DUMPSARENA

Developing AI Cloud Solutions on Azure

Microsoft AI-200

Version Demo

Total Demo Questions: 6

Total Premium Questions: 65

Buy Premium PDF

https://dumpsarena.co

sales@dumpsarena.co

sales@dumpsarena.co
dumpsarena.co

Topic Break Down

Topic	No. of Questions
Topic 1, Design AI solutions	5
Topic 2, Develop AI solutions	29
Topic 3, Deploy and maintain AI solutions	31
Total	65

QUESTION NO: 1

You are developing an AI API hosted in Azure Container Apps (ACA). The API uses database credentials stored in Azure Key Vault, which is configured to use Azure RBAC for authorization.

The security team periodically rotates the database credentials. The API must retrieve the latest version of each credential at runtime without redeploying the container or exposing secrets in application code or configuration.

Which three actions should you perform to implement this secure secret-access strategy? Each correct answer is part of the solution.

A. Configure a Key Vault RBAC role assignment.

B. Configure a Key Vault access policy.

C. Export the secret during deployment.

D. Assign a system-assigned managed identity.

E. Retrieve the secret at runtime by using the SDK.

ANSWER: A D E

Explanation:

Assigning a system-assigned managed identity gives the Azure Container App an automatically managed Microsoft Entra identity. The application can use that identity to authenticate to Azure services without storing a client secret, certificate, database password, or other credential in its source code or deployment configuration. The identity exists for the lifetime of the container app and can be selected through Azure Identity credentials such as `DefaultAzureCredential`.

Because this Key Vault uses the Azure RBAC permission model, a Key Vault RBAC role assignment must be granted to that managed identity at the vault scope or an appropriate narrower scope. For read access to secrets, the built-in `Key Vault Secrets User` role provides the required data-plane permission to retrieve secret values.

The API should retrieve each secret at runtime by using the Azure Key Vault SDK. Calling `GetSecret` without specifying a version requests the current version of the named secret. Consequently, after rotation, subsequent runtime retrievals obtain the newly current credential without a container deployment. If the application caches credentials, its cache refresh interval must be designed to re-query Key Vault often enough to recognize rotations. See Managed identities in Azure Container Apps and Azure RBAC for Key Vault.

QUESTION NO: 2 - (DRAG DROP)

DRAG DROP -

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 -

Proseware Inc. develops AI-powered knowledge management solutions for enterprise customers. The company is modernizing its platform to support semantic search, intelligent document retrieval, and real-time partner integrations.

The engineering team uses Python and Azure SDKs. The architecture is being redesigned to support containerized microservices, vector search workloads, and serverless backend processing.

Planned Application Architecture

Microservices are containerized by using Docker.

Code for containerized microservices and Azure Function apps is developed locally but stored in a GitHub repository.

Custom images for containerized microservices are stored in Azure Container Registry (ACR).

Base images are stored in Docker Hub. Custom images must be rebuilt automatically whenever their base images are updated.

Azure Cosmos DB for NoSQL stores documents, metadata, and vector embeddings.

Azure Functions generate vector embeddings of Azure Cosmos DB for NoSQL-hosted documents and send messages to Service Bus to trigger search index updates.

Azure Container Apps (ACA) apps host backend API services that provide semantic search across Azure Cosmos DB for NoSQL documents. API services process Service Bus messages and update search indexes.

Azure Kubernetes Service (AKS) processes batch vector embedding regeneration for existing Azure Cosmos DB for NoSQL documents (whenever the embedding model is changed).

An extranet-facing containerized webhook allows business partners to submit documents to be processed by internal AI workflows for semantic search and retrieval.

Monitoring -

Telemetry generated by Azure resources is sent to Azure Monitor.

A Log Analytics workspace is used to collect ACA apps logs, AKS container logs, and Azure Functions apps logs. Monitoring of Azure Functions is currently implemented by using Azure Application Insights SDK instrumentation. Business Requirements -

Embeddings for new or updated Azure Cosmos DB for NoSQL-hosted documents must be automatically generated.

Backend API services must scale automatically during business hours. Cold start delay of backend APIs must be minimized.

Secrets must be stored outside of container images.

