# Question 1 : Database Assignment [github repo link](https://github.com/harshiteuro/flockai-assignment.git) -https://github.com/harshiteuro/flockai-assignment.git

## 1. Lifetime Value of a User

SELECT u.user\_id, u.name, u.email, SUM(o.total\_amount) AS lifetime\_value FROM Users u JOIN Orders o ON u.user\_id = o.user\_id GROUP BY u.user\_id, u.name, u.email ORDER BY lifetime\_value DESC LIMIT 10;

## 2. Product Popularity Over Time

SELECT p.product\_id, p.name AS product\_name, DATE\_FORMAT(o.order\_date, '%Y-%m-01') AS sale\_month, SUM(oi.quantity) AS total\_quantity\_sold FROM OrderItems oi JOIN Orders o ON oi.order\_id = o.order\_id JOIN Products p ON oi.product\_id = p.product\_id WHERE o.order\_date >= CURDATE() - INTERVAL 1 YEAR GROUP BY p.product\_id, p.name, DATE\_FORMAT(o.order\_date, '%Y-%m-01') ORDER BY p.product\_id, sale\_month;

## 3. Customer Retention Rate

WITH user\_months AS (

SELECT

DISTINCT user\_id,

DATE\_FORMAT(order\_date, '%Y-%m') AS order\_month

FROM Orders

WHERE order\_date >= CURDATE() - INTERVAL 1 YEAR

),

retention AS (

SELECT

curr.order\_month AS month,

COUNT(DISTINCT curr.user\_id) AS current\_month\_users,

COUNT(DISTINCT prev.user\_id) AS retained\_users

FROM user\_months curr

LEFT JOIN user\_months prev

ON curr.user\_id = prev.user\_id

AND DATE\_FORMAT(DATE\_ADD(prev.order\_month, INTERVAL 1 MONTH), '%Y-%m') = curr.order\_month

GROUP BY curr.order\_month

)

SELECT

month,

ROUND((retained\_users / NULLIF(current\_month\_users, 0)) \* 100, 2) AS retention\_rate\_percentage

FROM retention

ORDER BY month;

## 4. Average Order Value and Order Frequency by User Segment

WITH user\_orders AS (

SELECT

user\_id,

SUM(total\_amount) AS lifetime\_value,

COUNT(\*) AS total\_orders,

TIMESTAMPDIFF(MONTH, MIN(order\_date), MAX(order\_date)) + 1 AS active\_months

FROM Orders

GROUP BY user\_id

),

ranked\_users AS (

SELECT

user\_id,

lifetime\_value,

total\_orders,

active\_months,

NTILE(10) OVER (ORDER BY lifetime\_value) AS decile

FROM user\_orders

),

segmented\_users AS (

SELECT

user\_id,

lifetime\_value,

total\_orders,

active\_months,

CASE

WHEN decile <= 3 THEN 'Low Spenders'

WHEN decile <= 7 THEN 'Medium Spenders'

ELSE 'High Spenders'

END AS segment

FROM ranked\_users

)

SELECT

segment,

ROUND(AVG(lifetime\_value / total\_orders), 2) AS avg\_order\_value,

ROUND(AVG(total\_orders / active\_months), 2) AS avg\_orders\_per\_month

FROM segmented\_users

GROUP BY segment;

## 5. Conversion Rate of Marketing Campaigns

SELECT

mc.campaign\_id,

mc.campaign\_name,

ROUND(

COUNT(DISTINCT o.user\_id) / COUNT(DISTINCT mc.user\_id) \* 100,

2

) AS conversion\_rate

FROM MarketingCampaigns mc

LEFT JOIN Orders o

ON mc.user\_id = o.user\_id

AND o.order\_date BETWEEN mc.campaign\_start\_date AND mc.campaign\_end\_date

GROUP BY mc.campaign\_id, mc.campaign\_name;

1. Query Design Choices

• All queries were written for MySQL compatibility.

• Standard SQL features like GROUP BY, JOINs, aggregate functions, and window functions were used.

• Common Table Expressions (CTEs) were used where MySQL version supported it.

• The queries aimed to be efficient and readable, supporting future scaling and clarity.

