coRecruit Platform Development

Q&A Session with Technical Research & Implementation Guide

Q1: Admin Roles Architecture

Intern Question: Right now, we have two admin types: one for basic technical tasks (System Admin) and one with full access (Full Admin). Do we really need two separate admin roles, or can we combine them into one Admin role with different access levels depending on responsibilities?

Research & Analysis: ✅ **Choose: One Admin role with different permission levels**

Why? Think of it like a smartphone:

- Instead of making separate phones for "basic users" and "power users"
- Smartphone makers create ONE phone with different apps and settings available based on user needs
- Same device, different capabilities

How It Works:

```
🏠 One Admin Role House
├── 🚪 Level 1 Doors (Basic Admin)
│   ├── Manage user accounts
│   ├── View basic reports
│   └── Handle job categories
├── 🚪 Level 2 Doors (Advanced Admin)
│   ├── Everything from Level 1
│   ├── System health monitoring
│   └── AI performance settings
└── 🚪 Level 3 Doors (Full Admin)
    ├── Everything from Level 1 & 2
    ├── Delete important data
    └── Change security settings
```

Benefits for Your Team:

- **Easier to code:** One user interface instead of two
- **Easier to test:** One system instead of two separate systems
- **Easier to grow:** Can add new permission levels easily
- **Less confusion:** Admins know there's just one "Admin" role

Technical Implementation Suggestions:

- **Implementation:** Use bitmask permissions (e.g., Basic=1, Advanced=3, Full=7) for efficient role checks
- **Database Schema:**

sql

```sql
CREATE TABLE users (
  id SERIAL PRIMARY KEY,
  email VARCHAR(255) UNIQUE,
  permission_mask INT  -- Stores combined permissions (e.g., 7 = Full Admin)
);
```

- **Middleware Example (Node.js):**

javascript

```javascript
function checkPermission(req, res, next, requiredPermission) {
  if (req.user.permission_mask & requiredPermission) next();
  else res.status(403).send("Forbidden");
}
```

- **Tools:** Django's built-in Permission model (Python), CASL (JavaScript) for frontend permission checks

---

Q2: AI Test and Interview Features

Intern Questions:

- Do we need a complete AI test environment (like actual coding or MCQ tests with scoring), or is it enough to just generate question sets for internal use?
- During interviews, should the AI give real-time support (like transcripts, scoring, follow-up questions), or just prepare interview kits in advance?

Research & Analysis: ❌ **Do not build real-time AI interview features now** (Why reinvent the wheel?) ✅ **If you really want, just prepare interview kits (before the interview)**

Why?

- Real-time AI means huge RAM usage, expensive infrastructure, and constant debugging
- Prepared kits (e.g., suggested questions, evaluation forms) are faster to build, cheaper, and enough for most interviews

Current Core Problems:

- Volume Overload (200-500 resumes per job)
- Time Management (80% time wasted on unqualified candidates)
- Bias and Inconsistency Issues
- Interview Scheduling Nightmare
- Skills Mismatch Problem (60% wrong recommendations)

AI Test Environment Problems:

- Adds massive complexity to your system
- Requires different AI models for different job types
- Needs secure test environment (cheating prevention)
- Requires integration with coding platforms
- Needs real-time proctoring systems

Future Integration:

- 🔗 API connections to HackerRank/Codility
- 📊 Import test results into your platform
- 🎯 Combine matching scores + test scores
- 📈 Provide unified candidate rankings

Technical Implementation Suggestions:

- **For Resume Parsing:** Use spaCy (local NLP) + Nomic Embed (lightweight embeddings) for semantic matching
- **Quantize models:** 4-bit precision (e.g., GGUF format) to fit in 16GB RAM
- **Interview Kits:** Generate PDFs with LaTeX (via pandoc) or WeasyPrint (HTML-to-PDF)
- **Templates:** Store in Markdown for version control
- **Future-Proofing:** Design plugin architecture to later integrate HackerRank API (e.g., REST webhooks)

Q3: Job Category Management Complexity

Intern Question: The admin dashboard includes "Job Category Management." We need to know how complex this feature is. Does this simply mean managing basic labels like Developer, Designer, etc., or are we handling detailed job frameworks, skill tags, and industry taxonomies?

Research & Analysis: ✅ **Go with a detailed job category system**

Think beyond just "Developer" — the AI and recruiters need more data to make better matches.

Structure should include:

- **Industry** (e.g., Tech, Finance, Healthcare)
- **Job Role** (e.g., Backend Developer, UX Designer)
- **Experience Level** (e.g., Entry, Mid, Senior)
- **Skills Tags** (e.g., Python, React, Communication)
- **Salary Ranges** (optional)

This improves AI matching, filters, and overall job-candidate fit.

Technical Implementation Suggestions:

- **Database:** Use PostgreSQL JSONB for flexible skill tags:

```sql
CREATE TABLE job_categories (
  id SERIAL PRIMARY KEY,
  industry VARCHAR(50),
  role VARCHAR(50),
  skills JSONB  -- {"required": ["Python"], "optional": ["AWS"]}
);
```

- **UI/UX:** React Select for multi-tag input
- **Scaling:** Elasticsearch for fast job-candidate matching (if scaling later)

Q4: Interview Process Flow

Intern Question: To design proper candidate tracking and status flows, we need to understand the full hiring process. How many interview rounds will each candidate typically go through?