Developers must be able to correlate telemetry across Azure Functions hosts and apps. All tracing must be implemented by using OpenTelemetry SDK instrumentation.

Development efforts must be minimized. Technical Requirements -

Container images must be built automatically and validated before code updates are merged into the main branch. Image build automation must run inside the Azure Container Registry, eliminating dependency on local developer machines and external build services.

Dependency of image builds on local developer machines must be eliminated.

Event-driven scaling in ACA must occur based on the number of pending messages in the Azure Service Bus queue.

Azure Cosmos DB for NoSQL RU consumption must be minimized.

Vector similarity search must use embeddings stored in Azure Cosmos DB for NoSQL. The partner-facing containerized webhook service must run on Azure App Service.

Secrets must NOT be stored in container images, source control, or application configuration directly. They must be accessed securely at runtime.

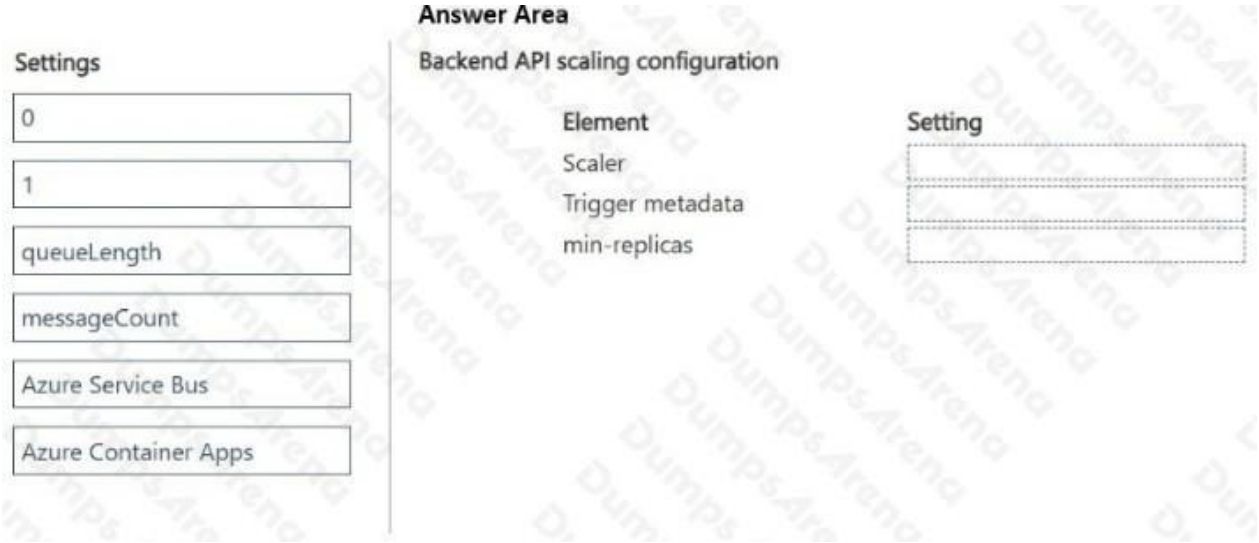
All secrets must be stored centrally in Azure Key Vault and accessed at runtime through a managed identity. Azure App Service must supply secrets at runtime without relying on external services.

Resources and workloads must be deployed by using Bicep templates through an automated, version-controlled pipeline. Local and command-line deployments must be eliminated to ensure repeatable, auditable deployments. Known Issues -

