Tasks:

Docker: Time Taken: 35 Mins

1. Create a sample docker container with a Node.js Express app and demonstrate the installation.

```
## Assignment Questions

# You should be able to find what system packages are needed by looking
through the app
To identify the system packages that are needed by a Node.js app, we can
analyse the dependencies and requirements specified within the app's code and
configuration files i.e. 'package.json' file.

# You should not need to change the app code in any way
For this I'm installing a library "express: 4.17.1"

# The app should be running as a non-privileged user
Within the Dockerfile, I have added the steps to
- Create a non-root user
- Set ownership of the application directory to the non-root user
- Switch to the non-root user


# The app should be automatically restarted if crashes or is killed
We can configure Docker restart policy to always to automatically restarted if
crashes or is killed
the command is: "docker run --restart always my-app:1.0"


# The app should maximize all the available CPUs
When running the Docker container, we can allocate the necessary resources and
enable CPU sharing across all the available cores.
the updated command is "docker run -d --name wnergy_app --cpuset-cpus "0-3" --
restart always my-app:1.0"


# Time zone should be in CST.
In the Dockerfile, I have added a step "ENV TZ=America/Chicago" to set the
time zone to CST.

# Follow best practices when writing a docker file.
The following best practices are implemented:
- Using a slim version of the base image to reduce the image size.
- Copying the package.json and package-lock.json files and running npm install
before copying the rest of the app code to take advantage of Docker's layer
caching mechanism.
- Using --only=production flag with npm install to only install production
dependencies and to reduce the image size.
```

```
- Exposing the port on which the app will run to make it clear to the user
which port to connect to.
- Using CMD instead of ENTRYPOINT to allow the command to be overridden if
necessary.
```

2. **Routine tasks**

a. Create a backup script inside the q1-backup directory using either PowerShell or bash.

- The output of the script is a zip archive of all the files in the ./data directory
- The output zip archive should be stored in the ./backups directory with a filename based on the current timestamp in UTC in the format q1-backup-yyyyMMdd-HHmmss.zip as per the example backup archive
- After creating the zip archive, the script should apply a retention policy that will keep only the newest 8 files (based on alphanumeric order) in the data directory, deleting the rest.
- Have the script accept the number of files to keep for retention as a parameter (or if not
  specified, default to 8 as above)

** **Bonus** - Have the script upload the newly created backup archive to Amazon S3 or Azure Blob storage - allow the credentials and bucket/container name to be passed as parameters to the script.

Time Taken: 2 Hrs.

Please refer to script in the folder .\WEnergy-Assignment\2. Routine Tasks\a. backup_script

```
To use the script "backup_script.sh", follow the steps below:

- Make the script executable by running the following command: chmod +x
backup_script.sh.
- Execute the script by running ./backup_script.sh [RETENTION_COUNT], where
[RETENTION_COUNT] is an optional parameter that specifies the number of files
to keep for retention. If not specified, it will default to 8.

After executing the script, it will perform the following actions:

- It will create a zip archive of all the files in the ./data directory. The
filename of the archive will be based on the current timestamp in UTC.
- The archive will be stored in the ./backups directory.
- The script will apply the retention policy by deleting the oldest files in
the ./data directory if the file count exceeds the retention count specified
or the default count of 8.

--------------------------------------------------------------------------
--------------------------------------------------------------------------
---------------
```

```
To ensure the script works correctly, please follow the steps mentioned below
for the AWS S3 addition (Bonus Task):

- Replace the placeholders YOUR_ACCESS_KEY, YOUR_SECRET_ACCESS_KEY, and
YOUR_BUCKET_NAME in the script with your actual AWS access key, secret access
key, and bucket/container name, respectively.

- Make sure you have the AWS CLI installed.

- Configure the AWS CLI with the appropriate credentials by running aws
configure --profile myprofile in the terminal. This command will create a
profile named "myprofile" in the AWS CLI configuration, which will be used in
the script.

- When prompted, enter your AWS Access Key ID, AWS Secret Access Key, default
region, and output format. Provide the corresponding values based on your AWS
account configuration.

- Once you have updated the script and configured the AWS CLI, you can run the
script using the following command:

./updated_backup_script.sh 10

- Replace 10 with your desired retention period. If you omit the retention
period parameter, it will default to 8 as specified in the script.

- After configuring the profile, the AWS CLI will utilize the specified
credentials when executing the "aws s3 cp" command in the script.
```

b. GIT (Time Taken: 15 Mins)

