Internal Design Document

HYPERSCREEN

University of British Columbia

CPSC 319 - Software Engineering Project

Date of Submission: April 7th, 2025

Instructor: Jerry Jim

By Group: Daine Yip, Tirth Choksey, James Reinhardt, Advika Kumari, Muneeba Ashiq, Steve Paul, Ki Bum Kim, Advay Rajguru

Version History				
Version	Approved By	Revision Date	Description of Change	Author
1.0			Document creation	All internal team members
2.0		April 5, 2025	Updates to Design	All internal team members

Stakeholder Sign Off

_____ April 7th, 2025

Marek Buchler

_____ April 7th, 2025

Peter Smith

_____ April 7th, 2025

Shashank Shivamurthy

_____ April 7th, 2025

Jerry Jim

Internal Sign Off

Advay Rajguru

_____ April 7th, 2025

Advay Rajguru

Daine Yip

_____ April 7th, 2025

Daine Yip

James Reinhardt _____ April 7th, 2025

James Reinhardt

Advika Kumari

_____ April 7th, 2025

Advika Kumari

Ki Bum Kim

_____ April 7th, 2025

Ki Bum Kim

Muneeba Ashiq

_____ April 7th, 2025

Muneeba Ashiq

Steve Paul

_____ April 7th, 2025

Steve Paul

Tirth Choksey

_____ April 7th, 2025

Tirth Choksey

Table of Contents

Introduction

Overview

As AWS postings continue to garner a large number of applicants, the team at AWS Vancouver would like to speed up the process and allow recruiters and hiring managers to become more effective in filling positions in the organization.

Hyperscreen is a web-based tool to optimize applicant screening. Leveraging technologies like Nextjs/react, Django, and Docker, the application will enable efficient job posting, applicant evaluation, and reporting with a user-centric design.

Goals

The HyperScreen project aims to address the challenges faced by AWS recruiters and hiring managers in managing high application volumes.

Key features that we aim to include are a customizable scoring criteria, detailed reporting, and streamlined recruiter workflows, targeting a 50% reduction in screening time and high candidate retention.

Assumptions

We work under the assumption that our solution will be able to support a maximum user base of 500 participants (recruiter, HR and hiring manager, applicants, etc).

Programming Environment

Programming Languages & Runtime

- Python 3.12.4
- Node.js 22.13.1
- Typescript ^5

Integrated Development Environment (IDE)

Primary IDE

- Jetbrains Pycharm (Latest version)
    - Student Developer Pack Required

Alternative Code Editors

- Visual Studio Code
- Vim/NeoVim

Version Control

- System: Git
- Repository Host: GitHub

Backend Framework & Dependencies

- Django 5.1.5
- Dependencies:

    - Django==5.1.5
    - psycopg[binary]==3.2.4
    - gunicorn==21.2.0
    - python-dotenv==1.0.1
    - djangorestframework==3.15.2
    - django-cors-headers==4.6.0
    - django-q2==1.7.6
    - PyMuPDF== 1.25.3
    - django-q2==1.7.6
    - Faker==37.0.0
    - reportlab==4.3.1
    - requests==2.32.3
    - python-docx==1.1.2
    - pypdf2==3.0.1
    - pypandoc==1.15
    - xlsxwriter==3.2.0
- Stretch Goal Dependencies
    - Ollama==0.6.4
    - Tinyllama 1.1b

Frontend Framework & Tools

- React==19.0.0
- Next.js=15.2.4
- Tailwind CSS==3.4.1
- Package manager: npm==10.9.2

Database

- PostgreSQL 17.2

Design Tools

- Figma
    - UML and Prototyping

Project Management

- Jira
    - Issue tracking and bug reporting
    - Backlog management
- Google Docs
    - Documentation drafting
    - Version history tracking

Production, Development and Test Environments

Production:

We intend to deploy hyperscreen as containerised services using Docker. Currently, it requires 4 docker containers with the corresponding architectures:

- Backend Server
    - Django application
    - Gunicorn as WSGI HTTP Server
    - REST API implementation using Django Rest framework
    - Reasoning
        - Seamless integration with SQL-based databases
        - Built-in CRUD functionality reduces development effort
        - Production-ready framework
- Frontend Server
    - React/Next.js application
    - Nodejs server provided by Nextjs build
    - Typescript-based implementation
    - Tailwind for styling
    - Reasoning
        - Team familiarity with React
        - Reliable and battle-tested framework
        - **Server-side rendering (SSR)** improves SEO for the applicant module
- Database
    - PostgreSQL 17.2
    - Reasoning
        - Popular choice with powerful features
        - Natively integrates with Django's ORM for easier development
- Nginx (Reverse Proxy)
    - Exposed on port 80
    - Requests to "/" are forwarded to the frontend server

    - Requests to “/proxy/” are forwarded to the backend server
    - Requests to “/media/” are used to serve the uploaded media files
    - Reasoning
        - High-performance and resource-efficient
        - Optimizes application server usage
        - Efficiently serves static files, including uploaded résumés
- Qcluster
    - A service provided by django-q2
    - Multiple workers spawned that look for tasks queued to postgres db by main backend server
    - Used to execute resume parsing in the background asynchronously.

All containers share the same internal network with only the nginx server being exposed to the internet to handle incoming requests and send out responses.

