MICROSOFT AZURE

AI-200 Study Guide

Developing AI Cloud Solutions on Azure

A concise revision guide for Azure AI cloud developers

Prepared by Jitendra Singh Malik

September 2026

Independent study resource

These notes were created while preparing for the AI-200 exam. They are not official Microsoft exam material. Features, previews, licensing and user-interface details may change, so validate current behaviour using Microsoft Learn and product documentation.

How to use this guide

This guide condenses AI-200's four skills-measured domains into a structured reference, grounded in Microsoft's official study guide and service documentation.

- Read chapters 1-4 in order to build up from container hosting through data, messaging, and finally security/observability.
- Use the code-identifier callouts (in monospace) to memorize exact CLI commands, YAML fields, and function names.
- Review the "Common confusion" callouts before practice exams - they target the exact distinctions exam scenarios test.
- Use the final quick-reference chapter as a last-day revision sheet.

Contents

Source note

This guide is based on FabricPrep's AI-200 deep-dive study guide pages, researched directly from Microsoft Learn's official AI-200 study guide and underlying Azure service documentation.

CHAPTER 1

Develop Containerized Solutions on Azure

Build and store images with Container Registry, host containers on App Service, and deploy and scale them with Container Apps and AKS.

Azure Container Registry: building, storing, and versioning images

Azure Container Registry (ACR) is the private registry almost every containerized AI workload on Azure pulls from — it stores the images your Container Apps, AKS, and App Service deployments run, alongside Helm charts and other OCI artifacts.

Repositories and tags

- An image is identified by `<registry>.azurecr.io/<repository>:<tag>`, for example `myregistry.azurecr.io/inference-api:1.4.0`.
- Tags aren't guaranteed unique over time unless you enforce it — enabling the `Immutable` tag setting on a repository prevents an existing tag from being overwritten, which matters once a tag is referenced from a production deployment.
- ACR supports geo-replication, letting a single registry serve multiple regions with local pull latency and one set of image names to manage.

ACR Tasks: building images without a local Docker daemon

- A *quick task* (`az acr build`) uploads your source and Dockerfile, builds the image in Azure, and pushes it to the registry on success — no local Docker installation required.
- A scheduled or triggered task automates that same build: on a Git commit, on a base image update (so a patched base image ripples into every dependent application image automatically), or on a timer.
- A *multi-step task* is defined in YAML and can chain several build/run/test/push operations — for example: build an image, run it, run a separate test container against it, and only push if the tests pass. Each step runs inside its own container, giving you composable, dependency-aware build pipelines entirely offloaded to Azure compute.

> **Common confusion**
> ACR Tasks is a build and automation service; it doesn't run your production workload continuously. Once an image lands in ACR, orchestration and hosting are separate concerns handled by App Service, Container Apps, or AKS.

Deploying containers to Azure App Service

App Service can run a single container or a small multi-container app directly from a registry, which is a lighter-weight option than Container Apps or AKS when you don't need event-driven scaling or Kubernetes-native features.

Container-specific configuration

- The App Service plan still determines compute size and scaling limits, exactly as it does for code-based deployments — containers don't bypass the underlying plan.
- `WEBSITES_PORT` (or the port your container exposes via `EXPOSE`) tells App Service which port to route traffic to inside the container.

- Continuous deployment can be wired to ACR so that pushing a new tag (or updating the `latest` tag, when webhooks are enabled) triggers an automatic restart with the new image.

Environment variables and secrets

- App settings are injected into the container as environment variables at startup — the same mechanism used for non-containerized App Service apps.
- Secrets shouldn't be pasted directly into app settings in plaintext for anything sensitive; a Key Vault reference (`@Microsoft.KeyVault(...)`) in an app setting lets App Service resolve the actual secret value from Key Vault at runtime using its managed identity, without the secret ever being stored in the App Service configuration itself.
- Deployment slots work the same way for containers as for code: a staging slot can run the new image and be validated before a slot swap promotes it to production with no downtime.

> **Common confusion**
> Changing an app setting on a container-based App Service app restarts the container (since environment variables are read at process startup) — there's no way to hot-reload a changed setting into a running container process.

