Project Documentation

End-to-End Automated Weekly Data Pipeline with Visual Reporting

From API to Dashboard in Metabase



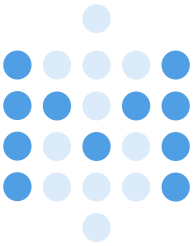
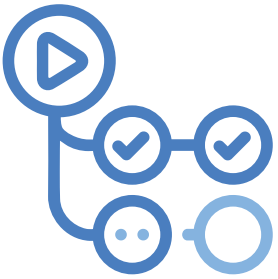



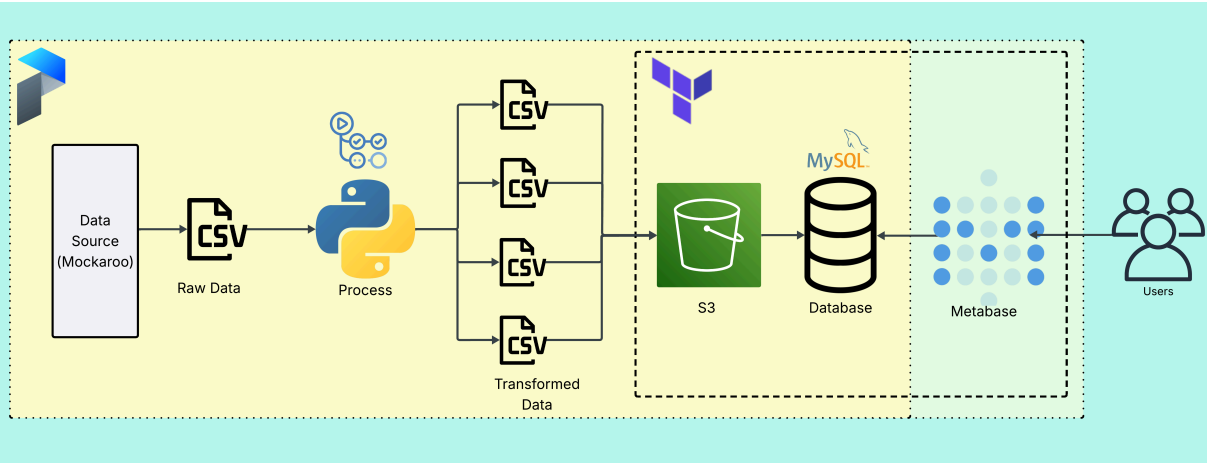
Table Of Contents

Project Overview

This project is designed to automate the process of generating, transforming, storing, and visualizing synthetic business data. The pipeline integrates external data generation, automated transformation, cloud-based storage, and business intelligence tooling into a seamless workflow. The infrastructure is deployed and managed using Terraform to ensure consistency, scalability, and reproducibility.

The goal is to enable a reliable data pipeline that can support analytical queries and dashboards through a self-hosted Metabase instance. This setup offers a robust foundation for internal analytics, experimentation, and data-driven decision-making.

Flow



- Data is generated using the **Mockaroo API** and retrieved in CSV format.
- A Python script processes the raw data, performing cleaning, transformation, and structuring operations.

- The transformed data is split into multiple CSV files, each representing a specific database table.
- **Prefect is used as the orchestration tool** to coordinate this process. It schedules and manages the execution of tasks such as data fetching, transformation, and uploading.
- These CSV files are uploaded to an Amazon S3 bucket which serves as a staging layer.
- A scheduled process running within the cloud environment fetches the CSV files from S3.
- The data is mapped and loaded into a MySQL database, aligning with the defined schema.
- A Metabase instance, connected to the database, allows users to query and visualize the data through dashboards and reports.

By streamlining the entire data lifecycle, this solution enhances operational efficiency, improves data reliability, and supports faster, insight-driven decisions, all while reducing the manual workload on technical teams.

Project Goal and Business Impact

The primary goal of this project is to build a secure, modular data pipeline that demonstrates how dashboards and reports can be automatically updated as new data becomes available. The system is designed to handle the full data flow, from ingestion and transformation to staging and loading, ensuring that data is consistently structured and delivered to a centralized database for analysis.

The orchestration of these steps reduces manual effort, increases reliability, and provides a clear example of how automated processes can support ongoing data operations without introducing unnecessary complexity.

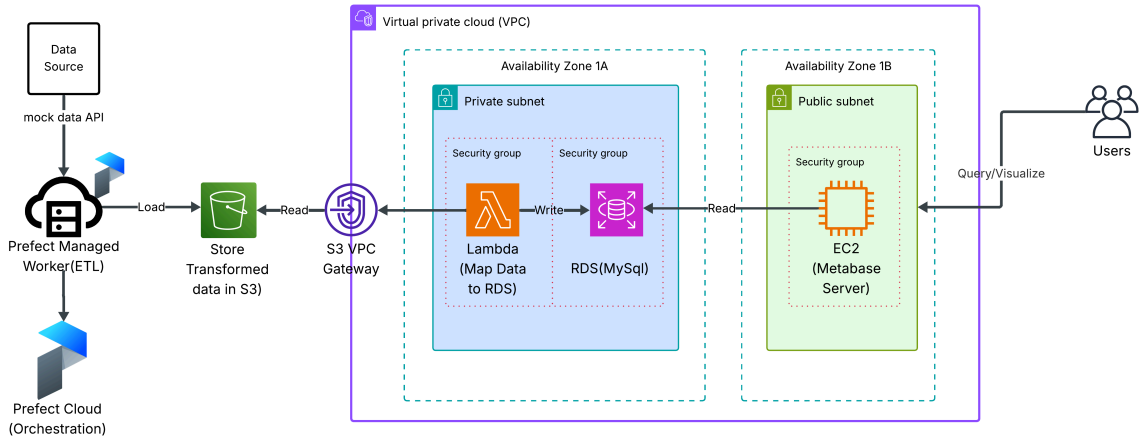
From a business perspective, this solution helps stakeholders:

- Understand how reporting systems can stay aligned with changing data in near real time

- Strengthen data processes by reducing the chances of missed updates or inconsistencies
- Improve collaboration between technical and non-technical teams by making insights more accessible through intuitive dashboards

This setup improves data visibility, supports timely decision-making, and strengthens the overall responsiveness of reporting systems. It is intended to complement existing workflows by helping teams access accurate and current information with less manual effort.

Architecture Overview



Services used

- **Mockaroo**
- **Prefect**
- **Amazon S3**
- **AWS Lambda**
- **Amazon RDS (MySQL)**
- **Amazon EC2**

1. **Prefect Cloud & Managed Worker**
    - Handles orchestration and execution of ETL jobs.

- Fetches data from the **mock data API**, transforms it, and loads it into **Amazon S3**.

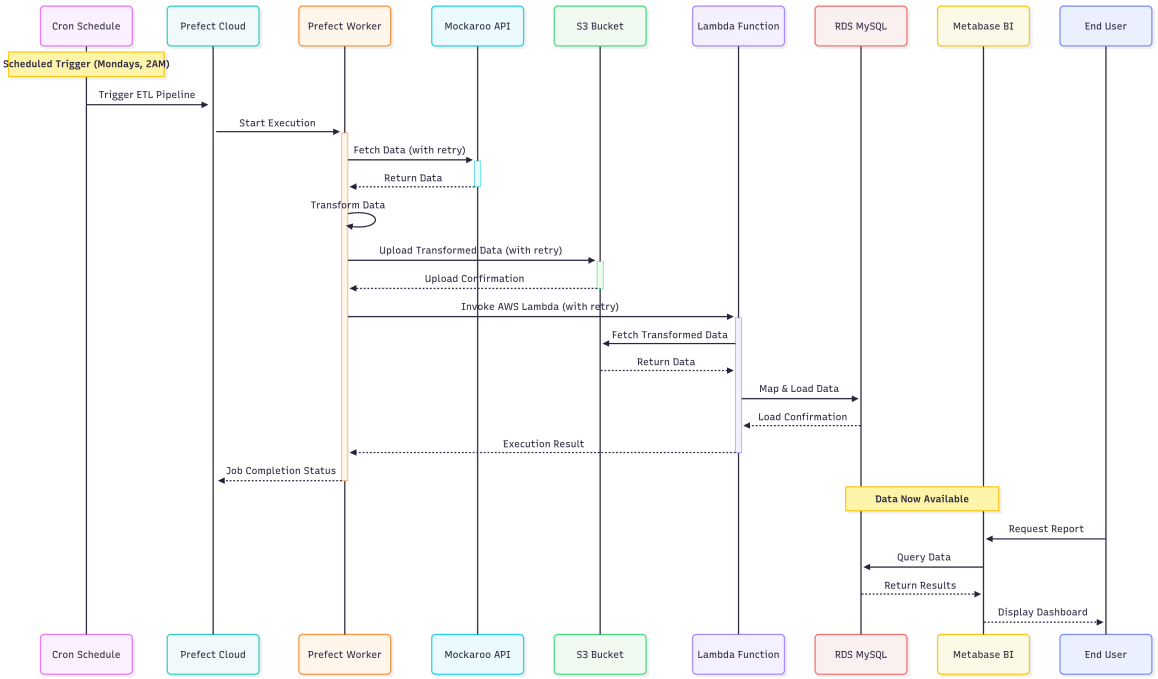
2. **Amazon S3**
    - Stores the transformed data temporarily before ingestion into the database.
    - Connected to your VPC through an **S3 VPC Gateway**.
3. **AWS Lambda**
    - Reads data from S3 and maps it into the **RDS MySQL** database.
    - Serverless, triggered programmatically.
4. **Amazon RDS (MySQL)**
    - Managed database instance where structured data is stored.
    - Placed inside a **private subnet** to restrict public access.
5. **Amazon EC2 (Metabase Server)**
    - Hosts Metabase for querying and visualizing data.
    - Deployed in a **public subnet** to allow secure user access.