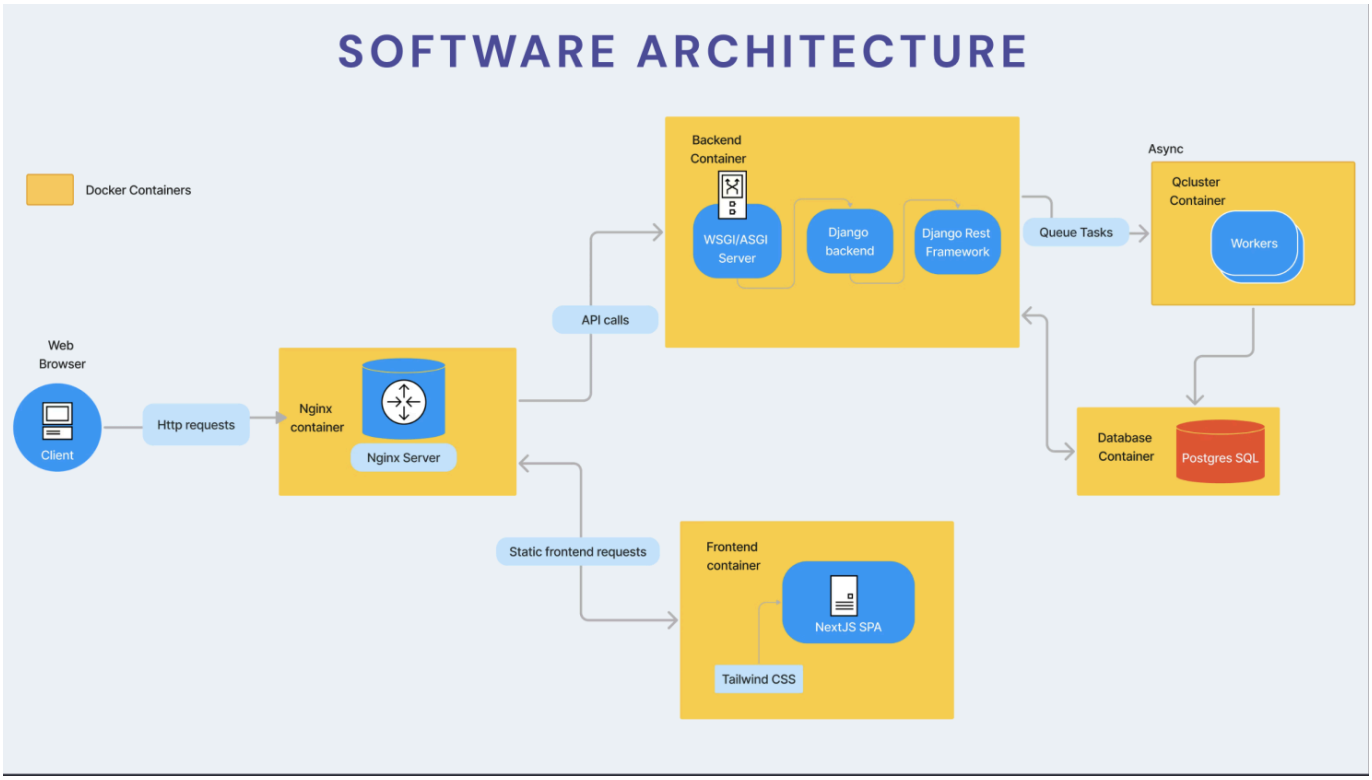
Testing & Development Environment

We intend on running automated tests locally on our dev setup as well as manual testing on the production environment.

Requirements for dev setup:

- Postgres 17.2 running locally on your machine with specific username and password
- Use django’s own development server in a virtual environment with the right packages installed on localhost:8000
    - Django’s unit tests can be run using the management module.
- Spawn the Qcluster process along with the django dev server to handle the background tasks.
- Use Next.js development server to run the frontend on localhost:3000
    - Use Next.js unit tests and end to end tests
- Do end to end manual testing on the application running on localhost:3000.

