Data Engineering Assessment Documentation

# Introduction

Autochek Africa is an Automotive Technology development company that builds solutions aimed at enabling and enhancing commerce within the automotive sector. We focus on solutions that improve access to Auto loans, quality maintenance / after-sales services, warranty solutions, and transactional marketplaces.

We are operational in the West, East, and North Africa, and we have also developed a partner-led retail footprint with over 500 service centers.

# Assumption

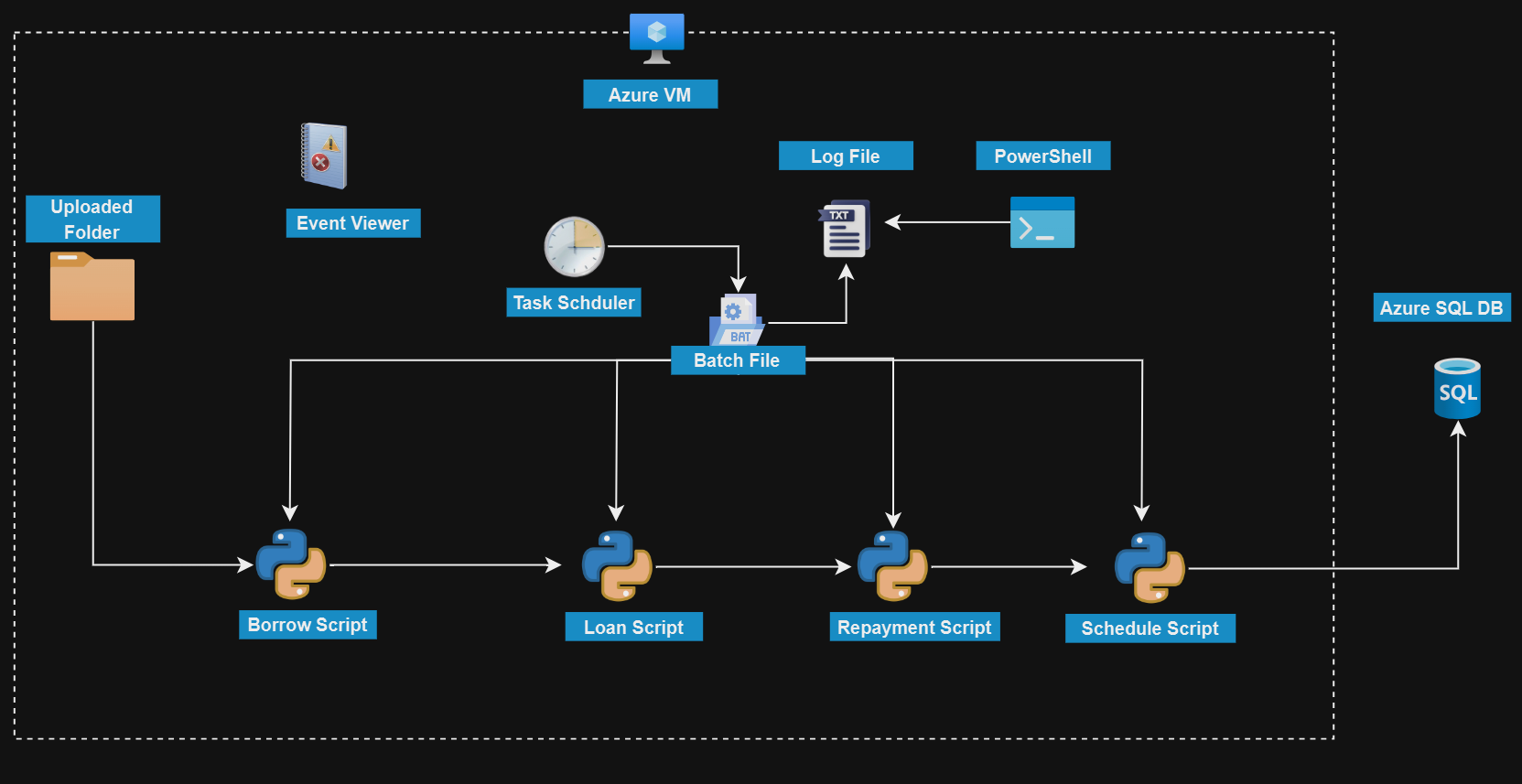
The following assumptions were carried out to achieve this project.

* *Data is uploaded to a particular folder daily.*
* *We know the exact time (Hour) the data is been loaded.*

# Project Architecture

We are tasked to create an ETL pipeline that reads data from a particular folder. Data are dropped daily, we need to create a logic that picks the latest file based on the last modified data, performs all necessary transformations, and loads the data into an Azure SQL Database (Data Warehouse).

The entire process needs to be automated and monitored. We are going to create a CRON job in a Windows Server (VM) and automate the Python script using a ***.bat*** file in the process.



*Project Architecture*

## Process Done

We will explain one process which will cover the entire process, we will break our Python code into multiple parts.

*The following step was used to achieve the desired ETL process:*

### Step 1: Import Necessary Libraries

The libraries below will be used in achieving the entire process.

|  |
| --- |
| import os import pandas as pd from sqlalchemy import create\_engine from dotenv import dotenv\_values from datetime import datetime |

### Step 2: Create a Function to Get the Latest File

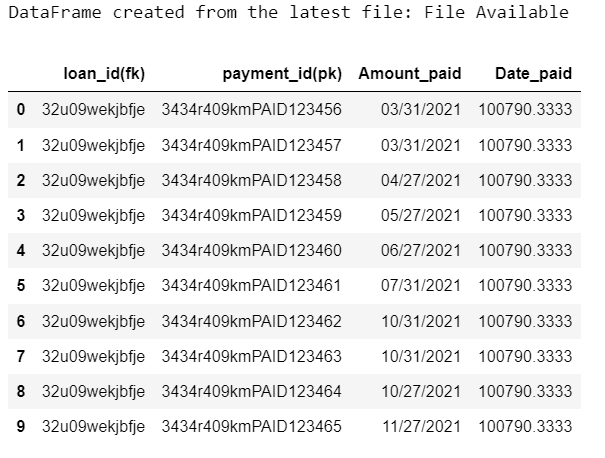
A function is created to get the latest file in the folder path and read the file as a dataframe.

|  |
| --- |
| def get\_latest\_file(folder\_path):  files = [os.path.join(folder\_path, file) for file in os.listdir(folder\_path)]    if files:  latest\_file = max(files, key=os.path.getmtime)  return latest\_file  else:  return None  def read\_latest\_file\_into\_dataframe(folder\_path):  latest\_file = get\_latest\_file(folder\_path)   if latest\_file:  if latest\_file.lower().endswith('.csv'):  df = pd.read\_csv(latest\_file)  elif latest\_file.lower().endswith(('.xls', '.xlsx')):  df = pd.read\_excel(latest\_file)  else:  print(f"Unsupported file format for {latest\_file}")  return None   return df  else:  print(f"No files found in {folder\_path}")  return None |

### Step 3: Define Folder Path

The file path will help the function get the latest file in the folder path. In this way, we can create an automated process if the data is being loaded daily.

|  |
| --- |
| # Define the folder path folder\_path = r"C:\Users\xxxxxxx\xxxxx\xxxxxx\Repayment\_Folder" latest\_df = read\_latest\_file\_into\_dataframe(folder\_path)  if latest\_df is not None:  print("DataFrame created from the latest file: File Available")    latest\_df.head(10) |



### Step 4: Transformation

Data is transformed and standardized at this point to its necessary component, by setting the appropriate datatypes and renaming necessary columns.