Security

- I used a **Virtual Private Cloud (VPC)** to keep everything isolated and secure within its own network.
- Inside the VPC, I split resources into:
    - A **private subnet** for Lambda and RDS (not exposed to the internet).
    - A **public subnet** for the Metabase EC2 instance (needs internet access for users).
- **Security groups** were used to control traffic:
    - Each service has its own security group.
    - RDS only accepts traffic from Lambda and EC2.
    - EC2 only accepts HTTP/HTTPS traffic from users.
    - Lambda only talks to S3 and RDS.
- I used an **S3 VPC Gateway** so that Lambda can read data from S3 without using the public internet, this keeps the data flow private and secure.

- RDS (MySQL) is protected from the public and only reachable by the internal services it needs to work with

Sequence Overview



- A **cron schedule** is configured to automatically trigger the **ETL pipeline** every Monday at 2:00AM
- **Prefect Cloud** receives the trigger and tells the **Prefect Worker** to start running the pipeline.
- The **Prefect Worker**:
  - Fetches raw data from the **Mockaroo API**.
  - Transforms the data as needed.
  - Uploads the transformed data to an **S3 bucket**.
  - Gets confirmation that the data was uploaded.
- After that, the worker **invokes the AWS Lambda Function** to begin loading data into the database.
- **Lambda Function**:
  - Reads the transformed data from **S3**.
  - Maps and loads it into **RDS MySQL**.

    - Sends back a load confirmation to Prefect Worker.
- The worker sends back the **execution result** to Prefect Cloud, which logs the **job completion status**.
- A 'retry' mechanism is configured on tasks that can fail due to temporary external issues. This includes **getting data from an API, uploading files to S3, and calling Lambda functions.** However, data transformation tasks generally don't have retries. If they fail, it's usually because of an error in the code, such as a syntax error, which needs to be fixed directly.

Comprehensive Overview of the Project Development Process

Step 1: Create IAM User

I started by creating a user (which is me) and granted AdministratorAcess permission to the user

In AWS, the root user is the main account owner with full access, but it's not recommended for daily use due to security risks. Instead, We create IAM users with specific permissions, like copies of the master key so they can access AWS safely without using the root account

- Log in as the **root user** and go to the IAM service.
- Create a new IAM user with programmatic and console access.
- Attach the user to a group with necessary permissions (e.g., AmazonRDSFullAccess, AmazonEC2FullAccess).
- Save the **access key** and **secret key** securely.

Step 2: Create RDS instance

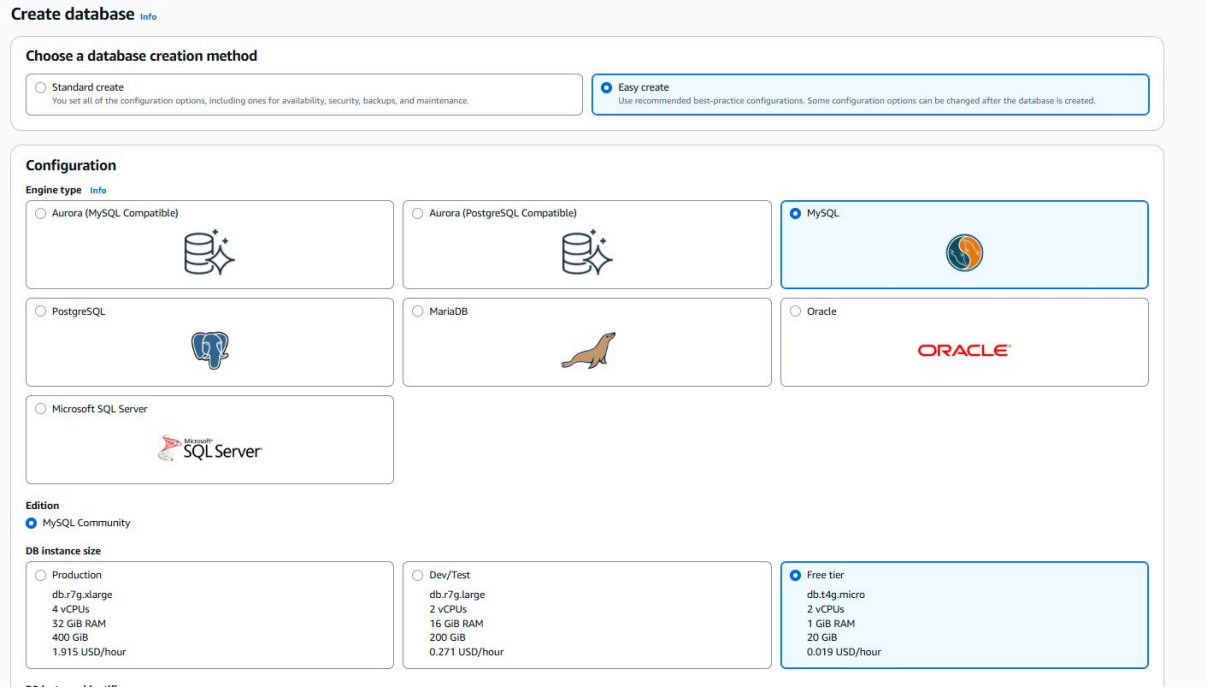
Amazon RDS is a managed service that lets you easily run, scale, and manage relational databases in the cloud without handling the underlying infrastructure.

- A **relational database** is a type of database that stores data in **tables** made up of **rows and columns**, and uses **relationships** between these tables to

organize and connect the data efficiently.

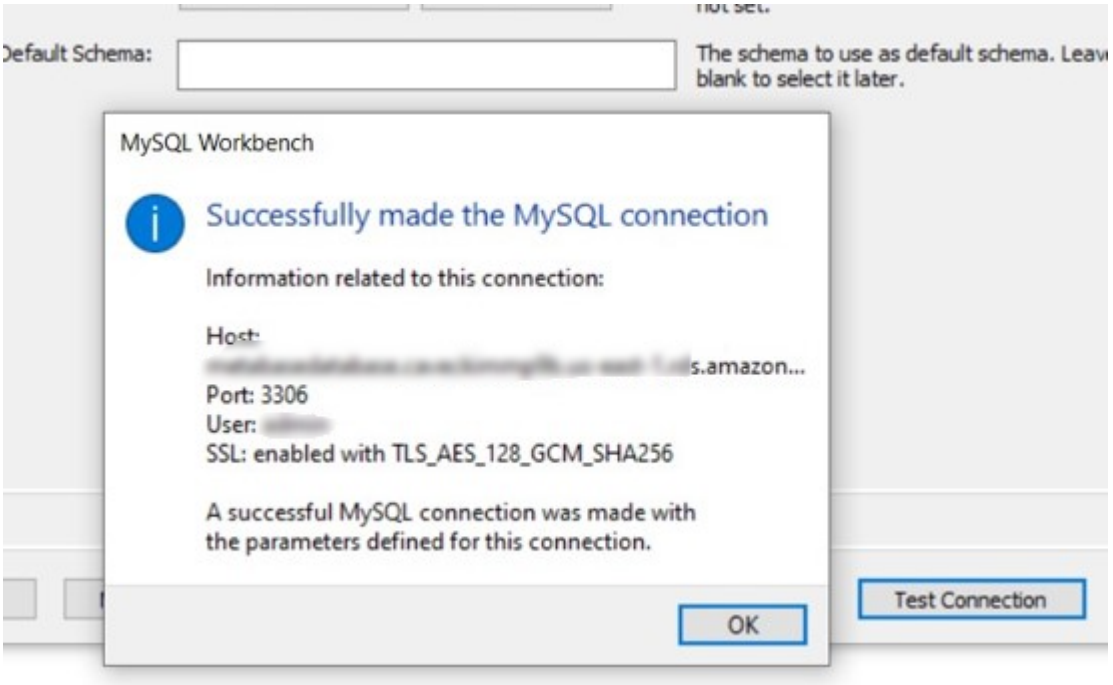
Create and Configure RDS

I created an Amazon RDS MySQL instance in using the **free tier (db.t4g.micro)** with a secure username and password