Research & Analysis: Keep it simple and start with **2 interview rounds:**

1. **Initial Screening** (optional) – AI/Recruiter shortlisting
2. **Final Interview** (required) – HR Manager + client/hiring team

Later, you can support 3+ rounds if needed (e.g., Technical Round, Culture Fit Round, etc.), but for MVP, assume max 2 rounds per candidate.

Technical Implementation Suggestions:

- **Database Schema:**

```sql
CREATE TABLE interviews (
  id SERIAL PRIMARY KEY,
  candidate_id INT REFERENCES users(id),
  round INT CHECK (round IN (1, 2)),
  status VARCHAR(20) CHECK (status IN ('scheduled', 'completed', 'rejected'))
);
```

- **API Endpoint:**

```rest
PATCH /api/interviews/{id}
Body: {"status": "completed"}
```

- **Frontend:** Use React DnD for drag-and-drop Kanban boards

---

Q5: User Roles Confirmation

Intern Question: We are assuming that coRecruit will support four user roles. Can we confirm these four roles are final? Should we design Figma screens for them?

Proposed Roles:

- HR Manager
- Recruiter/Screener
- System Admin
- Full Admin

Research & Analysis: ✅ Yes, the following four roles are final for MVP. Start Figma designs accordingly:

- **HR Manager** – Final decisions, full dashboard view
- **Recruiter/Screener** – Resume handling, shortlisting
- **System Admin** – User & job category management
- **Full Admin** – Has all access, config, monitoring, data

These roles cover all basic operations and future scaling.

Technical Implementation Suggestions:

- **JWT Flow:** Admin creates user with POST /api/users (generates temp password)
- **Security:** Rate-limit password reset endpoints (express-rate-limit)
- **Password Hashing:** Use Argon2id for password hashing (CPU/memory-hard)
- **System emails:** Temp link with JWT one-time token (expires in 24h)
- **Password Reset:** User resets password via PATCH /api/auth/reset-password

---

Q6: Admin Features Design Scope

Intern Question: To keep our initial Figma designs focused and manageable, we suggest limiting admin screens to these four key areas. Please confirm if this list looks correct and what should be shown in each one:

Proposed Areas:

- System Health Monitor: Uptime, system performance, alerts
- AI Performance Metrics: How well the AI is parsing, matching, and scoring
- User Account Management: Add/edit/delete users, assign roles

- Job Category Management: Add/edit job categories used in postings

Research & Analysis: ✅ **Confirmed. Here's what each should show:**

- **System Health Monitor** → Shows uptime, performance, and error alerts
- **AI Performance Metrics** → Displays resume parsing accuracy, matching scores, etc.
- **User Account Management** → Add/edit/remove users, reset passwords, assign roles
- **Job Category Management** → Manage job categories, skills, and filters used in job postings

Technical Implementation Suggestions:

- **Backend:** Expose Prometheus metrics (e.g., system_uptime, ai_accuracy)

```python
from prometheus_client import Gauge
AI_ACCURACY = Gauge('ai_matching_accuracy', 'Accuracy of resume-JD matches')
```

- **Dashboard:** Grafana (open-source) for visualizing metrics
- **Error Tracking:** Sentry for error tracking
- **Memory Management:** Run Redis for caching (reduce DB hits)

Q7: Recruiter Role Design

Intern Question: The recruiter's job seems to include many AI features like parsing resumes, screening candidates, and matching profiles. Since these tasks are handled by AI, do we still need separate Figma screens for the Recruiter, or should we merge them into the same interface used by the HR Manager (with limited access)?

Research & Analysis: ✅ Use one shared interface for both HR Manager and Recruiter, with different views based on permissions. Just hide advanced controls for Recruiters.

Section	HR View	Recruiter View
Dashboard	See all jobs & candidates	See only assigned candidates
AI Tools	Adjust AI settings, give feedback	View AI matches, rate resumes
Reports	See team performance	See personal stats only

Technical Implementation Suggestions:

- **Frontend:** Use role-based component rendering (React/Vue):

```jsx
{user.role === 'hr_manager' && <AdvancedFilters />}
```

- **API:** Add ?fields=basic or ?fields=detailed query params to control response size
- **UI Framework:** React + Material-UI for reusable permission-aware components

---

Q8: Employer Account Setup and Access

Intern Question: Since company employers (like HR Managers and Recruiters) will use organization-specific emails, we're wondering how their accounts should be created and managed. Do employers need a "Register," "Forgot Password," and "Reset Password" flow, or should their email and password be created by the system and assigned to them? If so, should this be handled by the Admin role?

Research & Analysis: ✅ Admins should create employer accounts manually. No self-registration or reset password screens needed.

Why?

- Ensures only real company users get access
- Avoids spam/fake accounts
- Gives full control over role assignment

Flow:

1. Admin enters employer's company email (e.g., hr@company.com)
2. System generates temporary password
3. Employer gets a welcome email
4. On first login, employer resets password

No need for:

- ❌ "Register" button
- ❌ "Forgot Password" link

Admins handle account creation and recovery. It's safer and more professional.

