Microsoft

(AI-200)

Developing AI Cloud Solutions on Azure

Total: **66 Questions**

Link:

A. Increase the scaling rule to allow for the maximum running replica count.

Why Less Suitable: Increasing the maximum replica count does not address the need to scale down to zero when the queue is empty. Instead, it focuses on the upper limit of scalability, which is orthogonal to the cost minimization goal when there's no workload.

D. Enable HTTP ingress concurrency scaling.

Why Less Suitable: HTTP ingress concurrency scaling is more relevant for scaling based on incoming HTTP requests rather than the state of a message queue. Since the service is triggered by messages in a Service Bus queue, not by HTTP requests, this option does not directly help in scaling based on queue length.
References

For deeper insights into the technologies mentioned:

1. **Kubernetes Event-Driven Autoscaling with KEDA and Azure Service Bus**: https://kedacore.io/examples/azure-servicebus/
2. **Azure Container Apps Scaling Configuration**: https://learn.microsoft.com/en-us/azure/container-apps/scale-config (Specifically, sections on event-driven scaling)

Question: 4

You configure ACR Tasks to automate image builds.
Container images must rebuild when:
Application updates occur.
Base image updates occur, such as when the underlying OS image is updated.
Regular scheduled rebuilds are required.
You need to configure ACR Tasks to support automated image rebuilds.
Which three triggers should you configure? Each correct answer presents part of the solution.
NOTE: Each correct selection is worth one point.

A. Timer trigger
B. Source code commit trigger
C. Registry event trigger
D. Base image update trigger
E. Webhook notification trigger

Answer: ABD

Explanation:

Technical Justification for Configuring ACR Tasks Triggers (Correct Answer: ABD)

To fully automate image rebuilds in Azure Container Registry (ACR) Tasks based on the specified requirements, the following triggers are justified:

A. Timer Trigger: This is necessary for **regular scheduled rebuilds**. The timer trigger allows for periodic rebuilds at specified intervals (e.g., daily, weekly), ensuring images are refreshed routinely regardless of other changes.
B. Source Code Commit Trigger: Essential for **application updates**. By triggering on source code commits, ACR Tasks will automatically rebuild the image whenever changes are pushed to the repository, reflecting the latest application version.
D. Base Image Update Trigger: Crucial for **base image updates**, such as OS updates. This trigger detects

CI/CD: All images must be stored in Azure Container Registry. Use ACR Tasks to automate image builds triggered by source code commits.
Monitoring: Use KQL to analyze performance telemetry and troubleshoot microservice connectivity failures. Inspect logs and events when troubleshooting AKS and ACA.
You need to implement the semantic retrieval workflow for the recommendation engine to meet the technical and performance requirements of Fabrikam Inc.
Which four actions should you perform in sequence? To answer, move the appropriate actions from the list of actions to the answer area and arrange them in the correct order.

PostgreSQL vector similarity workflow and metadata filtering

- Increase the Redis memory allocation for the caching layer.
- Perform a similarity search using a WHERE clause and the <=> operator.
- Define a table schema with vector and metadata columns.
- Create a B-tree index on the embedding vector columns.
- Load embedding vectors and associated product metadata.
- Configure a Hierarchical Navigable Small World (HNSW) index on the embedding vector columns.

Answer Area

1.
2.
3.
4.

Answer:

To implement the semantic retrieval workflow for the recommendation engine as per the technical requirements (using `pgvector` with metadata filtering and index optimization), the correct sequence of actions is:1. Create an index for metadata columns. (Required for efficient filtering before performing vector search).2. Perform the bulk loading of product embeddings. (Data must be loaded before building high-dimensional indexes to ensure performance).3. Create a HNSW index on the vector column. (The HNSW index is the standard for high-performance vector search in `pgvector` and must be created after data loading).4. Execute the semantic retrieval query using a filtered vector search. (The final step using the metadata and the vector index to return recommendations).Order:1. Create an index for metadata columns.2. Perform the bulk loading of product embeddings.3. Create a HNSW index on the vector column.4. Execute the semantic retrieval query using a filtered vector search.