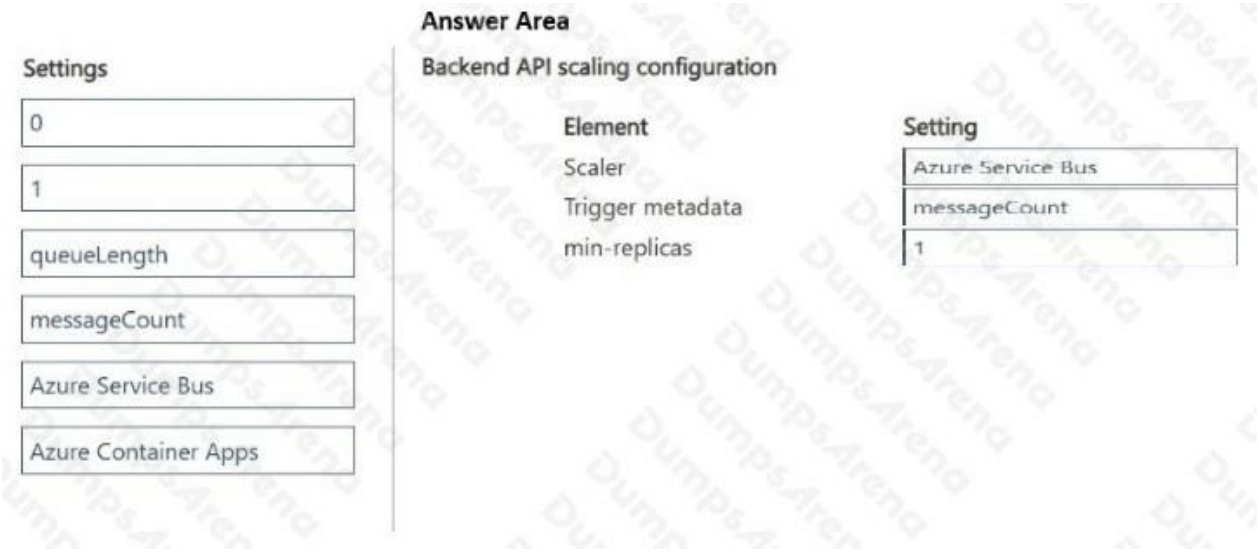
RU consumption spikes during vector similarity queries.

You need to configure event-driven scaling for the backend API services to meet the technical requirements. Which settings should you use for each element? To answer, move the appropriate settings to the correct elements. You may use each setting 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:



Explanation:

Azure Container Apps uses event-driven scale rules to increase or decrease the number of replicas according to an external event source. Here, the backend API services consume messages originating from an Azure Service Bus queue, and the technical requirement specifically states that scaling must occur according to the number of pending queue messages. The appropriate scaler is therefore **Azure Service Bus**.

The Azure Service Bus scale rule accepts **messageCount** as trigger metadata. This value defines the target number of active queue messages per replica. As pending messages accumulate, the platform uses that threshold to determine that additional backend API replicas are needed. This directly implements queue-based, event-driven scaling rather than relying on processor or memory utilization. Microsoft documents the Service Bus trigger metadata, including `messageCount`, in its Azure Container Apps scaling documentation.

The minimum replica count should be set to **1**. A minimum of one keeps an API container instance available, which reduces the cold-start delay that would occur if the service had fully scaled to zero. This aligns with the requirement to minimize cold starts for backend APIs during business hours while still allowing the service to scale out automatically when the Service Bus backlog grows. Azure Container Apps supports configuring minimum and maximum replica values alongside scale rules, as described in the scale applications in Azure Container Apps documentation.

Accordingly, configure the backend API scale rule with Azure Service Bus as the scaler, messageCount as its trigger metadata, and one as the minimum replica count.

QUESTION NO: 3 - (HOTSPOT)

HOTSPOT -

You are reviewing the Python tracing configuration for an application that must send distributed traces to Azure Monitor.

```
01 from opentelemetry import trace
02 from opentelemetry.sdk.trace import TracerProvider
03 from opentelemetry.sdk.trace.export import BatchSpanProcessor
04 from azure.monitor.opentelemetry.exporter import AzureMonitorTraceExporter
05
06 trace.set_tracer_provider(TracerProvider())
07 tracer = trace.get_tracer(__name__)
08
09 exporter = AzureMonitorTraceExporter(connection_string="<conn>")
10 span_processor = BatchSpanProcessor(exporter)
11 trace.get_tracer_provider().add_span_processor(span_processor)
```

For each of the following statements, select Yes if the statement is true. Otherwise, select No. NOTE: Each correct selection is worth one point.