Technical Implementation Suggestions:

- **Data Security:** Encrypt resumes at rest (AES-256) and in transit (TLS 1.3)
- **API Security:** Rate-limiting (express-rate-limit), CSRF tokens
- **Auth Security:** JWT expiration (1h), refresh tokens
- **Local Storage:** Use SQLite for lightweight staging environments

---

Dataset Requirement & RAG Implementation

200 CVs and 200 Resumes required for the RAG system.

Technical Implementation Suggestions:

- **Tooling:** Label Studio (open-source) to annotate resumes
- **Alternative:** spaCy Prodigy (if budget allows) for active learning
- **Preprocessing:** Use Python faker to anonymize sensitive data
- **Storage:** Store embeddings in FAISS (local similarity search)
- **AI Models:** Llama.cpp (4-bit quantized) for local LLM tasks
- **Speech Processing:** Whisper.cpp for speech-to-text (offline)

Tech Stack Recommendations

your approach

Component	Tool	Why?
Backend	Django/Node.js	RBAC support, scalable
Database	PostgreSQL	JSONB, ACID compliance
NLP	spaCy + Nomic Embed	Lightweight, local
Auth	JWT + Argon2id	Stateless, secure hashing
Monitoring	Prometheus + Grafana	Open-source, customizable
Frontend	React + Material-UI	Reusable permission-aware components

---

Development Timeline: 8-Week Sprint Plan

AI Developer Team Distribution (3 Developers)

Phase 1: Core Features (Weeks 1-6)

AI Developer 1: Resume Processing & Matching Engine

Weeks 1-2:

- Set up spaCy + Nomic Embed environment
- Build resume parsing pipeline (PDF/DOC to structured data)
- Create basic skill extraction module

Weeks 3-4:

- Implement job-resume matching algorithm
- Build semantic similarity scoring system
- Create candidate ranking system

Weeks 5-6:

- Integration with database (PostgreSQL + JSONB)
- API endpoints for resume processing
- Basic testing and optimization

AI Developer 2: Job Category Management & Classification

Weeks 1-2:

- Design job taxonomy database schema
- Build job category classification system
- Create skill tagging automation

Weeks 3-4:

- Implement industry-role-level mapping
- Build skill recommendation engine

- Create job description analysis tools

Weeks 5-6:

- Integration with admin panel APIs
- Multi-tag search and filtering
- Performance optimization

AI Developer 3: Interview Kits & Assessment Tools

Weeks 1-2:

- Research and design interview question templates
- Build question generation system based on job requirements
- Create evaluation criteria frameworks

Weeks 3-4:

- Implement PDF generation for interview kits
- Build assessment scoring algorithms
- Create candidate evaluation templates

Weeks 5-6:

- Integration with user roles and permissions
- API development for interview management
- Testing and validation

Phase 2: Accuracy Enhancement & Advanced Features (Weeks 7-8)

All Developers - Collaborative Phase:

Week 7: Model Fine-tuning & Optimization

- **Developer 1:** Fine-tune matching accuracy with collected data
- **Developer 2:** Optimize job categorization with real job postings
- **Developer 3:** Enhance interview question quality and relevance

Week 8: Integration & Performance Enhancement

- **Collaborative:** RAG system implementation with 200 CV dataset
- **Collaborative:** Cross-component testing and debugging
- **Collaborative:** Performance monitoring setup (Prometheus metrics)
- **Collaborative:** Final optimization and deployment preparation

Deliverables by Phase:

Phase 1 Deliverables (Week 6):

- ✅ Resume parsing and matching system
- ✅ Job category management with skill tagging
- ✅ Interview kit generation system
- ✅ Core API endpoints
- ✅ Basic admin panel integration

Phase 2 Deliverables (Week 8):

- ✅ Enhanced matching accuracy (target: 85%+)
- ✅ RAG-powered intelligent recommendations
- ✅ Advanced analytics and reporting

- ✅ Performance monitoring dashboard
- ✅ Production-ready deployment

Key Success Metrics:

- **Resume Parsing Accuracy:** 90%+
- **Job-Candidate Matching Accuracy:** 85%+
- **System Response Time:** <2 seconds
- **Memory Usage:** Within 16GB limit
- **API Reliability:** 99.5% uptime

Risk Mitigation:

- **Weekly sync meetings** between all developers
- **Shared code repository** with branch protection
- **Containerized development** environment (Docker)
- **Automated testing** pipeline
- **Performance benchmarking** at each milestone

Key Takeaways:

- Avoid over-engineering (e.g., real-time AI → pre-generated kits)
- Leverage open-source/local tools to stay within 16GB RAM
- Design for extensibility (e.g., plugin architecture for future HackerRank integration)
- Prioritize security (encryption, rate-limiting, audits)
- Focus on core MVP features first, then enhance accuracy and performance

This document was structured and formatted with the assistance of generative AI tools to ensure clarity, comprehensive technical coverage, and professional presentation standards.