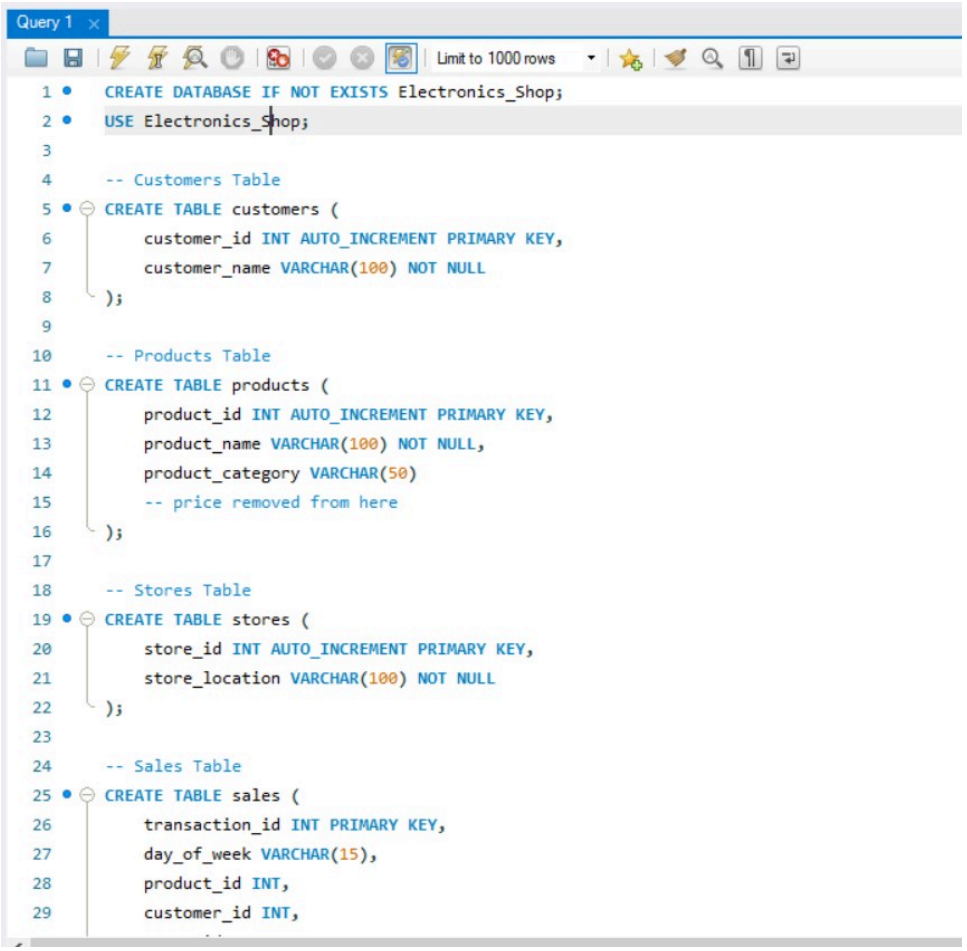


1. I initially set the database to **private (no public access)** for security reasons.
2. After creating the database, I tried connecting to the RDS instance from my local machine using MySQL Workbench, but the connection failed due to restricted access.
3. I changed RDS settings to **enable public access**, hoping to fix the connection issue.
4. Still **could not connect** after enabling public access.
5. Discovered the **security group** was blocking incoming traffic on MySQL port **3306**.
6. Updated security group rules to **allow inbound traffic** on port **3306** from the **public IP** where MySQL workbench is running(my local machine)

7. After updating **security groups**, the connection to the database **succeeded**.



8. In MySQL Workbench on my local machine, I intentionally created the database schema with empty tables. Consequently, the RDS instance currently contains only the empty SQL structure, which will be populated later.



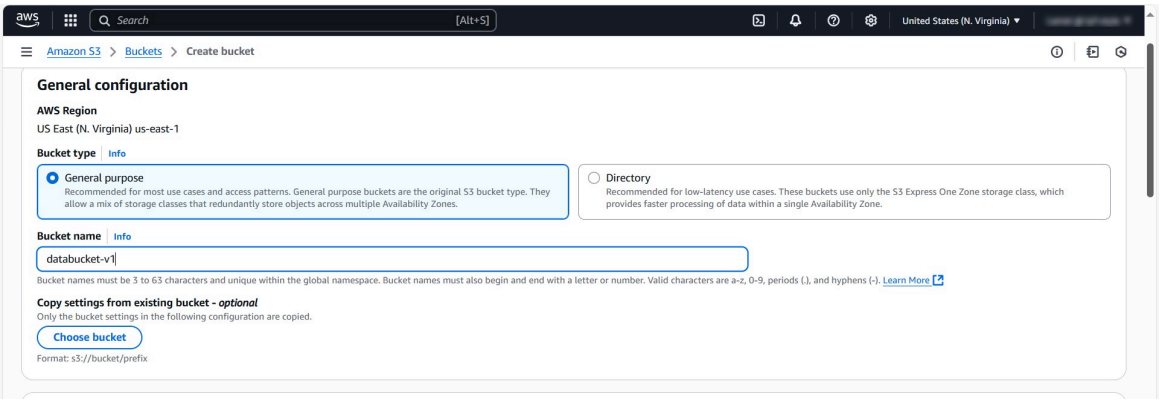
10. After creating my schema, I set the RDS instance back to private access and ensured the subnet is private as well. Now, the RDS is ready to be loaded with data.

Step 3:Create S3 bucket and Lambda Function

AWS S3 is a scalable object storage service for storing data, while **AWS Lambda** is a **serverless compute** service that runs code in response to events, e.g such as a file being uploaded to an **S3 bucket.**

▼ S3 Bucket: Acts as a Staging layer for storing transformed data & ETL flows

I created an S3 bucket that will serve as a staging layer for the transformed data. This bucket will also store the ETL flow code. Each time the pipeline is deployed, the updated code will be saved in this bucket. This process will be explained more clearly in Step 4.

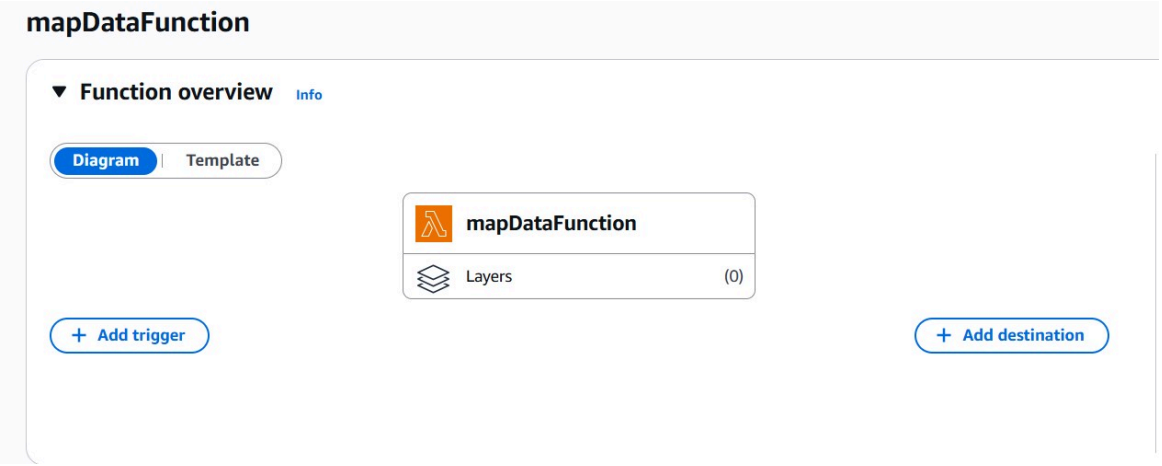


▼ Lambda Function: MapDataFunction.

I created a Lambda function that runs within a **private subnet** alongside the RDS instance. The Lambda has its own security group configured to allow **inbound access** from the **RDS security group**, ensuring secure and controlled **network communication** between the Lambda and the database.

Lambda Function Purpose:

This Lambda function maps transformed CSV files from S3 to their respective tables in the final storage, which is an RDS MySQL database.



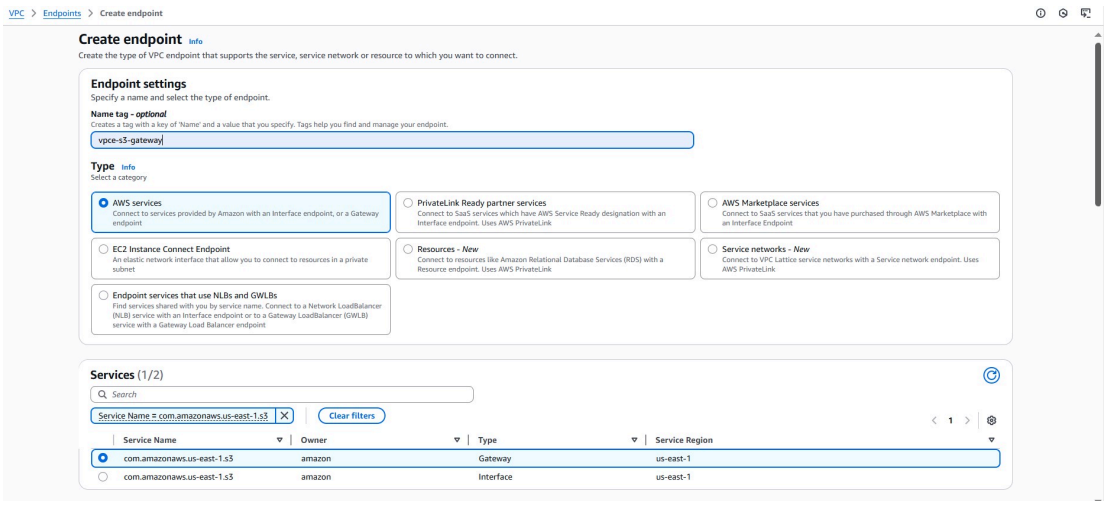
- This Lambda function is deployed as a zipped package that includes the function code along with the required `mysql` package.
- Since this Lambda function fetches data from S3, it requires an **IAM role** with permissions to access the specified S3 bucket, along with basic execution permissions for logging to **CloudWatch**. Below is the S3 permission policy attached to the role

```
{
    "Version": "2012-10-17",
    "Statement": [
        {
            "Effect": "Allow",
            "Action": [
                "s3:GetObject",
                "s3:ListBucket"
            ],
            "Resource": [
                "arn:aws:s3:::databucket-v1",
                "arn:aws:s3:::databucket-v1/*"
            ]
        }
    ]
}
```

S3 VPCE Gateway

Since this Lambda function is deployed in a private subnet, it cannot access S3 by default because private subnets do not have a route to the internet, and accessing S3 typically requires outbound internet access through an **Internet Gateway**.

- One common workaround is to use a **NAT Gateway**, which enables resources in a private subnet to initiate outbound connections to the internet (e.g., to reach S3). However, NAT Gateways incur additional costs, and since this project aims to stay within the AWS Free Tier, I opted for a **free alternative**.
- Instead, I configured an **S3 VPC Gateway Endpoint (VPCE)**. This allows the Lambda function to privately and securely access S3 over the AWS internal network without using the internet by routing traffic directly to S3 through the VPC. This setup is both **secure** and **cost-effective**, making it ideal for workloads within private subnets.