Azure Container Apps: environments, revisions, and event-driven scaling

Container Apps is the serverless, Kubernetes-based hosting option built for microservices and event-driven AI workloads — it gives you scaling behavior similar to AKS without requiring you to manage the cluster.

Environments and revisions

- A Container Apps *environment* is the security and networking boundary around one or more container apps — apps in the same environment share a virtual network and can communicate directly.
- Every update to a container app's template (image, scale rules, or configuration such as CPU/memory) creates a new *revision*, an immutable snapshot. Revision mode can be single (only the latest revision serves traffic) or multiple (several revisions serve traffic simultaneously, useful for gradual rollout or A/B testing with traffic-splitting weights).
- Not every change creates a revision — updating a secret's value, for example, applies to existing revisions in place, while changing the container image always creates a new one.

KEDA-based event-driven scaling

- Container Apps scaling is powered by KEDA (Kubernetes Event-Driven Autoscaling) under the hood. Scale rules fall into three categories: HTTP (concurrent request count), TCP (concurrent connections), and custom — CPU/memory or an event source such as Azure Service Bus, Event Hubs, Kafka, or Redis.
- A custom scale rule maps directly to a KEDA *scaler* specification (for example, `type: azure-servicebus` with a `queueName` and `messageCount` threshold) — this is exactly the mechanism you'd use to scale a queue-consuming AI inference worker out as messages pile up, and back to zero when the queue is empty.
- Multiple scale rules can be defined on one app; the app scales as soon as *any* rule's condition is met. Scaling to zero (min replicas = 0) means the app costs nothing while idle, at the expense of a cold start on the next request.

> **Common confusion**

KEDA scale rules control how many replicas run; they don't change compute size per replica (CPU/memory), which is set separately in the container app's resource configuration.

Deploying and managing applications on AKS with manifest files

AKS gives you a fully managed Kubernetes control plane — you manage the worker nodes and workloads, Azure manages the API server, etcd, and control-plane upgrades. Compared to Container Apps, AKS trades some operational simplicity for full Kubernetes API access and ecosystem compatibility.

Manifest-based deployment

- Workloads are described declaratively in YAML manifests applied with `kubectl apply -f`. A typical AI service needs at least a `Deployment` (desired pod count, container image, resource requests/limits) and a `Service` (stable network endpoint routing traffic to the pods, whether ClusterIP, NodePort, or LoadBalancer).
- Deployments manage rolling updates automatically: updating the image reference in a Deployment manifest and re-applying it replaces pods gradually according to the rollout strategy, rather than taking the whole app down at once.
- Namespaces let you partition a single cluster (for example, separating a dev environment from production, or isolating tenants) with independent RBAC and resource quotas.
- ACR integrates with AKS through `az aks update --attach-acr`, which grants the cluster's managed identity pull access to the registry without embedding registry credentials in a Kubernetes secret.

Common confusion

A Deployment ensures pods exist and restarts them on failure, but a Service is what makes them reachable at a stable address — deleting a Deployment's Service doesn't stop the pods, and creating pods without a Service leaves them unreachable from outside the cluster.

Monitoring and troubleshooting AKS and Container Apps

Diagnosing a failing containerized deployment means correlating three signal types: logs (what the app said), events (what the platform did), and connectivity (whether traffic can actually reach the workload).

Container Apps

- The Log Stream and the `az containerapp logs show` command tail console output (stdout/stderr) from a running revision in near real time — the first stop for a crashing or misbehaving container.
- System logs (revision provisioning, scaling events, restarts) are separate from application console logs and are what tells you *why* a revision failed to become active, as opposed to what the app printed.
- Provisioning and health probe failures (a container failing its readiness or liveness probe) show up as revision-level status, distinct from an app that started fine but is returning errors to callers.

AKS

- `kubectl get events --sort-by=.lastTimestamp` surfaces scheduling failures, image pull errors, and probe failures at the cluster level, often revealing the root cause before you ever look at application logs.
- `kubectl logs <pod> [-c <container>] [--previous]` retrieves a container's logs, with `--previous` critical for a pod that already crashed and restarted (`CrashLoopBackOff`) — the current container's logs may be empty while the previous attempt's logs hold the actual error.