Software Architecture

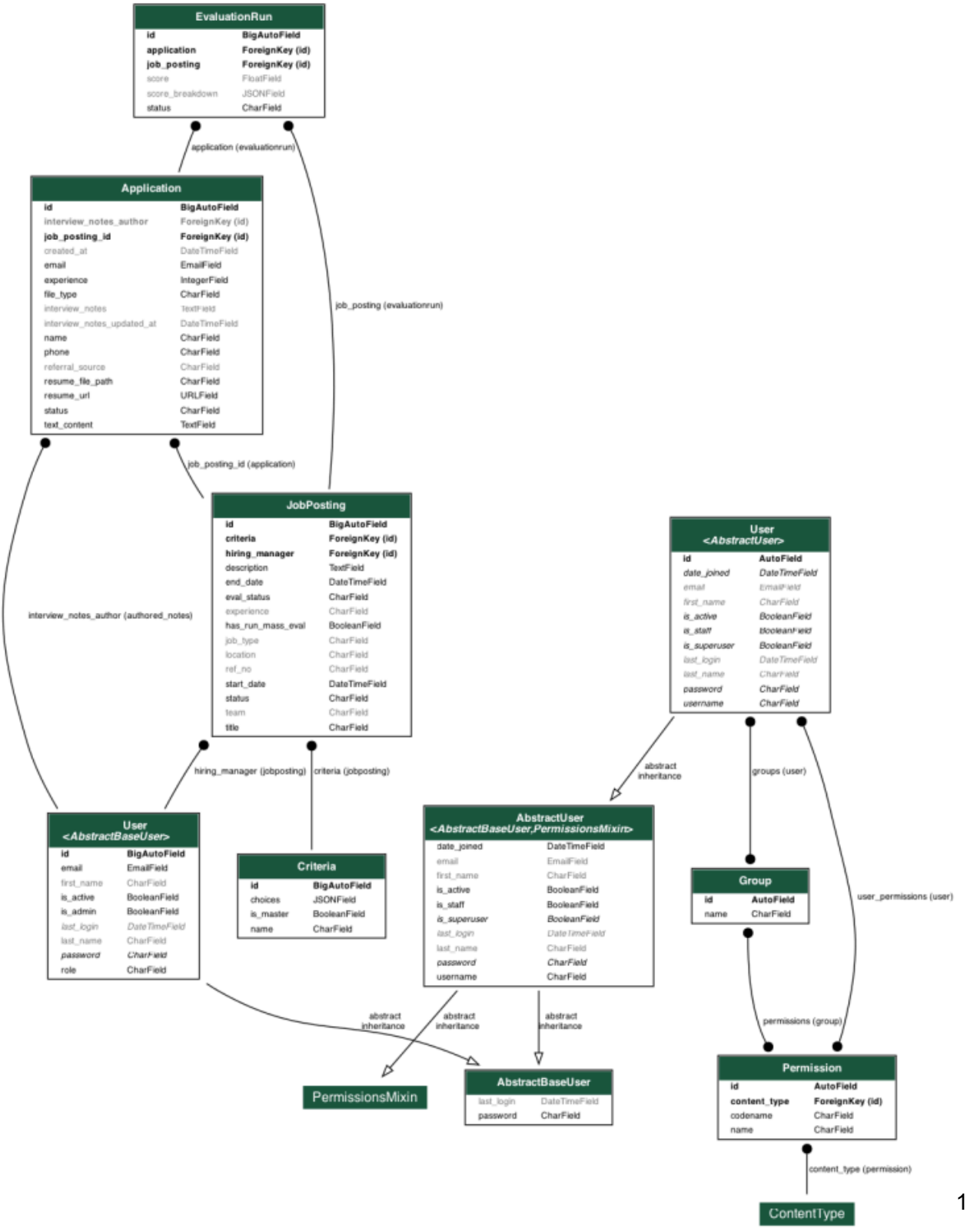


The client's web browser sends HTTP requests to the Nginx container.

Presentation & User Interface	Nginx Requests to "/" are forwarded to the frontend container. React, Next JS, Server Side Rendering
Application Layer	Django Rest Framework (DRF) for REST API implementation, facilitating communication between frontend/backend.
Domain Layer	Django application. Implements core rules, such as job matching, applicant tracking, and access control.
Data Layer	PostgreSQL 17.2 Database, Managing storage for job listing, applicants, resumes, profiles