- A **bucket policy** was configured to work with the S3 VPC Gateway Endpoint.
- **Access is restricted** to requests originating from a my specific VPC (`vpc-032e*********`).
- This allows services inside the VPC such as a **Lambda function in a private subnet** to access the bucket via the endpoint.
- 📂 The policy grants permission to:
  - `s3:ListBucket` – list objects in the bucket
  - `s3:GetObject` – read/download objects
  - `s3:DeleteObject` – remove objects
- All requests from **outside the VPC** are **explicitly denied**, even if they come with valid AWS credentials.
- This ensures **private, secure, and cost-effective** access to S3 from within the VPC without needing a NAT Gateway or Internet Gateway.

- I then attached the **S3 Gateway Endpoint to the private route table**, which is associated with the private subnet that contains both the

> Lambda function and the RDS instance. The public route table, on the other hand, is associated with the EC2 instance since it requires internet access via an Internet Gateway. I manually created the route table to properly separate public and private network traffic.

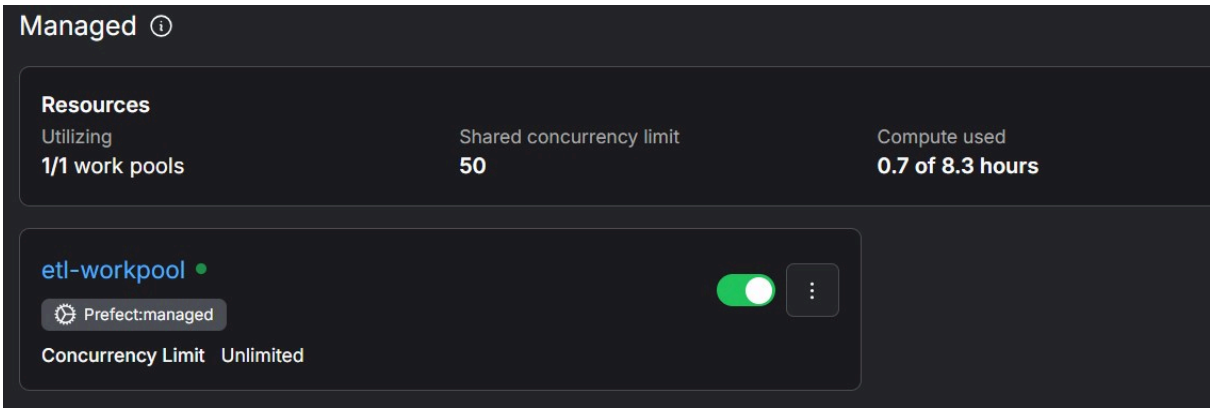
Step 3: Configuring Prefect & CI/CD

Prefect is a modern **workflow orchestration tool** that helps you build, schedule, and monitor data pipelines (ETL/ELT jobs) in Python

I installed **Prefect** in my local environment. This allows me to decorate my **Python scripts** with Prefect **tasks**, create deployments, and **monitor** my workflows directly from the Prefect Cloud UI.

The first step was to create a **work pool**. A work pool is a process that manages the execution of **ETL flows**. In this project, I used Prefect's **managed infrastructure** to run the flows, which means the work pool runs **serverlessly** on Prefect Cloud. While Prefect also supports self-hosted work pools, I chose the **serverless** option for simplicity and ease of deployment.

How does the worker **execute** my flow? .....I choose where to store the flow, **GitHub, S3, or any blob storage.** I used S3 as the storage location. The worker then fetches the flow from that location and runs it using **Prefect's infrastructure.**



With the work pool created, I then decorated my Python scripts using Prefect's `@task` and `@flow` decorators to define tasks and workflows that Prefect can orchestrate.

```python
@task(retries=1, retry_delay_seconds=10, name="📥 Extract from Mockaroo", cache_policy=NO_CACHE)
def extract_from_mockaroo(api_key: str, save_path: str = "raw_sales_data.csv", count: int = 500) -> pd.DataFrame:
    """
    Fetches synthetic CSV data from Mockaroo, saves it locally, and returns it as a DataFrame.
    Retries on failure up to 3 times.
    """
```

```python
@task(name="🔄 Transform Sales Data", cache_policy=NO_CACHE)
def transform_sales_data(df_raw):
    logger = get_run_logger()
    logger.info("🔧 Starting transformation...")
```

```python
@task(name="Data modeling")
def model_sales_data(df: pd.DataFrame, raw_df: pd.DataFrame):
    """
    Splits the cleaned dataframe into four dimension/fact tables:
```

```python
@task(name="Load Customers to S3-Bucket", retries=1, retry_delay_seconds=10, cache_policy=NO_CACHE)
def upload_customers_to_s3(customers_df: pd.DataFrame, bucket: str, s3):
    return upload_df_to_s3(customers_df, bucket, "transformed_data/customers.csv", s3)


@task(name="Load Products to S3-Bucket", retries=1, retry_delay_seconds=10, cache_policy=NO_CACHE)
def upload_products_to_s3(products_df: pd.DataFrame, bucket: str, s3):
    return upload_df_to_s3(products_df, bucket, "transformed_data/products.csv", s3)


@task(name="Load Stores to S3-Bucket", retries=1, retry_delay_seconds=10, cache_policy=NO_CACHE)
def upload_stores_to_s3(stores_df: pd.DataFrame, bucket: str, s3):
    return upload_df_to_s3(stores_df, bucket, "transformed_data/stores.csv", s3)


@task(name="Load Sales to S3-Bucket", retries=1, retry_delay_seconds=10, cache_policy=NO_CACHE)
def upload_sales_to_s3(sales_df: pd.DataFrame, bucket: str, s3):
    return upload_df_to_s3(sales_df, bucket, "transformed_data/sales.csv", s3)


@task(name="Load Raw Data to S3-Bucket", retries=1, retry_delay_seconds=10, cache_policy=NO_CACHE)
def upload_raw_df_to_s3(raw_df: pd.DataFrame, bucket: str, s3):
```

By doing this, I add observability to my functions and Prefect **logs** the task's run **status** (such as success or failure), execution duration, and results in the UI. It also enables features like automatic **retries** and error handling.

The flow is now ready to be deployed.

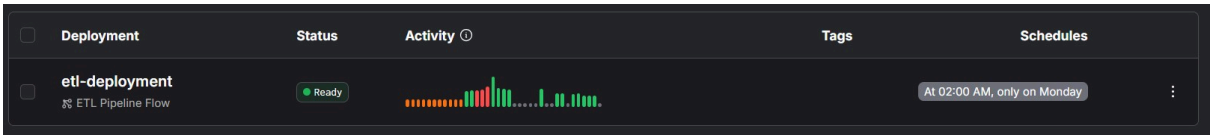
- I used GitHub actions **CI/CD** for deploying, which means the flow is deployed to production only if the CI/CD pipeline **passes** successfully. The pipeline checks for **pytest scripts**, configurations, credentials, and other requirements. Since I use S3 as the storage for the flow, the **CI/CD process** will deploy and push the flow to **S3** only if all tests **pass**. This ensures that broken code never reaches the main pipeline.

```yaml
    - name: Install dependencies
      run: |
        pip install -r requirements.txt
        pip install pytest

    - name: Run tests
      run: pytest tests/

    - name: Prefect Cloud Login
      run: prefect cloud login --key ${{ secrets.PREFECT_API_KEY }} --workspace "lamel/default"

    - name: Deploy Flow to Prefect
      if: github.event_name == 'push'
      run: prefect deploy --all
```

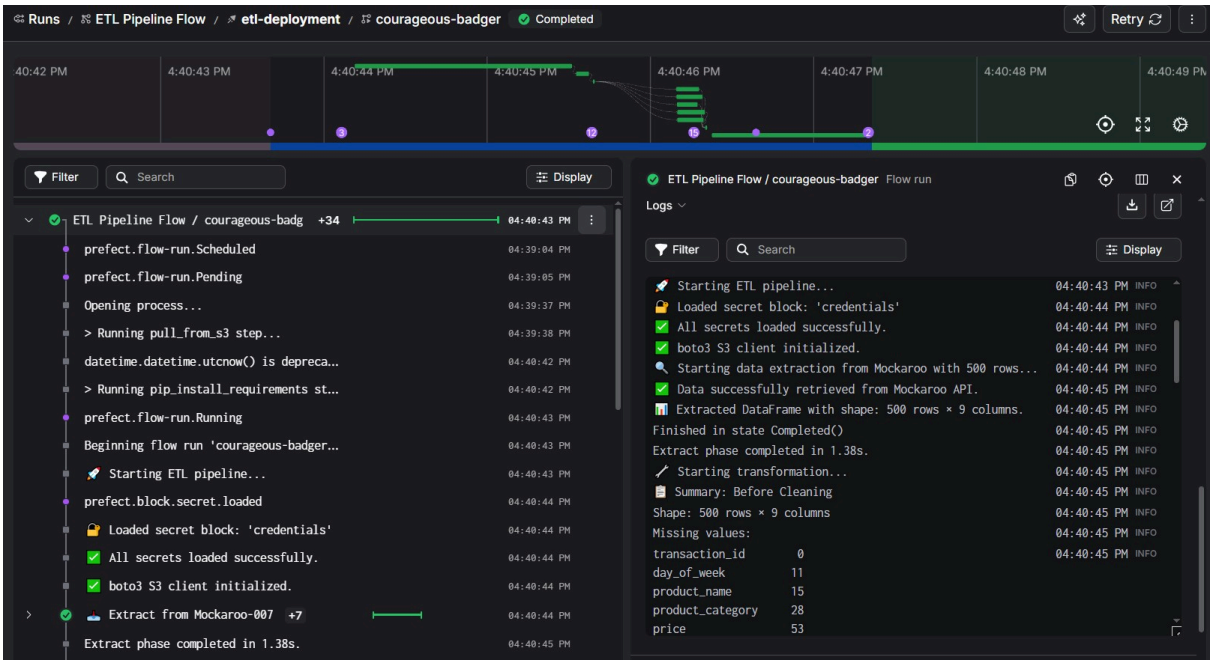
- The image above shows a section of my GitHub Actions workflow. It handles logging into Prefect and automatically deploys the flow whenever code is pushed to the repository.
- The command `prefect deploy -all` also triggers my `prefect.yaml` file, which pushes the flow to my S3 bucket.

```yaml
push:
  - prefect_aws.deployments.steps.push_to_s3:
      id: push_code
      requires: prefect-aws>=0.5
      bucket: databucket-v1
      folder: flows
      credentials: "{{ prefect.blocks.aws-credentials.aws-credentials }}"

pull:
  - prefect_aws.deployments.steps.pull_from_s3:
      id: pull_code
      requires: prefect-aws>=0.5
      bucket: '{{ push_code.bucket }}'
      folder: '{{ push_code.folder }}'
      credentials: "{{ prefect.blocks.aws-credentials.aws-credentials }}"

  - prefect.deployments.steps.pip_install_requirements:
      directory: "{{ pull_code.directory }}"
      requirements_file: requirements.txt

deployments:
- name: etl-deployment
  flow_name: main
  entrypoint: etl_pipeline/etl_flow.py:main
  parameters: {}
  work_pool:
    name: etl-workpool
    work_queue_name: null
    job_variables:
      env:
        AWS_ROLE_ARN: arn:aws:iam::215848077260:role/service-role/ExtractData-LambdaRole
  schedule:
    cron: "0 2 * * 1"  # Every Monday at 2 AM
    timezone: "Africa/Johannesburg"
```