- `kubectl describe pod <pod>` shows recent events for a specific pod, including why a container was restarted or why scheduling failed (for example, insufficient CPU/memory on any node).
- End-to-end connectivity issues (a pod that starts but can't reach a dependency) are diagnosed by checking Service selectors match pod labels, network policies, and DNS resolution inside the cluster (`kubectl exec` into a pod to test with `curl` or `nslookup`).

Common confusion

A pod stuck in `Pending` is a scheduling problem (no node has room, or a required resource isn't available); a pod stuck in `CrashLoopBackOff` is a runtime problem (the container starts and then exits) — the two point troubleshooting in very different directions.

CHAPTER 2

Develop AI Solutions Using Azure Data Management Services

Build the data layer for AI apps with Cosmos DB, Azure Database for PostgreSQL, and Azure Managed Redis, including vector storage and retrieval.

Azure Cosmos DB for NoSQL: SDK access, RU cost, and indexing

Cosmos DB for NoSQL is a common backing store for AI applications because it combines flexible, schema-free JSON documents with low, predictable latency at any scale — and, increasingly, native vector search alongside the rest of an item's data.

Connecting and querying

- The SDK (Python, .NET, Java, JS) connects via a `CosmosClient` constructed from an endpoint and key, or — the recommended production pattern — Microsoft Entra ID with a managed identity, avoiding a long-lived key in configuration.
- Every operation is scoped to a container within a database; a container's *partition key* determines how items are distributed across physical partitions and is the single most consequential design decision for both cost and query performance.
- Queries that don't filter on the partition key become *cross-partition* queries — still possible, but more expensive in Request Units (RUs) than a query scoped to one logical partition.

Request Units and consistency levels

- Every read and write consumes RUs, a currency abstracting CPU, memory, and IO cost. A point read (by id and partition key) is the cheapest possible operation; a large cross-partition query with `ORDER BY` costs substantially more.
- Consistency level is a deliberate trade-off along a spectrum: *Strong* (always read the latest committed write, highest latency/cost) → *Bounded staleness* → *Session* (the default — a client always sees its own writes) → *Consistent prefix* → *Eventual* (lowest latency/cost, no ordering guarantee). Most AI applications default to Session consistency and only tighten it where correctness genuinely requires it.
- Indexing policy controls which paths are indexed; by default every property is indexed automatically, which is convenient but not free — excluding paths you never query on reduces both storage and write RU cost.

> **Common confusion**
> RU consumption is billed the same whether a query returns zero results or many — a query's *cost* is driven by how much data it has to scan and process, not by how much it returns, which is why a well-chosen partition key and indexing policy matter even for queries with small result sets.

Vector search and change feed in Cosmos DB for NoSQL

Cosmos DB's native vector search lets you store an item's embedding alongside its source data and metadata in the same document, avoiding the operational overhead of syncing a separate vector store.

Vector storage and search

- A container's vector policy declares which path holds the embedding, its dimensionality, and the distance function (cosine, dot product, or Euclidean). A vector *indexing* policy then determines how that path is indexed for search — separate from, but coordinated with, the vector policy.
- Three vector index types trade off accuracy against cost: `flat` is a brute-force, 100%-accurate scan capped at 505 dimensions; `quantizedFlat` compresses vectors for lower RU cost and latency at a small accuracy loss, still brute-force; `diskANN` builds an approximate-nearest-neighbor graph (Microsoft Research's DiskANN algorithm) that scales to millions of vectors with the lowest latency and RU cost, at the cost of being approximate rather than exact.
- A search is expressed with the `VectorDistance` system function in a normal NoSQL query, for example: `SELECT TOP 10 c.title, VectorDistance(c.contentVector, @queryVector) AS score FROM c ORDER BY VectorDistance(c.contentVector, @queryVector)`. Always including a `TOP N` clause matters — without it, the engine tries to rank and return far more candidates than any application needs, at unnecessary RU cost.
- Because `diskANN` and `quantizedFlat` are approximate, the exact same query can return slightly different ordering across runs once your recall-vs-cost tuning favors speed over exhaustiveness — that's expected behavior, not a bug to chase.