Data Design

Relational Schema



SQL Code

```
BEGIN;
--
-- Create model Permission
--
CREATE TABLE "auth_permission" ("id" integer NOT NULL PRIMARY KEY GENERATED BY DEFAULT AS IDENTITY, "name" varchar(50) NOT NULL, "content_type_id" integer NOT NULL, "codename" varchar(100) NOT NULL);
--
-- Create model Group
--
CREATE TABLE "auth_group" ("id" integer NOT NULL PRIMARY KEY GENERATED BY DEFAULT AS IDENTITY, "name" varchar(80) NOT NULL UNIQUE);
CREATE TABLE "auth_group_permissions" ("id" bigint NOT NULL PRIMARY KEY GENERATED BY DEFAULT AS IDENTITY, "group_id" integer NOT NULL, "permission_id" integer NOT NULL);
--
-- Create model User
--
CREATE TABLE "auth_user" ("id" integer NOT NULL PRIMARY KEY GENERATED BY DEFAULT AS IDENTITY, "password" varchar(128) NOT NULL, "last_login" timestamp with time zone NOT NULL, "is_superuser" boolean NOT NULL, "username" varchar(30) NOT NULL UNIQUE, "first_name" varchar(30) NOT NULL, "last_name" varchar(30) NOT NULL, "email" varchar(75) NOT NULL, "is_staff" boolean NOT NULL, "is_active" boolean NOT NULL, "date_joined" timestamp with time zone NOT NULL);
CREATE TABLE "auth_user_groups" ("id" bigint NOT NULL PRIMARY KEY GENERATED BY DEFAULT AS IDENTITY, "user_id" integer NOT NULL, "group_id" integer NOT NULL);
CREATE TABLE "auth_user_user_permissions" ("id" bigint NOT NULL PRIMARY KEY GENERATED BY DEFAULT AS IDENTITY, "user_id" integer NOT NULL, "permission_id" integer NOT NULL);
ALTER TABLE "auth_permission" ADD CONSTRAINT "auth_permission_content_type_id_codename_01ab375a_uniq" UNIQUE ("content_type_id", "codename");
ALTER TABLE "auth_permission" ADD CONSTRAINT "auth_permission_content_type_id_2f476e4b_fk_django_co" FOREIGN KEY ("content_type_id") REFERENCES "django_content_type" ("id") DEFERRABLE INITIALLY DEFERRED;
CREATE INDEX "auth_permission_content_type_id_2f476e4b" ON "auth_permission" ("content_type_id");
```

```
CREATE INDEX "auth_group_name_a6ea08ec_like" ON "auth_group" ("name"
varchar_pattern_ops);
ALTER TABLE "auth_group_permissions" ADD CONSTRAINT
"auth_group_permissions_group_id_permission_id_0cd325b0_uniq" UNIQUE ("group_id",
"permission_id");
ALTER TABLE "auth_group_permissions" ADD CONSTRAINT
"auth_group_permissions_group_id_b120cbf9_fk_auth_group_id" FOREIGN KEY ("group_id")
REFERENCES "auth_group" ("id") DEFERRABLE INITIALLY DEFERRED;
ALTER TABLE "auth_group_permissions" ADD CONSTRAINT
"auth_group_permissio_permission_id_84c5c92e_fk_auth_perm" FOREIGN KEY
("permission_id") REFERENCES "auth_permission" ("id") DEFERRABLE INITIALLY
DEFERRED;
CREATE INDEX "auth_group_permissions_group_id_b120cbf9" ON "auth_group_permissions"
("group_id");
CREATE INDEX "auth_group_permissions_permission_id_84c5c92e" ON
"auth_group_permissions" ("permission_id");
CREATE INDEX "auth_user_username_6821ab7c_like" ON "auth_user" ("username"
varchar_pattern_ops);
ALTER TABLE "auth_user_groups" ADD CONSTRAINT
"auth_user_groups_user_id_group_id_94350c0c_uniq" UNIQUE ("user_id", "group_id");
ALTER TABLE "auth_user_groups" ADD CONSTRAINT
"auth_user_groups_user_id_6a12ed8b_fk_auth_user_id" FOREIGN KEY ("user_id")
REFERENCES "auth_user" ("id") DEFERRABLE INITIALLY DEFERRED;
ALTER TABLE "auth_user_groups" ADD CONSTRAINT
"auth_user_groups_group_id_97559544_fk_auth_group_id" FOREIGN KEY ("group_id")
REFERENCES "auth_group" ("id") DEFERRABLE INITIALLY DEFERRED;
CREATE INDEX "auth_user_groups_user_id_6a12ed8b" ON "auth_user_groups" ("user_id");
CREATE INDEX "auth_user_groups_group_id_97559544" ON "auth_user_groups" ("group_id");
ALTER TABLE "auth_user_user_permissions" ADD CONSTRAINT
"auth_user_user_permissions_user_id_permission_id_14a6b632_uniq" UNIQUE ("user_id",
"permission_id");
ALTER TABLE "auth_user_user_permissions" ADD CONSTRAINT
"auth_user_user_permissions_user_id_a95ead1b_fk_auth_user_id" FOREIGN KEY ("user_id")
REFERENCES "auth_user" ("id") DEFERRABLE INITIALLY DEFERRED;
ALTER TABLE "auth_user_user_permissions" ADD CONSTRAINT
"auth_user_user_permi_permission_id_1fbb5f2c_fk_auth_perm" FOREIGN KEY
("permission_id") REFERENCES "auth_permission" ("id") DEFERRABLE INITIALLY
DEFERRED;
CREATE INDEX "auth_user_user_permissions_user_id_a95ead1b" ON
"auth_user_user_permissions" ("user_id");
CREATE INDEX "auth_user_user_permissions_perm
```

```
BEGIN;
--
-- Create model Application
--
CREATE TABLE "api_application" ("id" bigint NOT NULL PRIMARY KEY GENERATED BY DEFAULT AS IDENTITY, "status" varchar(55) NOT NULL, "created_at" timestamp with time zone NOT NULL, "email" varchar(254) NOT NULL, "resume_url" varchar(200) NOT NULL, "resume_file_path" varchar(100) NOT NULL, "file_type" varchar(50) NOT NULL, "text_content" text NOT NULL, "name" varchar(255) NOT NULL, "phone" varchar(20) NOT NULL, "experience" integer NOT NULL);
--
-- Create model Criteria
--
CREATE TABLE "api_criteria" ("id" bigint NOT NULL PRIMARY KEY GENERATED BY DEFAULT AS IDENTITY, "name" varchar(255) NOT NULL, "choices" jsonb NOT NULL, "is_master" boolean NOT NULL);
--
-- Create model User
--
CREATE TABLE "api_user" ("id" bigint NOT NULL PRIMARY KEY GENERATED BY DEFAULT AS IDENTITY, "password" varchar(128) NOT NULL, "last_login" timestamp with time zone NULL, "email" varchar(254) NOT NULL UNIQUE, "is_active" boolean NOT NULL, "first_name" varchar NULL, "last_name" varchar NULL, "is_admin" boolean NOT NULL, "role" varchar(20) NOT NULL);
--
-- Create model JobPosting
--
CREATE TABLE "api_jobposting" ("id" bigint NOT NULL PRIMARY KEY GENERATED BY DEFAULT AS IDENTITY, "ref_no" varchar(15) NOT NULL UNIQUE, "title" varchar NOT NULL, "description" text NOT NULL, "status" varchar(20) NOT NULL, "start_date" timestamp with time zone NOT NULL, "end_date" timestamp with time zone NOT NULL, "location" varchar(255) NULL, "job_type" varchar(10) NULL, "team" varchar(10) NULL, "experience" varchar(10) NULL, "eval_status" varchar(20) NOT NULL, "criteria_id" bigint NOT NULL, "hiring_manager_id" bigint NOT NULL);
--
-- Create model EvaluationRun
--
CREATE TABLE "api_evaluationrun" ("id" bigint NOT NULL PRIMARY KEY GENERATED BY DEFAULT AS IDENTITY, "score" double precision NULL, "status" varchar(10) NOT NULL, "application_id" bigint NOT NULL, "job_posting_id" bigint NOT NULL);
--
-- Add field job_posting_id to application
--
```

```
ALTER TABLE "api_application" ADD COLUMN "job_posting_id_id" bigint NOT NULL CONSTRAINT
"api_application_job_posting_id_id_c031a9bc_fk_api_jobposting_id" REFERENCES
"api_jobposting"("id") DEFERRABLE INITIALLY DEFERRED; SET CONSTRAINTS
"api_application_job_posting_id_id_c031a9bc_fk_api_jobposting_id" IMMEDIATE;
--
-- Raw SQL operation
--
CREATE EXTENSION IF NOT EXISTS pg_trgm;
CREATE INDEX "api_user_email_9ef5afa6_like" ON "api_user" ("email" varchar_pattern_ops);
ALTER TABLE "api_jobposting" ADD CONSTRAINT
"api_jobposting_criteria_id_44356428_fk_api_criteria_id" FOREIGN KEY ("criteria_id") REFERENCES
"api_criteria" ("id") DEFERRABLE INITIALLY DEFERRED;
ALTER TABLE "api_jobposting" ADD CONSTRAINT
"api_jobposting_hiring_manager_id_b4c84eb5_fk_api_user_id" FOREIGN KEY ("hiring_manager_id")
REFERENCES "api_user" ("id") DEFERRABLE INITIALLY DEFERRED;
CREATE INDEX "api_jobposting_ref_no_b728c667_like" ON "api_jobposting" ("ref_no"
varchar_pattern_ops);
CREATE INDEX "api_jobposting_criteria_id_44356428" ON "api_jobposting" ("criteria_id");
CREATE INDEX "api_jobposting_hiring_manager_id_b4c84eb5" ON "api_jobposting"
("hiring_manager_id");
ALTER TABLE "api_evaluationrun" ADD CONSTRAINT
"api_evaluationrun_application_id_842525df_fk_api_application_id" FOREIGN KEY ("application_id")
REFERENCES "api_application" ("id") DEFERRABLE INITIALLY DEFERRED;
ALTER TABLE "api_evaluationrun" ADD CONSTRAINT
"api_evaluationrun_job_posting_id_d1d48e16_fk_api_jobposting_id" FOREIGN KEY ("job_posting_id")
REFERENCES "api_jobposting" ("id") DEFERRABLE INITIALLY DEFERRED;
CREATE INDEX "api_evaluationrun_application_id_842525df" ON "api_evaluationrun"
("application_id");
CREATE INDEX "api_evaluationrun_job_posting_id_d1d48e16" ON "api_evaluationrun"
("job_posting_id");
CREATE INDEX "api_application_job_posting_id_id_c031a9bc" ON "api_application"
("job_posting_id_id");
COMMIT;
```