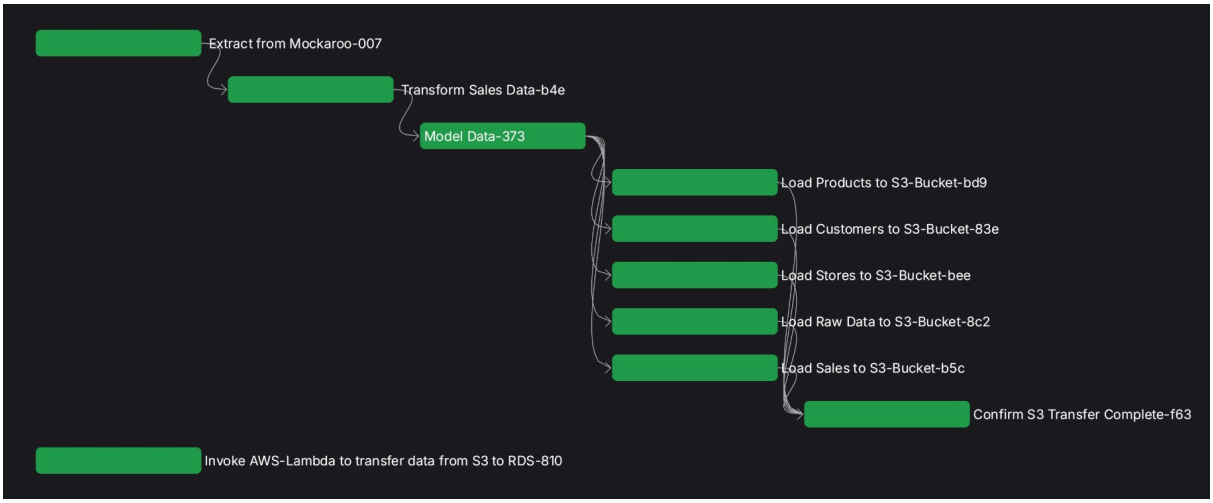
- The image above shows my `prefect.yaml` file, which defines where the flow is deployed. When the CI/CD pipeline runs successfully, it packages and pushes the flow to an S3 bucket, as specified in the configuration. The file also defines the `etl-deployment`, which becomes visible in the Prefect UI. When this deployment is triggered manually or via a schedule in the UI, Prefect pulls the flow from S3 and executes it using the `etl-workpool`.



- The flow is automatically triggered every Monday at 2 AM, but I can also manually trigger it from the **Prefect UI** and observe the run in real time.



- Prefect allows me to monitor each stage of my ETL flow through detailed logs and visualizations, making it much easier to identify failures or performance bottlenecks.



- This image is a **visual representation of the flow**, showing each task from **data extraction to loading**.
- **Prefect** can even run some tasks **asynchronously** for better performance.
- As shown, the flow begins by **extracting**, **transforming**, and **modeling** the data before **loading it into S3**.
- It then performs a **confirmation check** to ensure the transformed data has been successfully uploaded.
- Once that confirmation task (`Confirm S3 Transfer Complete`) finishes, the next task — `(Invoke AWS-Lambda to transfer data from S3 to RDS)` is triggered
  - This is the Lambda function that was created in the previous step `(mapDataFunctiom)`
- As long as the **AWS credentials** are properly configured in **Prefect**, the **Lambda function can be securely invoked** directly from your flow
- This **Lambda function** is responsible for **mapping the transformed data** and **loading it into the RDS** database.

In this pipeline, **S3 effectively acts as a *staging area*** between the transformation step and final database storage, And now that the data is in the **MySQL database**, it's **ready to be queried and visualized.**

> 💡 Remember, we initialized the RDS instance with an empty schema. Now that we have tested the `etl_flow` by executing it, the MySQL database in RDS is populated with data.

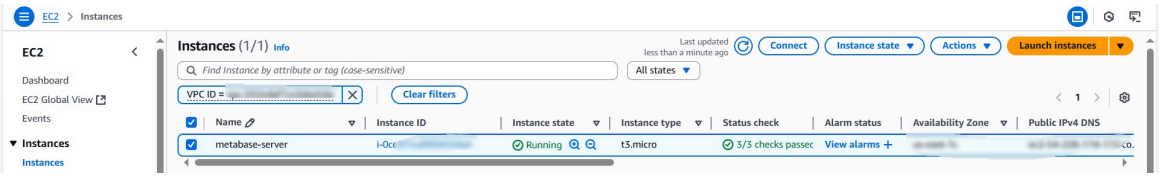
Step 5:Create EC2 for Hosting user Application

Amazon EC2 (Elastic Compute Cloud) is a web service that provides **resizable virtual servers** in the cloud, allowing you to run applications without owning physical hardware.

▼ Launching EC2

I launched a **t3.micro EC2 instance** running **Ubuntu**, intended to host the **user-facing** application **Metabase**, a **business intelligence (BI)** tool.

- I also created a **pem** file which serves as a access key to the EC2 server.
- This EC2 instance also has an **Elastic IP address** associated with it. This ensures that the **public IP** remains **constant,** even if the instance is **stopped** or **rebooted**. By doing so, users can maintain uninterrupted access to the hosted Metabase application without needing to update connection settings or DNS records after system restarts.



▼ Installing Metabase on the Server

To install Metabase on the instance, I had to install **Java** first. After that, I proceeded with the **Metabase** installation.

To access the server via SSH and install applications, I had to add a new **inbound SSH rule** for my **IP address**. Additionally, I opened **port 3000** to allow **access** to Metabase, which uses that port.

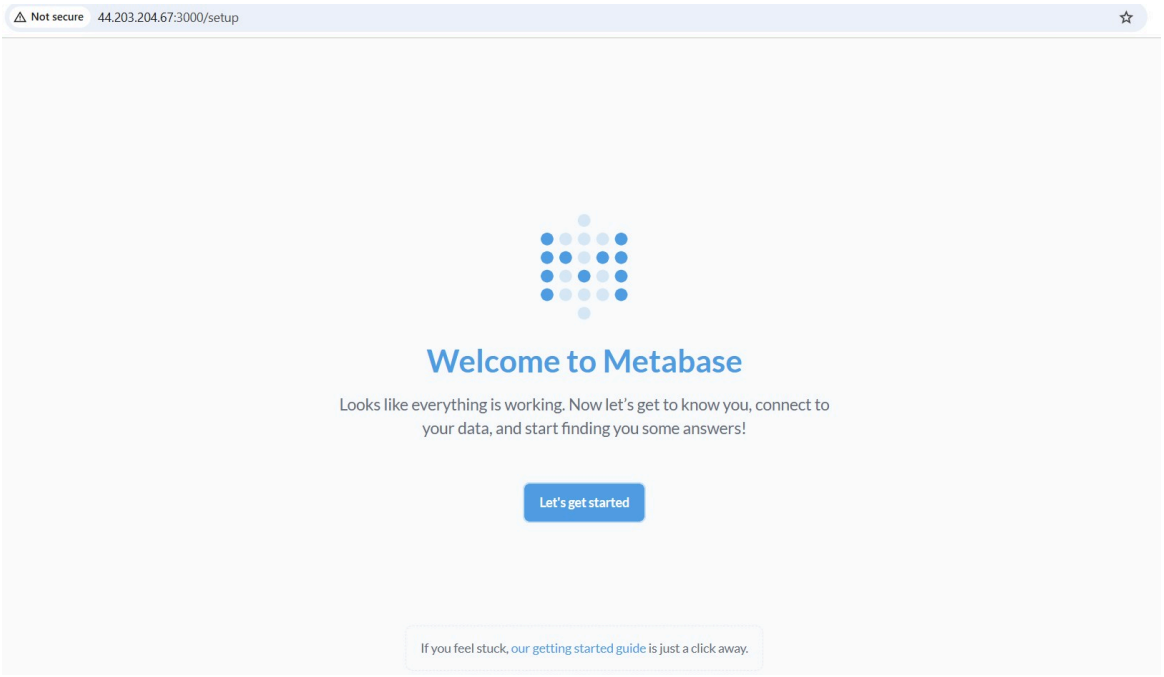
Checking applications

```
ubuntu@ip-[redacted] ~
ubuntu@ip-[redacted]:~$ java --version
java 21.0.7 2025-04-15 LTS
Java(TM) SE Runtime Environment (build 21.0.7+8-LTS-245)
Java HotSpot(TM) 64-Bit Server VM (build 21.0.7+8-LTS-245, mixed mode, sharing)
ubuntu@ip-[redacted]:~$ ls -l ~/metabase.jar
-rw-rw-r-- 1 ubuntu ubuntu 361445417 Dec 20  2023 /home/ubuntu/metabase.jar
ubuntu@ip-[redacted].~$
```