|  |
| --- |
| def transformation(latest\_df):  # Step 1: Rename columns  latest\_df.rename(columns={  'loan\_id(fk)': 'loan\_id',  'payment\_id(pk)': 'payment\_id',  'Amount\_paid': 'date\_paid',  'Date\_paid': 'amount\_paid'  }, inplace=True)    # Step 2: Convert 'date\_paid' to datetime with time component  if 'date\_paid' in latest\_df.columns:  try:  # Convert to datetime  latest\_df['date\_paid'] = pd.to\_datetime(latest\_df['date\_paid'], format='%m/%d/%Y', errors='coerce')    # Set the time to 00:00:00  latest\_df['date\_paid'] = latest\_df['date\_paid'].dt.normalize()    print("Column 'date\_paid' converted to datetime with time component.")  except Exception as e:  print(f"Error converting 'date\_paid': {str(e)}")    return latest\_df  # Apply the transformation function transformed\_df = transformation(latest\_df)  # Display the first few rows of the 'date\_paid' column print(transformed\_df['date\_paid'].head(10))  # Verify the datatype of the 'date\_paid' column print(transformed\_df['date\_paid'].dtype)  # Check for any null values print(transformed\_df['date\_paid'].isnull().sum())  # Display a sample value in the desired format print(transformed\_df['date\_paid'].iloc[0].strftime('%Y-%m-%d %H:%M:%S.%f')[:-3]) |

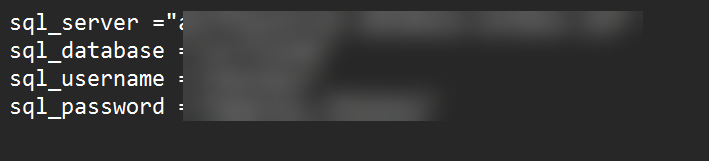
Attaching a new column to the data frame known as ***ingestion\_time*** will help track the exact time data is loaded to the Azure Database.

|  |
| --- |
| # Adding the ingestion\_date column with the current UTC timestamp transformed\_df['ingestion\_date'] = datetime.utcnow()  # Display the updated DataFrame transformed\_df.head() |

### Step 5: Security

For security, we will be using the System environment variable which will provide us with a higher level of security for the entire process.

|  |
| --- |
| # Step 3: Load environment variables from .env file env\_dir = r'C:\Users\xxxxxx\xxxxx\Test\_Question\Credentials' env\_values = dotenv\_values(os.path.join(env\_dir, '.env')) |



After defining the credential filepath use the following to achieve this.

|  |
| --- |
| # Function to load environment variables from .env file def get\_db\_credentials(env\_dir):  env\_values = dotenv\_values(os.path.join(env\_dir, '.env'))    sql\_server = env\_values.get("sql\_server")  sql\_database = env\_values.get("sql\_database")  sql\_username = env\_values.get("sql\_username")  sql\_password = env\_values.get("sql\_password")  sql\_driver = 'ODBC Driver 17 for SQL Server'  schema\_name = 'autocheck'  table\_name = 'repayment'    return {  'sql\_server': sql\_server,  'sql\_database': sql\_database,  'sql\_username': sql\_username,  'sql\_password': sql\_password,  'sql\_driver': sql\_driver  } |

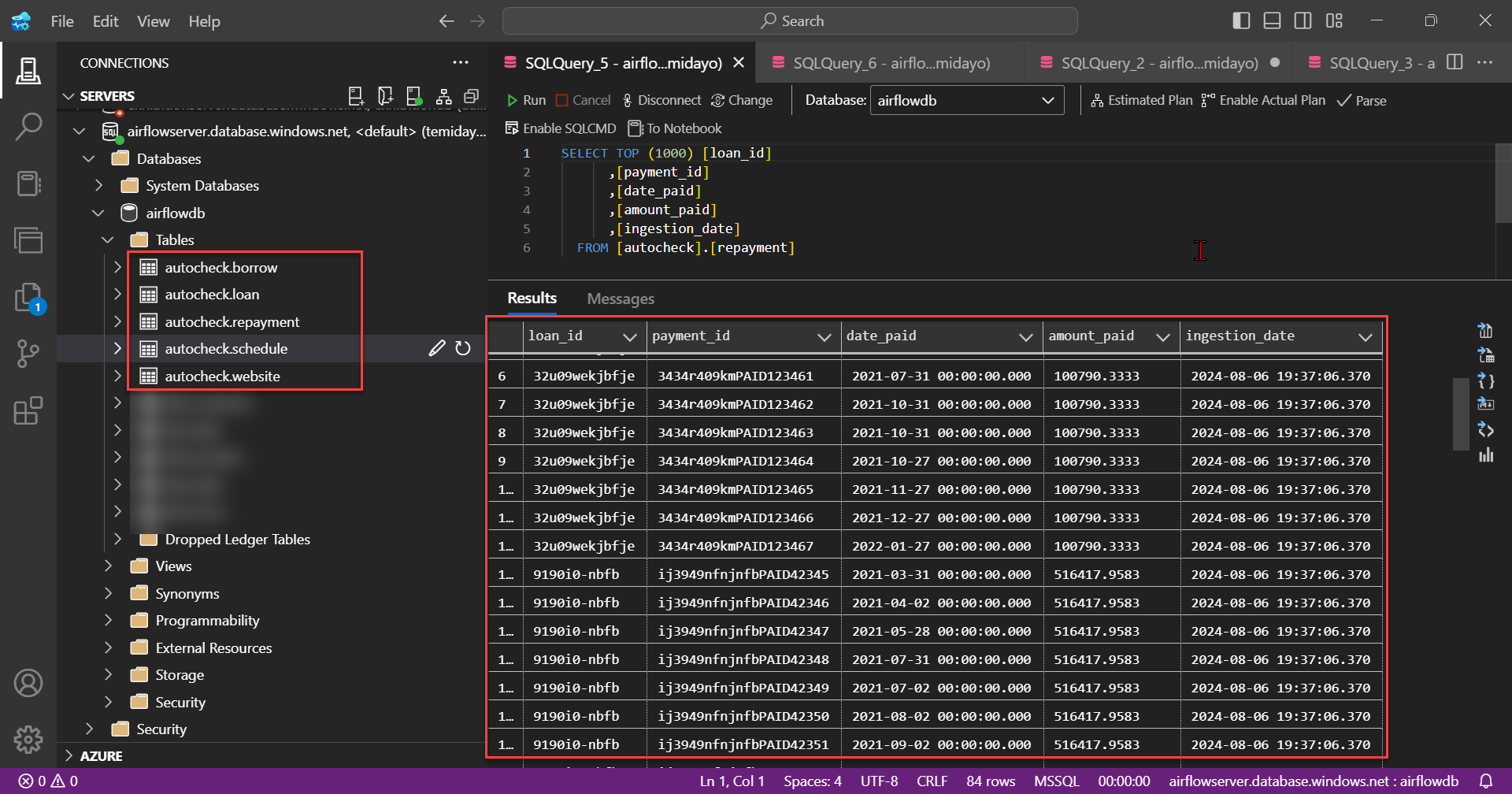
### Step 6: Load Data to Azure SQL Database

The function below will help in creating a function for loading data to the Azure SQL Database. This is concerning the following credentials.

|  |
| --- |
| # Function to load DataFrame to SQL database def load\_data\_to\_sql(transformed\_df, table\_name, env\_dir, schema\_name='autocheck'):  try:  credentials = get\_db\_credentials(env\_dir)  connection\_string = f"mssql+pyodbc://{credentials['sql\_username']}:{credentials['sql\_password']}@{credentials['sql\_server']}/{credentials['sql\_database']}?driver={credentials['sql\_driver']}"  engine = create\_engine(connection\_string)    # Append DataFrame to SQL table  transformed\_df.to\_sql(table\_name, engine, schema=schema\_name, if\_exists='append', index=False)    print(f"Data appended to Azure SQL Database table {schema\_name}.{table\_name} successfully.")  except Exception as e:  print(f"Error writing to SQL Database: {str(e)}")  # Check if DataFrame is available before attempting to load it to SQL if transformed\_df is not None:  table\_name = 'repayment' # Define your table name here  load\_data\_to\_sql(transformed\_df, table\_name, env\_dir) |

### Step 7: Confirm Data Migration

In your Azure SQL Database use the Select statement to view the loaded tables.