Answer Area

Configuring the OpenTelemetry exporter and span processor in Python

Statement	Yes	No
The code configures the tracer provider before any spans are created.	○	○
The configuration exports traces synchronously.	○	○
The configuration enables export to Azure Monitor.	○	○

ANSWER:

Answer Area

Configuring the OpenTelemetry exporter and span processor in Python

Statement	Yes	No
The code configures the tracer provider before any spans are created.	●	○
The configuration exports traces synchronously.	○	●
The configuration enables export to Azure Monitor.	●	○

Explanation:

The correct selections are **Yes** for configuring the tracer provider before spans are created, **No** for synchronous trace export, and **Yes** for enabling Azure Monitor export. The tracing pipeline is set up in the appropriate order:

`trace.set_tracer_provider(TracerProvider())` installs the application's provider first, and `trace.get_tracer(__name__)` retrieves a tracer after that provider has been registered. The shown code contains no call that starts or creates a span before this setup. This ensures that subsequent spans created by the retrieved tracer are associated with the configured provider.

The configured span processor is `BatchSpanProcessor`. This processor collects ended spans into a queue and sends them in batches using a background worker, rather than making the application thread wait for every individual trace export operation to complete. This behavior is useful for application performance because telemetry transmission can be performed efficiently outside the immediate span-completion path. OpenTelemetry documents this batching behavior in its Python trace export documentation.

The destination is enabled by constructing `AzureMonitorTraceExporter` with the Azure Monitor connection string and passing that exporter to the batch processor. Registering the processor on the active tracer provider completes the route from newly created OpenTelemetry spans to Azure Monitor. Azure Monitor's OpenTelemetry guidance describes using the Azure Monitor exporter and a connection string to send telemetry to the Azure Monitor backend; see Enable Azure Monitor OpenTelemetry for Python applications. Thus, spans produced after this configuration can be batched and exported to the Azure Monitor resource identified by the supplied connection string.

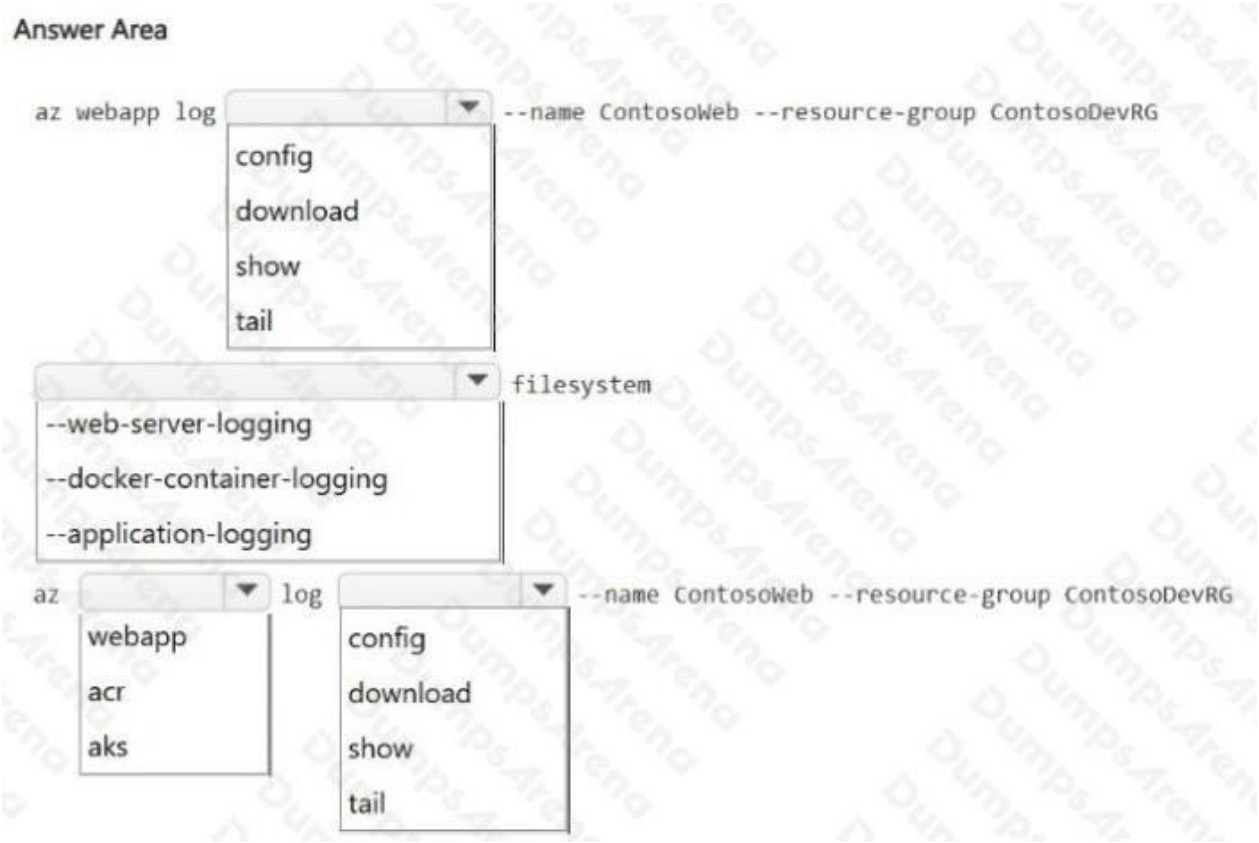
QUESTION NO: 4 - (HOTSPOT)