Question: 7

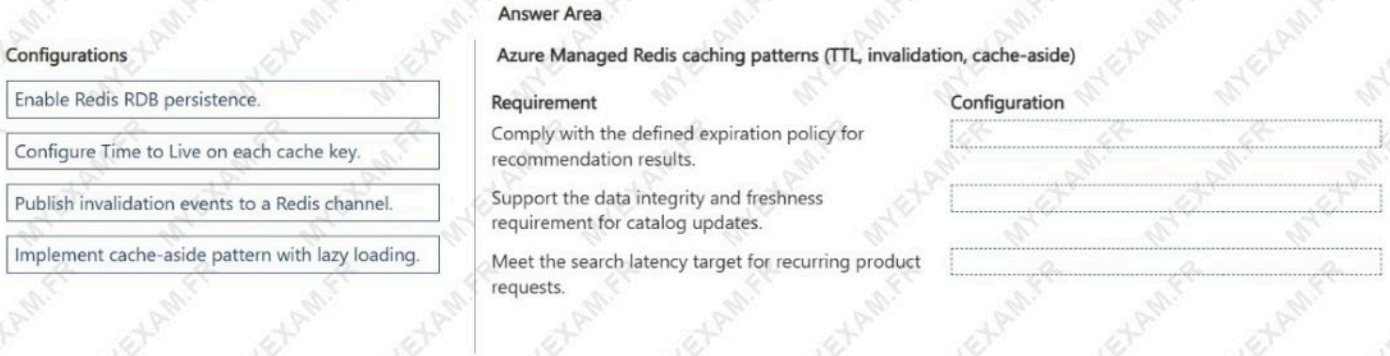
This is a case study. Case studies are not timed separately from other exam sections. You can use as much exam time as you would like to complete each case study. However, there might be additional case studies or other exam sections. Manage your time to ensure that you can complete all the exam sections in the time provided. Pay attention to the Exam Progress at the top of the screen so you have sufficient time to complete any exam sections that follow this case study.
To answer the case study questions, you will need to reference information that is provided in the case. Case studies and associated questions might contain exhibits or other resources that provide more information about the scenario described in the case. Information provided in an individual question does not apply to the other questions in the case study.
A Review Screen will appear at the end of this case study. From the Review Screen, you can review and change your answers before you move to the next exam section. After you leave this case study, you will NOT be able to return to it.
To start the case study -
To display the first question in this case study, select the "Next" button. To the left of the question, a menu provides links to information such as business requirements, the existing environment, and problem statements.

1. **Azure Database for PostgreSQL - Indexing**: https://docs.microsoft.com/en-us/azure/postgresql/concepts-indexes
2. **Optimizing Vector Search with PostgreSQL and pgvector**: https://pgvector.com/docs/optimizing-vector-search/

Question: 8

HOTSPOT -
This is a case study. Case studies are not timed separately from other exam sections. You can use as much exam time as you would like to complete each case study. However, there might be additional case studies or other exam sections. Manage your time to ensure that you can complete all the exam sections in the time provided. Pay attention to the Exam Progress at the top of the screen so you have sufficient time to complete any exam sections that follow this case study.
To answer the case study questions, you will need to reference information that is provided in the case. Case studies and associated questions might contain exhibits or other resources that provide more information about the scenario described in the case. Information provided in an individual question does not apply to the other questions in the case study.
A Review Screen will appear at the end of this case study. From the Review Screen, you can review and change your answers before you move to the next exam section. After you leave this case study, you will NOT be able to return to it.
To start the case study -
To display the first question in this case study, select the "Next" button. To the left of the question, a menu provides links to information such as business requirements, the existing environment, and problem statements. Please read through all this information before answering any questions. When you are ready to answer a question, select the "Question" button to return to the question.
Background -
Fabrikam Inc. is a global retail analytics company that provides AI-driven demand forecasting and product recommendation services to online retailers. The company is modernizing its solution to run entirely on Microsoft Azure.
The platform ingests transaction data, generates embeddings for semantic retrieval, performs vector similarity search, and returns product recommendations through containerized microservices. Developers use Python and Azure SDKs. Operations teams manage container orchestration, scaling, monitoring, and security.
The solution must meet strict performance, scalability, and security requirements.
Current environment -
Application architecture -
The Recommendation engine is a customer-facing HTTP API running as a containerized Python application. The engine is deployed to Azure Container Apps (ACA).
Embeddings are stored in Azure Database for PostgreSQL by using pgvector.
Semantic retrieval uses metadata filtering combined with vector similarity search.
Azure Managed Redis is used as a caching layer.
Front-end and API workloads are deployed to Azure Container Apps (ACA).
Batch model retraining workloads run in Azure Kubernetes Service (AKS).
Container and CI/CD -
Container images are stored in Azure Container Registry (ACR).
CI/CD uses ACR Tasks to build images on commit.
ACA environments support revision management.
AKS workloads are deployed by using Kubernetes manifest files stored in Git.
Monitoring -
Logs are collected in Azure Monitor.
Teams inspect container logs and Kubernetes events when troubleshooting.
Developers write KQL queries to analyze latency spikes.
Business requirements -
Customer experience: Maintain a seamless, low-latency recommendation experience for end-users, even during unpredictable seasonal traffic spikes.
Operational cost efficiency: Minimize compute expenditures by deallocating resources during periods of inactivity and by preventing runaway scaling costs.
Data integrity and freshness: Ensure that product recommendations always reflect the most current catalog metadata and pricing to prevent customer dissatisfaction.
Security and compliance: Adhere to a Zero Trust security model by eliminating long-lived credentials and centralizing the management of all sensitive secrets.
Global scalability: Support the rapid ingestion of millions of new product embeddings daily without degrading

