An interim report for a dissertation that will be submitted in partial fulfilment of a University of Greenwich Master's Degree

Big Data-Driven Geospatial and Predictive Analysis of UK IT Employment Trends

Venkata Samba Siva Reddy Chilamuru

001410264

MSc Data Science

Date Proposal Submitted:

Submission Date:

Supervisor: Jia Wang

Topic Area: Big Data Analytics, Data Science, Geospatial Analysis, Employment Market Analysis

Keywords associated with the project: Big Data Analytics, Geospatial Data Analysis, UK IT Employment Trends, Machine Learning for Job Market Insights, GIS for Labour Market Analysis, Employment Pattern Visualization.

MSc Modules studied that contribute towards this project: Machine learning, Data Visualization, Big data analysis

```
import json
import boto3
import requests
from bs4 import BeautifulSoup
import csv
from datetime import datetime
import time
import re
from io import StringIO
import os
import psycopg2
from psycopg2.extras import execute_batch
from time import sleep
from typing import Tuple, Optional
```

```
class LinkedInRecentITJobsScraper:
def __init__(self):
self.headers = {
'User-Agent': 'Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/537.36 (KHTML, like Gecko)
Chrome/120.0.0.0 Safari/537.36',
'Accept': 'text/html,application/xhtml+xml,application/xml;q=0.9,image/webp,*/*;q=0.8',
'Accept-Language': 'en-US,en;q=0.5',
'Connection': 'keep-alive',
}
```

These headers make your script look like a **real user/browser** so LinkedIn doesn't block it as a bot.

Key Elements:

- **User-Agent**: Pretends to be a real Chrome browser — helps avoid bot detection.
- **Accept & Accept-Language**: Tells the server what type of content/language we expect.
- **Connection: keep-alive**: Keeps the connection open for better performance during scraping.

"In the constructor, I defined browser-like headers to make my scraper look like a normal user. This helps avoid getting blocked by LinkedIn and ensures I receive the correct HTML content when I send requests."

```
self.it_jobs = [
"Cloud Engineer",
"Full Stack Developer",
"Backend Developer",
"Frontend Developer",
"Machine Learning Engineer"
]
```

What It Does:

This is a list of **specific IT job titles** you want to search for on LinkedIn.

Why It's Needed:

When the scraper runs, it loops through each of these job titles and performs a separate LinkedIn search. This makes your results focused only on **in-demand IT roles**.

```
# Add database configuration
self.db_config = {
'dbname': os.environ['DB_NAME'],
'user': os.environ['DB_USER'],
'password': os.environ['DB_PASSWORD'],
'host': os.environ['DB_HOST'],
'port': os.environ['DB_PORT']
}
```

What It Does:

It loads the **PostgreSQL connection details** from environment variables, not directly in the code.

Key	Description
DB_NAME	Your database name
DB_USER	Username to log in
DB_PASSWORD	Secure password
DB_HOST	Hostname or IP address (like RDS endpoint)
DB_PORT	Usually 5432 for PostgreSQL

Why It's a Good Practice:

- Keeps your **credentials safe** (not hardcoded)
- Lets you **deploy securely** on Lambda or servers
- Makes your code **portable** (easily moved to another machine/environment)

"I defined a list of IT job roles I want to target like Cloud Engineer and ML Engineer. Then I set up my database connection details using environment variables so I can store the scraped data into PostgreSQL securely and without hardcoding sensitive information."

```
self.geocoding_cache = {}  # Simple in-memory cache
```

This creates an **empty dictionary** that you'll use to store **geocoding results** (latitude and longitude for cities/locations).

When your scraper later calls the function get_coordinates(location), it:

- Checks this cache first to see if the lat/lon for a location is already available.
- If not, it sends a request to the **OpenStreetMap Nominatim API** to fetch it.

Once fetched, it stores the result in this cache so it doesn't call the API again for the same location.