API Design

Applicant Module

1. Get All Jobs (Paginated)

Endpoint: GET /api/application/jobs

Description: Retrieves a paginated list of all jobs that exist on the platform.

Response:

- **Return Type:** List<Job>
- **Status Codes:**
    - 200 OK – Successfully retrieved the job list.
    - 400 Bad Request – Invalid pagination parameters.

2. Get a Specific Job

Endpoint: GET /api/get_jobInfo

Description: Retrieves details of a specific job using its job ID.

Path Parameters:

Parameter	Type	Required	Description
jobId	int	Yes	Unique identifier of the job

Response:

- **Return Type:** Job
- **Status Codes:**
    - 200 OK – Successfully retrieved the job details.
    - 404 Not Found – Job ID does not exist.

3. Search for a Job (Paginated)

Endpoint: GET /api/application/jobs

Description: Search for jobs by keyword, reference number, or title.

Query Parameters:

Parameter	Type	Required	Description
keyword	string	No	Keyword to search within job titles and descriptions
location	string	No	Reference number of the job

Response:

- **Return Type:** List<Job>
- **Status Codes:**
    - 200 OK – Successfully retrieved the matching job list.
    - 400 Bad Request – Invalid search query.

4. Check If an Application Exists

Endpoint: GET /api/application/{email}/{jobID}

Description: Checks whether a user has already applied for a specific job.

Path Parameters:

Parameter	Type	Required	Description
email	string	Yes	Email of the applicant
jobID	int	Yes	Unique identifier of the job
email	string	Yes	Unique identifier for user

Response:

- **Return Type:** Boolean
    - true – Application exists
    - false – No application found
- **Status Codes:**
    - 200 OK – Successfully checked application status.
    - 400 Bad Request – Invalid email or job ID format.

5. Create a Job Application

Endpoint: POST /api/application/jobs/{jobId}

Description: Creates a new application for a job, submitting both the CV and application data.

Request Body:

Content-Type: form-data

```
{
   "email": "string",
   "name": "string",
   "phone": "string",
   “experience”: int,
   "resume": "file"
}
```

Response:

- **Return Type:** Application
- **Status Codes:**
    - 201 Created – Application successfully submitted.
    - 400 Bad Request – Invalid input data.
    - 404 Not Found – Job ID does not exist.

6. Generate Interview Questions

Endpoint: GET /api/interview-questions/{application_id}/

Description: Generates a list of interview questions for a particular job

Path Parameters:

Parameter	Type	Required	Description

application_Id	string	Yes	Application ID used to get JobPosting for interview question generation

Response:

- **Return Type:** String
- **Status Codes:**
  - 200 OK – Interview questions generated successfully
  - 400 Bad Request – Invalid input data.
  - 404 Not Found – Job posting or Application not found
  - 500 Internal Error - Error with interview question model

Hiring Manager Module

1. Get List of Jobs Created by a Specific HM or Admin (Paginated)

Endpoint: GET /api/job_postings/

Description: Retrieves a list of job postings based on user role:

- **Admin**: Gets all job postings.
- **Hiring Manager (HM)**: Gets only the jobs they have created.

Path Parameters:

Parameter	Type	Required	Description
email	string	Yes	Unique identifier of the user
page	int	No	Page number for pagination (default: 1)
limit	int	No	Number of jobs per page (default: 10)

Response:

- **Return Type:** List<Job>

- **Status Codes:**
    - 200 OK – Successfully retrieved the job list.
    - 403 Forbidden – Unauthorized access.

2. Get Job Data by Job ID

Endpoint: GET /api/job_postings/{id}

Description: Retrieves all details related to a specific job.

Path Parameters:

Parameter	Type	Required	Description
id	int	Yes	Unique identifier of the job

Response:

- **Return Type:** Job
- **Status Codes:**
    - 200 OK – Successfully retrieved job data.
    - 404 Not Found – Job ID does not exist.

3. Delete Job Posting by ID

Endpoint: DELETE /api/job_postings/{id}

Description: Deletes all data associated with a job, including scoring criteria.

Path Parameters:

Parameter	Type	Required	Description
id	int	Yes	Unique identifier of the job

Response:

- **Status Codes:**
    - 200 OK – Job deleted successfully.
    - 404 Not Found – Job ID does not exist.
    - 403 Forbidden – Unauthorized access.

4. Start Evaluation for a Specific Job

Endpoint: POST /api/eval/queue/<str:jobID>/

Description: Runs evaluation on all applicants:

- If **processing**, returns 0.
- If **not processing**, drops all associated row instances from the Runs table and runs evaluations.

Path Parameters:

Parameter	Type	Required	Description
jobid	str	Yes	Unique identifier of the job

Response:

- **Return Type:** List<Evaluated_Applications>
- **Status Codes:**
    - 200 OK – Evaluation started successfully.
    - 400 Bad Request – Evaluation cannot proceed.

5. Mass Evaluation of All Applicants for a Job

Endpoint: POST /api/eval/queue/<str:jobID>/?mass=true

Description: Runs evaluation on all applicants:

- If **processing**, returns 0.
- If **not processing**, drops all associated row instances from the Runs table and runs evaluations.