Examples of design choices include:

- Using SUM() + GROUP BY to compute lifetime value (LTV)

- Using DATE\_FORMAT() to group by months

- Using window functions like NTILE() for user segmentation into spend tiers

- Using dynamic date filtering with CURDATE() for time-based reporting

2. INDEXING STRATEGIES

Recommended indexes to optimize performance:

Table Recommended Index

------------------ ---------------------------------------------------

Orders (user\_id), (order\_date)

OrderItems (order\_id), (product\_id)

Products (product\_id)

Users (user\_id)

MarketingCampaigns (user\_id), (campaign\_start\_date, campaign\_end\_date)

3. Query-Specific Assumptions

• Dates are assumed to be clean and consistent (e.g., no NULL order dates).

• Purchases are inferred from records in the Orders and OrderItems tables.

• Retention is based on the assumption that purchasing in back-to-back months means retention.

• Campaign conversion assumes a "conversion" is any purchase made during the campaign period.

• Each marketing campaign is associated with a single user.

• The database does not track order status (e.g., canceled orders were not considered).

# Question 2 : Operating Systems and Threading

import threading

import time

import queue

from collections import defaultdict, deque

# Task object

class Task:

def \_\_init\_\_(self, task\_id, priority, processing\_time):

self.task\_id = task\_id

self.priority = priority

self.processing\_time = processing\_time

self.remaining\_time = processing\_time

def \_\_repr\_\_(self):

return f"Task(id={self.task\_id}, priority={self.priority}, remaining={self.remaining\_time})"

# Task Scheduler class implementation

class TaskScheduler:

def \_\_init\_\_(self, time\_slice=1, num\_workers=2):

self.time\_slice = time\_slice

self.task\_queues = defaultdict(deque) # priority -> deque of tasks

self.lock = threading.Lock()

self.running = True

self.completed\_tasks = []

self.monitor\_thread = threading.Thread(target=self.monitor)

self.workers = [threading.Thread(target=self.worker\_loop) for \_ in range(num\_workers)]

# adding task in priority queue

def add\_task(self, task):

with self.lock:

self.task\_queues[task.priority].append(task)

print(f"[ADD] {task}")

# return the highest priority task in FIFO order

def get\_next\_task(self):

with self.lock:

for priority in sorted(self.task\_queues.keys()):

if self.task\_queues[priority]:

task = self.task\_queues[priority].popleft()

return task

return None

# context switching

def requeue\_task(self, task):

with self.lock:

self.task\_queues[task.priority].append(task)

# running queue + ready queue

def worker\_loop(self):

while self.running:

task = self.get\_next\_task()

if task:

exec\_time = min(self.time\_slice, task.remaining\_time)

print(f"[EXEC] {task.task\_id} for {exec\_time}s")

time.sleep(exec\_time)

task.remaining\_time -= exec\_time

if task.remaining\_time <= 0:

print(f"[DONE] {task.task\_id}")

with self.lock:

self.completed\_tasks.append(task)

else:

self.requeue\_task(task)

else:

time.sleep(0.1)

# monitoring the task status asynchronously

def monitor(self):

while self.running:

with self.lock:

queue\_status = {p: len(q) for p, q in self.task\_queues.items()}

print(f"[MONITOR] Queue sizes: {queue\_status}, Completed: {len(self.completed\_tasks)}")

time.sleep(2)

# starting monitor thread and both workers thread to run asynchronously

def start(self):

self.monitor\_thread.start()

for w in self.workers:

w.start()

# stopping the main thread until all workers threads and monitor thread executes completely

def stop(self):

self.running = False

for w in self.workers:

w.join()

self.monitor\_thread.join()

# main thread - Round Robin(RR) CPU Scheduling

if \_\_name\_\_ == "\_\_main\_\_":

scheduler = TaskScheduler(time\_slice=1, num\_workers=2)

scheduler.start()

scheduler.add\_task(Task("T1", priority=2, processing\_time=5))

scheduler.add\_task(Task("T2", priority=1, processing\_time=3))

scheduler.add\_task(Task("T3", priority=2, processing\_time=4))

scheduler.add\_task(Task("T4", priority=1, processing\_time=6))

scheduler.add\_task(Task("T5", priority=3, processing\_time=2))

time.sleep(15)

scheduler.stop()

### **Task Scheduler Documentation**

**1. Design Choices**

a. Task Structure

- Each task has a unique task\_id, priority (lower number = higher priority), and processing\_time (in seconds).

- Simulates real-world tasks with different urgency and duration.

b. Scheduling Algorithm

- Used priority queues (dictionary of deques) where keys are priority levels.

- Within each priority, tasks are scheduled using round-robin (FIFO order).

- Ensures fairness within priority while maintaining high-priority preference.

c. Time Slicing

- Fixed time slice (1 second) used to simulate round-robin execution.

