**Assistance of AI with Rearc Data Quest challenge**

I used AI tool as a reference and explanation tool to assist in debugging my Python scripts, AWS S3 integrations.

1. **Part 1 – AWS S3 & Sourcing Datasets**

**Prompts that I used –**

* Explain how to upload a dataset from a public URL to an AWS S3 bucket using Python
* I am thinking of using below python libraries for my script, does it look correct?
* Simplify the logic for connecting to the open dataset and uploading to S3

**What It Got Wrong or Needed Correction –**

* Initially suggested incorrect S3 upload method (upload\_fileobj instead of upload\_file for local files)
* Provided code that didn’t automatically handle region specification (us-east-2), which I had to correct
* Missed creating the target folder structure in S3 🡪 I manually fixed it
* Some explanations were overly generic, so I rewrote parts in my own words to match my understanding
* **Over-simplified HTML parsing → 0 files discovered**
  + **AI assumption:** a basic href="..." regex would capture all links.
  + **Reality:** the BLS index sometimes uses single quotes, relative paths (./pr.series), subpaths (pr/pr.series), and includes query/folder links.
  + **Impact:** discovery returned an empty list → nothing uploaded → user saw empty S3 bucket and “0 files”.
  + **Fix:** A more tolerant parser that -
    - accepts ' **or** " quotes,
    - normalizes relative paths,
    - ignores query/folder links,
    - filters to pr.\* names.
* **Ignoring provider anti-scraping behavior → 403 on directory index**
  + **AI assumption:** setting a simple custom User-Agent is enough.
  + **Reality:** BLS occasionally returns **403** for non-browser clients hitting the **directory index**, even with a UA. However, direct file URLs often still work.
  + **Impact:** the “list” step failed silently or looped retries; notebooks looked like they were “still running”.
  + **Fixes:**
    - Use a **browser-like requests.Session** (headers + keep-alive).
    - Add **retries** for 429/5xx.
    - Set **connect/read timeouts** and a small **rate limit**.
    - Add a **fallback list of known pr.\* filenames** when the index can’t be parsed. We then fetch files directly (which succeeds even when index is blocked).

**What I Did independently -**

* Created Free tier AWS account
* Manually created and configured the AWS S3 bucket rearc-dataquest-harpreet and IAM user
* Created IAM User as rearc-dataquest-user and attach permissions policies as [AmazonS3FullAccess](https://us-east-1.console.aws.amazon.com/iam/home?region=us-east-2#/policies/details/arn%3Aaws%3Aiam%3A%3Aaws%3Apolicy%2FAmazonS3FullAccess) (lets your script create/read/write objects) and [CloudWatchLogsFullAccess](https://us-east-1.console.aws.amazon.com/iam/home?region=us-east-2#/policies/details/arn%3Aaws%3Aiam%3A%3Aaws%3Apolicy%2FCloudWatchLogsFullAccess) (optional, helpful for debugging Lambda later) to it
* Fetch Access Key and Secret Access Key for this IAM User and stored it locally as CSV.
* Configured my local computer to talk to AWS services using this Access Key and Secret Access Key using Powershell script as –

pip install awscli

aws configure

* Tested code locally in Jupyter Notebook to verify file upload success.
* Added exception handling and logging based on my AWS and Python knowledge.
* Rewrote the explanations and markdown comments in my own style.

**If I had another week, I would do below things -**

* Emit a **run manifest** (manifest-YYYYMMDDTHHMMSSZ.json) to S3 with file actions & hashes
* Add **S3 Inventory**/Athena table to track object metadata over time
* Build a small **CLI** + **Lambda** (EventBridge schedule) and alert on diff anomalies (new/missing files)
* Add **SSE-KMS** support via flag for stricter orgs

**What the AI Should Have Done Differently**

* **Probe first, assume later**  
  Start with a 3-step diagnostic (index fetch, single file fetch, S3 list). Adjust strategy based on results.
* **Design for hostile/variable HTML**  
  Use tolerant parsing or offer a fallback path when the index is inaccessible.
* **Use browser-like headers for public data portals**  
  Pair with retries, timeouts, and small rate limits.