Why This Is Smart:

- **Reduces API calls** (important because Nominatim is rate-limited to 1 request/second).
- **Speeds up your scraping** process.
- **Avoids hitting limits or getting blocked** by the geocoding service.

```
def clean_location(self, location):
"""Clean location string to extract only city name"""
location = re.sub(r', England, United Kingdom$', '', location)
location = re.sub(r', United Kingdom$', '', location)
location = re.sub(r', UK$', '', location)
location = re.sub(r' Area, United Kingdom$', '', location)
location = re.sub(r' Area$', '', location)
location = re.sub(r'Greater ', '', location
if ',' in location:
location = location.split(',')[0].strip()
return location
```

Purpose:

This function **cleans and standardizes messy location strings** from LinkedIn job listings. The goal is to extract just the **main city name** — so that it's easier to group and analyze later (especially for maps).

If LinkedIn gives you a location like:

"Greater London Area, United Kingdom" → This method turns it into just **"London"**

- Removes ", England, United Kingdom" if it's at the end
- Removes other common suffixes that clutter the location
- Keeps only the first part if there's a comma — in case it's like "London, England"

All locations will be cleaned to just the **city name**, which helps in:

- Grouping jobs by city
- Geocoding correctly
- Displaying clean names on maps or dashboards

"LinkedIn often gives location names in long or inconsistent formats, like 'Greater London Area, UK'. So I created a function to clean those strings and extract just the city name, like 'London'. This makes the data more accurate for grouping and mapping job locations."

```
def determine_experience_level(self, job_description):
"""Determine if the job is entry-level, mid-level, or senior"""
text = job_description.lower()

if any(word in text for word in ['senior', 'lead', 'principal', 'manager', 'head of']):
return 'Senior Level'
elif any(word in text for word in ['junior', 'entry', 'graduate', 'trainee']):
return 'Entry Level'
else:
return 'Mid Level'
```

determine_experience_level(self, job_description)

This function checks the job description to figure out whether the job is:

- **Entry Level**
- **Mid Level**
- **Senior Level**

How:

- Converts the description to lowercase
- Looks for keywords like:
  - "senior", "lead", "manager" → **Senior Level**

- "junior", "entry", "graduate" → **Entry Level**
- If neither appears → **Mid Level** by default

"I used simple keyword matching to classify the job into entry, mid, or senior level based on words in the job description. It helps filter job roles more accurately in my dashboard."

```
def is_remote(self, job_description):
"""Check if the job is remote"""
text = job_description.lower()
if any(word in text for word in ['remote', 'work from home', 'wfh', 'hybrid']):
if 'hybrid' in text:
return 'Hybrid'
return 'Remote'
return 'On-site'
```

is_remote(self, job_description)
Determines whether the job is:

- **Remote**
- **Hybrid**
- **On-site**

How:

- Looks for keywords like "remote", "work from home", "hybrid" in the description
- If "hybrid" is found, returns "Hybrid"
- If any of the others are found, returns "Remote"
- If none of those are present → returns "On-site"

"I created a function to classify the work type — whether it's remote, hybrid, or on-site — based on keywords found in the job description. This lets me show work-style trends in my dashboard."

```
def is_recent_job(self, posted_date_text):
"""Check if the job was posted recently (within last 3 days)"""
text = posted_date_text.lower()
if 'hours ago' in text or 'hour ago' in text:
return True
elif 'day ago' in text or '1 day ago' in text:
return True
elif '2 days ago' in text:
return True
elif '3 days ago' in text:
return True
return False
```

is_recent_job(self, posted_date_text)
Filters out **only the most recent jobs** (last 3 days).
How:

- Checks if the LinkedIn job listing says things like:

"1 day ago", "2 days ago", "3 days ago", or "hours ago"

- If yes → returns True (keep the job)
- Otherwise → False (skip it)

“To make sure I’m collecting only fresh job data, I wrote a filter to keep jobs posted in the last 3 days. This helps me keep the dataset real-time and relevant.”

def get_coordinates(self, location: str) -> Tuple[Optional[float], Optional[float]]:

This function **gets the latitude and longitude** for a given city/location using the **OpenStreetMap Nominatim API**.
Why?
You need coordinates to:

- **Map jobs** on a UK map
- Group/filter jobs by region
- Create location-based visualizations in Power BI

if not location:
return None, None

Check if location is empty
If no location is provided, it returns None for both lat and lon.