- Tasks are re-queued if not completed within the current time slice.

**2.** **Synchronization Strategies**

a. Thread Safety

- Shared resources like task\_queues and task\_status are protected using threading.Lock().

- Ensures no race conditions during concurrent access.

b. Worker Threads

- Fixed pool of worker threads fetch tasks based on priority and round-robin.

- Threads run in infinite loop and handle task execution respecting time slices.

c. Monitor Thread

- A separate thread logs the current status of tasks (queued, running, completed).

- Uses locking to read shared data safely.

**3. Assumptions Made**

- Time slice set to 1 second.

- Tasks are manually added using add\_task() method (extendable to support APIs).

- All tasks are assumed to be finite and eventually complete.

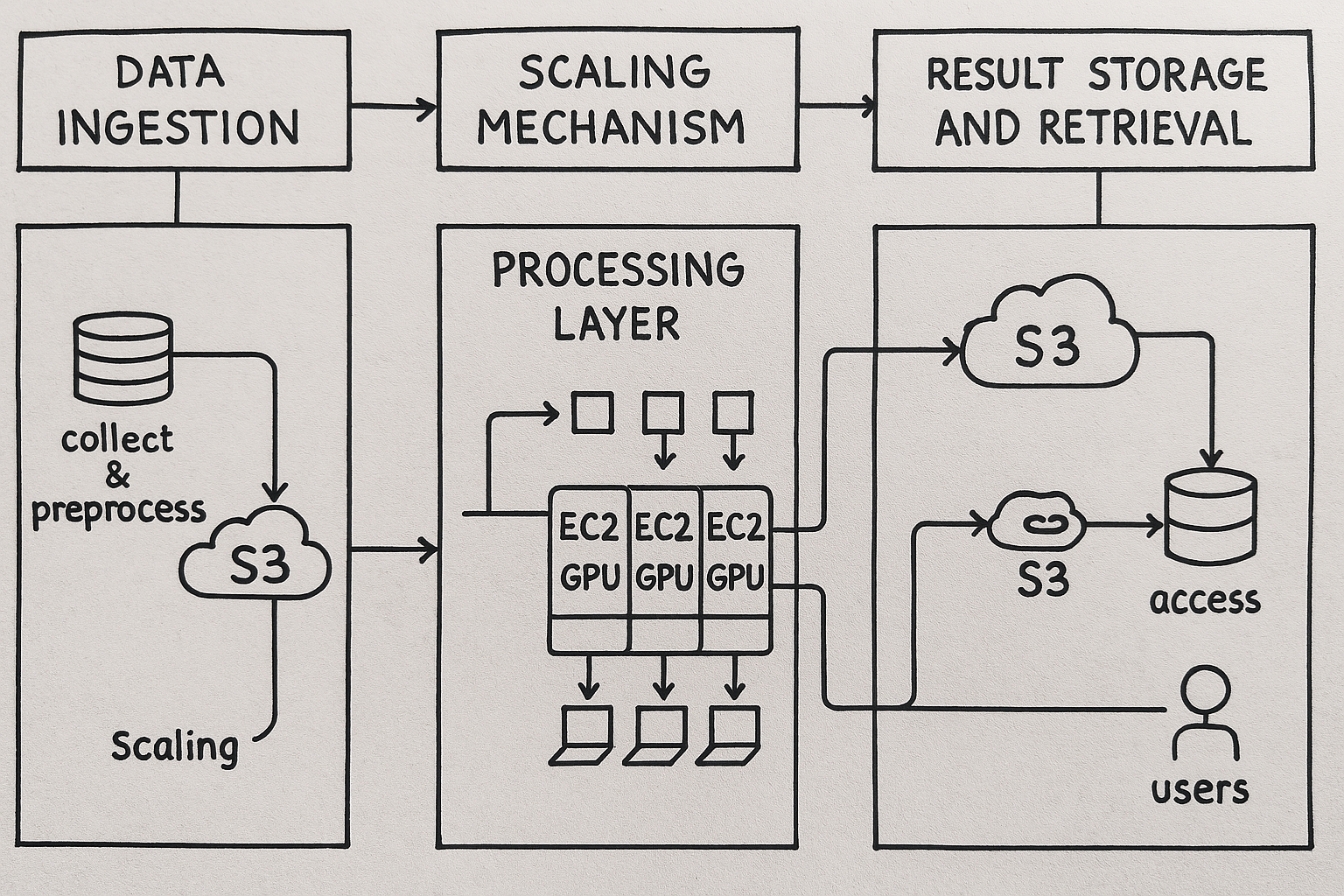
- Designed for limited worker threads (2-4) in demo context.