Path Parameters:

Parameter	Type	Required	Description
jobid	str	Yes	Unique identifier of the job

Response:

- **Return Type:** List<Evaluated_Applications>

- **Status Codes:**
    - 200 OK – Evaluation started successfully.
    - 400 Bad Request – Evaluation cannot proceed.

6. Create a New Job Posting

Endpoint: POST /api/job_postings

Description: Creates a new job posting:

- Associates the job with a **Hiring Manager**.
- Pulls in **Master Scoring Criteria**.

Response:

- **Return Type:** Job
- **Status Codes:**
    - 201 Created – Job created successfully.
    - 400 Bad Request – Invalid job data.

7. Update Job Details

Endpoint: PUT /api/update_job/?jobid={jobid}

Description: Updates job details such as location, title, etc. Works for both populated and unpopulated fields.

Query Parameters:

Parameter	Type	Required	Description
jobid	int	Yes	Unique identifier of the job

Response:

- **Status Codes:**
    - 200 OK – Job updated successfully.
    - 400 Bad Request – Invalid job data.
    - 404 Not Found – Job ID does not exist.

8. Get Scoring Criteria for a Job

Endpoint: GET /api/scoring_criteria/{id}

Description: Retrieves the scoring criteria associated with a job.

Path Parameters:

Parameter	Type	Required	Description
id	int	Yes	Unique identifier of the job

Response:

- **Return Type:** ScoringCriteria
- **Status Codes:**
    - 200 OK – Successfully retrieved scoring criteria.
    - 404 Not Found – Job ID does not exist.

9. Update Scoring Criteria for a Job

Endpoint: PUT /api/update_scoring_criteria/{jobid}

Description: Updates existing scoring criteria for a job. The Hiring Manager submits a template taken from **Master Scoring Criteria**.

Path Parameters:

Parameter	Type	Required	Description
jobid	int	Yes	Unique identifier of the job

Body:

{“criteria”: <Criteria fields to update>}

Response:

- **Status Codes:**
    - 200 OK – Scoring criteria updated successfully.
    - 400 Bad Request – Invalid data.
    - 404 Not Found – Job ID does not exist.

10. Get all applicants for a job

Endpoint: GET /api/job_postings/<int:jobId>/applicants/

Description: Gets all applicants for a give job.

Path Parameters:

Parameter	Type	Required	Description
jobid	int	Yes	Unique identifier of the job

Response:

- **Return Type:** List<Applications>
- **Status Codes:**
    - 200 - returns a list of applications for that job
    - 404 Not found – No job found

11. Get all applicants for a job

Endpoint: GET /api/get_jobInfo/

Description: fetch info on a job

Path Parameters:

Parameter	Type	Required	Description
jobid	int	Yes	Unique identifier of the job

Response:

- **Return Type:** Job
- **Status Codes:**
    - 200 - returns info on a job
    - 404 Not found – No job found

12. Get scores for all applicants

Endpoint: GET /api/scores/<int:jobID>/

Description: get scores for all applications (direct or otherwise) related to this job.

Used to show the results of mass evaluation.

Path Parameters:

Parameter	Type	Required	Description
jobid	int	Yes	Unique identifier of the job

Response:

- **Return Type:** List<EvaluationRuns>
- **Status Codes:**
    - 200 - returns list of evaluation runs
    - 404 Not found – No job found

12. Get scores for only applicants who applied

Endpoint: GET /api/scores-filtered/<int:jobID>/

Description: get scores for applicants who only applied for this job.

Used to show the results of regular evaluations.

Path Parameters:

Parameter	Type	Required	Description
jobid	int	Yes	Unique identifier of the job

Response:

- **Return Type:** List<EvaluationRuns>
- **Status Codes:**
    - 200 - returns list of evaluation runs
    - 404 Not found – No job found

13. Get and add interview notes to an application

Endpoint: GET, PUT /api/application/<int:application_id>/interview-notes/

Description: get, add or update the interview notes associated with an application

Path Parameters:

Parameter	Type	Required	Description

application_id	int	Yes	Unique identifier of the application

Response:

- **Return Type:** Application
- **Status Codes:**
    - 200 - returns an application as is, updated or added
    - 404 Not found – No application found

14. Get resume

Endpoint: POST /api/get_resume/

Description: get the pdf file that the applicant used to apply with, if the user uploaded docx, it converts it into pdf and returns that.

query Parameters:

Parameter	Type	Required	Description
path	str	Yes	The path to the file to retrieve

Response:

- **Return Type:** FileResponse
- **Status Codes:**
    - 200 - returns a file
    - 404 Not found – No file found

15. Check the evaluation status

Endpoint: GET /api/eval/check/<str:jobID>/

Description: Check what the status of the current evaluation of a job is (both mass and regular). Returns status of either init, Complete or Processing

Path Parameters:

Parameter	Type	Required	Description
jobid	int	Yes	Unique identifier of the job

Response:

- **Return Type:** status as json
- **Status Codes:**
    - 200 - returns a json
    - 404 Not found – No job found

15. Get Application

Endpoint: GET /api/application/<int:id>/

Description : Get all information on an application by the application id

Path Parameters:

Parameter	Type	Required	Description
application_id	int	Yes	Unique identifier of the application

Response:

- **Return Type:** Application
- **Status Codes:**
    - 200 - returns an object of type Application
    - 404 Not found – No application found

16. Generate Interview questions (experimental)

Endpoint: GET /api/interview-questions/<int:application_id>/

Description: generates interview questions using an llm based on the job description and the resume of the applicant. This is not supported on the main branch and only is available on another. Had to do so due to resource constraints on the server. Does work on our personal laptops.

Path Parameters:

Parameter	Type	Required	Description
application_id	int	Yes	Unique identifier of the application

Response:

- **Return Type:** interview questions as json
- **Status Codes:**
    - 200 - returns a json
    - 404 Not found – No application found

Admin Module

1. Start Evaluation for a Specific Job

Endpoint: POST /api/eval/{jobid}

Description: Checks job status:

- If **processing**, returns 0.
- If **not processing**, runs evaluation, drops all associated row instances from the Runs table, and evaluates applicants.

