toggle	✓
5	Single resume upload	✓
6	Multiple resume upload (CSV/XLSX/PDFs)	✓
7	Multi-JD matching	✓
8	Photo link extraction from spreadsheet	✓
9	Candidate name extraction from resume	✓
10	Elapsed time counter	✓
11	Excel export (color-coded)	✓
12	API key config via UI	✓
13	Retry logic with exponential backoff	✓
14	Batch concurrency (15 parallel)	✓
15	Responsive design (desktop + mobile)	✓

#	Feature	Status
16	Custom favicon and branding	✓
17	Production deployment (Vercel + Render)	✓
18	MIT License	✓

9. Files Modified Summary

File	Changes Made
backend/main.py	Multi-JD endpoints, batch size 15, photo column detection, SSE streaming, branding
backend/utils/scorer.py	gpt-4o-mini, parallel scoring, retry logic, client singletons, compact prompt
backend/utils/extractor.py	PyMuPDF primary, PyPDF2 fallback, TXT support, warning suppression
backend/requirements.txt	Pinned versions, replaced pdfplumber with PyMuPDF
frontend/src/app/page.js	Timer, footer branding
frontend/src/app/layout.js	ATS.ai title, meta description, favicon
frontend/src/components/Header.js	ATS.ai branding
frontend/src/components/FileUpload.js	Single/Multiple resume toggle
frontend/src/components/JobDescription.js	Multi-file JD upload
frontend/src/components/ResultsTable.js	Matched Role column
README.md	Full rewrite with features, benchmarks, deployment guide
LICENSE	MIT License added

10. Conclusion

The ATS.ai project evolved from a basic resume scoring tool with critical errors into a production-grade, high-performance AI recruitment platform. Through systematic optimization — switching LLM models, implementing batch concurrency, adding retry logic, and upgrading PDF parsers — we achieved a 63% reduction in processing time (from 15+ minutes to ~6 minutes for 120 resumes) while simultaneously improving reliability from a ~5% failure rate to zero failures.

The platform is now deployed, branded, and ready for real-world use. Any user can bring their own API keys and start scoring resumes immediately.

— Project by **Krishna & Arya** —