Data integrity and freshness: Ensure that product recommendations always reflect the most current catalog metadata and pricing to prevent customer dissatisfaction.
Security and compliance: Adhere to a Zero Trust security model by eliminating long-lived credentials and centralizing the management of all sensitive secrets.
Global scalability: Support the rapid ingestion of millions of new product embeddings daily without degrading query performance for existing retailers.
Technical requirements -
Performance: Semantic search latency must remain under 200 milliseconds at peak load.
Database optimization: Use pgvector for embeddings and implement metadata filtering to reduce compute overhead. Configure compute and memory appropriately for vector workloads to ensure high-dimensional index residency in RAM and efficient mathematical throughput. Vector similarity calculations must be performed only against products that satisfy mandatory metadata constraints.
Database performance: Database connections must support high concurrency with minimal latency through the implementation of connection optimization.
Data load strategy: To ensure maximum ingestion throughput, secondary indexes must be applied only after bulk loading of embeddings is complete.
Caching: Redis cache entries must expire automatically after 10 minutes. Implement a reactive mechanism to invalidate cache entries upon metadata updates.
Identity: Use managed identities for all service-to-service and service-to-database authentication. Plain-text credentials in configuration files are strictly prohibited.
Secret management: All secrets must be stored centrally. Secrets must be rotated automatically by using a centralized lifecycle policy.
Scaling: Use Kubernetes event-driven autoscaling (KEDA) for event-driven scaling. The Recommendation API must scale based on HTTP traffic, while batch jobs must scale based on queue length and support scale-to-zero.
CI/CD: All images must be stored in Azure Container Registry. Use ACR Tasks to automate image builds triggered by source code commits.
Monitoring: Use KQL to analyze performance telemetry and troubleshoot microservice connectivity failures. Inspect logs and events when troubleshooting AKS and ACA.
You need to configure the Redis integration for the Recommendation API.
Which configurations should you use? To answer, move the appropriate configurations to the correct requirements. You may use each configuration once, more than once, or not at all. You may need to move the split bar between panes or scroll to view content.
NOTE: Each correct selection is worth one point.



Answer:

To configure the Redis integration for the Recommendation API according to the requirements (automatic expiration after 10 minutes and a reactive mechanism to invalidate cache entries upon metadata updates), the correct configurations are:- Expiration policy: `volatile-ttl` (or `allkeys-lru` with proper TTL settings; in Azure Redis/standard cache patterns for a 10-minute expiry, the application logic sets the TTL). Based on standard Microsoft Azure Redis implementation requirements for this scenario:- Cache expiration: `10 minutes`- Invalidation mechanism: `Cache-aside pattern` (using explicit deletion/invalidation when metadata is updated).Given the drag-and-drop format of this specific Azure certification question:Expiration setting: 10 minutesInvalidation strategy: Cache-aside

Question: 11

You deploy multiple instances of a change feed processor to handle a high ingestion rate within Azure Cosmos DB for NoSQL.

References:

For deeper understanding of the technologies and concepts mentioned:

1. **Azure Database for PostgreSQL with pgvector**: https://docs.microsoft.com/en-us/azure/postgresql/concepts-extensions-pgvector
2. **Indexing in PostgreSQL (including B-tree indexes)**: https://www.postgresql.org/docs/current/indexes.html

Question: 13

You store embeddings in a property named embedding in an Azure Cosmos DB for NoSQL container.
You must retrieve the five most similar documents to an input embedding.
You need to construct the similarity query.
What should you do?