Path Parameters:

Parameter	Type	Required	Description
jobid	int	Yes	Unique identifier of the job

Response:

- **Return Type:** List<Evaluated_Applications>
- Status Codes:
    - 200 OK – Evaluation started successfully.
    - 400 Bad Request – Evaluation cannot proceed.

2. Mass Evaluation of All Applicants for a Job

Endpoint: POST /api/mass-eval/{jobid}

Description: Runs evaluation on all applicants:

- If **processing**, returns 0.
- If **not processing**, drops all associated row instances from the Runs table and runs evaluations.

Path Parameters:

Parameter	Type	Required	Description
jobid	int	Yes	Unique identifier of the job

Response:

- **Return Type:** List<Evaluated_Applications>
- Status Codes:
    - 200 OK – Evaluation started successfully.
    - 400 Bad Request – Evaluation cannot proceed.

3. Get Master Scoring Criteria

Endpoint: GET /api/Mastercriteria

Description: Retrieves the master scoring criteria used for evaluating job applicants.

Response:

- **Return Type:** ScoringCriteria
- Status Codes:
    - 200 OK – Successfully retrieved scoring criteria.
    - 404 Not Found – No master scoring criteria found.

4. Update Master Scoring Criteria

Endpoint: POST /api/Mastercriteria

Description: Updates the master scoring criteria

Request Body:

Content-Type: form-data
{
“Education:: {
“Bachelors”: int
“Masters”: int
},
“Location”: {
“Vancouver”: int
….
}

Response:

- Status Codes:
  - 200 OK – Scoring criteria updated successfully.
  - 400 Bad Request – Invalid data.

5. Get List of Hiring Managers

Endpoint: GET /api/hrmanager/

Description: Retrieves a list of all hiring managers.

Response:

- **Return Type:** List<User>
- Status Codes:
  - 200 OK – Successfully retrieved list.
  - 404 Not Found – No users found.

6. Update Hiring Manager Information

Endpoint: POST /api/hrmanager{hmid}

Description: Updates details of a hiring manager.

Content-Type: form-data
{
"email": "string",
"firstname": "string",
"lastname": "string",
"role": "string"

}

Path Parameters:

Parameter	Type	Required	Description
hmid	int	Yes	Unique identifier for admin user

Response:

- Status Codes:
    - 200 OK – Hiring manager updated successfully.
    - 400 Bad Request – Invalid update data.
    - 404 Not Found – Hiring manager ID does not exist.

7. Create a New Hiring Manager

Endpoint: POST /api/hrmanager/add

Description: Creates a new hiring manager profile.

Response:

- **Return Type:** User
- Status Codes:
    - 201 Created – Hiring manager added successfully.
    - 400 Bad Request – Invalid data.

8. Delete a Hiring Manager

Endpoint: DELETE /api/hrmanager/{id}

Description: Deletes a hiring manager by ID.

Path Parameters:

Parameter	Type	Required	Description
hmid	int	Yes	Unique identifier for admin user

Response:

- Status Codes:
    - 200 OK – Hiring manager deleted successfully.
    - 404 Not Found – Hiring manager ID does not exist.

9. Update Application status

Endpoint: PUT api/application/{id}/status

Description: Updates the application status

Path Parameters:

Parameter	Type	Required	Description
id	int	Yes	Unique identifier for application

Response:

- Status Codes:
    - 200 OK – Status updated successfully.
    - 400 Bad request – Status not found, invalid status
    - 404 Not found - Application ID not found

10. Update HR manager password

Endpoint: PUT /api/hrmanager/{id}/password/

Description: Update a hiring manager's password. Only accessible by recruiters

Path Parameters:

Parameter	Type	Required	Description
user_id	int	Yes	Unique identifier for HR manager user

Response:

- Status Codes:
    - 200 OK – Password updated successfully for HR manager user.
    - 400 Bad Request – Password not found
    - 403 Forbidden - Only recruiters can update passwords
    - 404 Not found - User not found

Reporting Module

1. Export Report Data for Applicant Scores

Endpoint: GET
/api/export/scores/<int:jobId>/"

Description: Generate excel sheet for reports on individual applicant scores.

Response:

- **Return Type:** Excel sheet
- **Status Codes:**
    - 200 OK – Successfully retrieved rankings.
    - 400 Not Found – job id not found
    - 500 Bad Request.

2. Export Report Data for Applicant Status

Endpoint: GET
/api/export/status/<int:jobId>/"

Description: Generate Excel sheet for report on individual applicant job status (rejected, accepted, submitted, interviewed, and submitted).

Response:

- **Return Type:** Excel sheet
- **Status Codes:**
    - 200 OK – Successfully retrieved rankings.
    - 400 Not Found – job id not found
    - 500 Bad Request.

3. Export Report Data for Applicant Experience

Endpoint: GET
/api/export/experience/<int:jobId>/"
Description: Generate report for applicant years of experience.

Response:

- **Return Type:** Excel sheet
- **Status Codes:**
    - 200 OK – Successfully retrieved rankings.
    - 400 Not Found – job id not found
    - 500 Bad Request.

4. Export Report Data for Applicant Referral Source Information

/api/export/referral/<int:jobId>/"

Description: Generate report for where applicants heard about posting.

- **Return Type:** Excel sheet
- **Status Codes:**
    - 200 OK – Successfully retrieved rankings.
    - 400 Not Found – job id not found

500 Bad Request.

5. Export Report Data for Overall Interview Stats

/api/export/interview/<int:jobId>/"

Description: Generate report for stats about overall applicant acceptance rate.

- **Return Type:** Excel sheet
- **Status Codes:**
    - 200 OK – Successfully retrieved rankings.
    - 400 Not Found – job id not found
    - 500 Bad Request.

6. Export Report Data for Overall Interview Stats

/api/export/overview/<int:jobId>/"

Description: Exports report data in xlxs format for individual applicant scores.