- Using the priority to put task in ready queue instead of arrival time.

- In-memory task management only (no persistent storage).

- Stopping the main thread to track and monitor the task statuss

# Question 3 : System Architecture Assignment

  
  
2. Detailed Design Explanation

### Data Ingestion Layer

* Collection & Preprocessing:
  + Data is ingested from various sources (APIs, IoT devices, files).
  + Preprocessing includes filtering, normalization, batching, and format conversion.
* Technologies & Tools:
  + AWS S3: Raw data is stored temporarily.
  + AWS Kinesis / Apache Kafka: Real-time streaming ingestion.
  + AWS Lambda: Lightweight preprocessing.
  + AWS Glue: For more complex ETL processes.

### Processing Layer

* EC2 GPU Instances Usage:
  + GPU-based EC2 instances handle ML inference, simulations, or video rendering tasks.
  + Each task is containerized using Docker for portability.
* Recommended Instances:
  + g5.2xlarge/g5.12xlarge (NVIDIA A10G) for ML and graphics workloads.
  + p4d.24xlarge (NVIDIA A100) for deep learning at scale.
* Workload Distribution:
  + Jobs distributed via AWS SQS or ECS Task Queues.
  + AWS ECS or Kubernetes (EKS) to orchestrate containers across GPU instances.

### Scaling Mechanism

* Scaling Parameters:
  + GPU/CPU utilization (CloudWatch metrics).
  + SQS queue length.
  + Processing time per job.
* Auto-Scaling Implementation:
  + EC2 Auto Scaling Group with custom scaling policies.
  + AWS Lambda monitors SQS metrics and dynamically adjusts instance count.
  + Use EKS Horizontal Pod Autoscaler if using Kubernetes.
* Handling Demand Spikes:
  + Pre-warm instances or maintain a minimal number of idle GPU nodes.
  + Use spot instances with fallback to on-demand for cost efficiency.
  + Multi-AZ deployment for HA.

### Result Storage and Retrieval

* Storage:
  + AWS S3 for storing results (versioned buckets).
  + Amazon RDS / DynamoDB for metadata indexing and querying.
* Retrieval:
  + API Gateway + Lambda or Fargate containers for user queries.
  + S3 pre-signed URLs for secure result downloads.
* Optimization & Caching:
  + Use Amazon CloudFront to cache frequently accessed results.
  + Redis (via ElastiCache) for real-time result lookup.

## 3. Implementation Details

### AWS Services and Tools

* Compute: EC2 GPU (G5, P4), Lambda, ECS/EKS
* Storage: S3, RDS/DynamoDB
* Ingestion: S3, Kinesis, Kafka, AWS Glue
* Monitoring: CloudWatch, X-Ray
* Scaling: Auto Scaling Groups, Lambda Triggers
* Security: IAM, VPC, Security Groups, KMS

### Configuration and Setup

1. Create VPC, subnets, route tables.
2. Setup S3 buckets and IAM policies.
3. Configure Kinesis streams and SQS queues.
4. Deploy ECS/EKS cluster with GPU-enabled AMIs.
5. Setup Auto Scaling policies with thresholds.
6. Use AWS Glue for ETL and Lambda for triggers.
7. Deploy API Gateway for user-facing endpoints.

### Monitoring and Logging

* CloudWatch Metrics: For EC2, Lambda, GPU usage.
* CloudWatch Logs: For ECS tasks, Lambda functions.
* X-Ray: For tracing API calls.
* ELK Stack (optional): Centralized log analysis.

### Security Considerations

* Use IAM roles with least privilege.
* All services inside private subnets with NAT access.
* Enable encryption at rest and in transit (S3, RDS, etc.).
* Use VPC endpoints for secure S3 access.
* API Gateway protected via Cognito / API keys.

## 4. Challenges and Solutions

| Challenge | Solution |
| --- | --- |
| Handling large data volumes | Stream processing (Kinesis), S3 for scalable storage, Glue for batching |
| GPU resource allocation | ECS task scheduling, SQS-based job dispatching |
| Seamless scaling | Auto Scaling Groups, CloudWatch alarms, warm pool |
| Latency/performance | CloudFront caching, Redis caching layer, instance pre-warming |