Change feed processor

- The change feed is a persistent, ordered log of every insert and update to a container (deletes aren't included by default). It's the mechanism for reacting to new or changed items — for example, generating an embedding the moment a new document lands and writing it back to the same item.
- The change feed *processor* (available in the .NET and Java SDKs; Python and Node.js use the lower-level pull model instead) distributes leases — one per logical partition range — across however many compute instances are running, and checkpoints progress per lease so processing can resume without reprocessing everything after a restart.
- Processing guarantees are *at-least-once*: if your handler throws partway through a batch, the same batch is retried from the last checkpoint, so handler logic needs to be idempotent (safe to run twice on the same data).

> **Common confusion**
> The change feed only reflects changes going forward from when a processor's lease container was first initialized (or a specific configured start time) — it isn't a way to replay history from before that unless you explicitly enable an all-versions-and-deletes read mode or start from the beginning of the container's lifetime.

Azure Database for PostgreSQL: schema, indexing, and connection efficiency

Azure Database for PostgreSQL (Flexible Server) gives AI applications a relational option — useful when data already has a natural tabular shape, or when the app needs both structured querying and vector search over the same rows via the `pgvector` extension.

Schema and data types

- Choosing precise column types (`numeric` vs. `float`, `text` vs. `varchar(n)`, native `jsonb` for semi-structured fields) affects both storage footprint and query performance — `jsonb` in particular lets you keep flexible metadata columns alongside strict relational columns without a full NoSQL migration.
- Standard indexing (B-tree for equality/range lookups, GIN for `jsonb` or full-text) applies exactly as in any PostgreSQL deployment; vector similarity search needs its own, separate index type (covered next).

Connection optimization

- Each PostgreSQL connection is a real backend process, materially more expensive to open than a typical NoSQL connection — an AI application issuing many short-lived requests (common in serverless or Functions-triggered workloads) can exhaust available connections quickly without pooling.
- PgBouncer (available as a built-in connection-pooling option on Flexible Server) sits in front of the database and multiplexes many client connections onto a smaller pool of actual backend connections, which is usually the single highest-leverage fix for throughput and latency problems caused by connection churn.
- SDKs should also be configured with sensible client-side pool sizes and timeouts rather than opening a fresh connection per request.

> **Common confusion**
> Adding more compute (vCores/memory) to a PostgreSQL server doesn't fix a connection-exhaustion problem by itself — the number of usable connections is bounded by the `max_connections` setting and by memory per connection, so pooling is often a bigger lever than scaling up.

pgvector: indexing strategies and RAG patterns

The `pgvector` extension adds a native `vector` column type and similarity operators to PostgreSQL, letting embeddings live next to the row they describe instead of in a separate system.

Distance operators and access methods

- `<=>` computes cosine distance, `<->` computes L2 (Euclidean) distance, and `<#>` computes negative inner product — the query must use the operator matching the index's configured operator class (for example `vector_cosine_ops` for `<=>`) or the planner won't use the index at all.
- Without any index, `pgvector` performs an exact, brute-force nearest-neighbor scan — perfectly accurate, but linear in the number of rows.

Choosing an index type

- **IVFFlat** partitions vectors into *lists* built during a one-time training pass over existing data; a query only searches the closest `probes` lists. Fast to build and memory-light, but needs data present before indexing for good list boundaries, and has a lower speed/recall ceiling than graph-based indexes.
- **HNSW** builds a multi-layer navigable-graph structure. It has no training step (so it can be built on an empty table, unlike IVFFlat) and generally gives a better speed-vs-recall trade-off, at the cost of longer build time and higher memory use. Its two build-time knobs are `m` (connections per layer) and `ef_construction` (candidate list size during build); `ef_search` tunes the accuracy/speed trade-off per query or connection.
- **DiskANN** is designed to stay fast when the index doesn't fit in memory, since it's optimized for data resident on SSD — the right choice as embedding volume grows past what HNSW can comfortably hold in RAM.
- `pgvector` indexes are capped at 2,000 dimensions; higher-dimensional embeddings must be reduced in dimensionality or left unindexed (falling back to a brute-force scan, or partitioning/sharding for scale).