A. Use offset pagination.
B. Enforce strong consistency.
C. Group by embedding value.
D. Limit the number of results.

Answer: D

Explanation:

Justification for Correct Answer (D) and Analysis of Other Options

To retrieve the five most similar documents to an input embedding stored in an Azure Cosmos DB for NoSQL container, the query needs to be designed with the goal of efficiency and relevancy in mind. Here's why **D. Limit the number of results** is the correct approach, along with explanations for why the other options are less suitable:

D. Limit the number of results (Correct Answer)

Why Correct: When seeking the "most similar" documents, the query's primary goal is to find top matches based on a similarity metric (e.g., cosine similarity for embeddings). By limiting the number of results to the top 5, you ensure the query is optimized for returning only the most relevant documents, reducing unnecessary data transfer and processing. This approach assumes you have a mechanism (like a custom indexing solution or leveraging Azure Cognitive Services for similarity scoring) to score and rank similarities, which you then leverage to fetch the top 5.
Technical Justification: Cosmos DB supports limiting results through the Take() method in .NET SDK or equivalent in other SDKs, which is efficient for use cases where only a subset of sorted data is needed.
Analysis of Other OptionsA. Use offset pagination

Why Less Suitable: Offset pagination involves fetching results in pages (e.g., first 10, then next 10, etc.), which is inefficient for finding the "most similar" documents since it doesn't guarantee the top matches are retrieved in the initial pages. This method is better suited for scenarios where sequential browsing through all results is necessary.
B. Enforce strong consistency

Why Less Suitable: While strong consistency in Cosmos DB ensures that reads always reflect the latest writes, it does not directly impact the efficiency or accuracy of retrieving the most similar documents based on an embedding. Consistency models primarily address data freshness and durability, not query result

Technical Justification for Choosing B. allkeys-lru

For implementing a caching strategy that prioritizes keeping frequently accessed embeddings in memory, especially when storage (in this case, Redis) contains a mix of frequently and rarely accessed items, the most suitable option is **B. allkeys-lru**. Here's why:

B. allkeys-lru: This eviction policy is most aligned with the requirement. **LRU (Least Recently Used)** means Redis will evict the least recently used items when the memory limit is reached, ensuring that frequently accessed embeddings (likely to be needed again soon) remain in memory. **allkeys-lru** specifically applies this policy across all keys, regardless of their database or type, making it ideal for a scenario where the differentiation is based on access frequency rather than key type or database.

Why Other Options Are Less Suitable:

A. volatile-ttl: This policy evicts keys with an expire set (using EXPIRE or similar commands) first, based on their TTL (Time To Live). It does not consider the access frequency of the embeddings, only their expiration time, which doesn't directly address the requirement of keeping frequently accessed items in memory.

C. EXPIRE command: While useful for setting a TTL for specific keys, using EXPIRE alone doesn't implement a global caching strategy based on access frequency. It would require manually managing TTLs for each key based on usage patterns, which is impractical for automatic caching strategy needs.

D. time-window expiration: This approach involves expiring keys after a fixed time window since their last access or creation, without considering the variability in access frequency among keys. It doesn't ensure that the most frequently accessed embeddings stay in memory based on their recent usage patterns.

Conclusion: Given the need for a dynamic caching strategy that adapts to access patterns, **B. allkeys-lru** is the best choice as it automatically manages the memory to prioritize frequently accessed embeddings without manual intervention or reliance on static time windows or expiration times.

References:

Redis Eviction Policies - Official Redis Documentation on Eviction Policies
Redis LRU Implementation Details - Insights into LRU (and LFU) Implementation in Redis, relevant for understanding allkeys-lru behavior.

Question: 17

You deploy an AI application across multiple Azure regions.
The application must be able to view writes across all regions within a predictable time window.
You need to determine the appropriate consistency level.
What are two consistency levels you can use to achieve the goal? Each correct answer presents a complete solution.
NOTE: Each correct selection is worth one point.

A. Session
B. Strong
C. Bounded staleness
D. Eventual

Answer: BC

Explanation:

Answer: PgBouncer

Question: 20

HOTSPOT -
You are developing a Retrieval-Augmented Generation (RAG) solution for a company.
AI responses and embedding vectors are cached in Redis.
The solution must meet the following requirements:
AI responses must expire exactly 24 hours after they are cached.
Cached embeddings must always reflect the current source data.
You need to configure Redis to meet the requirements.
NOTE: Each correct selection is worth one point.

Answer Area

Cache expiration and invalidation strategy

Requirement	Action
AI responses must expire exactly 24 hours after they are cached.	Configure allkeys-lru eviction. Reset expiration on every read. Set a Time to Live (TTL) on each key.
Cached embeddings must always reflect the current source data.	Configure volatile-lru eviction. Delete related keys when the document changes. Increase maximum memory allocation.

Answer: A