Assignment 4 problems：

Part 1 Database design

**Why choose dynamodb:**

## Scalability

* Seamless scaling: DynamoDB automatically scales to handle traffic spikes during peak ski season without manual intervention
* No capacity planning: Can handle millions of lift ride records without database restructuring
* Partition-based: Distributes ski resort data across many partitions for parallel processing

## Performance

* Single-digit millisecond responses: Critical for real-time lift ride recording
* Consistent performance: Maintains speed regardless of data size (whether tracking 100 or 100,000 skiers)
* Global tables: Can replicate data across regions for multi-resort operations

## Schema Flexibility

* Schema-less design: Easy to add new attributes (like equipment rentals or lesson data) without migrations
* Varied data types: Accommodates different data models for different ski resort operations

## Operational Benefits

* Fully managed: No database administration overhead
* High availability: 99.99% availability SLA with multi-AZ replication
* Pay-per-use pricing: Cost effective for seasonal ski resort traffic patterns
* Automated backups: Built-in point-in-time recovery

## Use Case Alignment

* Time-series data: Ideal for lift ride tracking which is time-series in nature
* Key-based access patterns: Your queries match DynamoDB's access pattern strengths
* Sparse data: Only store attributes that exist (not all skiers use all lifts)

## Compared to Alternatives:

vs. RDS:

* No need for complex joins (your queries are key-based)
* Better handles write-heavy workloads (constant lift ride recording)
* No schema rigidity when business requirements change

vs. Redis:

* Better persistence guarantees (Redis is primarily in-memory)
* Superior horizontal scaling for large datasets
* More sophisticated querying with GSI and LSI options
* Built-in backup and disaster recovery

Cons:

Must design the schema with target query in mind. RDS is more flexible with different queries.

**Why skierId as partition key and combination of dayId#liftId#timestamp as sort key:**

Most of the query are based on skierId.

Concatenation of the attributes gives more freedom on making queries using “begin with” “end with”

**Why choosing resort-day and skier-day as GSI:**

The query is asking for unique skiers for resort/season/day. There are probably more resorts than seasons and days. Making season/day to be GSI partition key may make the partition too “hot”. why day is sort key than season -> same reason.

why we use skier-day GSI rather than make it LSI or use query “begin with”?:

### Global Secondary Index (current approach)

* Pros for fast reads:
  + Direct lookup using exact equality on both skierId and dayId
  + No need for begins\_with operations which can be less efficient
  + Smaller index size due to projected attributes
  + Separate capacity from base table (can provision higher read capacity)
* Cons:
  + Eventually consistent by default (can request strongly consistent reads at higher cost)
  + Additional write capacity needed to maintain the index

### Local Secondary Index (alternative)

* Pros for fast reads:
  + Strongly consistent reads available
  + Can be slightly faster than GSIs due to being co-located with base table items
* Cons:
  + Must be created at table creation time
  + Shares provisioned throughput with the base table
  + Still has the 10GB limit per partition

### Primary Table with begins\_with (no additional index)

* Pros:
  + Strongly consistent reads
  + No additional cost or complexity
* Cons:
  + Less efficient query pattern (range scan with begins\_with rather than equality)
  + Must retrieve all attributes (unless using projection expressions)

System design:

· **For GET EC2 Instances**:

* Implement three-level caching:
  + Local in-memory cache (very short TTL)
  + Redis distributed cache (medium TTL)
  + DynamoDB with eventual consistency

· **For POST EC2 Instances**:

* Implement request validation
* Use fire-and-forget message publishing with minimal blocking
* Queue depth monitoring to detect issues

· **For Consumer Instances**:

* Implement idempotent processing (safe to retry)
* Batch writes where possible
* Dead letter queue handling for failed messages

# System Architecture Risks Professor Questions

## Potential Risks

### Single Points of Failure

* Each component has only 1 instance, creating multiple single points of failure ----multiple server & rmq & consumer
* If RabbitMQ goes down, your entire message queue is unavailable --simply return 404 or storing locally in a blocking queue or having local persistent store (SQLite, LevelDB, or files)
* DynamoDB access failure would affect both server and consumer --retry/ Dead letter queue/ Get only from redis
* Redis failure could impact caching and server performance -- read from db

### Scalability Concerns

* Single instances of each component limit horizontal scaling
* Potential bottlenecks during high traffic periods
* No apparent load balancing strategy

### Data Consistency and Durability

* Message delivery guarantees between server and consumer via RabbitMQ
* Potential data loss if RabbitMQ fails before messages are consumed
* Redis data persistence configuration (if used for more than caching)

### Network Issues

* Network partitions between distributed components
* Latency between server and DynamoDB or RabbitMQ
* Connection pool management under load

### Security Vulnerabilities

* Access control between components
* Data encryption in transit and at rest
* Authentication mechanisms between services

## Questions a Professor Might Ask

1. "What happens if RabbitMQ crashes while processing messages?"
2. "How do you ensure data consistency between DynamoDB writes from both server and consumer?"
3. "What Redis eviction policies have you implemented and why?"
4. "Explain your strategy for handling backpressure when the consumer can't keep up with message production."
5. "How would you scale this architecture if traffic increased 10x?"
6. "What monitoring and alerting have you implemented?"
7. "How do you handle idempotency for messages that might be processed multiple times?"
8. "What's your disaster recovery plan for each component?"
9. "How do you manage database connection pools efficiently?"
10. "What circuit breaking patterns have you implemented to prevent cascading failures?"
11. "How do you handle schema evolution in DynamoDB?"
12. "What's your strategy for dead letter queues in RabbitMQ?"