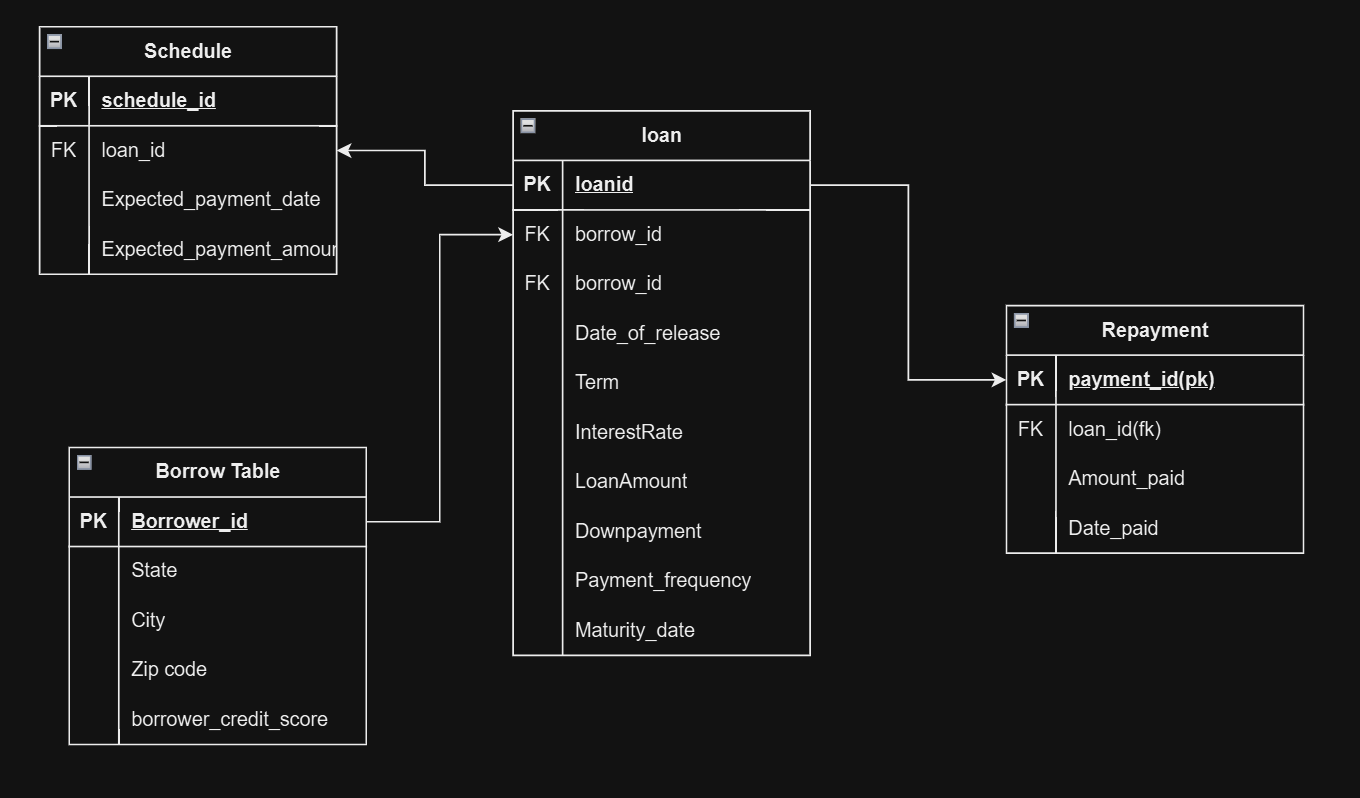


**NOTE:** The same process will be used for the other tables.

# Data Modeling

We need to set the relationship between the different tables in the SQL Database.

The entity-relationship explains the entire process used for this process.



|  |
| --- |
| /\* Task 1. Alter Table to be a Primary Key \*/ -- Load Table ALTER TABLE [autocheck].[loan] ADD CONSTRAINT PK\_load\_id PRIMARY KEY (loan\_id);  -- Borrow Table ALTER TABLE [autocheck].[borrow] ADD CONSTRAINT PK\_borrow\_id PRIMARY KEY (Borrower\_Id);  -- Repayment Table ALTER TABLE [autocheck].[repayment] ADD CONSTRAINT PK\_payment\_id PRIMARY KEY (payment\_id);  -- Schedule Table ALTER TABLE [autocheck].[schedule] ADD CONSTRAINT PK\_schedule\_id PRIMARY KEY (schedule\_id);   /\* Task 2: Create Foreign Keys Relationships \*/ -- Repayment Table(loan\_id\_fk) ==Loan Table(loan\_id)  ALTER TABLE [autocheck].[repayment] ADD CONSTRAINT FK\_load\_id FOREIGN KEY (loan\_id) REFERENCES [autocheck].[loan] (loan\_id);  -- ALTER TABLE [autocheck].[schedule] ADD CONSTRAINT FK\_schedule\_id FOREIGN KEY (loan\_id) REFERENCES [autocheck].[loan] (loan\_id);  -- ALTER TABLE [autocheck].[loan] ADD CONSTRAINT FK\_borrow\_id FOREIGN KEY (Borrower\_id) REFERENCES [autocheck].[borrow] (Borrower\_id); |

# Task

*Using the dimensional tables above, write queries to Calculate PAR Days - Par Days*

*means the number of days the loan was not paid in full. Eg If the loan repayment*

*was due on the 10th of Feb 2022 and payment was made on the 15th of Feb 2022 the*

*par days would be 5 days (NOTE: For each day, a customer missed a payment*

*the amount\_at\_risk is the total amount of money we are expecting from the customer as*

*at that time, for instance, if the customer owes for 5000 from month 1 and 10000 for*

*current month the amount\_at\_risk will be the total amount 5000 + 10000 = 15000)*

|  |
| --- |
| WITH LatestPayments AS (  -- Get the latest payment date for each loan  SELECT loan\_id, MAX(Date\_paid) AS LastPaymentDate  FROM [autocheck].[repayment]  GROUP BY loan\_id ), ExpectedPayments AS (  -- Get the earliest expected payment date that hasn't been fully paid  SELECT s.loan\_id, MIN(s.Expected\_payment\_date) AS EarliestExpectedDate,  SUM(s.Expected\_payment\_amount) AS TotalExpectedAmount  FROM [autocheck].[schedule] s  LEFT JOIN [autocheck].[repayment] r ON s.loan\_id = r.loan\_id AND s.Expected\_payment\_date = r.Date\_paid  GROUP BY s.loan\_id  HAVING SUM(s.Expected\_payment\_amount) > COALESCE(SUM(r.Amount\_paid), 0) ) SELECT   l.Borrower\_id,  l.loan\_id,  l.Date\_of\_release,  l.Maturity\_date,  ep.EarliestExpectedDate AS FirstMissedPaymentDate,  DATEDIFF(day, ep.EarliestExpectedDate, GETDATE()) AS PAR\_Days,  ep.TotalExpectedAmount AS Amount\_at\_Risk,  COALESCE(lp.LastPaymentDate, l.Date\_of\_release) AS LastPaymentOrReleaseDate FROM   [autocheck].[loan] l JOIN ExpectedPayments ep ON l.loan\_id = ep.loan\_id LEFT JOIN LatestPayments lp ON l.loan\_id = lp.loan\_id WHERE   GETDATE() > ep.EarliestExpectedDate ORDER BY   PAR\_Days DESC, Amount\_at\_Risk DESC; |

## 

## Orchestrate ETL Process

There are multiple ways of achieving this, but we will be using the ***Task Scheduler*** ***and Batch file*** automation available in the Windows server due to cost. This will be housed in a ***Virtual Machine (IaaS) on Azure.***

## Create DAG Process Using Batch File

Windows servers already have inbuilt automation tools that help automate takes. For this, we will use the Batch file in the Windows server and create a DAG process scenario.

