

Architectural Blueprint and Implementation Strategy for a Bespoke Applicant Tracking System in Small-Scale Organizational Contexts

1. Executive Summary

The modernization of Human Resources (HR) infrastructure has traditionally been a privilege reserved for large enterprises capable of sustaining significant capital expenditure and dedicated engineering teams. However, the operational exigencies of small-to-medium organizations—specifically regarding process integrity, data synchronization, and talent acquisition efficiency—are no less acute. The primary research material provided for this analysis, the "HR System Development Plan," outlines a rigorous technical strategy for a bespoke Applicant Tracking System (ATS).¹ This system is architected to automate the ingestion of resumes via email, parse unstructured candidate data into a relational schema, and facilitate high-fidelity synchronization between internal HR administrators and external client stakeholders.

This research report presents an exhaustive analysis of implementing this specific HR system, tailored expressly for a small organization with limited user count but high data integrity requirements. The central thesis of this analysis is that a **Modular Monolith architecture**, supported by **PostgreSQL's advanced native features** (Row-Level Security, Full-Text Search) and a **Python-based Event-Driven** ingestion layer, offers the optimal balance of performance, security, and maintainability. This approach circumvents the operational overhead of microservices while delivering enterprise-grade isolation and scalability.

Furthermore, the report synthesizes a granular operational roadmap for a single developer implementation. It argues that by leveraging high-abstraction tools such as **Refine.dev** for the frontend and **SendGrid Inbound Parse** for ingestion, a single engineer can successfully deploy and maintain a system that would traditionally require a multi-disciplinary team. The analysis delves into the physics of email ingestion, the mathematics of fuzzy logic for candidate deduplication, and the rigid state management required to enforce the "Leaver" exclusion loop, providing a comprehensive blueprint for modernizing HR infrastructure in resource-constrained environments.

2. The Strategic Context of HR Digital Transformation

in Small Organizations

The digital transformation of HR mandates a paradigm shift from disparate, manual communication channels to unified, data-driven platforms.¹ In small organizations, recruitment processes are often fragmented across email threads, spreadsheets, and shared drives. This fragmentation leads to data desynchronization, loss of candidate history, and significant compliance risks, particularly regarding the "Right to be Forgotten" and the tracking of former employees who may be ineligible for rehire.

2.1 The "Build vs. Buy" Dilemma for Niche Requirements

Small organizations typically face a choice between purchasing off-the-shelf SaaS ATS solutions or building custom software. Commercial tools often suffer from feature bloat, high per-user licensing costs, and a lack of flexibility regarding specific business rules—such as the strict "Leaver" feedback loop described in the research material.¹

The proposed custom solution addresses specific complex business logic that commercial tools rarely support out-of-the-box:

1. **Automated Ingestion:** The direct conversion of emails into database records without manual data entry.
2. **Dual-Interface Model:** A high-privilege view for HR and a restricted, friction-free view for external Clients.
3. **Closed Feedback Loop:** The systematic flagging of terminated employees to prevent future surfacing in recruitment searches.

By building a bespoke Modular Monolith, the organization retains full data sovereignty and can enforce these specific workflows without the recurring costs and limitations of SaaS platforms.

3. Architectural Paradigm: The Modular Monolith

The foundational decision in any software engineering project is the architectural pattern. For an ATS aiming to serve a small organization with high reliability, the industry standard often erroneously leans toward microservices. However, the research material explicitly and correctly advocates for a **Modular Monolith**.¹

3.1 The Microservices Fallacy in Small-Scale Contexts

In the contemporary software landscape, microservices are frequently touted as the solution to scalability. This architecture decomposes an application into loosely coupled services, each with its own database and deployment lifecycle. While this offers fault isolation for massive distributed teams, it introduces catastrophic complexity for a small organization or a single

developer.