HOTSPOT -

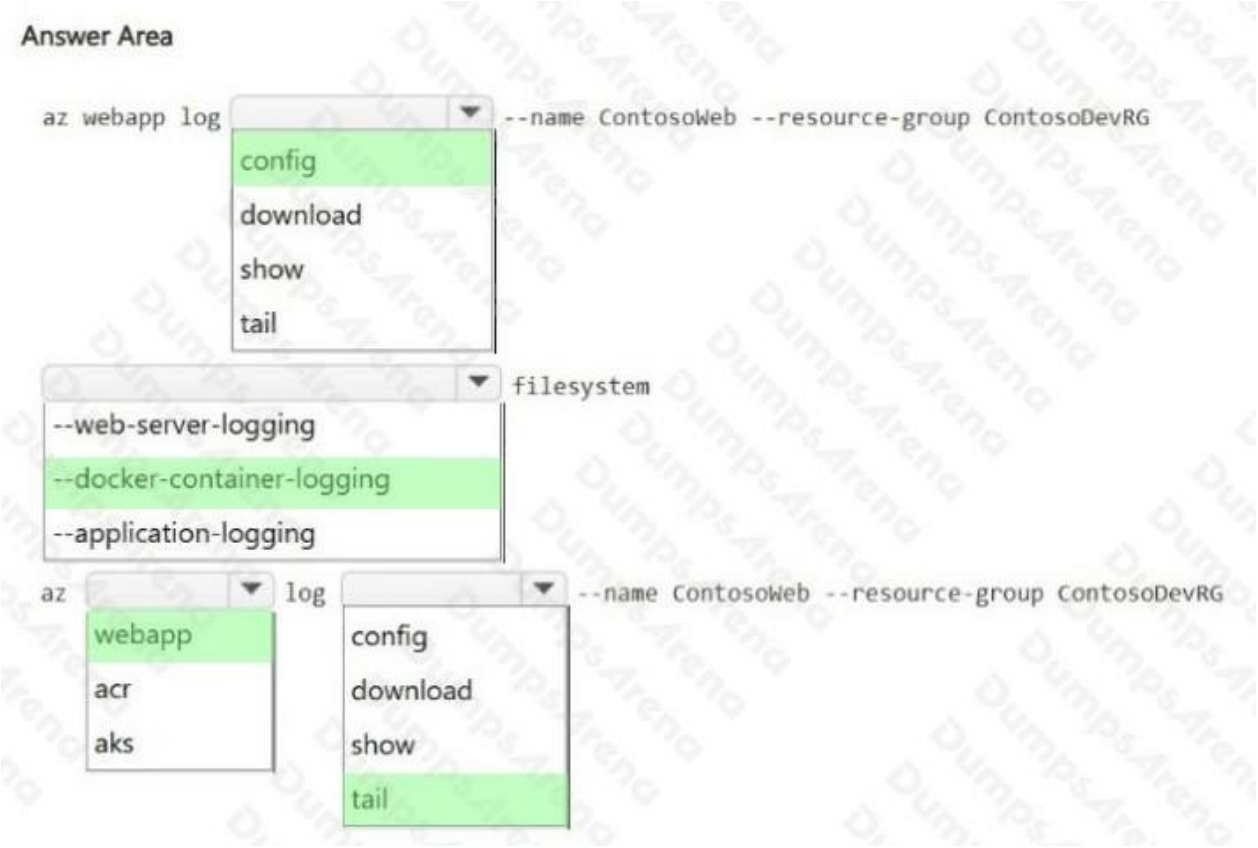
You plan to deploy a web app to App Service on Linux. You create an App Service plan. You create and push a custom Docker image that contains the web app to Azure Container Registry.