A batch file is a script that contains a series of commands to be executed by the command-line interpreter in Windows. Think of it as a list of instructions your computer follows individually.

### Batch File Script

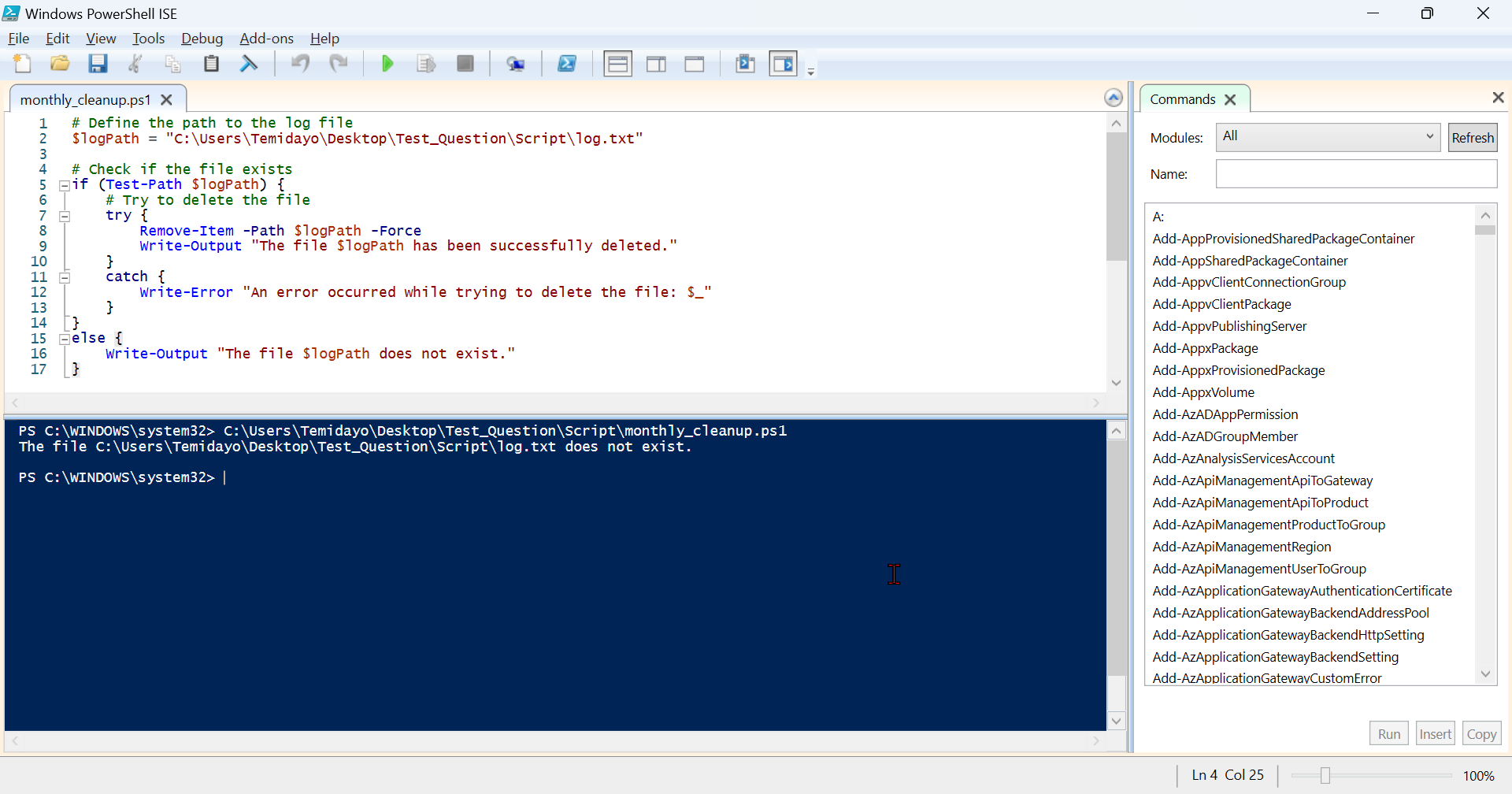
Ensure you save the file extension as ***.bat***

|  |
| --- |
| @echo off REM Define the paths to the Python interpreter and scripts set python\_path="C:\Users\Temidayo\anaconda3\python.exe" set borrow\_path="C:\Users\Temidayo\Desktop\Test\_Question\Script\Borrow\_Scripts\borrow.py" set loan\_path="C:\Users\Temidayo\Desktop\Test\_Question\Script\Loan\_Scripts\loan.py" set repayment\_path="C:\Users\Temidayo\Desktop\Test\_Question\Script\Repayment\_Scripts\Repayment.py" set schedule\_path="C:\Users\Temidayo\Desktop\Test\_Question\Script\Schedule\_Scripts\schedule.py"  REM Define the path for log files set log\_path="C:\Users\Temidayo\Desktop\Test\_Question\Script\log.txt"  REM Get current date and time for /f "tokens=1-4 delims=/ " %%a in ('date /t') do (set mydate=%%c-%%b-%%a) for /f "tokens=1-4 delims=:. " %%a in ('time /t') do (set mytime=%%a:%%b:%%c)  REM Run borrow script and log output with date and time echo %mydate% %mytime% Running borrow script >> %log\_path% 2>&1 %python\_path% %borrow\_path% >> %log\_path% 2>&1 set borrow\_exit\_code=%errorlevel%  REM Check if borrow script was successful if %borrow\_exit\_code% equ 0 (  REM Borrow script was successful, now run loan script  echo %mydate% %mytime% Running loan script >> %log\_path% 2>&1  %python\_path% %loan\_path% >> %log\_path% 2>&1  set loan\_exit\_code=%errorlevel%    REM Check if loan script was successful  if %loan\_exit\_code% equ 0 (  REM Loan script was successful, now run repayment script  echo %mydate% %mytime% Running repayment script >> %log\_path% 2>&1  %python\_path% %repayment\_path% >> %log\_path% 2>&1  set repayment\_exit\_code=%errorlevel%    REM Check if repayment script was successful  if %repayment\_exit\_code% equ 0 (  REM Repayment script was successful, now run schedule script  echo %mydate% %mytime% Running schedule script >> %log\_path% 2>&1  %python\_path% %schedule\_path% >> %log\_path% 2>&1  set schedule\_exit\_code=%errorlevel%    REM Optionally, you can check schedule script exit code here if needed  if %schedule\_exit\_code% equ 0 (  echo %mydate% %mytime% All scripts completed successfully. >> %log\_path%  ) else (  echo %mydate% %mytime% Schedule script failed. >> %log\_path%  )  ) else (  echo %mydate% %mytime% Repayment script failed. Schedule script will not be executed. >> %log\_path%  )  ) else (  echo %mydate% %mytime% Loan script failed. Repayment and Schedule scripts will not be executed. >> %log\_path%  ) ) else (  echo %mydate% %mytime% Borrow script failed. Loan, Repayment, and Schedule scripts will not be executed. >> %log\_path% )  echo %mydate% %mytime% Task completed. Check log at %log\_path% |

### PowerShell Script

PowerShell provides a means of automating tasks on Windows servers. Ensure files are saved as ***.ps1***