After successfull installation of metabase, I also configured a **systemd file**, to always **auto restart** the application ,especially in scenarios where the EC2 instance **reboots** or the application **stops** unexpectedly, this minimizes **downtime**.

```
ubuntu@ip-[redacted]~$ cat /etc/systemd/system/metabase.service
[Unit]
Description=Metabase Server
After=network.target
Wants=network-online.target

[Service]
User=ubuntu
WorkingDirectory=/home/ubuntu
ExecStart=/usr/bin/java -Xms256m -Xmx512m -jar /home/ubuntu/metabase.jar
Restart=always
RestartSec=10
Environment=MB_JETTY_PORT=3000

# Limit memory usage and ensure automatic cleanup
MemoryMax=700M
TimeoutStartSec=60

[Install]
WantedBy=multi-user.target
```

▼ Running The Application



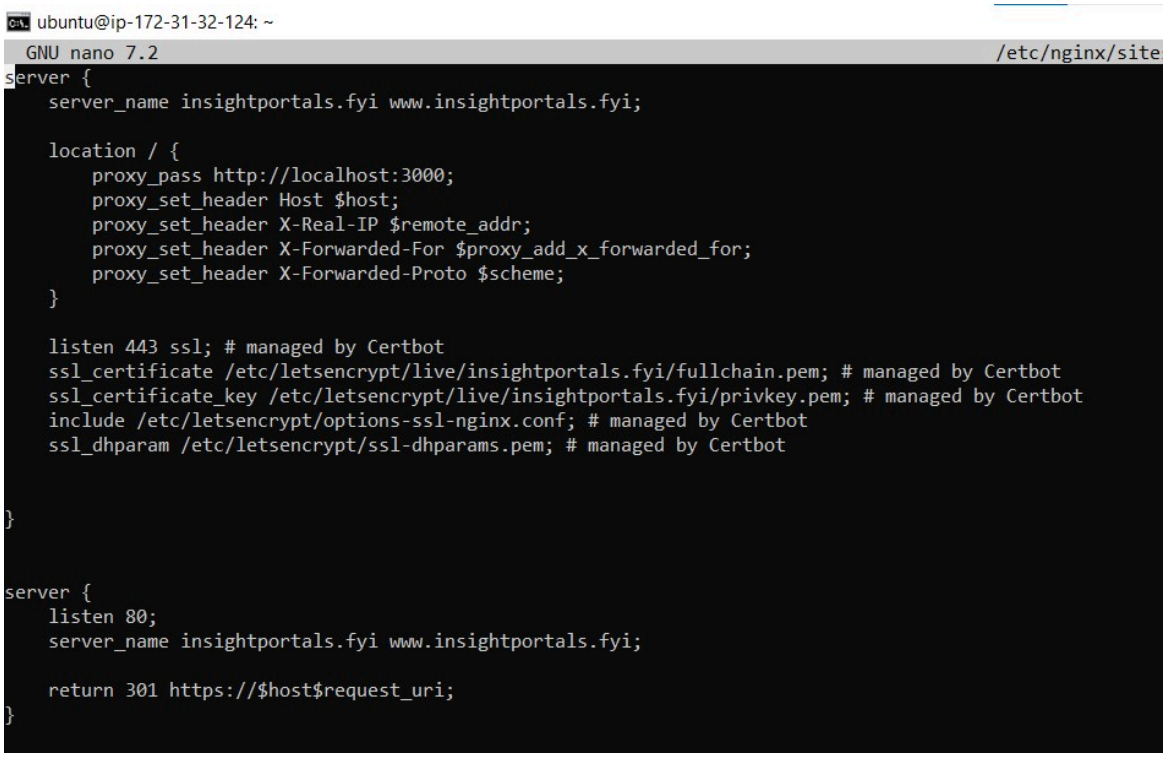
I accessed the Metabase application by navigating to port 3000 in my browser after opening that port in the security group settings. Currently, it shows as 'not secure' because I'm accessing the application via **HTTP** instead of **HTTPS**.

To secure the application, I configured a **reverse proxy** using Nginx and set up HTTPS with a free **SSL certificate** from **Let's Encrypt**.

A
reverse proxy is like a middleman between users and a web application. When someone visits a website, the reverse proxy receives their **request** first, then forwards it to the actual app running on the server (like Metabase). It then sends the app's response back to the user.

- **In the config file**, I set up a nginx reverse proxy so that requests to my domain (**insightportals.fyi**) are forwarded to Metabase running on port **3000**
- I installed **SSL certificates** for my domain using **Let's Encrypt** and **Certbot** to make the site **secure** with **HTTPS**.
- To make this work, I also updated the **DNS** settings at **my domain provider**(Porkburn) to point the domain (and www subdomain) to the **public IP address** of my EC2 instance. This way, when someone visits

insightportals.fyi , the request goes to my EC2 server, where Nginx **forwards** it to Metabase securely using SSL.

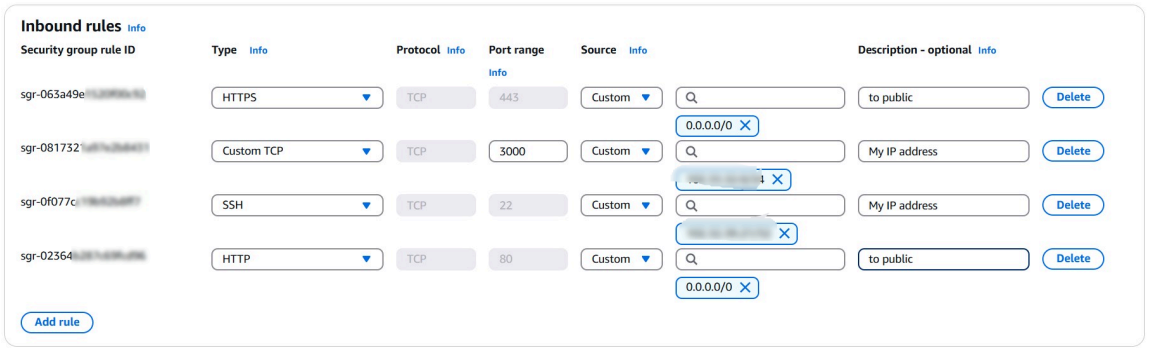


The Nginx setup has two parts:

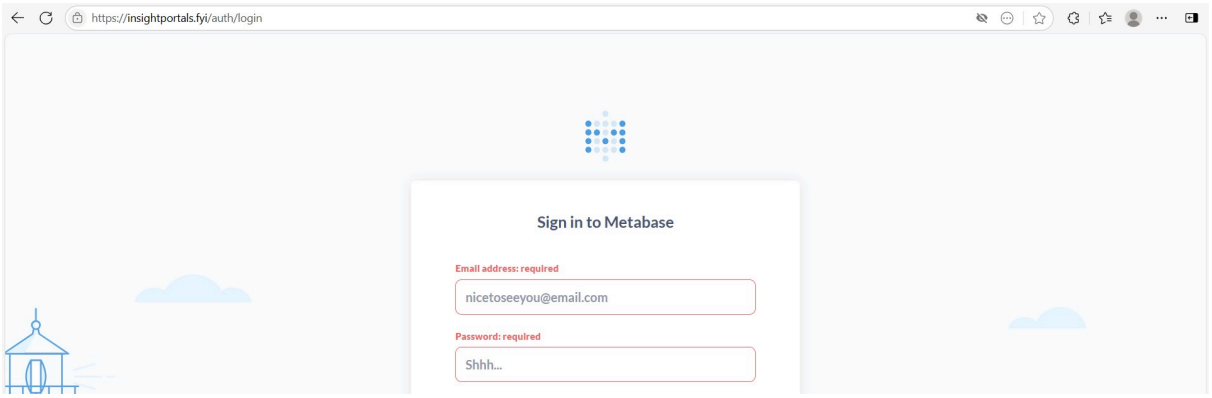
- One listens on **port 443 (HTTPS)** and forwards requests securely to Metabase on port 3000.
- The other listens on **port 80 (HTTP)** and automatically redirects users to the secure HTTPS site.

To allow traffic to the Metabase app through Nginx and my domain, I updated the EC2 Security Group inbound rules to allow HTTP and HTTPS traffic.