The "Microservices Tax" involves substantial overhead:

- **Network Latency:** Internal method calls are replaced by network requests, introducing serialization/deserialization latency.
- **Distributed Consistency:** Maintaining data integrity across multiple databases requires complex transaction patterns like Sagas or Two-Phase Commits (2PC), which are prone to failure and difficult to debug.
- **Operational Complexity:** Instead of monitoring a single application, the developer must manage a fleet of services, a service mesh, an API gateway, and distributed tracing infrastructure.

For an ATS where the business logic demands strict referential integrity between a Candidate profile, their Application status, and the Client feedback, a distributed architecture poses significant risks of data desynchronization. The research material cites Amazon Prime Video's strategic reversion from microservices to a monolith, which reduced infrastructure costs by over 90%, as definitive evidence of the efficiency of monolithic architectures for high-throughput, tightly coupled applications.¹

3.2 Defining the Modular Monolith

The proposed Modular Monolith is not a "Spaghetti Monolith"—a codebase where dependencies are entangled and unstructured. Rather, it is a design pattern that structures the application into distinct, encapsulated domains within a single deployable unit.

Domain Boundaries:

1. **Ingestion Domain:** Responsible for the entry of raw data, specifically resume parsing and file handling.
2. **Candidate Management Domain:** Manages the core profile data, skills taxonomy, and employment history.
3. **Client Portal Domain:** Manages external access, job requirements, and the feedback loop.
4. **Notification Domain:** Handles transactional emails and real-time Server-Sent Events (SSE).

Operational Advantage:

For a single developer, this approach allows for code that is structured like microservices—with clear boundaries and defined interfaces—but executes as a monolith. Data sharing occurs via efficient in-memory function calls or database JOINs rather than network-bound API calls. This eliminates the class of errors related to network failures between services, significantly reducing the debugging burden and infrastructure costs.

3.3 Event-Driven Architecture (EDA) for Asynchronous Ingestion

While the core application logic (CRUD operations, status updates) operates synchronously, the ingestion of resumes via email is inherently asynchronous and burst-prone. To prevent blocking the main application threads during heavy recruitment drives, the system integrates an Event-Driven Architecture component at the ingress layer.¹

The Hybrid Approach:

This architecture utilizes message queues (Redis backed by Celery) to decouple the reception of an email from its processing.

- **Ingress:** When an email arrives via the webhook, the system simply acknowledges receipt and places the payload onto a queue.
- **Processing:** Specialized worker processes consume these messages to perform CPU-intensive tasks such as Optical Character Recognition (OCR) and Natural Language Processing (NLP).

This separation of concerns ensures that the user-facing interfaces remain responsive even if the backend is processing hundreds of incoming resumes simultaneously, a critical requirement for maintaining user experience during peak loads.

4. The Ingestion Subsystem: Physics of Automated Entry

The foundational requirement of the system is the automated conversion of unstructured resume files (PDF, DOCX) attached to emails into structured database records containing Name, Employment History, Date of Birth (DOB), Skills, and Cost to Company (CTC).¹

4.1 Email Ingestion Protocol: Webhooks vs. IMAP

There are two primary methodologies for programmatically receiving emails: polling a mail server via IMAP (Internet Message Access Protocol) or utilizing a Push-based Webhook from an email service provider.

Comparative Analysis of Ingestion Protocols:

Feature	IMAP Polling	Inbound Parse Webhook (SendGrid)
Mechanism	The app connects to the mail server every X seconds to check for new mail.	The provider accepts the mail and POSTS data to the app URL immediately.

Latency	High (dependent on poll interval).	Near Real-Time.
Complexity	High. Requires managing socket connections, SSL, and manual MIME parsing.	Low. Provider handles MIME decoding and sends parsed JSON/Multipart data.
Scalability	Limited by single-thread polling constraints.	Highly Scalable (Serverless/Load Balanced HTTP endpoints).
Reliability	Prone to connection timeouts and "unread" status sync issues.	Robust retry mechanisms built into the provider.