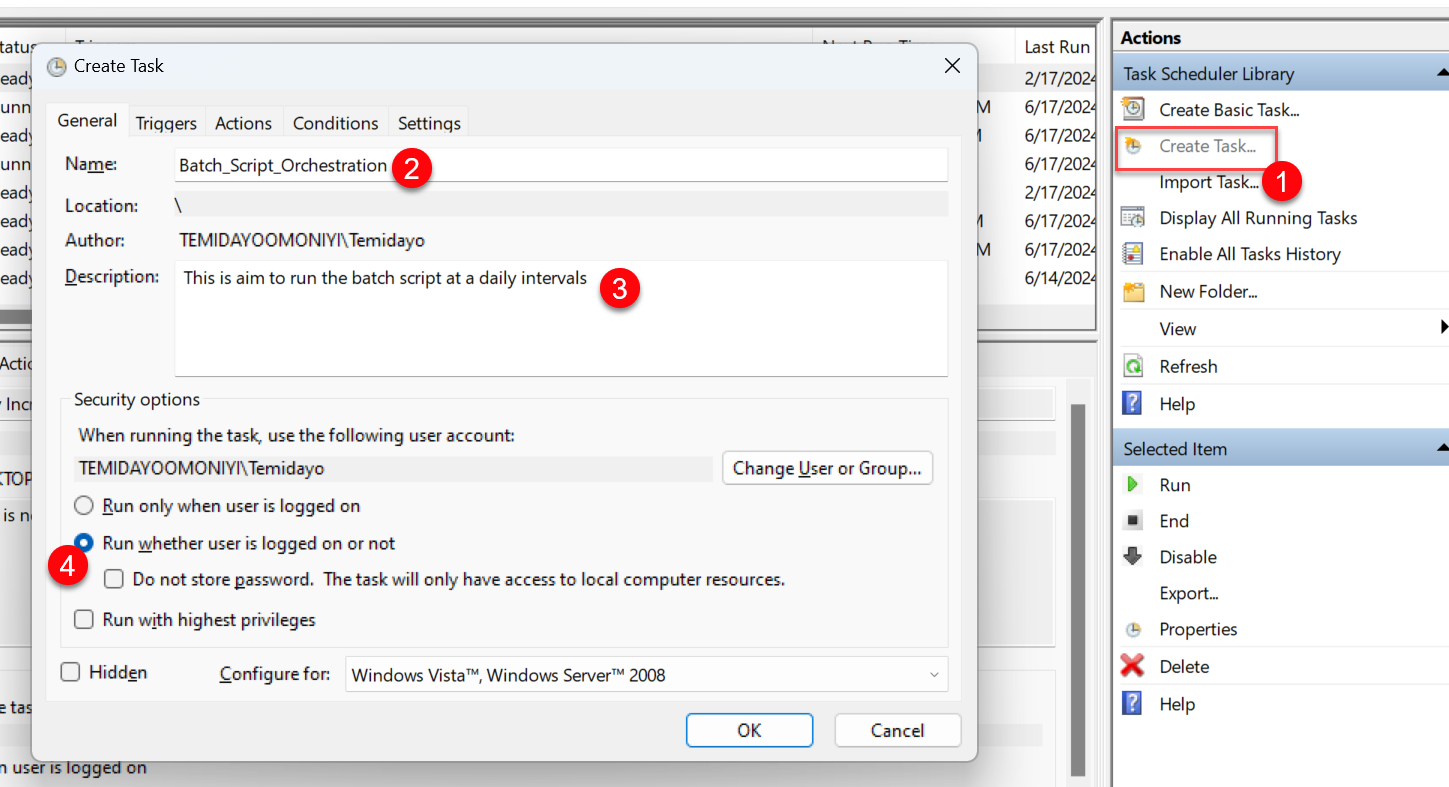
|  |
| --- |
| # Define the path to the log file $logPath = "C:\Users\Temidayo\Desktop\Test\_Question\Script\log.txt"  # Check if the file exists if (Test-Path $logPath) {  # Try to delete the file  try {  Remove-Item -Path $logPath -Force  Write-Output "The file $logPath has been successfully deleted."  }  catch {  Write-Error "An error occurred while trying to delete the file: $\_"  } } else {  Write-Output "The file $logPath does not exist." } |



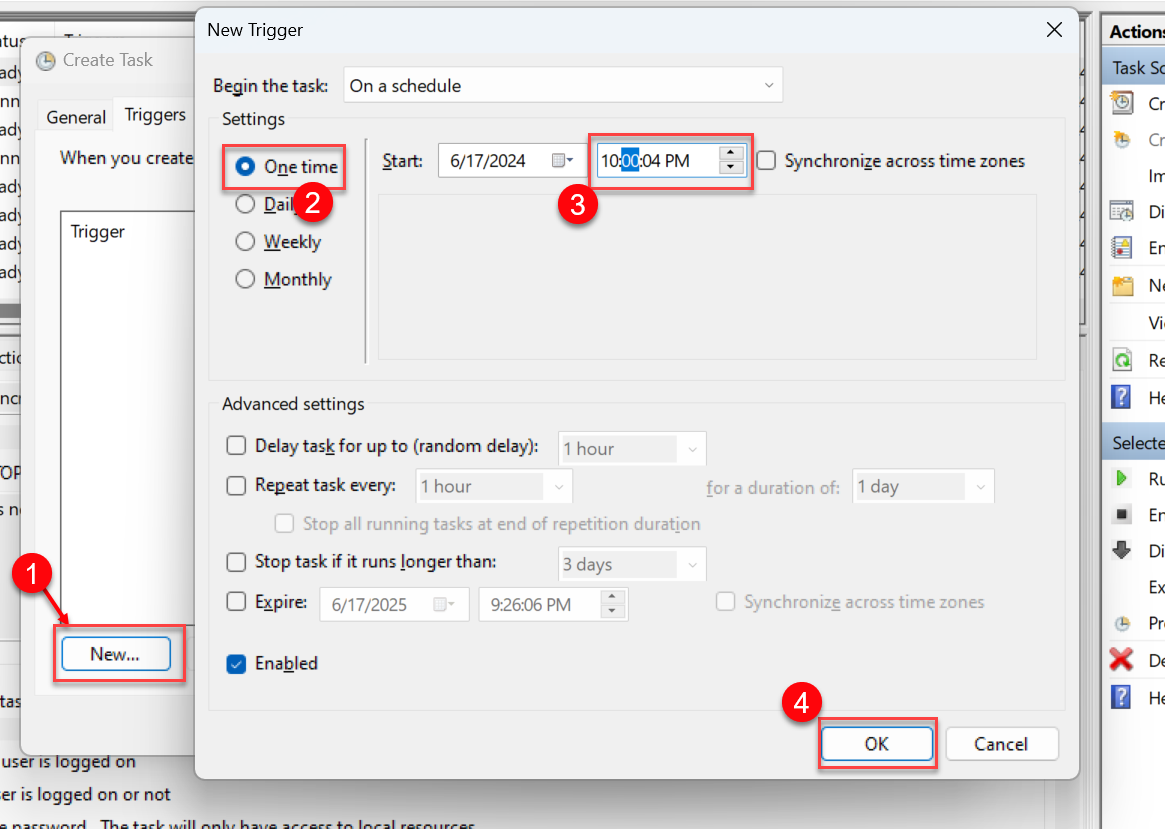
## Create CRON Job Using Task Scheduler

We will be using the Windows Task Scheduler in our Virtual Machine which serves as an ***Infrasture as a Service (IaaS)***.