You need to access the console logs generated from inside the container in real-time.

How should you complete the Azure CLI command? To answer, select the appropriate options in the answer area. NOTE: Each correct selection is worth one point.



ANSWER:



Explanation:

For a custom Docker container running in Azure App Service on Linux, the container's console output must be enabled as App Service container logging before it can be streamed. The appropriate configuration is `az webapp log config` together with `--docker-container-logging filesystem`. This directs the Docker container logs to the App Service file system, making the standard output and standard error written by the process inside the container available through App Service logging.

Once Docker container logging is configured, use `az webapp log tail` for the named app and resource group. The `tail` command establishes a live stream of the log output in the Azure CLI session, which satisfies the requirement to access console logs in real time. The completed commands are:

```
az webapp log config --name ContosoWeb --resource-group ContosoDevRG --docker-container-logging filesystem

az webapp log tail --name ContosoWeb --resource-group ContosoDevRG
```

Microsoft documents that enabling Docker container logging to the file system supports viewing container logs, and the App Service log-tail command streams the active logs. See Access diagnostic logs for custom containers and the Azure CLI reference for az webapp log tail.

QUESTION NO: 5

Proseware must deploy a batch workload that regenerates vector embeddings for existing Azure Cosmos DB for NoSQL documents when the embedding model changes. According to the planned architecture, this workload runs on Azure Kubernetes Service (AKS). What should be used to define and deploy the batch embedding workload?

A. YAML-formatted files

B. kubectl run and create commands

C. XML-formatted files

D. az aks commands

ANSWER: A

Explanation:

YAML-formatted files are the appropriate way to define the batch embedding workload for AKS. AKS is a managed Kubernetes service, and Kubernetes workloads are declaratively described through manifest files, which are commonly written in YAML. A manifest specifies the desired state of Kubernetes resources, including the container image, command arguments, environment configuration, resource requests and limits, restart behavior, and identity-related settings. For this scenario, the batch process can be modeled as a Kubernetes Job so that each embedding-regeneration run executes to completion; a CronJob can be used if regeneration must occur on a schedule. The YAML definition can be stored alongside the application source in GitHub, reviewed and validated before merging, and applied by the automated deployment pipeline. This provides a repeatable, version-controlled specification for running the custom image from Azure Container Registry on AKS. Kubernetes Jobs are specifically intended for finite batch work and track completion of one or more pods, making them well suited to regenerating embeddings for an existing document set after a model change. See Kubernetes Jobs documentation and Deploy applications in AKS.

QUESTION NO: 6 - (DRAG DROP)