Strategic Decision:

The system utilizes SendGrid Inbound Parse Webhook.¹ This configuration involves pointing the MX records of a subdomain (e.g., parse.recruiting-system.com) to mx.sendgrid.net. SendGrid processes all incoming mail, strips the headers, extracts the body and attachments, and transmits a multipart/form-data payload to the backend API. This eliminates the need for the application to maintain persistent connections to a mail server, significantly reducing resource consumption.

4.2 Intelligent Parsing Logic and Tool Selection

Upon receipt of the resume file, the system must extract specific entities. The requirement explicitly lists Name, Employment, DOB, Skills, and CTC.

4.2.1 Text Extraction and OCR

Resumes arrive in various formats. The first step is to convert binary files into plain text.

- **PDF Processing:** The system utilizes **pdfminer.six**, a robust Python library for extracting text from PDF documents.¹ Unlike simpler libraries, pdfminer.six analyzes the layout to distinguish between columns. This is vital for resumes that often feature sidebars for skills; a standard linear extraction would merge the text from the sidebar with the main body, rendering it nonsensical.
- **OCR Fallback:** A significant minority of resumes are image-based (scanned). If pdfminer returns a low character count, the pipeline falls back to **Tesseract OCR** (via pytesseract) to optically recognize text from the page images.

4.2.2 Natural Language Processing (NLP) Strategy

Once text is obtained, the system employs Named Entity Recognition (NER) to identify the required fields. The report recommends a hybrid approach starting with Open Source tools to minimize costs while allowing for future scalability.¹

Tool Selection:

- **spaCy & pyresparser:** The system leverages spaCy for its industrial-strength NLP capabilities and pyresparser for specific resume-parsing logic.
- **Skills Extraction:** pyresparser extracts skills by matching tokens against a comprehensive CSV database of technical terms.
- **Employment History:** The parser looks for date ranges (using regex patterns like (19|20)\d{2}) associated with ORG entities identified by spaCy models (en_core_web_sm or trf for accuracy).¹
- **CTC (Cost to Company):** This specific field is often non-standard (e.g., "12 LPA", "\$80,000", "50k/month"). Standard parsers often miss this. The system implements a custom Regular Expression Post-Processor specifically designed to identify currency symbols and numerical patterns associated with keywords like "CTC", "Salary", "Remuneration", or "Package".

4.3 The "Do Not Hire" Exclusion Mechanism

A critical business rule is the automated exclusion of former employees ("leavers"). This requires a mechanism to identify individuals regardless of minor variations in their resume details (e.g., "Bob Smith" vs. "Robert Smith").

Implementation Logic:

1. **Identity Hashing:** Upon parsing, the system generates a "Candidate Hash" based on normalized immutable identifiers: SHA256(normalize(email) + normalize(mobile_number)).¹
2. **The Exclusion Check:** Before a candidate is indexed for search, this hash is checked against the database. If the status returned is LEFT_COMPANY or BLACKLISTED, the incoming resume is flagged.
3. **Fuzzy Deduplication:** To prevent circumvention (e.g., applying with a new email), the system utilizes PostgreSQL's pg_trgm (Trigram) extension. This allows for similarity matching on the full_name and date_of_birth columns. If a high-confidence match (>90% similarity) is found against a "Left" candidate, the new profile is flagged for manual review.¹

5. Database Schema and Advanced Data Modeling

The database is the backbone of the system, enforcing data integrity and enabling the

complex search and security requirements. **PostgreSQL (v15+)** is the chosen technology due to its ability to handle structured relational data alongside semi-structured data (JSONB) and its advanced text search capabilities.¹

5.1 Entity-Relationship Model (ERD)

The schema is designed to separate global candidate data from specific client applications, enabling a many-to-many relationship where a candidate can be applied to multiple jobs over time.

5.1.1 Core Tables

Table: candidates

The central repository of talent.