# Check cache first
if location in self.geocoding_cache:
return self.geocoding_cache[location]
This avoids unnecessary API calls — saves time and prevents rate limit issues.

try:
# Add UK to the location query to improve accuracy
search_location = f"{location}, United Kingdom"

# Using Nominatim API
url = https://nominatim.openstreetmap.org/search

Set up the API request
This formats the location and targets the **Nominatim API** from OpenStreetMap.

headers = {
'User-Agent': 'LinkedInJobScraper/1.0', # Required by Nominatim
'Accept-Language': 'en-US,en;q=0.9',
}
params = {
'q': search_location,
'format': 'json',
'limit': 1
}
Set up the API request
This formats the location and targets the **Nominatim API** from OpenStreetMap.

```
response = requests.get(url, headers=headers, params=params)
sleep(1)  # Respect rate limit - 1 request per second
```

Send the request

- Sends a GET request
- Adds sleep(1) to avoid being blocked (Nominatim allows 1 request/second)

```
if response.status_code == 200:
results = response.json()
if results:
lat = float(results[0]['lat'])
lon = float(results[0]['lon'])
```

Parse the result

If the response is successful and has data, extract latitude and longitude.

```
# Cache the result
self.geocoding_cache[location] = (lat, lon)
return lat, lon
```

Store it in cache

This saves the coordinates so next time it won't call the API again for the same location.

```
# Cache negative result
self.geocoding_cache[location] = (None, None)
return None, None
except Exception as e:
print(f"Error geocoding location '{location}': {str(e)}")
return None, None
```

If no result, cache and return None

"This function gets latitude and longitude for each job location using the OpenStreetMap API. I add ', United Kingdom' to help accuracy, and then parse the JSON response. I also added caching so the API isn't called repeatedly for the same location — which saves time and avoids hitting usage limits. These coordinates are later used to plot jobs on a map in Power BI."

```
def scrape_linkedin_jobs(self):
```

This function **loops through all your IT job titles**, goes to LinkedIn search results, and **scrapes the job listings** that were posted in the **last 3 days only**.

```
all_jobs = []
max_pages_per_category = 1  # Reduced for Lambda execution time constraints
for job_title in self.it_jobs:
print(f"\nScraping recent jobs for: {job_title}")
page = 0
consecutive_old_jobs = 0
```

Loop Through Each Job Title

- You have a list like "Cloud Engineer", "ML Engineer", etc.
- Each one gets searched separately on LinkedIn.

```
while page < max_pages_per_category:
try:
encoded_title = requests.utils.quote(job_title)
url =
f"https://www.linkedin.com/jobs/search/?keywords={encoded_title}&location=United%20Ki
ngdom&start={page*25}&f_TPR=r86400"
print(f"Scraping page {page + 1}...")
```

Set Page Limit

- Limits scraping to **1 page per role** to keep it **Lambda-friendly and fast**.
- Each page has around **25 jobs**.

Build Search URL

- This **encodes the job title** so it can be safely used in a URL.
- For example:
  - "Cloud Engineer" becomes "Cloud%20Engineer"
- If you don't encode it, spaces or special characters can break the URL.

"I used URL encoding to safely include the job title in the LinkedIn search URL."

- This builds the **actual LinkedIn job search URL** using:
  - keywords={encoded_title} → the role to search (e.g., "Cloud Engineer")
  - location=United%20Kingdom → restricts search to the UK
  - start={page*25} → for pagination (page 1 = 0, page 2 = 25, etc.)
  - f_TPR=r86400 → filter for jobs posted in the last **24 hours**

"I dynamically built the LinkedIn URL with job title, UK location, and a filter to only get recent jobs."

print(f"Scraping page {page + 1}...")

- Just a log message that tells which page the scraper is on
- Helpful for debugging or watching the scraping process in real time

"This line just prints which page is being scraped — useful for tracking progress or debugging."

response = requests.get(url, headers=self.headers)

- This **sends a GET request** to LinkedIn using the search URL built earlier.
- headers=self.headers makes the request look like it's coming from a browser (to avoid being blocked).

"I sent a GET request to the LinkedIn job search URL, using headers to make it look like a real browser visit."

```
if response.status_code != 200:
print(f"Failed to fetch page {page + 1}. Status code: {response.status_code}")
break
soup = BeautifulSoup(response.text, 'html.parser')
```

soup = BeautifulSoup(response.text, 'html.parser')

- Parses the HTML content of the page using **BeautifulSoup**.
- This turns the raw HTML into a structured format so we can extract data easily

"I used BeautifulSoup to turn the LinkedIn HTML page into something I can search through programmatically."

job_cards = soup.find_all('div', class_='base-card')

- **job_cards = soup.find_all('div', class_='base-card')**
  - Searches for all the job listings on the page.
  - LinkedIn job cards are stored in elements with the class "base-card".