RAG with metadata filtering

- A retrieval-augmented generation pattern typically combines a vector similarity ORDER BY with a normal `WHERE` clause on metadata columns (tenant id, document category, date range) — letting you scope semantic search to only the rows a given user or use case should ever see, in the same query as the similarity ranking.

- Sizing compute, memory, and storage for a vector workload isn't just about row count: index build time and query latency both scale with vector dimensionality and the chosen index's memory footprint, so a workload with large embeddings needs headroom beyond what row count alone would suggest.

> **Common confusion**
> IVFFlat's `lists` parameter is chosen relative to the number of rows *at index build time* — adding a large amount of new data afterward without rebuilding the index degrades recall, since the list boundaries no longer reflect the data's actual distribution. HNSW doesn't have this problem since it has no training phase.

Azure Managed Redis: caching and vector indexing

Azure Managed Redis gives AI applications a very low-latency layer for two related but distinct jobs: general-purpose caching, and — via the RediSearch module — vector similarity search close to the application.

Caching, expiration, and invalidation

- Standard Redis data operations (`SET`/`GET`, hashes, sorted sets) work as an application-level cache in front of a slower backing store — for example, caching a Cosmos DB or PostgreSQL read, or caching an LLM response for a repeated prompt (semantic caching).
- A time-to-live (`EXPIRE`, or `SET ... EX`) on a key evicts it automatically, which is the simplest invalidation strategy for data that's naturally time-bounded (a session token, a rate-limit counter).
- For data that changes on a schedule you control rather than a fixed TTL, explicit invalidation (deleting or overwriting the key when the source data changes) keeps the cache from serving stale results indefinitely.

Vector search with RediSearch

- Vector search requires the RediSearch module, which must be enabled *when the Azure Managed Redis instance is created* — modules can't be added to an existing instance afterward, so this is a provisioning-time decision, not something you can bolt on later.
- Not every tier supports it: Memory Optimized, Balanced, and Compute Optimized tiers support RediSearch; Flash Optimized does not.
- Vectors are stored in hash or JSON structures alongside metadata (document id, source, tenant), and indexed with either a `FLAT` (exact, brute-force) or `HNSW` (approximate, graph-based) index, searched via K-nearest-neighbor or range queries, and combinable with metadata filters for hybrid search — the same core pattern as Cosmos DB and pgvector, just backed by Redis's in-memory performance profile.
- Because vector search shares the instance with caching, session storage, and rate limiting, Azure Managed Redis is often chosen less for raw vector scale and more for keeping a very-low-latency retrieval step (like a semantic cache or a real-time recommendation lookup) physically close to other hot application data.

> **Common confusion**
> Redis vector search trades some capability for speed — features like Cosmos DB's DiskANN-scale approximate search over huge datasets aren't the point of Redis; it's optimized for low-latency lookups over data that comfortably fits the tier's memory profile, not for being the system of record for a large embedding corpus.

CHAPTER 3

Connect to and Consume Azure Services

Wire AI back-end services together with Service Bus and Event Grid for messaging, and Azure Functions for serverless APIs.

Azure Service Bus: queues, topics, and dead-letter handling

Service Bus is the durable, ordered message broker of choice for back-end operations that must not be dropped — a classic fit for queuing an AI inference request, a document-processing job, or any workload where "eventually, exactly once effectively" beats "immediately, best effort."

Queues vs. topics

- A *queue* is point-to-point: each message is received and processed by exactly one consumer, ideal for distributing work across a pool of identical workers.
- A *topic* is publish/subscribe: a message sent to the topic is delivered to every *subscription* on it, each of which behaves like its own independent queue. This is the right shape when the same event (say, "document uploaded") needs to trigger several independent downstream processes (indexing, embedding generation, notification) without the publisher knowing about any of them.
- A subscription can have a SQL or correlation *filter*, so only messages matching a condition are delivered to it — letting one topic serve several different consumer types without every consumer having to filter messages itself.