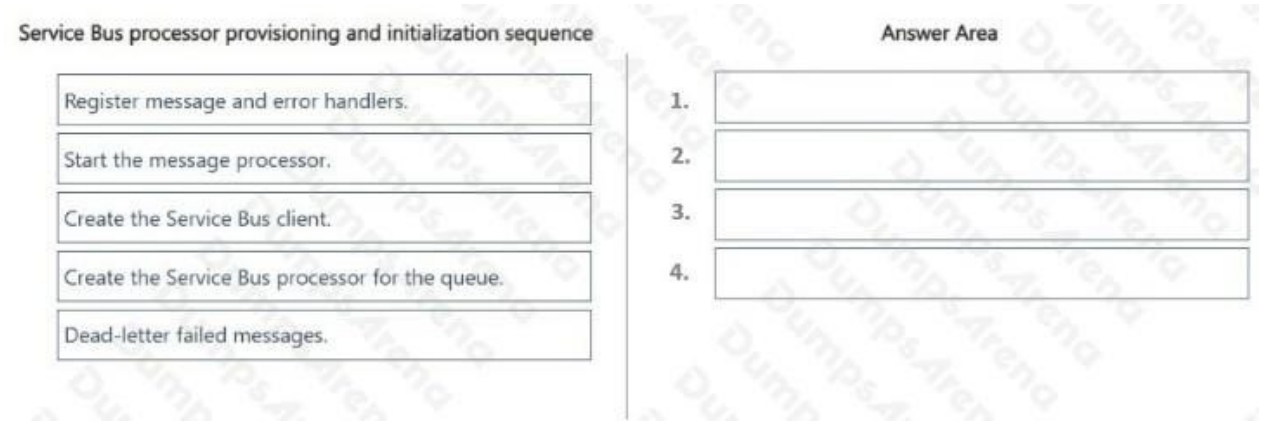
DRAG DROP -

You are provisioning and configuring a Service Bus processor for AI batch jobs.

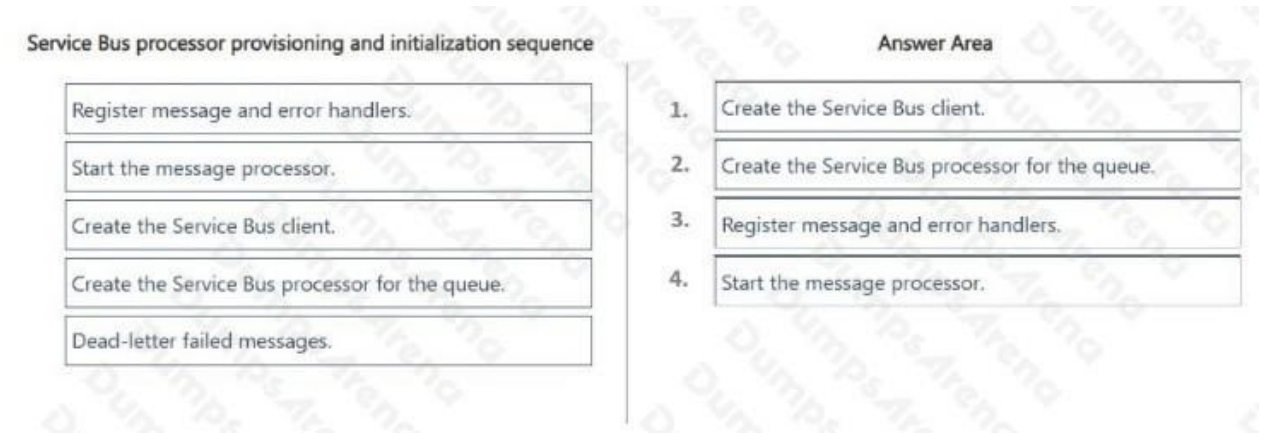
The processor must connect to an existing queue, register handlers for message and error processing, and then begin receiving messages.

You need to provision and configure the Service Bus processor for message and error handling.

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.



ANSWER:



Explanation:

A Service Bus processor is initialized in a deliberate order so that its connection, receive scope, and callback behavior are all ready before message delivery starts. First, create the Service Bus client. In the Azure.Messaging.ServiceBus SDK, `ServiceBusClient` is the top-level client that establishes access to the Service Bus namespace and is used to create senders, receivers, and processors.

Next, create the processor for the queue. Calling `CreateProcessor` with the existing queue name produces a `ServiceBusProcessor` configured to receive from that specific queue. At this point the processor exists, but it does not yet actively receive messages.

Then register the message and error handlers. The processor uses the `ProcessMessageAsync` callback for each received message and `ProcessErrorAsync` for exceptions arising during processing or receive operations. Registering these handlers before startup ensures that every message and processor error is routed to the application's intended logic as soon as processing begins.

Finally, start the message processor by calling `StartProcessingAsync`. This starts the background receive loop and allows the processor to invoke the registered callbacks for messages arriving in the queue. Microsoft documents this same lifecycle: create a client, create and configure a processor, assign the processing handlers, and start processing. See Use Azure Service Bus queues with .NET and the ServiceBusProcessor.StartProcessingAsync API reference.