- id: UUID (Primary Key).
- full_name: VARCHAR.
- email: VARCHAR (Unique, Indexed).
- phone: VARCHAR.
- dob: DATE (Required for verification).
- ctc_current: NUMERIC (Parsed value).
- skills: JSONB (Array of strings, e.g., `[]`).
- resume_s3_url: VARCHAR (Path to object storage).
- search_vector: TSVECTOR (Pre-computed field for FTS).
- is_blacklisted: BOOLEAN (Default FALSE) - The Core "Leaver" Flag.
- metadata: JSONB (Stores the raw output from pyresparser).

Table: employment_history

Stores the one-to-many relationship of a candidate's career.

- id: UUID.
- candidate_id: UUID (Foreign Key).
- organization: VARCHAR.
- designation: VARCHAR.
- start_date: DATE.
- end_date: DATE.

Table: clients

Represents the external entities (Clients) accessing the system.

- id: UUID.
- company_name: VARCHAR.
- tenant_id: UUID (Critical for Security/RLS).
- contact_email: VARCHAR.

Table: applications

The pivot table managing the workflow.

- id: UUID.
- candidate_id: UUID.
- client_id: UUID.
- job_role: VARCHAR.
- status: ENUM (SUBMITTED, CLIENT REVIEW, INTERVIEW_SCHEDULED, SELECTED, JOINED, LEFT_COMPANY).
- client_feedback: TEXT.
- interview_date: TIMESTAMP WITH TIME ZONE.
- joining_date: DATE.
- created_at: TIMESTAMP.

5.2 Native Full-Text Search vs. External Engines

The HR interface requires searching candidates by "Skill" and "City." While engines like ElasticSearch are powerful, they introduce the "dual source of truth" problem, where the database and the search index can drift out of sync.¹

Strategic Choice: PostgreSQL Native Full-Text Search.

PostgreSQL provides enterprise-grade search capabilities sufficient for databases with tens of millions of records.

- **Implementation:** A generated column named search_vector is added to the candidates table. This column automatically concatenates full_name, skills, current_location (extracted from resume), and employment_history into a tsvector.
- **Indexing:** A GIN (Generalized Inverted Index) is applied to search_vector.
- **Ranking:** The search query utilizes ts_rank to score results. For instance, a candidate with "Python" in both their current job title and skills list will rank higher than one with "Python" only in an old project description.¹

Filtering Logic:

The query explicitly handles the exclusion requirement:

SQL

```
SELECT *, ts_rank(search_vector, query) as rank
FROM candidates, to_tsquery('english', 'Python & London') query
WHERE search_vector @@ query
AND is_blacklisted = FALSE -- Satisfies the "Exclude Leavers" requirement
ORDER BY rank DESC;
```

6. Security Architecture: Multi-Tenancy and Row-Level Security (RLS)

A paramount requirement is that Clients must view profiles sent to them but never access the global candidate pool or other Clients' data. Relying solely on application-level filtering (e.g., WHERE client_id =...) is prone to developer error, potentially leading to critical data leaks.

6.1 Database-Level Isolation with RLS

PostgreSQL's Row-Level Security policies enforce access control at the database engine level.¹

Implementation Strategy:

1. **Policy Definition:** A policy is defined on the applications and candidates tables.

SQL

```
CREATE POLICY client_isolation_policy ON applications
FOR ALL
USING (client_id = current_setting('app.current_client_id')::uuid);
```

2. **Session Context:** When the API processes a request from a Client, it authenticates the user and sets a session variable (e.g., SET app.current_client_id = 'client_uuid_123') before executing the query.
3. **Enforcement:** Even if the SQL query attempts to SELECT * FROM applications, the database will automatically filter the rows, returning only those belonging to that specific client ID. This guarantees that a Client can never accidentally view unauthorized data, acting as a failsafe against application logic errors.¹

7. Backend Application Logic and State Management

The backend serves as the orchestrator of the recruitment workflow, managing the complex state transitions required by the interaction between HR and Clients.