So HTTP will redirect the traffic to HTTPS

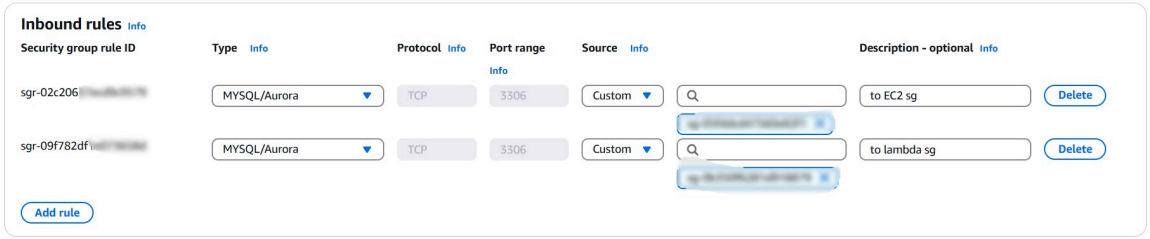


Now the application site is secured

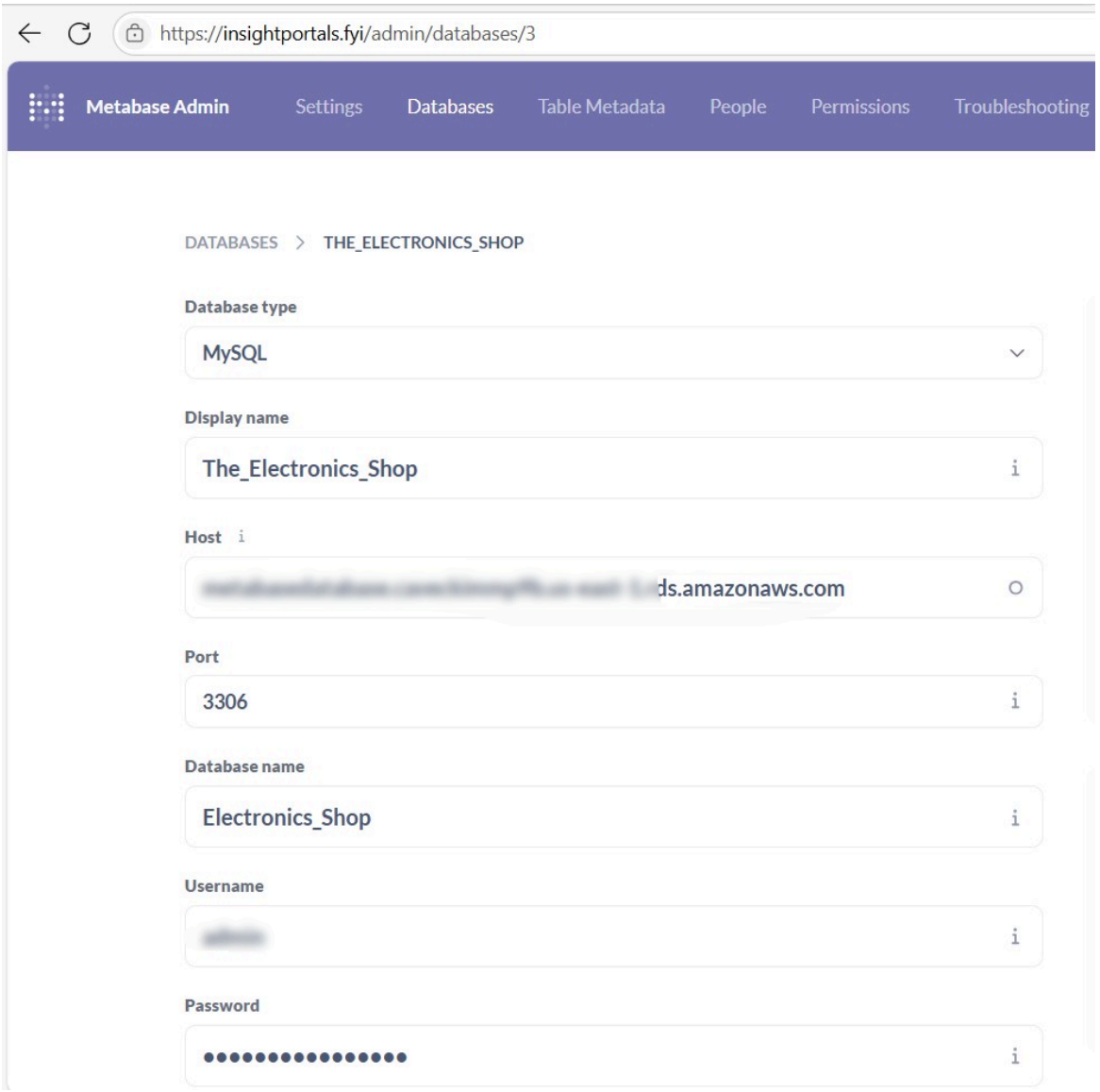


▼ Connecting The Application To The Backend

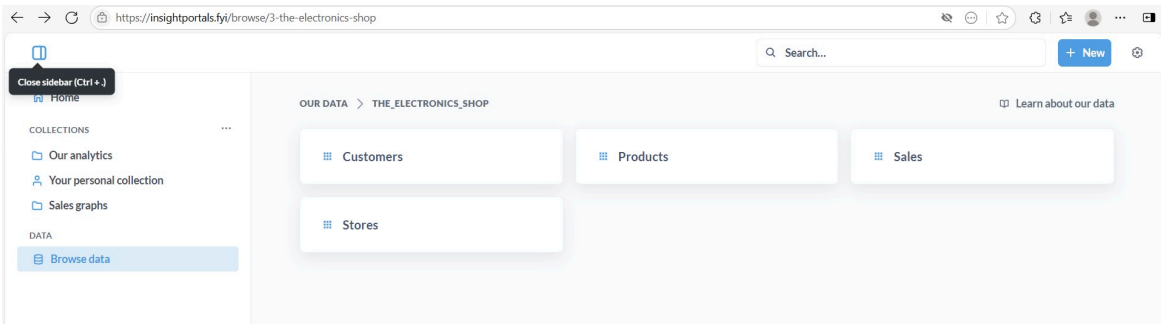
To connect the Metabase application to the database, I added a new **inbound rule** in the **RDS security group** that allows connections from the **EC2 security group**.



After updating the inbound rules, I returned to the Metabase site, created a new user profile, and connected the database by entering the RDS endpoint, username, and password.



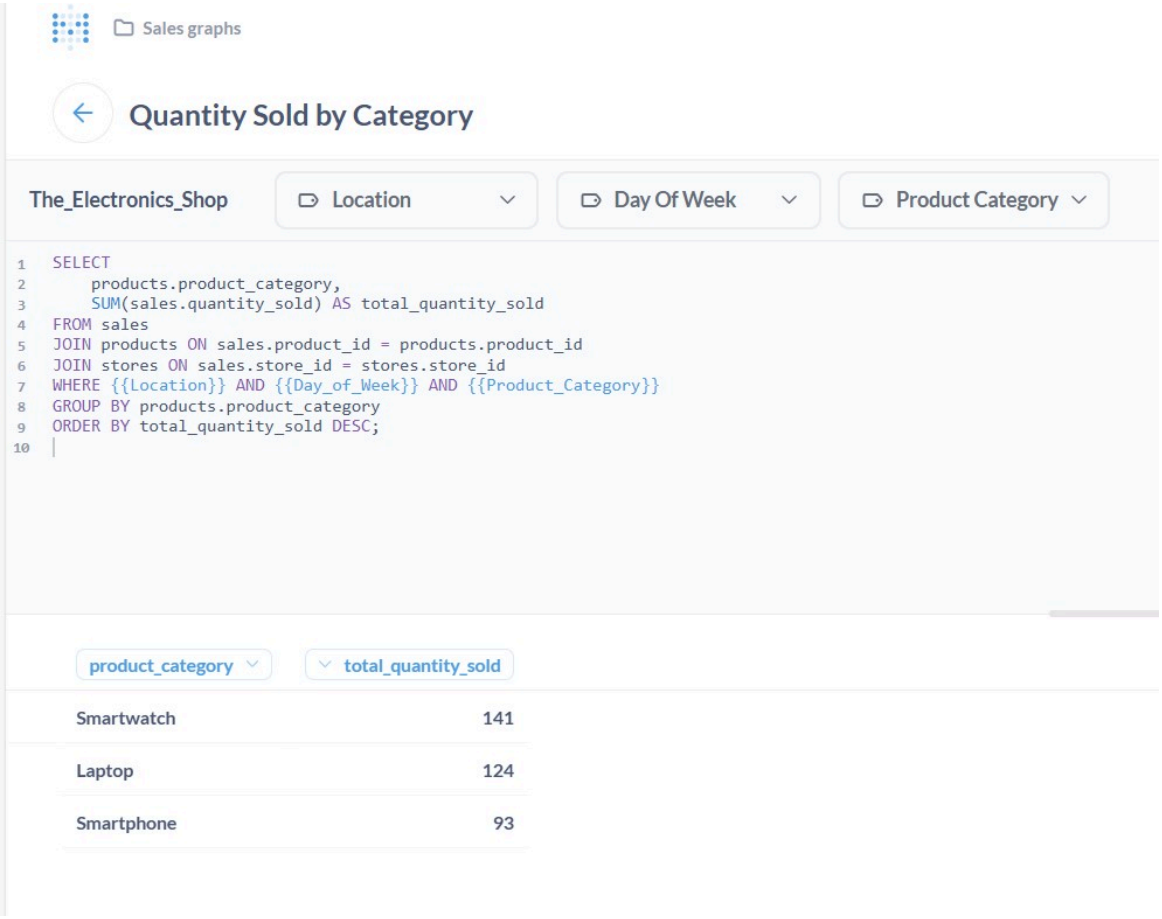
Now for the exiting part! It is time to see if the application has connected to the database



The database connected successfully and can now be queried.

Remember, it was initially empty, but running the etl_flow populated it with data.