Response:

- **Return Type:** Excel sheet
- **Status Codes:**
    - 200 OK – Successfully retrieved rankings.
    - 400 Not Found – job id not found
    - 500 Bad Request.

Algorithms

The following algorithm will be used for evaluating and scoring candidates for a particular job.

Phase 1: Defining Scoring Criteria (Admin/Hiring Manager Input)

1. **Criteria Setup**
    - The Admin/Hiring Manager will set an evaluation criteria across categories. Some of these categories include:
        - **Education**: Selected from a predefined list (e.g., High School, Bachelor's, Master's, PhD).
        - **Years of Experience**: Admin/Hiring Manager will input the required experience as an integer.
        - **Hard Skills**: Will be manually inputted as a comma-separated list.
        - **Soft Skills**: Will be manually inputted as a comma-separated list.
2. **Weight Assignment**
    - Each selected criterion will be assigned a weight (scale: 1-10) to reflect its importance.
    - Weights will influence the final score of a candidate.

Phase 2: Resume Data Collection and Preprocessing

1. **Applicant Data Collection**
    - The candidates will submit the following information while applying for a job:
        - Name, email, phone number (excluded from processing for privacy).
        - Years of experience (self-reported to avoid extraction complexity).
        - Information about where they heard about the job.
        - Resume file either a PDF or DOCX (parsed for text extraction).

2. **Preprocessing**
    - The system will extract text from the resume.
        - PyMuPDF package will be used to extract text from the PDF files.
        - Python-docx package will be used to extract text from the DOCX files.

Phase 3: Initial Resume Matching (Keyword-Based Approach)

Once the criteria has been established by the Admin or Hiring Manager, they have the option to initiate evaluations for a specific job, either on all resumes stored in the database or only on those submitted for that particular job. Upon initiating the evaluation, the system will execute the following keyword matching algorithm.

1. **Predefined Categories**
    - A JSON-based dictionary will map variations of predefined list categories like education and location etc.
        - For example:

```
{
  "Bachelor’s": [
    "Bachelor’s Degree Holder",
    "Graduate with Bachelor's Degree",
    "University Graduate",
    "Undergraduate Student"
  ]
}
```

    - If any of the mapped phrases appeared in the resume, the corresponding category will be marked **1**; otherwise, **0**.
2. **Years of Experience Matching**
    - The applicant’s self-reported experience will be compared to the required experience provided by the Admin/Hiring Manager.
    - If it met or exceeded the required experience, this category will be marked **1**; otherwise, **0**.
3. **Hard/Soft Skill Matching (Exact Match)**
    - Skills will be matched against resume text using exact keyword matching.
    - If a specified skill appears, it will be marked **1**; otherwise, **0**.
4. **Score Calculation**
    - All matched categories will produce a **binary vector** per candidate.
    - Each **binary value** will be multiplied by the **assigned weight**.
    - Results of each match and its weight will be stored in a python dictionary.
    - The **final score** will be the weighted sum of all matched values.

Phase 4: Score Breakdown

As the keyword matching process from Phase 3 is executed, the system will simultaneously generate a **score breakdown** to provide detailed insight into how each candidate's overall score was calculated.

This breakdown is stored in a **Python dictionary** and made available to the Admin or Hiring Manager on the frontend. It displays the individual contribution of each evaluation category, offering full transparency into the decision-making process.

Structure of Score Breakdown

```
score_breakdown = {
   "Education": 0,
   "Experience": 0,
   "Hard Skills": 0,
   "Soft Skills": 0,
   "Location": 0,
}
```

How It Works:

- For each matched category in Phase 3, the **assigned weight** is added to its corresponding field in the **score_breakdown**.
- The final **score_breakdown** represents a **weighted distribution** of the candidate's score across different criteria.
- This breakdown is **separate from** the final cumulative score but is calculated in parallel using the same logic.

Notable Trade Offs

- Parsing accuracy with word-matching: using LLM for parsing can increase accuracy but add complexity and longer development. A simple word matching is easier to implement and provides a minimally viable product feature.
- Security of private information: implementing robust encryption or masking can slow down development. Utilizing a basic masking framework will be a minimally viable solution to protect applicant information.
- Handling users beyond 500 or 10 concurrent users: current requirement of maximum of 500 participants (administrators, HR, hiring manager and applicants) and 10 concurrent users are appropriate for this project's scope.

Performance versus Development Speed

- **Database Selection**
    - Chosen Approach: PostgreSQL
    - Trade-off: Better relational capabilities but requires more set up and maintenance
    - Alternative: NoSQL database
- **Parsing Implementation**
    - Chosen Approach: Simple word-matching system
    - Trade-off: A simple word matching is easier to implement and provides a minimally viable product feature.
    - Alternative: LLM integration. Needs to be a local model to avoid leaking user data to cloud services. May reduce performance and be tricky to implement, but would improve screening functionality
- **Data Protection**
    - Chosen Approach: basic masking framework
    - Trade-off: Utilizing a basic masking framework will be a minimally viable solution to protect applicant information.
    - Alternative: Advanced encryption would be more secure but slow down development time.

Notable Risks

Technical Risks & Mitigations

- Poor resume parsing: failing to parse key words in the resume will lead to inaccurate scoring for applicants and undermine the applicant ranking system.
    - In the event that our resume parsing algorithm does not correctly parse the necessary data to perform accurate scoring evaluations, we include the most important fields as manual input fields for the applicant.
- Security measures: potential breach of postgres database data could cause leak of applicants personal information.
    - If the assessed privacy risk is too great, we can migrate to AWS RDS for data storage and make use of their security offerings to protect user data.
- Using a third-party Language Model like openAi or Anthropic could improve performance, but also risks leaking applicant data.
- Participation of 500 participants and concurrent users: can lead to slower response time and system failure with a large user base.
    - We can implement a more robust infrastructure using kubernetes clusters to orchestrate multiple nodes of our services at the same time.