"I searched the parsed HTML for all job listings by finding divs with the class 'base-card'."

```
if not job_cards:
print("No more jobs found on this page")
break
```

if not job_cards:

- If no job listings were found (maybe the page is empty), it prints a message and **stops** scraping further pages.
  "If there are no jobs found, I stop scraping to save time and resources."

recent_jobs_found = False

- **recent_jobs_found = False**
  - This is a flag to track if **any recent jobs** were found on the page.
  - If not, it may stop the loop after a few consecutive old pages.
    "I reset the flag for checking if this page has any recent job listings."

```
for job in job_cards:
try:
posted_date = job.find('time', class_='job-search-card__listdate')
if posted_date:
posted_date_text = posted_date.text.strip()
if not self.is_recent_job(posted_date_text):
continue
```

Check Job Posting Date

- Extracts the posting date (e.g., "1 day ago", "2 days ago").
- If it's **not within 3 days**, skip this job using continue.

"I checked the posted date of each job and only continued if it was posted in the last 3 days."

```
recent_jobs_found = True
```

Mark That Recent Jobs Were Found

- Keeps track that at least one recent job was found on this page.

```
title = job.find('h3', class_='base-search-card__title').text.strip()
company = job.find('h4', class_='base-search-card__subtitle').text.strip()
raw_location = job.find('span', class_='job-search-card__location').text.strip()
```

Extract Job Title, Company, and Location

- Gets the job **title**, **company name**, and **raw location**
- Then uses clean_location() to clean it up (e.g., "Greater London Area" → "London")

"I extracted the job's title, company, and location — and cleaned the location to make it suitable for mapping."

```
location = self.clean_location(raw_location)
job_url = job.find('a', class_='base-card__full-link')['href']
```

Get the Job URL

- Extracts the **direct URL** to the full job posting.

```
job_response = requests.get(job_url, headers=self.headers)
job_soup = BeautifulSoup(job_response.text, 'html.parser')
job_description = job_soup.find('div', class_='show-more-less-html__markup').text.strip() if
job_soup.find('div', class_='show-more-less-html__markup') else ""
```

Visit the Job Page to Get Description

- Visits the job's detailed page
- Parses it with BeautifulSoup
- Tries to extract the full **job description text**

"I visited each job's detailed page to get the full description — which I use to classify experience level and work type."

```
experience_level = self.determine_experience_level(job_description)
work_type = self.is_remote(job_description)
```

Classify Experience Level and Work Type

Uses two functions you already defined:

- determine_experience_level() → Entry, Mid, or Senior
- is_remote() → On-site, Remote, or Hybrid

```
all_jobs.append({
'Job Title': title,
'Company': company,
'Location': location,
'Experience Level': experience_level,
'Work Type': work_type,
'Category': job_title,
'Posted Date': posted_date_text if posted_date else 'Recently',
'Job URL': job_url,
'Date Scraped': datetime.now().strftime("%Y-%m-%d %H:%M:%S")
})

except Exception as e:
print(f"Error parsing job: {str(e)}")
continue

if not recent_jobs_found:
consecutive_old_jobs += 1
if consecutive_old_jobs >= 2:
break
else:
consecutive_old_jobs = 0

page += 1
time.sleep(1)  # Reduced sleep time for Lambda

except Exception as e:
print(f"Error fetching page: {str(e)}")
break

return all_jobs
```

Add Everything to all_jobs[]

- Adds it to the all_jobs list
- Includes the **scrape timestamp** for tracking

"Finally, I saved all the job details into a dictionary and added it to the list of results, so I can later export them to S3 or my database."

```
def save_to_s3(self, jobs):
```

This function saves all the job data to a CSV file and uploads it to AWS S3 cloud storage.

```
if not jobs:
print("No jobs to save")
return
```

Check if there's any data to save

- If no job data is available, it exits early.