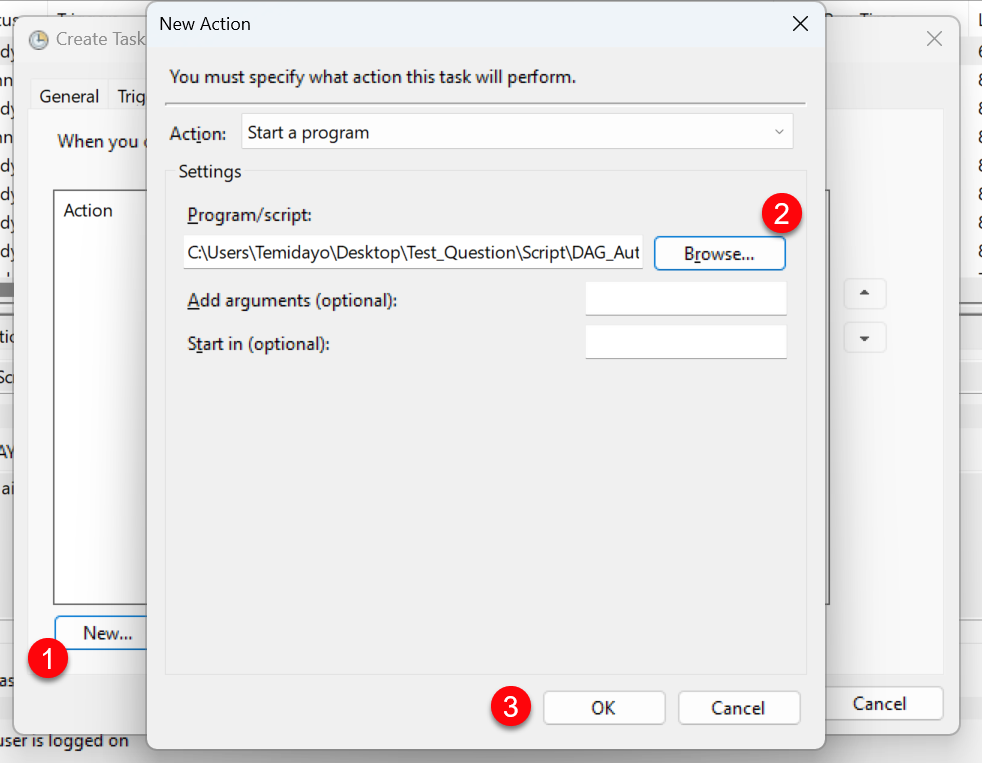
In your Windows server search for a task scheduler and create a new task



To set a trigger create a new trigger and pick the time you want it to run on a daily.



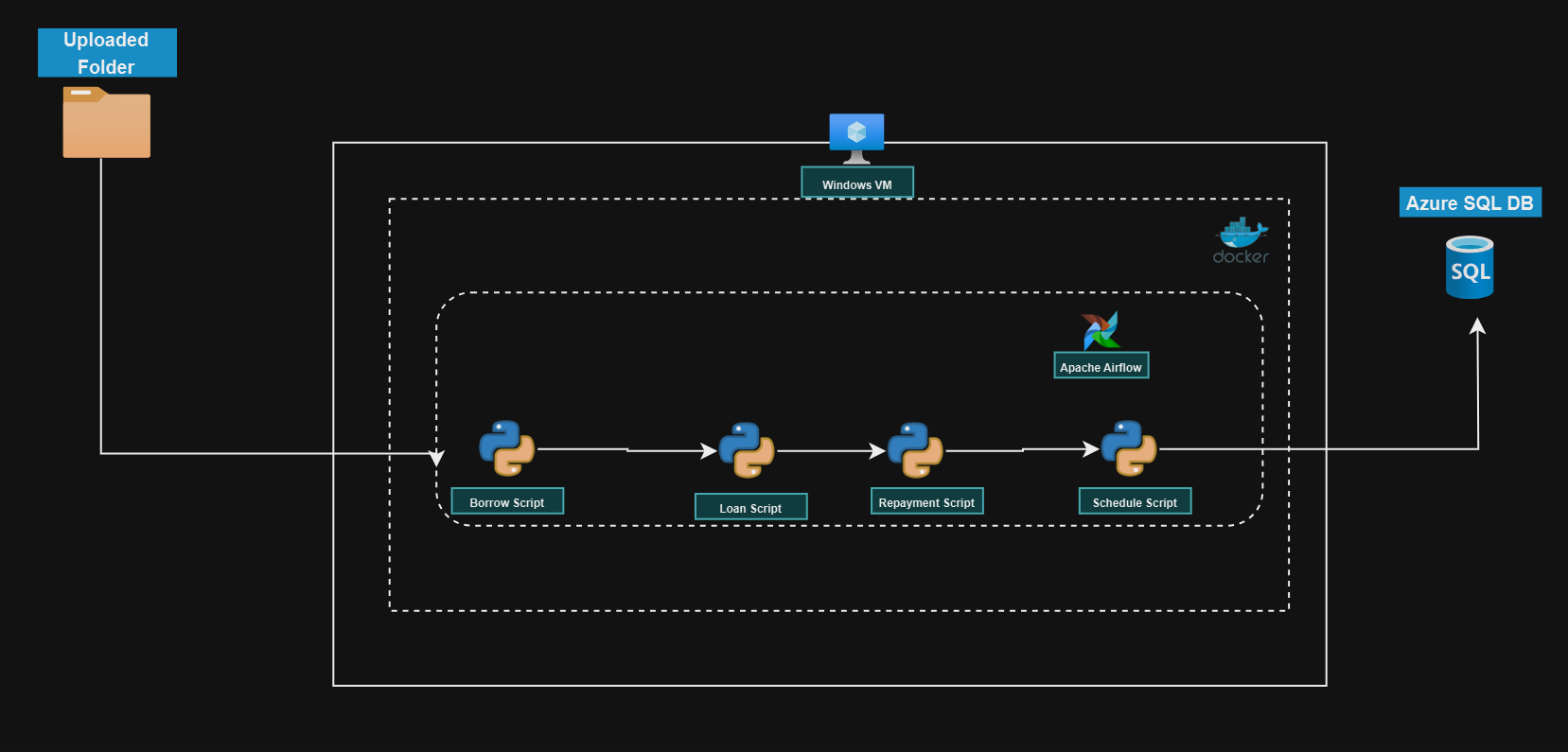
For the action, we want to run the ***.bat*** file that will be used to trigger the other Python scripts.



Save the process and wait for the scheduled time to run.

# Alternative Architecture

The ***IaaS*** approach was used due to cost saving, you can consider other approaches like ***modern data stack*** or ***Platform as a Service (AWS, Azure, or GCP).***



