# Project Plan: Automated Greenhouse Job Data Pipeline

Date: October 24, 2025

Objective: To design and implement a scalable, automated data pipeline that extracts job postings from various company Greenhouse boards and loads them into a central AWS RDS PostgreSQL database.

## 1. Project Overview

This pipeline will replace the current manual process of running a local Python script. It will be automated, secure, and "serverless," meaning it will run on a schedule without requiring a dedicated server or manual intervention. The goal is to create a reliable "Extract, Load, Transform" (ELT) process.

## 2. Data Source (Extract)

* **Source:** Public Greenhouse Job Board APIs.
* **Endpoint:** https://boards-api.greenhouse.io/v1/boards/{company\_name}/jobs
* **Initial Companies:** Coinbase, Stripe, Notion, Airbnb, Uber, Lyft, Figma, Plaid, Brex, Canva.

## 3. Data Destination (Load)

* **Database:** AWS RDS for PostgreSQL
* **Endpoint:** user.c0fo0cumqafz.us-east-1.rds.amazonaws.com
* **Database Name:** postgres

### 3.1. Database Schema

A table named greenhouse\_jobs will be created to store the job data. The job\_id from the API will be used as the PRIMARY KEY to prevent duplicate entries.

**SQL CREATE TABLE Statement:**

CREATE TABLE IF NOT EXISTS greenhouse\_jobs (  
 job\_id BIGINT PRIMARY KEY,  
 company\_name VARCHAR(100),  
 title VARCHAR(255),  
 location VARCHAR(255),  
 job\_url TEXT,  
 updated\_at TIMESTAMPTZ,  
 raw\_data JSONB -- Stores the complete, original JSON for future analysis  
);  
  
-- Optional: Index for faster sorting  
CREATE INDEX IF NOT EXISTS idx\_jobs\_updated\_at ON greenhouse\_jobs (updated\_at DESC);

## 4. Proposed Architecture

We will use a modern, serverless AWS architecture. This design is cost-effective (pay-per-use), scalable, and highly secure as it eliminates the need for a public-facing database or an SSH bastion host for automation.

The pipeline consists of three core AWS services:

### Component 1: Amazon EventBridge (Scheduler)

* **Purpose:** This is the pipeline's trigger. It acts as a serverless cron job.
* **Function:** We will configure an EventBridge rule to run on a set schedule (e.g., "every day at 8:00 AM"). When triggered, it will automatically invoke our AWS Lambda function.

### Component 2: AWS Lambda (Compute)

* **Purpose:** This service will run our Python code without managing a server.
* **Function:** The Python script (detailed in Section 6) will be deployed as a Lambda function.
* **Key Configuration:** The Lambda will be configured to run **inside the same VPC as the RDS database**. This is the most important part of the architecture.
  + **Benefit:** Because it's in the same VPC, the Lambda can connect *directly* to the database's private endpoint. It does not need an SSH tunnel or a bastion host.

### Component 3: AWS Secrets Manager (Security)

* **Purpose:** To securely store the database password.
* **Function:** We will store the postgres user's password in Secrets Manager. The Python script will be given permission to retrieve this secret at runtime.
* **Benefit:** This eliminates hardcoding passwords in the script, which is a major security best practice.

## 5. Automated Workflow (Step-by-Step)

1. **Trigger:** At 8:00 AM daily, the Amazon EventBridge rule is triggered.
2. **Invoke:** EventBridge invokes the greenhouse-job-etl Lambda function.
3. **Run:** The Lambda function starts.
4. **Fetch Credentials:** The script retrieves the database password from AWS Secrets Manager.
5. **Extract:** The script loops through the list of company names and calls the Greenhouse API for each, collecting all job listings into a single list.
6. **Load:** The script connects *privately* to the RDS instance (since it's in the same VPC).
7. **Upsert:** It executes an INSERT ... ON CONFLICT query for each job. This adds new jobs and updates existing ones (based on job\_id), ensuring the data is always current and free of duplicates.
8. **Complete:** The Lambda function logs its success and shuts down.