1. **Part 2 – Working with the DataUSA API Dataset (Data Analytics)**

**How I Used AI**

For this part, I wanted to fetch open data directly from the **DataUSA API** and publish it to my S3 bucket. I used AI mostly as a guide to understand the right way to call an API, handle JSON responses and make the code more readable.  
Some of the prompts I used were things like:

* How to connect to the DataUSA API and store the results in a JSON file?
* Show me how to handle API pagination and check for response status codes in Python
* What’s a clean way to upload this API output to S3?

AI explanations helped me understand the flow — fetch the data → validate → save → upload and it suggested using the requests and boto3 libraries, which made sense for this task.

**What It Got Wrong or Needed Tuning**

The AI’s code examples worked, but they weren’t perfect. A few things I noticed:

* It sometimes assumed the API endpoint returned raw JSON, while in reality DataUSA wraps data inside a "data" object
* The code didn’t include error handling for bad responses (e.g., 404 or 500), so I added try/except and raise\_for\_status() myself
* It suggested saving the file without UTF-8 encoding, which caused formatting issues
* It didn’t automatically convert the API results into a Pandas DataFrame for readability, I added that step myself

**What I Did Myself**

I built a simple but clean Python script that -

* Fetches population data from the DataUSA API endpoint.
* Validates that the JSON response contains the expected keys.
* Saves the results to a file called population\_data.json.
* Uploads that file to my S3 bucket (rearc-dataquest-harpreet) with proper logging.

I also added a small delay between requests to avoid rate-limit errors.

**Reflection**

This part really helped me get comfortable with **API integrations and error handling**. AI gave me a direction but I had to debug and polish the logic to make it production-friendly.

1. **Part 3 – Fetching and Uploading Multiple Files from the BLS Dataset**

**How I Used AI**

In Part 3, I had to crawl multiple files from the **BLS** open dataset, extract links and upload everything to S3. I used AI to design the logic step-by-step — especially how to list files programmatically and handle relative links.  
Prompts I used included -

* How to scrape multiple file URLs from a public directory index using Python?
* Explain how to download each file and upload to S3
* Why does my requests.get loop keep running forever?

**What Went Wrong**

AI initially suggested a **very naive HTML parser**, using only a regex for href="...", which failed because the BLS index used both single and double quotes, relative paths like ./pr.series, and even nested folders.  
As a result, my first few runs uploaded **zero files** - the script looked stuck because it was retrying endlessly.

It also missed a subtle point: the BLS site sometimes blocks automated requests (403 errors) unless you mimic a browser. AI didn’t catch that.

**My Fixes**

I improved the parsing logic myself -

* Used requests.Session() with browser-like headers and a short delay between requests
* Allowed both ' and " in href patterns
* Normalized relative paths with urljoin
* Ignored query and folder links
* Added retries for HTTP 429 and 5xx errors

After these fixes, I was able to download all valid .series files and upload them to my S3 bucket successfully.

**Reflection**

This part taught me that **blindly trusting AI-generated code for web scraping isn’t safe** -you have to test, inspect and adapt. It was a good debugging experience.

1. **Part 4 – Automating with Terraform**

**How I Used AI**

For the final part, I used AI mainly to help me automate S3 bucket and IAM user creation.  
Prompts I used included -

* Show me a basic Terraform script to create an S3 bucket and attach IAM permissions
* How to store my Terraform state safely?

**What It Got Wrong or Missed**

AI’s Terraform examples were generally correct but slightly outdated in syntax. For example, it used older provider versions and missed a few required arguments like bucket region alignment with my existing resources.  
Also, the AI didn’t clearly mention that running terraform apply will **actually create resources in AWS**, which can incur cost — I verified that manually before applying.

**What I Did Myself**

I customized the script so that -

* It matched my bucket name (rearc-dataquest-harpreet)
* Used the correct region (us-east-2)
* Applied tags for identification
* Verified the terraform plan output before deployment
* Stored my credentials securely in terraform.tfvars

I ran both terraform validate and terraform plan successfully and confirmed that the configuration was error-free, though I didn’t apply it to avoid costs.

**Reflection**

Part 4 made me appreciate how **infrastructure-as-code** works in real projects. AI helped me understand the structure but I learned to review and reason through each resource definition myself.

1. **Final Thoughts**

Across all parts, AI was like a **helpful co-pilot** since it accelerated learning and debugging but still required my judgment to correct assumptions and handle edge cases.  
I especially liked how it helped me reason through AWS and Terraform concepts, even if I had to refine the implementation for real-world correctness.