*Alternative Architecture*

Data Gathering

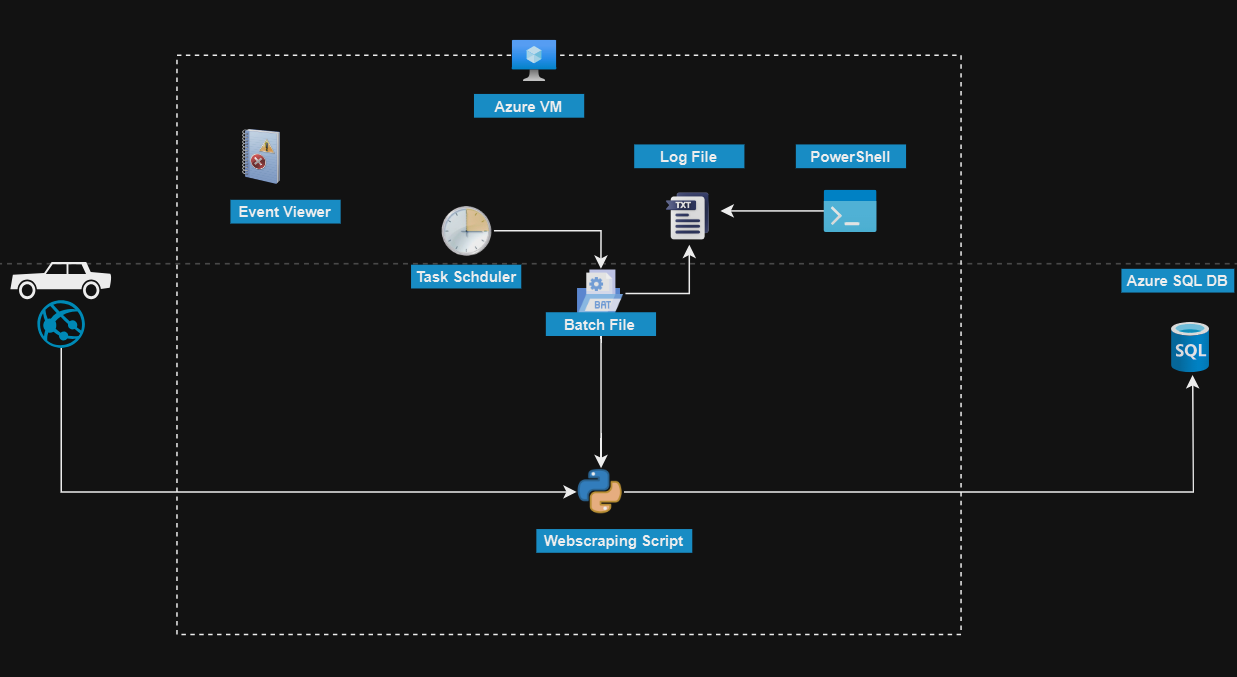
# Introduction

We are expected to perform data by scraping data from an auto-mobile site and getting the necessary fields. The scraped data is standardized and loaded into the Azure SQL Database.

Using the required Libraries, we need to automate the entire process. For this, we will be using the IaaS approach where all the necessary components are stored in a ***Virtual Machine*** of ***Azure Infrastructure as a Service***.

# Project Architecture

The image below explains the entire process that will be carried out. A CRON job process is done on the Batch file which is used to run the Web scraping script and push the data into Azure SQL Database and at the same create a log file for the entire process.



*Project Architecture*

## Code Implementation

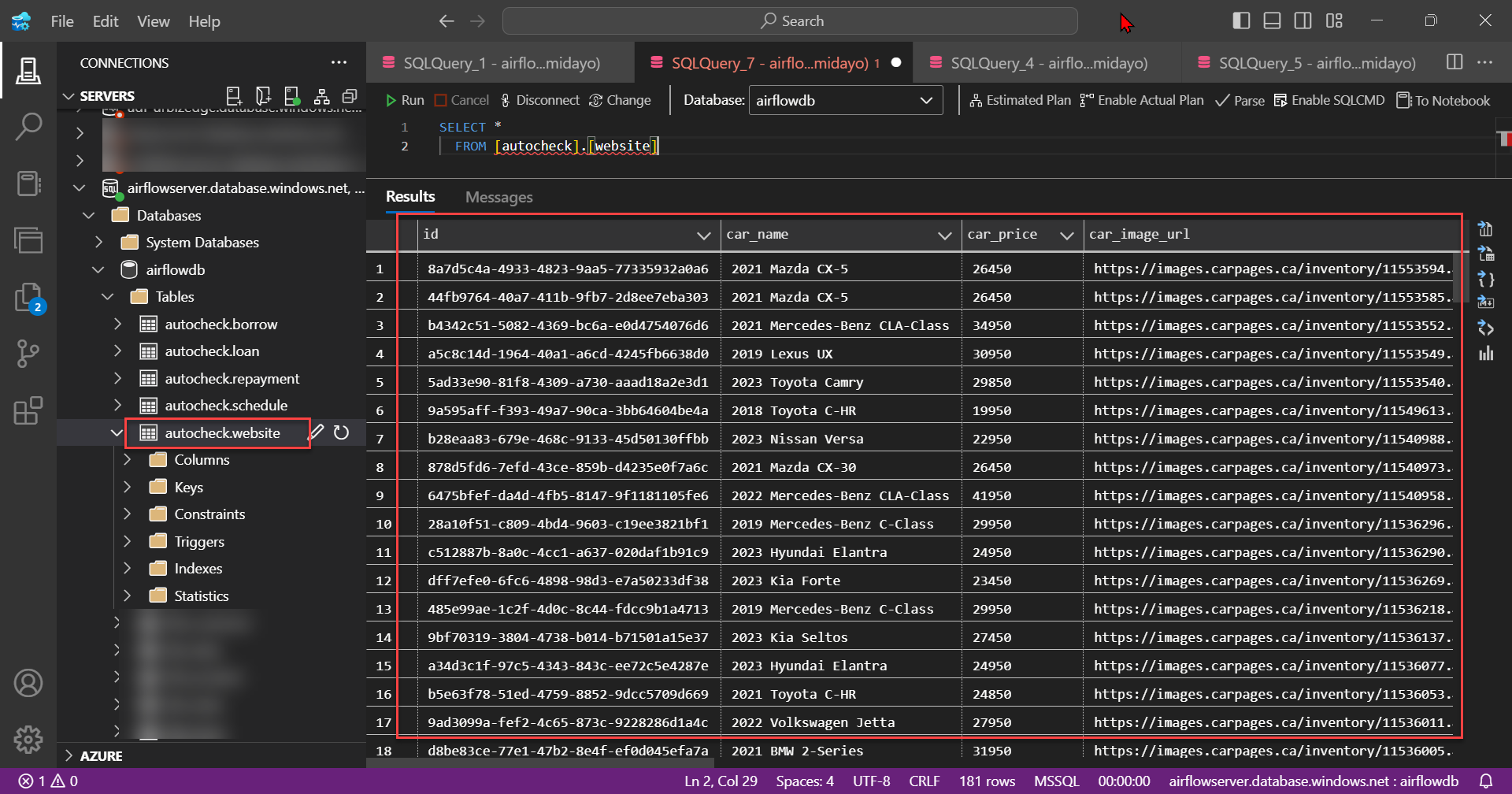
The code below is used in scraping data from <https://www.autorama.ca/vehicles/> website then standardizing it and inserting it into a database.

***Beautiful Soup*** is a Python library designed to simplify the process of extracting data from HTML and XML documents. It creates a parse tree for parsed pages that can be used to navigate, search, and modify the document. It makes it easier to work with the messy structure of web pages.