7.1 Finite State Machine (FSM) for Candidate Lifecycle

The lifecycle of a candidate involves rigid transitions triggered by different actors (HR vs. Client). This is best modeled using a **Finite State Machine** pattern to prevent invalid states (e.g., setting a candidate to "Left" before they have "Joined").¹

State Definitions:

- POOL: Candidate parsed and available in DB.
- SUBMITTED: Profile sent by HR to Client.
- UNDER REVIEW: Client has opened/viewed the profile.

- INTERVIEW_SCHEDULED: Client has set an interview date.
- OFFER_EXTENDED: Client has selected the candidate.
- JOINED: Candidate has accepted and started. (Sync point with DB).
- LEFT_COMPANY: Candidate has terminated employment. (Trigger point for Blacklisting).
- REJECTED: Application closed.

Transition Logic & Side Effects:

- **Transition:** SUBMITTED -> INTERVIEW_SCHEDULED
 - **Actor:** Client.
 - **Input:** interview_date, interview_mode.
 - **Side Effect:** Trigger SSE notification to HR; Send email confirmation to Candidate.
- **Transition:** JOINED -> LEFT_COMPANY
 - **Actor:** Client.
 - **Input:** exit_date, reason.
 - **Side Effect (Critical):** Execute a database transaction that updates the Candidate's master record, setting is_blacklisted = TRUE. This fulfills the requirement to exclude them from future searches.¹

7.2 Technology Stack: Python (FastAPI)

Python is the optimal choice for the backend due to its dominance in the NLP/Data Science ecosystem. Using Python for both the API (**FastAPI**) and the Parsing Service (**spaCy**) allows for code sharing (e.g., shared data models) and unified dependency management.

- **Framework: FastAPI.** It provides high performance (comparable to NodeJS) due to its asynchronous nature (Starlette), which is vital for handling concurrent I/O operations like database queries and SSE streams.¹
- **ORM: SQLAlchemy (Async).** It provides the necessary abstraction to interact with PostgreSQL while supporting the asynchronous drivers required by FastAPI.
- **Schema Validation: Pydantic.** Used for strict validation of incoming data from the Resume Parser and Client inputs.

8. Frontend Architecture: HR and Client Portals

The system requires two distinct user interfaces with vastly different User Experience (UX) requirements: a high-density, power-user dashboard for HR, and a simple, streamlined portal for Clients.

8.1 HR Interface: The Power Dashboard

HR users need to manage thousands of records, perform complex filtering, and view documents.

Framework Selection: Refine.dev (React)

Building a data-heavy admin panel from scratch is resource-intensive. Refine.dev is a React-based meta-framework specifically engineered for internal tools and CRUD applications.¹

- **Headless Architecture:** Unlike template-based admin panels (e.g., React-Admin), Refine is headless. It handles the logic—routing, networking, state management, authentication—but leaves the UI rendering to component libraries.
- **UI Library:** The system integrates **Ant Design** with Refine. Ant Design provides sophisticated table components with built-in sorting, filtering, and pagination, which are essential for the HR "Search Candidates" requirement.
- **Data Hooks:** Refine's useTable hook connects directly to the API. It automatically manages the URL query parameters for search filters (e.g., ?skills=python&city=new_york), ensuring that search states are shareable and persistent.

Key Features Implementation:

- **Search Interface:** A faceted search bar allowing HR to filter by "Skills" (using the JSONB column) and "City." A toggle switch "Include Ex-Employees" is strictly disabled/hidden by default to comply with the exclusion rule.
- **Document Preview:** A split-screen view where the left panel shows the parsed data form (editable by HR for corrections) and the right panel renders the original PDF using react-pdf, fetching the secure URL from S3.

8.2 Client Interface: The Frictionless Portal

Clients are external users who should not be burdened with complex login procedures.