## 6. Implementation Code

### 6.1. Python Script for AWS Lambda

This script is modified to run within Lambda. It removes the SSH tunnel (no longer needed) and includes logic to fetch the password from Secrets Manager (best practice).

import requests  
import json  
from datetime import datetime  
import psycopg2  
import os  
import boto3  
  
# --- Database credentials will be loaded from Environment Variables ---  
# These are set in the Lambda configuration  
DB\_HOST = os.environ['RDS\_ENDPOINT']  
DB\_USER = os.environ['DB\_USER']  
DB\_NAME = os.environ['DB\_NAME']  
SECRET\_NAME = os.environ['SECRET\_NAME'] # e.g., 'prod/postgres/credentials'  
  
def get\_db\_password():  
 """Retrieves the database password from AWS Secrets Manager."""  
 session = boto3.session.Session()  
 client = session.client(service\_name='secretsmanager')  
   
 try:  
 get\_secret\_value\_response = client.get\_secret\_value(SecretId=SECRET\_NAME)  
 secret = get\_secret\_value\_response['SecretString']  
 return json.loads(secret)['password']  
 except Exception as e:  
 print(f"Error retrieving secret: {e}")  
 raise e  
  
def fetch\_greenhouse\_jobs(company\_name):  
 """Fetches job listings from a company's Greenhouse Job Board API."""  
 api\_url = f"[https://boards-api.greenhouse.io/v1/boards/](https://boards-api.greenhouse.io/v1/boards/){company\_name}/jobs?content=true"  
   
 try:  
 response = requests.get(api\_url, timeout=10)  
 response.raise\_for\_status()  
 data = response.json()  
 jobs = data.get('jobs', [])  
 for job in jobs:  
 job['company\_name'] = company\_name.capitalize()  
 return jobs  
 except requests.exceptions.RequestException as req\_err:  
 print(f"An error occurred for {company\_name}: {req\_err}")  
 except json.JSONDecodeError:  
 print(f"Failed to parse JSON for {company\_name}.")  
 return []  
  
  
def load\_jobs\_to\_db(job\_list, db\_password):  
 """Connects to the Postgres DB and 'upserts' jobs."""  
 if not job\_list:  
 print("No jobs to load.")  
 return  
  
 try:  
 print(f"Connecting to database: {DB\_HOST}...")  
 with psycopg2.connect(  
 host=DB\_HOST,  
 user=DB\_USER,  
 password=db\_password,  
 dbname=DB\_NAME  
 ) as conn:  
   
 with conn.cursor() as cur:  
 print(f"Successfully connected. Preparing to load {len(job\_list)} jobs...")  
   
 upsert\_query = """  
 INSERT INTO greenhouse\_jobs (  
 job\_id, company\_name, title, location, job\_url, updated\_at, raw\_data  
 )  
 VALUES (%s, %s, %s, %s, %s, %s, %s)  
 ON CONFLICT (job\_id) DO UPDATE SET  
 title = EXCLUDED.title,  
 location = EXCLUDED.location,  
 updated\_at = EXCLUDED.updated\_at,  
 raw\_data = EXCLUDED.raw\_data;  
 """  
   
 jobs\_processed = 0  
 for job in job\_list:  
 job\_data = (  
 job.get('id'),  
 job.get('company\_name'),  
 job.get('title'),  
 job.get('location', {}).get('name', 'N/A'),  
 job.get('absolute\_url'),  
 job.get('updated\_at'),  
 json.dumps(job) # Store the full JSON  
 )  
   
 # Filter for jobs posted this month before loading  
 current\_month = datetime.now().month  
 current\_year = datetime.now().year  
 job\_date = datetime.fromisoformat(job.get('updated\_at').replace('Z', '+00:00'))  
  
 if job\_date.year == current\_year and job\_date.month == current\_month:  
 cur.execute(upsert\_query, job\_data)  
 jobs\_processed += 1  
  
 conn.commit()  
 print(f"Database load complete. {jobs\_processed} jobs from this month were inserted/updated.")  
   
 except Exception as e:  
 print(f"Database connection error: {e}")  
 raise e  
  
def lambda\_handler(event, context):  
 """  
 Main entry point for the AWS Lambda function.  
 'event' and 'context' are passed by AWS.  
 """  
 companies\_to\_scrape = [  
 "coinbase", "stripe", "notion", "airbnb", "uber", "lyft",   
 "figma", "plaid", "brex", "canva"  
 ]  
   
 all\_jobs = []  
   
 print("--- Starting Job Fetch ---")  
 for company in companies\_to\_scrape:  
 print(f"Fetching jobs for: {company}...")  
 jobs = fetch\_greenhouse\_jobs(company)  
 if jobs:  
 all\_jobs.extend(jobs)  
 print(f"--- Job Fetch Complete: {len(all\_jobs)} total jobs found ---\n")  
   
 # Get password and load data  
 try:  
 password = get\_db\_password()  
 load\_jobs\_to\_db(all\_jobs, password)  
   
 return {  
 'statusCode': 200,  
 'body': json.dumps(f'Successfully processed {len(all\_jobs)} jobs.')  
 }  
 except Exception as e:  
 print(f"Pipeline failed: {e}")  
 return {  
 'statusCode': 500,  
 'body': json.dumps(f'Error: {e}')  
 }