|  |
| --- |
| # -\*- coding: utf-8 -\*- """ Created on Wed Aug 7 08:06:53 2024  @author: Temidayo """  import time from functools import wraps from typing import Dict, List import re import uuid from datetime import datetime from dotenv import dotenv\_values import os import requests from bs4 import BeautifulSoup as bs import pandas as pd from sqlalchemy import create\_engine import json  BASE\_URL = "https://www.autorama.ca/vehicles/"  def retry\_on\_failure(retries=3, delay=2, exceptions=(requests.exceptions.RequestException,)):  def decorator(func):  @wraps(func)  def wrapper(\*args, \*\*kwargs):  attempt = 0  while attempt < retries:  try:  return func(\*args, \*\*kwargs)  except exceptions as e:  print(f'Attempt {attempt + 1} failed: {e}')  attempt += 1  time.sleep(delay)  raise Exception(f'Failed after {retries} retries')  return wrapper  return decorator   @retry\_on\_failure(retries=3, delay=3) # Retries 5 times with a 3-second delay def fetch\_page\_html(\*, url):  headers = {'User-Agent': 'Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/58.0.3029.110 Safari/537.3'}  response = requests.get(url, headers=headers)  response.raise\_for\_status() # Raises an HTTPError for bad responses  return response.content   def parse\_html\_as\_beautifulsoup(\*, html\_content: bytes)-> bs:  soup\_content = bs(html\_content, "lxml")  return soup\_content    def get\_cars\_info\_soup(\*, soup\_page\_content: bs)-> List:  main\_content\_list = soup\_page\_content.find\_all(name="main")  cars\_content\_soup = [car for main\_content in main\_content\_list   for car in main\_content.find\_all(attrs={"class": "row Vehicle py-4"})]    return cars\_content\_soup   def get\_car\_image\_url(\*, car\_info\_soup: bs)-> str:  image\_info\_list = car\_info\_soup.find\_all(name= "img")    image\_urls = [img\_tag.get("src", "") for img\_tag in image\_info\_list if "src" in img\_tag.attrs]    return " ".join(image\_urls)   def get\_car\_full\_description\_url(\*, car\_info\_soup: bs)-> str:  anchor\_tag\_content = car\_info\_soup.find\_all("a")  anchor\_links = [anchor\_tag.get("href", "") for anchor\_tag in anchor\_tag\_content]  description\_urls = list(set(anchor\_links))  return " ".join(description\_urls)   def get\_car\_name(\*, car\_info\_soup: bs)-> str:  name\_unparsed = car\_info\_soup.h4.text  car\_name = " ".join(name\_unparsed.split())  return car\_name   def get\_car\_interior\_features(\*, car\_info\_soup: bs)-> str:  return car\_info\_soup.small.text   def get\_car\_price(\*, car\_info\_soup: bs)-> float:  pattern = r"\$\d{1,3}(?:,\d{3})\*(?:\.\d{2})?"  price\_list = re.findall(pattern, car\_info\_soup.p.text)  if price\_list:  unparsed\_price = price\_list[0]  price = unparsed\_price.replace("$", "").replace(",", "")  return price    return 0.00    def get\_car\_current\_mileage(\*, car\_info\_soup: bs) -> float:  pattern = r"\d{1,3}(?:,\d{3})\*"  mileage\_list = car\_info\_soup.find\_all(name="p", attrs={"class": "h5"})   mileage\_val = []  for mileage in mileage\_list:  # Find all mileage values in the text and flatten the list  found\_mileage = re.findall(pattern, mileage.text)  mileage\_val.extend(found\_mileage)   # Convert to floats after removing commas  mileage\_val = [float(m.replace(',', '')) for m in mileage\_val]    if mileage\_val:  return mileage\_val[0]  return 0.00   def get\_car\_exterior\_features(\*, car\_info\_soup: bs)-> Dict[str, str]:  features\_soup\_list = car\_info\_soup.find\_all(name = "p", attrs = {"class": "mb-1"})  features\_text\_list = [feature\_soup.text for feature\_soup in features\_soup\_list]  features\_dict = {feature.split(":")[0] : feature.split(":")[1] for feature in features\_text\_list}  features\_dict = {feature.replace("\n", "").strip() : value.replace("\n", "").strip() for feature, value in features\_dict.items()}  return features\_dict   def get\_finance\_promise(\*, car\_info\_soup: bs)-> str:  promise\_soup\_list = car\_info\_soup.find\_all(name = "ul")  promise\_text\_list = [soup\_text.text.replace("\n", "") for soup\_text in promise\_soup\_list]  promise\_text\_list = [text.replace("check icon", "").replace("\n", "") for text in promise\_text\_list]  promise\_text = "".join(promise\_text\_list)  return " ".join(promise\_text.split())   def initialize\_data\_dict()->Dict[str, List]:  data\_dict = {  "id": [],  "car\_name": [],  "car\_price": [],  "car\_image\_url": [],  "full\_description\_url": [],  "interior\_features": [],  "exterior\_features": [],  "current\_mileage": [],  "finance\_promise": [],  "creation\_time": []  }  return data\_dict  def get\_last\_page(\*, soup\_page\_content: bs) -> int:  page\_numbers\_soup\_list = soup\_page\_content.find\_all(name="a", attrs={"class": "page-numbers"})    page\_numbers = [int(soup.text) if soup.text.isdigit() else 0 for soup in page\_numbers\_soup\_list]    last\_page = max(page\_numbers, default=0)    return int(last\_page)   def fill\_data\_dict\_with\_page\_content(\*, data\_dict: Dict[str, List], cars\_info: List[bs])-> Dict[str, List]:  for car\_info in cars\_info:  data\_dict["id"].append(str(uuid.uuid4()))    data\_dict["car\_name"].append(get\_car\_name(car\_info\_soup = car\_info))    data\_dict["car\_price"].append(get\_car\_price(car\_info\_soup = car\_info))    data\_dict["current\_mileage"].append(get\_car\_price(car\_info\_soup = car\_info))    data\_dict["car\_image\_url"].append(get\_car\_image\_url(car\_info\_soup = car\_info))    data\_dict["full\_description\_url"].append(get\_car\_full\_description\_url(car\_info\_soup = car\_info))    data\_dict["interior\_features"].append(get\_car\_interior\_features(car\_info\_soup = car\_info))    data\_dict["exterior\_features"].append(get\_car\_exterior\_features(car\_info\_soup = car\_info))    data\_dict["finance\_promise"].append(get\_finance\_promise(car\_info\_soup = car\_info))    data\_dict["creation\_time"].append(datetime.now())    return data\_dict  """PAGE CONTENT EXTRACTION"""  data\_dict = initialize\_data\_dict()  html\_page\_content = fetch\_page\_html(url = BASE\_URL)  soup\_page\_content = parse\_html\_as\_beautifulsoup(html\_content=html\_page\_content)  last\_page = get\_last\_page(soup\_page\_content= soup\_page\_content)  for page\_num in range(1, last\_page + 1):  print(f"Working on content for page {page\_num}")    page\_url = BASE\_URL + f"?dsp\_page={page\_num}"    html\_page\_content = fetch\_page\_html(url = page\_url)   soup\_page\_content = parse\_html\_as\_beautifulsoup(html\_content=html\_page\_content)  print(f"Content on page {page\_num} extracted successfully")    print("Parsing the data and extracting car content")  page\_cars\_info = get\_cars\_info\_soup(soup\_page\_content = soup\_page\_content)    fill\_data\_dict\_with\_page\_content(data\_dict= data\_dict, cars\_info= page\_cars\_info)  print(f"Page {page\_num} completed\n")   final\_df = pd.DataFrame(data\_dict)    # Step 3: Load environment variables from .env file env\_dir = r'C:\Users\Temidayo\Desktop\Test\_Question\Credentials' env\_values = dotenv\_values(os.path.join(env\_dir, '.env'))  # Function to load environment variables from .env file def get\_db\_credentials(env\_dir):  env\_values = dotenv\_values(os.path.join(env\_dir, '.env'))    sql\_server = env\_values.get("sql\_server")  sql\_database = env\_values.get("sql\_database")  sql\_username = env\_values.get("sql\_username")  sql\_password = env\_values.get("sql\_password")  sql\_driver = 'ODBC Driver 17 for SQL Server'  schema\_name = 'autocheck'  table\_name = 'website'    return {  'sql\_server': sql\_server,  'sql\_database': sql\_database,  'sql\_username': sql\_username,  'sql\_password': sql\_password,  'sql\_driver': sql\_driver  }  # Function to load DataFrame to SQL database def load\_data\_to\_sql(final\_df, table\_name, env\_dir, schema\_name='autocheck'):  try:  # Convert the exterior\_features column to a JSON string  final\_df['exterior\_features'] = final\_df['exterior\_features'].apply(json.dumps)   credentials = get\_db\_credentials(env\_dir)  connection\_string = f"mssql+pyodbc://{credentials['sql\_username']}:{credentials['sql\_password']}@{credentials['sql\_server']}/{credentials['sql\_database']}?driver={credentials['sql\_driver']}"  engine = create\_engine(connection\_string)    # Append DataFrame to SQL table  final\_df.to\_sql(table\_name, engine, schema=schema\_name, if\_exists='append', index=False)    print(f"Data appended to Azure SQL Database table {schema\_name}.{table\_name} successfully.")  except Exception as e:  print(f"Error writing to SQL Database: {str(e)}")  # Check if DataFrame is available before attempting to load it to SQL if final\_df is not None:  table\_name = 'website' # Define your table name here  load\_data\_to\_sql(final\_df, table\_name, env\_dir) |

## Confirm Result

After successfully running the script, login into your Azure SQL Database and perform the select statement to confirm the run.



## Alternative

An alternative model can be done depending on the level of expertise and cost the organization is willing to take on.