Now lets query and create The dashboard

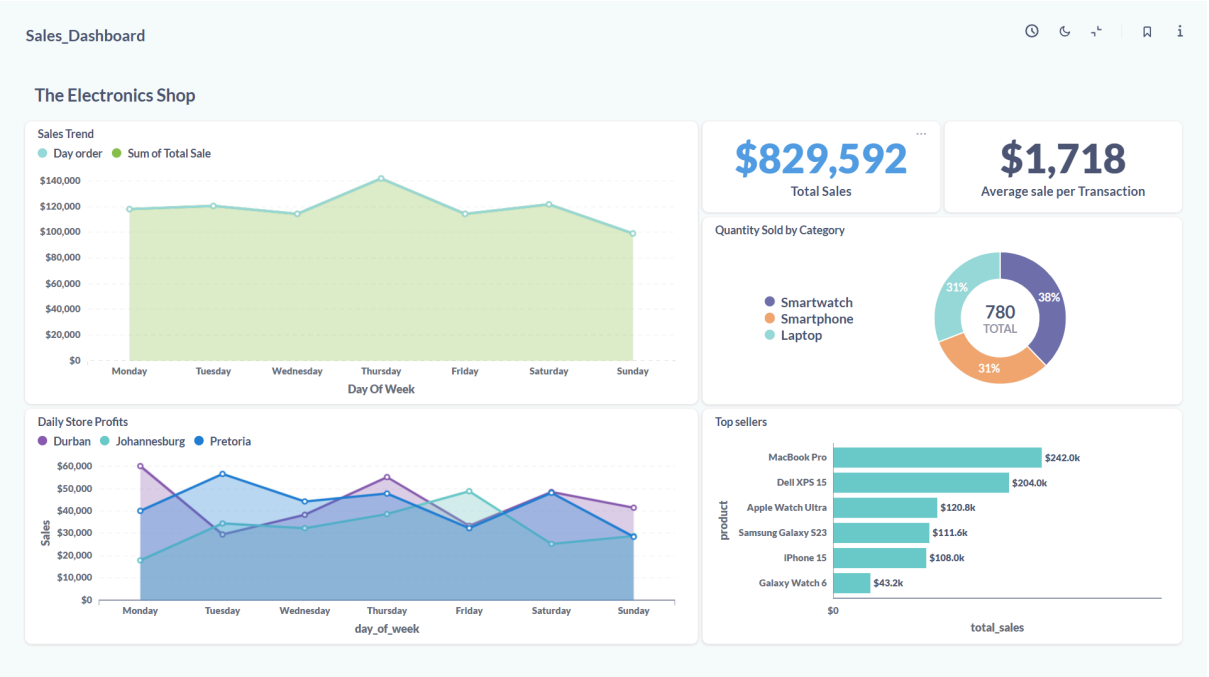


The app allows us to query the Data from RDS and also generate visuals from the queries.

The dashboard

- I created a **dashboard** to visualize the **weekly sales trends** using the mocked data.
- The dashboard is designed to **automatically refresh each week**, reflecting the latest data processed through the ETL pipeline

- I updated the flow **cron schedule** to trigger the flow within the next 5 minutes as a test to see if the dashboard would update accordingly. The test was successful the dashboard refreshed with new data.



Access to the dashboard is restricted from the public to prevent potential overload, as it's hosted on a free-tier server that may crash under high traffic.

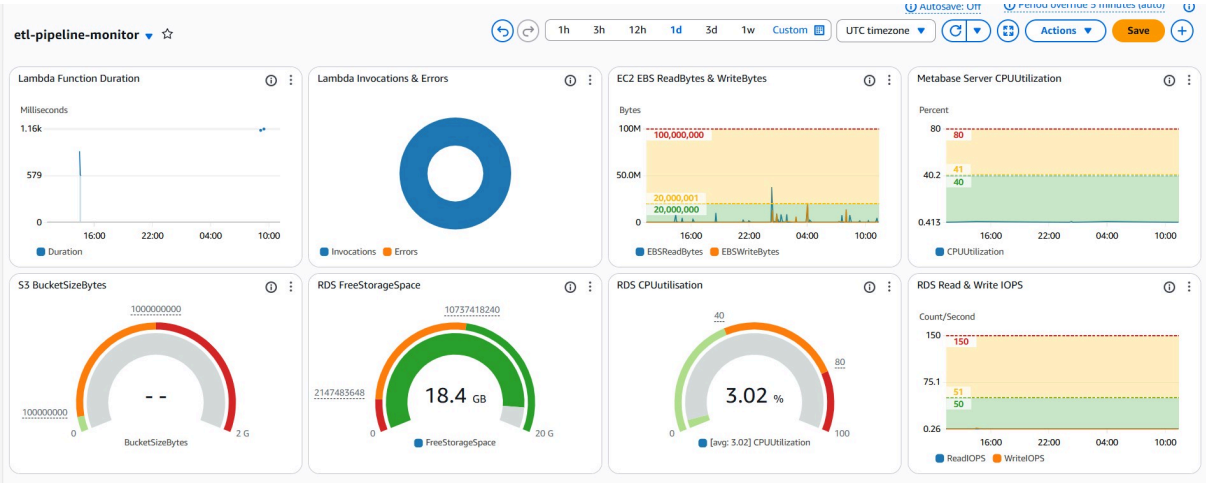
If you'd like access for demo or review purposes, please feel free to

contact me directly at lamel466@gmail.com to view the dashboard at https://insightportals.fyi/

Monitoring

AWS Services

For monitoring the resources, I used AWS CloudWatch metrics to track the health and performance of the services. This provides insights into the current state of the services and indicates when scaling might be necessary. However, since this is a free-tier project, no scaling actions have been implemented.



This dashboard provides a concise, 24-hour(can be filtter to desired timeframe) overview of key performance indicators for our ETL pipeline and its AWS components.

- **Lambda Function Duration:**
  - Displays Lambda execution time in milliseconds. Noted a significant spike over 1.16 seconds around 4 PM, otherwise low.
- **Lambda Invocations & Errors:**
  - The chart displays the ratio of successful Lambda invocations versus errors. Since there was only one Lambda run visible in the duration chart; this chart reflects that single successful execution with no errors
- **EC2 EBS ReadBytes & WriteBytes:**

    - Tracks EBS read and write data volume in bytes. Primarily write-heavy operations (EBSWriteBytes), with lower read activity (EBSReadBytes).
- **Metabase Server CPUUtilization:**
    - Shows Metabase server CPU usage as a percentage with generally low utilization for most of the day, indicating the server is likely idle or lightly used for the majority of the time.
- **S3 BucketSizeBytes:**
    - Indicates current S3 bucket size in bytes
- **RDS FreeStorageSpace:**
    - Displays free storage space on RDS in GB. Substantial 18.4 GB free out of 20 GB, showing healthy capacity.
- **RDS CPUUtilisation:**
    - Shows RDS instance CPU utilization as a percentage. Very low at 3.02%, indicating light load.
- **RDS Read & Write IOPS:**
    - Tracks RDS read/write I/O operations per second. Both read and write IOPS are consistently low.

PREFECT SERVICES

When using Prefect with its serverless worker infrastructure, monitoring focuses on the operational health and performance of data pipelines via Prefect Cloud, since infrastructure management is abstracted away.

- **Prefect Cloud Interface:**

  Acts as the primary dashboard for real-time and historical monitoring of workflows, showing flow runs, task states, and event logs without requiring direct monitoring of the underlying infrastructure.
- **Key Monitoring Areas:**
    1. **Flow Run Status & History:** Track success/failure rates, duration, retries, and detect stuck or slow flows.
    2. **Task Run Details:** Inspect individual task outcomes, durations, retries, and logs to pinpoint issues.

3. **Deployment Health:** Verify flows are scheduled and running as expected, checking deployment status and errors.
4. **Event Streams & Logs:** Use aggregated logs and error messages for troubleshooting and performance tuning.

This approach reduces operational overhead by relying on Prefect's serverless platform for infrastructure, allowing focus on data processing logic and pipeline reliability.

What I've Learned from This Project

- **Security groups**: It's crucial to define specific inbound and outbound rules instead of allowing open access. This minimizes security risks and ensures only necessary traffic is permitted.
- **SSL Certificates**: Securing the website with **HTTPS** using SSL/TLS certificates helps protect user data and build trust with users.
- **VPC Endpoints**: When services like **Lambda** run in **private subnets**, they need **VPC endpoints** to access services like **S3** without using the public internet.
- **Prefect:** I learned how to orchestrate, schedule, and monitor data workflows efficiently using Prefect
- **Lambda packages:** When deploying code that depends on external packages, you need to either include those dependencies in the Lambda deployment package (ZIP file) or attach them as a Lambda layer. This ensures the runtime environment has access to all required libraries during execution

---

Recommendations and Improvements

To enhance the scalability, security, and reliability of this project, the following improvements are recommended:

1. **Enhance Security with NACLs and Secrets Manager**

Implement **Network ACLs (NACLs)** to provide an extra layer of subnet-level security. Additionally, store sensitive configuration values and environment variables in **AWS Secrets Manager** to keep credentials secure and centralized.

2. **Improve Application Deployment with ECS Fargate**

   Migrate the EC2-hosted application to **Amazon ECS with Fargate** for a fully managed, serverless container deployment model. This reduces operational overhead and improves scalability.

3. **Adopt Scalable Architecture (Horizontal or Vertical Scaling)**

   Design the architecture to **scale vertically** (by upgrading instance sizes) or **horizontally** (by adding more instances or containers), depending on resource demands and cost-efficiency.

4. **Leverage Multi-AZ Deployments for High Availability**

   Run critical services such as RDS, ECS, and Lambda across **multiple Availability Zones** to ensure high availability and fault tolerance, minimizing the impact of AZ-level failures.

5. **Migrate to AWS Glue for Unified Orchestration**

   Consider migrating from Prefect to **AWS Glue**, which provides native integration with other AWS services. This can simplify orchestration, improve data cataloging, and reduce external dependencies by consolidating the workflow under a single cloud ecosystem.

6. **Strengthen S3 Security with Advanced Encryption**

   Use stronger encryption practices for the S3 bucket, such as **AWS KMS-managed keys (SSE-KMS)**, and implement **bucket policies** and **object-level encryption** to enforce tighter access control.

7. **Upgrade to Amazon Aurora for Database Performance**

   Replace the current MySQL RDS instance with **Amazon Aurora**, a high-performance, fault-tolerant relational database that offers better scalability, automatic backups, and faster failover capabilities.

THE END!