1. How do you set up a script to run every time a repository receives new commits through push?

```
a)
To automatically run a script whenever a repository receives new commits
through a push, you can use Git hooks feature. Git hooks are scripts that Git
can execute at specific points during the repository's lifecycle. In this
case, you can use the post-receive hook, which is triggered after new commits
are received through a push.

Follow these steps to set up the post-receive hook:

- Locate the .git folder within the repository's directory. If hidden files
are not visible, you might need to enable their display.

- Inside the .git folder, navigate to the hooks subdirectory.
```

```
- Create a new file named post-receive (without any file extension) within the hooks directory.

- Make the file executable by running the following command:

chmod +x post-receive

- Open the post-receive file through text editor and write a script that should be executed and save the file. Here you can use any scripting language such as Bash, Python, Ruby.

#!/bin/bash

echo "New commits received!"

Now, whenever someone pushes a new commit to the repository, the script "post-receive" hook will be automatically executed.
```

2. How do you find a list of files that have changed in a particular commit?

```
b)

To find a list of files that have changed in a particular commit, you can use the git diff command.

git diff <commit_id>

The above command will show the detailed changes along with the file names for the specified commit.

You can also use the below command to find the list of files changed in a specified commit.

git diff --name-only <commit_id>
```

C. Generate an export for all instances of running ec2 machines over multiple account mapped to a root account in AWS. Use your choice of language/platform.

Here I will be using Python with Boto3 Library. Boto3 provides easy-to-use APIs for interacting with various AWS services, including EC2.
Time Taken: 4.5 Hrs.

```
To generate an export of all instances of running EC2 machines across multiple
AWS accounts mapped to a root account, you can use the AWS SDK for Python
(Boto3) and assume a role in each account to retrieve the EC2 instances.

In the script, please replace 'account_id1', 'account_id2', 'account_id3' with
the AWS account IDs, and also update role name at:
f'arn:aws:iam::{account_id}:role/YourRoleName'

To run the script:

Install Python, Boto3 library.
Run the command "python 1.all_ec2_running_status.py"

The script will start running, assuming the roles in each AWS account,
retrieving the running EC2 instances information for each account.
```

3. Case study Time Taken: 6 Hrs

Customer wants to build a product for log analysis on AWS, which they want to take to market. The basic features of the MVP will be as follows:

1) Customer Data Management
2) Access to various features based on the plan purchased by the customer.
3) Agent node that can be deployed on the windows or Linux boxes for log collection.
4) Agents will then push the logs to centralized system which will be processed on the server side.
5) UI which will provide various analytics based on the plan to the customer.
6) Expected 100GB per day to be ingested by the server for all logs.
7) Assume that each plan will limit the size the logs that system will allow for ingestion.

Pl create an approach note which will provide the following for the deployment on AWS:

1) Tech Stack for each component
2) Justification for each item in tech stack on why it was considered.
3) Cost considerations on the solution
4) Deployment Architecture Diagram of the solution and cost of the Infra for utilization

For the above case study

- Tech Stack

Tech Stack for Log Analysis Application	
Customer Data Management	Amazon RDS
	Amazon DynamoDB
User Management	Amazon Cognito
Agent Node for Log Collection	AWS Systems Manager (SSM)
Centralized Log Processing	Amazon Kinesis Data Firehose
	Amazon S3
	Amazon Glue
	Amazon Athena
User Interface	AWS Amplify
	React.js
Ingestion of 100GB	Amazon Kinesis Data Firehose
	Amazon S3
Log Ingestion Limits per Plan	AWS API Gateway
	AWS Lambda

- Justification

Customer Data Management – Amazon RDS, Amazon Dynamo DB

Amazon RDS: Amazon RDS provides a managed database service that supports various database engines. It offers scalability, high availability, automated backups, and automated software patching. RDS is suitable for managing customer data securely and efficiently, ensuring data integrity and reliability.

Amazon DynamoDB: DynamoDB is a fully managed NoSQL database service provided by AWS. It offers seamless scalability, high availability, and low latency, making it suitable for handling customer data efficiently. DynamoDB's flexible schema allows easy storage and retrieval of customer-related information.

There are some limitations with DynamoDB compared to RDS:

- Query Flexibility: DynamoDB is designed for key-value pattern and querying data based on non-key attributes can be more challenging.
- Join Operations: DynamoDB does not support SQL join operations.
- Cost: DynamoDB's pricing can be higher compared to RDS

User Management – Amazon Cognito

Amazon Cognito: It is an identity management service that provides user authentication and authorization for web and mobile applications. It offers secure sign-up, sign-in, and access control features, allowing you to define user roles and permissions based on the purchased plan. Cognito integrates well with other AWS services and provides scalability and security for user management.