Authentication Strategy: Passwordless "Magic Links"

To maximize adoption and security, the Client Portal utilizes a Magic Link authentication flow.¹

1. **Trigger:** HR selects a candidate and clicks "Send to Client."
2. **Token Generation:** The backend generates a cryptographically signed JWT (JSON Web Token). This token contains the client_id (for RLS) and the specific application_ids being shared.
3. **Access:** The Client receives an email: "New Profiles for Review. Click here to view."
4. **Session:** Clicking the link authenticates the Client for a fixed session duration (e.g., 24 hours), granting access strictly to the shared profiles.

Feedback & Status Management UI:

- **Kanban View:** Clients view candidates as cards moving through columns (To Review, Interview, Selected).
- **Action Modals:**
 - **Select for Interview:** Opens a calendar input to set the interview_date.
 - **Reject:** Opens a form requiring a "Reason for Rejection" (Dropdown + Text Area).
 - **Terminate (Leaver):** Available only in the "Joined" tab. This action triggers the "Left"

Company" state transition.

9. Real-Time Synchronization via Server-Sent Events (SSE)

The prompt requires that actions taken by the Client (e.g., selecting a candidate) "sync status with HR." Traditional polling (checking the server every 30 seconds) is inefficient and creates data lag.

9.1 Technology Selection: SSE vs. WebSockets

WebSockets provide full-duplex communication but introduce complexity regarding connection state management, firewall traversal, and load balancing (requiring sticky sessions).

Server-Sent Events (SSE) provide a simplex (Server-to-Client) channel over standard HTTP. Since the primary requirement is notifying the HR dashboard of updates occurring on the server (initiated by the Client), SSE is the superior architectural choice.¹

- **Efficiency:** SSE uses a single long-lived HTTP connection.
- **Reconnection:** The EventSource API in browsers handles auto-reconnection natively.
- **Protocol:** It operates over standard HTTP/2, making it compatible with existing load balancers and firewalls without special configuration.

9.2 Implementation Strategy

1. **Event Stream Endpoint:** The API exposes an endpoint /api/stream protected by HR authentication.
2. **Frontend Connection:** The HR Dashboard (Refine app) initializes an EventSource connection on load.
3. **Backend Event Loop:**
 - When a Client updates an application (e.g., PATCH /applications/123 setting status to INTERVIEW), the backend commits the change to PostgreSQL.
 - Simultaneously, the backend publishes an event message to a **Redis Channel** (Pub/Sub).
 - The API process listening to this channel pushes the event to the connected HR clients via the SSE stream.
 - **Payload:** {"type": "STATUS_UPDATE", "id": "123", "new_status": "INTERVIEW", "client": "Acme Corp"}.
4. **UI Reaction:** The Refine app receives the event and uses React Query's invalidateQueries method to automatically refresh the data grid, instantly reflecting the new status.¹

10. Infrastructure and DevOps Plan

To ensure scalability and maintainability, the system will be deployed using containerization best practices.

10.1 Containerization Strategy

- **Docker:** All services (API, Worker, Database, Redis) are containerized.
- **Orchestration:** For the initial deployment, **Docker Compose** is sufficient. For production scaling, the system migrates to Kubernetes (K8s) or a managed service like AWS ECS.

10.2 Service Breakdown

1. app-api: FastAPI service handling REST requests and SSE streams.
2. app-worker: Python Celery worker handling Resume Parsing (CPU heavy) and Email Processing.
3. db: PostgreSQL 15 container with persistent volume.
4. redis: Message broker for Celery and Pub/Sub channel for SSE.
5. web-hr: React/Refine application (served via Nginx).
6. web-client: React application for the Client Portal.

10.3 CI/CD Pipeline

A CI/CD pipeline (using GitHub Actions) ensures code quality.

- **Validation:** On commit, run pytest for backend logic (specifically testing the State Machine transitions and RLS policies) and ESLint for frontend code.
- **Build:** Build Docker images and push to a container registry (ECR/Docker Hub).
- **Deploy:** Rolling update to the production environment to ensure zero downtime.

11. Single Developer Implementation Roadmap

The adaptation of this enterprise-grade system for a small organization with a single developer requires a disciplined, phased approach. The key is to leverage high-leverage tools that abstract complexity.