"First, I check if there are any jobs to save — if not, I skip the process."

```
try:
# Create CSV in memory
csv_buffer = StringIO()
fieldnames = ['Job Title', 'Company', 'Location', 'Latitude', 'Longitude',
'Experience Level', 'Work Type', 'Category', 'Posted Date',
'Job URL', 'Date Scraped']
```

Create an in-memory CSV file

- StringIO() lets you create a **CSV file in memory** (not on disk).
- DictWriter writes each job as a row using the field names.

"I use StringIO to create the CSV in memory — which is needed for Lambda functions since we don't use local files."

```
# Add coordinates to jobs data
jobs_with_coordinates = []
for job in jobs:
job_copy = job.copy()
lat, lng = self.get_coordinates(job['Location'])
job_copy['Latitude'] = lat
job_copy['Longitude'] = lng
jobs_with_coordinates.append(job_copy)
time.sleep(0.1)
# Rate limiting
writer = csv.DictWriter(csv_buffer, fieldnames=fieldnames)
writer.writeheader()
writer.writerows(jobs_with_coordinates)
```

Add Coordinates to Each Job

- For each job, it fetches the **latitude and longitude** using your get_coordinates() function.
- Adds this to the job dictionary.

"Before saving, I enrich each job entry with latitude and longitude — so it can be visualized on a map later."

```
# Upload to S3
s3 = boto3.client('s3')
bucket_name = os.environ['S3_BUCKET_NAME']
timestamp = datetime.now().strftime("%Y%m%d_%H%M%S")
file_name = f'linkedin_recent_it_jobs_{timestamp}.csv'

s3.put_object(
Bucket=bucket_name,
```

```
Key=file_name,
Body=csv_buffer.getvalue(),
ContentType='text/csv'
)
```

Upload the CSV to S3

- Connects to AWS S3 using the boto3 library
- Gets the bucket name from an environment variable
- Creates a unique file name using a timestamp
- Uploads the CSV file content to the specified S3 bucket and key

"I connect to my S3 bucket and upload the CSV file with a timestamped name. This keeps each upload separate and easy to track."

```
print(f"\nSaved {len(jobs)} jobs to S3: {bucket_name}/{file_name}")
return f"s3://{bucket_name}/{file_name}"
```

Print Confirmation

- Displays a message confirming how many jobs were saved and where.

```
except Exception as e:
print(f"Error saving to S3: {str(e)}")
return None
```

If There's an Error?

- If something goes wrong during upload, it logs the error and returns None.

"After scraping the jobs, I convert the data into a CSV file in memory, add geolocation info, and upload it to an S3 bucket using the boto3 library. Each file is named with a timestamp to make it unique. This makes my project scalable and cloud-based."

```
def save_to_postgres(self, jobs):
```

This function connects to a PostgreSQL database, creates a table (if it doesn't exist), and inserts the scraped job data into it.

```
if not jobs:
print("No jobs to save to database")
return
```

Check if there are any jobs to save

If no job data exists, it exits early to avoid unnecessary database operations.

```
conn = None
cur = None
try:
print("Connecting to database...")
conn = psycopg2.connect(**self.db_config)
cur = conn.cursor()
```

Connect to the PostgreSQL database

- Connects using credentials stored in self.db_config (which are loaded from environment variables).
- conn is the connection, and cur is the cursor used to run SQL commands.

"I connect to the PostgreSQL database securely using credentials from environment variables."

```
# Modified table creation to include latitude and longitude
create_table_query = """
CREATE TABLE IF NOT EXISTS linkedin_jobs (
id SERIAL PRIMARY KEY,
job_title VARCHAR(255),
company VARCHAR(255),
location VARCHAR(255),
latitude DECIMAL(10, 8),
longitude DECIMAL(11, 8),
experience_level VARCHAR(50),
work_type VARCHAR(50),
category VARCHAR(100),
posted_date VARCHAR(100),
job_url TEXT,
date_scraped TIMESTAMP,
created_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP
);
"""
```

Create the table (if it doesn't exist)

- Ensures the linkedin_jobs table exists with all necessary columns
- Includes latitude and longitude for geospatial work
- Adds created_at timestamp for audit purposes

"I created a table structure that includes job details and location coordinates, so I can do geospatial analysis and track data changes over time."

```
cur.execute(create_table_query)

# Add unique constraint if it doesn't exist
try:
cur.execute("""
ALTER TABLE linkedin_jobs
ADD CONSTRAINT unique_job_url UNIQUE (job_url);
""")
```