Agent Node for Log Collection – AWS Systems Manager (SSM)

AWS Systems Manager provides a suite of tools for managing resources on EC2 instances, including agent-based systems management capabilities. You can deploy the Systems Manager Agent on Windows or Linux boxes to collect logs from the respective operating systems. SSM simplifies agent management, provides secure communication, and offers centralized control over agent configuration and deployment.

Centralized Log Processing – Amazon Kinesis Data Firehose, Amazon S3, Amazon Glue, Amazon Athena

Amazon Kinesis Data Firehose enables real-time streaming ingestion of logs and automatically loads the data into data stores for processing. Firehose can directly deliver logs to Amazon S3 for long-term storage. Amazon Glue and Amazon Athena can be used for server-side log processing, data cataloging, and querying. This combination provides a scalable and cost-effective solution for log processing, storage, and analysis.

User Interface – AWS Amplify, React.js

AWS Amplify is a development framework for building scalable and secure web and mobile applications. It simplifies the integration with AWS services, including authentication and authorization with Cognito. React.js is a popular JavaScript library for building user interfaces. It provides a component-based approach, making it easier to develop dynamic and interactive UIs. Using Amplify with React.js enables efficient development of the user interface for log analytics.

100 GB per day Data Ingestion – Amazon Kinesis Data Firehose, Amazon S3

Amazon Kinesis Data Firehose is designed to handle large-scale data ingestion and can handle the expected 100GB per day. Firehose can buffer and batch the log data before delivering it to Amazon S3

for storage. Amazon S3 provides virtually unlimited storage capacity and high durability, making it suitable for storing logs at scale.

Log Ingestion Limits per Plan: AWS API Gateway, AWS Lambda

AWS API Gateway can be used to define and enforce rate limits on log ingestion APIs based on the customer's purchased plan. API Gateway integrates well with AWS Lambda, which can serve as the backend for implementing the rate limiting logic. This combination enables fine-grained control over log ingestion limits, ensuring that each plan adheres to its specified log size restrictions.

- Cost Considerations

Amazon RDS: The pricing of Amazon RDS is determined by factors such as the selected database engine, instance size, storage capacity, and data transfer. It is important to consider the expected storage needs for customer data and choose a suitable instance type accordingly. If necessary, provisions for IOPS or a Multi-AZ deployment should also be taken into consideration.

Amazon Cognito: Amazon Cognito offers a free tier, but beyond that, costs are typically based on the number of monthly active users and the amount of data stored. Estimate the expected number of active users and their usage patterns to determine the pricing impact.

AWS Systems Manager (SSM): AWS Systems Manager is generally included in the AWS Management Console free tier. However, data transfer costs may apply depending on the amount of data collected by the Systems Manager Agent.

Amazon Kinesis Data Firehose: Its pricing is based on data ingestion volume, data delivery destinations, and any optional data transformation or compression. Calculate the expected daily data ingestion volume (100GB) and choose appropriate delivery destinations (e.g., Amazon S3).

Amazon S3: Amazon S3 pricing includes costs for storage, data transfer, and any additional features used (e.g., data retrieval, replication). Assess the storage requirements for log data and determine the expected data transfer volume. Additional costs may occur fir using data life cycle management and cross-region replication.

Amazon Glue and Amazon Athena: Amazon Glue pricing is based on the number of Data Processing Units (DPUs) and crawler runtime, while Amazon Athena pricing is determined by the amount of data scanned during queries. Estimate the required DPUs based on the log processing workload and consider the data scanned during analytics queries.

AWS Amplify and React.js: The cost of AWS Amplify is primarily based on the backend resources utilized, such as AWS Lambda, API Gateway, and storage services. React.js itself is an open-source library and does not incur direct costs.

AWS API Gateway and AWS Lambda: AWS API Gateway pricing is determined by factors such as API requests, data transfer, and additional features used (e.g., caching, custom domains). AWS Lambda pricing is based on the number of function invocations and compute time. Estimate the expected API requests and data transfer, as well as the compute requirements for Lambda functions.

- Approx. cost of the application:

	Min. (In $)	Max. (In $)
Amazon RDS	300	1000
Amazon Cognito	10	30
AWS Systems Manager	Free	
Amazon Kinesis Data Firehose (100GB)	200	600
Amazon S3	30	150
Amazon Glue & Athena	50	200
AWS Amplify	20	100
AWS API Gateway & Lambda	20	100
Total (Approx.)	630	2180

- Architecture