Phase 1: The Core Foundation (Weeks 1-2)

Goal: Establish the Modular Monolith structure and Database.

- **Task 1 (Project Setup):** Initialize a Monorepo. Configure Docker Compose with PostgreSQL and Redis.
- **Task 2 (Database Modeling):** Define SQLAlchemy models. Implement the core Candidate and Application tables with the JSONB skills column.

- **Task 3 (API Scaffold):** Set up FastAPI. Create the basic CRUD endpoints. Implement the **RLS Policy** in SQL migrations immediately to bake security in from the start.

Phase 2: The Ingestion Engine (Weeks 3-4)

Goal: Automate data entry.

- **Task 1 (Email Pipeline):** Configure SendGrid Inbound Parse. Write the webhook handler in FastAPI that pushes raw JSON to Redis.
- **Task 2 (Worker Setup):** Implement the Celery worker. Write the pdfminer wrapper to extract text from PDFs.
- **Task 3 (Parser Logic):** Implement the regex patterns for Contact Info and CTC. Integrate pyresparser for basic skill extraction.

Phase 3: The HR Workspace (Weeks 5-6)

Goal: Enable HR to view and search data.

- **Task 1 (Frontend Init):** Initialize the Refine.dev project. Connect it to the API.
- **Task 2 (Data Grid):** Use Ant Design tables to display candidates. Wire up the useTable hook to the Search API.
- **Task 3 (Full-Text Search):** Implement the tsvector column and search query in the backend. Verify that "Leavers" are excluded from results.

Phase 4: The Client Loop & Logic (Weeks 7-8)

Goal: Enable external access and strict state management.

- **Task 1 (FSM Implementation):** Write the State Machine logic in Python. Ensure JOINED -> LEFT triggers the blacklist update.
- **Task 2 (Magic Links):** Implement the JWT generation logic and the email trigger.
- **Task 3 (Client UI):** Build the simplified Client Portal with the Kanban view.

Phase 5: Polish & Real-Time (Weeks 9-10)

Goal: Synchronization and Deployment.

- **Task 1 (SSE):** Add the SSE endpoint and Redis Pub/Sub logic. Connect the Frontend to listen for updates.
- **Task 2 (Deployment):** Configure GitHub Actions. Deploy to a PaaS provider (e.g., Render, Railway, or AWS App Runner) which simplifies operations for a single developer compared to raw K8s.

11.1 Failure Scenarios and Mitigation

Parsing Inaccuracies:

- *Risk:* The parser fails to identify a salary format (e.g., "50k") or misinterprets a company

name.

- *Mitigation:* The database schema is flexible (JSONB). The system assigns a "Confidence Score" to the parse. If the score is low, the status is set to MANUAL REVIEW. HR users can view the raw text side-by-side with the form and correct the data. These corrections can be used to retrain the parser or update the regex rules.

Email Ingestion Bursts:

- *Risk:* Receiving thousands of resumes in a short period (e.g., after a job fair) could overwhelm the parsing service.
 - *Mitigation:* The decoupling provided by the Redis queue ensures the Webhook receiver never times out. The system can scale the number of app-worker containers horizontally to process the backlog without affecting the app-api performance.
-

12. Conclusion

This architectural plan delivers a robust, secure, and highly automated HR system tailored to the user's specific requirements. By choosing a Modular Monolith with PostgreSQL, the system ensures data integrity and simplifies the complex query logic required to exclude former employees. The integration of SendGrid for ingestion, Python/spaCy for parsing, Refine.dev for the frontend, and Server-Sent Events for real-time synchronization creates a cohesive ecosystem that balances cost, performance, and user experience. The rigorous application of Row-Level Security guarantees that client data remains isolated, addressing the critical security implications of a multi-tenant client portal. This blueprint provides a clear, actionable path for a single engineering resource to build, deploy, and scale the solution.¹

Works cited

1. HR System Development Plan - Google Docs.pdf