Add unique constraint on job_url

- Prevents duplicate entries of the same job (based on job URL)

"I added a unique constraint on the job URL to avoid inserting the same job multiple times."

```
except psycopg2.errors.DuplicateTable:
conn.rollback()

conn.commit()

insert_query = """
INSERT INTO linkedin_jobs
(job_title, company, location, latitude, longitude, experience_level, work_type,
category, posted_date, job_url, date_scraped)
VALUES (%s, %s, %s, %s, %s, %s, %s, %s, %s, %s, %s)
ON CONFLICT (job_url) DO NOTHING
"""
```

Insert query for new jobs

- This safely inserts each job into the table.
- If the job already exists (same URL), it skips it (no error).

"I built this function to store my scraped jobs in a PostgreSQL database with proper structure, location mapping, and uniqueness checks. It helps with long-term storage, querying, and integration with analytics tools."

```
# Modify job data to include coordinates
job_data = []
for job in jobs:
lat, lng = self.get_coordinates(job['Location'])
job_data.append((
job['Job Title'],
job['Company'],
job['Location'],
lat,
lng,
job['Experience Level'],
job['Work Type'],
job['Category'],
job['Posted Date'],
job['Job URL'],
datetime.strptime(job['Date Scraped'], "%Y-%m-%d %H:%M:%S")
))
# Add a small delay to avoid hitting API rate limits
time.sleep(0.1)
```

- Loops through each job
- Gets its **coordinates**
- Packs all data into a **tuple** (ordered values) and stores them in job_data[]
- Adds a **small delay** (0.1 sec) between requests to avoid overloading the geocoding API

```
execute_batch(cur, insert_query, job_data)
conn.commit()
```

- Inserts all job records into PostgreSQL in **bulk** using execute_batch() — this is faster than inserting one-by-one.
- Commits the changes.

"I prepared a list of job data with location coordinates, then used execute_batch to insert everything efficiently into my PostgreSQL table. I also added a delay to avoid hitting the location API too fast."

```
print(f"\nSaved {len(jobs)} jobs to PostgreSQL database")
except Exception as e:
print(f"Error saving to PostgreSQL: {str(e)}")
if conn:
conn.rollback()
finally:
if cur:
cur.close()
if conn:
conn.close()
```

Error Handling and Cleanup:

- If there's an error: print it and roll back the database changes
- Always closes the DB connection (good practice)

```
def lambda_handler(event, context):
```

This is the **function AWS automatically runs** when triggered (e.g., by EventBridge on a schedule).

```
try:
print("Starting LinkedIn job scraper...")
scraper = LinkedInRecentITJobsScraper()
jobs = scraper.scrape_linkedin_jobs()
```

Start the scraper

"I create the scraper object and run the scraping function."

```
# Save to S3
s3_path = scraper.save_to_s3(jobs)

# Save to PostgreSQL
scraper.save_to_postgres(jobs)
```

Saves job data to:
AWS S3 as CSV
PostgreSQL as structured records

```
# Prepare summary
categories = {}
for job in jobs:
category = job['Category']
if category not in categories:
categories[category] = 0
categories[category] += 1

summary = {
'total_jobs': len(jobs),
'jobs_by_category': categories,
'output_file': s3_path,
'timestamp': datetime.now().strftime("%Y-%m-%d %H:%M:%S")
}
```

Prepare a summary report

- Builds a breakdown by job category (e.g., how many ML jobs, backend jobs, etc.)
- Includes total jobs, file path in S3, and timestamp

```
return {
'statusCode': 200,
'body': json.dumps(summary)
}
```

Return the response

- Returns a success response with a summary payload (for logs or API integrations)

```
except Exception as e:
```

If there's an error:

- Captures any issues during scraping, saving, or uploading and returns a 500 error with details.

```
print(f"Error in lambda execution: {str(e)}")
return {
'statusCode': 500,
'body': json.dumps({
'error': str(e)
})
}
```

“The lambda_handler() is the entry point AWS uses. It runs the full pipeline: scraping jobs, saving them to S3 and PostgreSQL, then returning a summary. If anything goes wrong, it catches the error and prints a message — otherwise, it gives back how many jobs were processed and where the data is saved.”