Dead-letter queues

- Every queue and subscription has an associated dead-letter sub-queue (`$deadletterqueue`) where messages land automatically after exceeding `MaxDeliveryCount` failed delivery attempts, expiring (TTL), or being explicitly dead-lettered by application code.
- Dead-lettering exists specifically so a single poison message (one that reliably crashes its consumer) doesn't block the queue forever by being redelivered in an infinite retry loop — it's set aside for separate inspection while healthy messages keep flowing.
- Consumers should use `PeekLock` receive mode (lock a message, process it, then explicitly complete or abandon it) rather than `ReceiveAndDelete` for anything where losing a message on a crash mid-processing is unacceptable — `ReceiveAndDelete` removes the message the instant it's received, before you know processing succeeded.

> **Common confusion**
> A message that keeps failing and is redelivered isn't lost — it's still in the queue being retried — until it's either dead-lettered after too many attempts or its TTL expires. Monitoring dead-letter queue depth is a standard health signal precisely because a growing DLQ means something downstream is systematically broken, not just occasionally slow.

Azure Event Grid: event-driven workflows

Event Grid is built for a different shape of problem than Service Bus: reacting to discrete events from Azure services or your own application, at very high scale and low latency, rather than reliably queuing units of work for guaranteed processing.

Core model

- An event source publishes events to a *topic* (a system topic for built-in Azure resource events, like a new blob landing in storage, or a custom topic for application-defined events).
- An *event subscription* routes matching events from a topic to a handler — a Function, a webhook, Service Bus, Event Hubs, and more.
- A *filter* on a subscription narrows delivery by event type or subject prefix/suffix, so a single topic covering "everything happening in this storage account" can still let a subscriber receive only, say, blob-created events under a specific folder prefix.

Custom events and reliability

- Applications publish custom events in the CloudEvents or Event Grid schema, which is how an AI pipeline stage signals "I finished my part" to whichever downstream stages care, without those stages being hardwired to know about each other.
- Event Grid retries failed deliveries with exponential backoff up to a configurable retry policy, and can be configured with a dead-letter destination (typically a storage container) for events that exhaust their retries — conceptually similar to Service Bus's DLQ, but for events rather than queued work items.

> **Common confusion**
> Event Grid notifies you that something happened; it doesn't guarantee a consumer will process it exactly once, and it isn't a work queue you pull batches from — Service Bus (or Event Grid delivering *into* Service Bus) is the better fit when you need ordered, exactly-once-processed units of work rather than fan-out notifications.

Azure Functions: serverless APIs, triggers, and bindings

Functions is the serverless compute layer that ties the rest of an AI back end together — often the glue code that runs when a queue message arrives, an event fires, or an HTTP request comes in, without you provisioning or managing a server.

Triggers and bindings

- A *trigger* is what causes a function to run — exactly one per function: HTTP request, a new Service Bus message, a new Event Grid event, a Cosmos DB change feed entry, a timer, and others.
- *Bindings* are declarative input/output connections to other services, configured (not hand-coded) so a function can, for example, read a Cosmos DB document and write a Service Bus message without writing SDK connection boilerplate for either — the runtime supplies the connected object or return-value wiring based on configuration.
- Building a serverless API means using the HTTP trigger, with the route, allowed methods, and auth level defined per function; each function effectively becomes one API endpoint without you standing up a web server.

Configuring and deploying function apps

- A function app is the deployment and scaling unit — one or more functions packaged and hosted together, sharing a runtime version, app settings, and (on Consumption or Premium plans) the same automatic, event-driven scaling behavior.

- The hosting plan matters for AI workloads specifically: Consumption scales to zero and is billed per execution but has cold starts and execution time limits; Premium keeps instances warm and supports VNet integration; Functions can also run on Container Apps for full control over the container image, GPU-backed hosting, and scaling that shares KEDA rules with the rest of your Container Apps environment.
- App settings (including Key Vault references, exactly as with App Service) configure connection strings and secrets for a function app's triggers and bindings without embedding them in code.

Common confusion

A trigger and an input binding sound similar but are different: a trigger starts the function's execution and a function has exactly one; input/output bindings are optional, can be multiple, and only supply or receive data during a run that's already been triggered by something else.

CHAPTER 4

Secure, Monitor, and Troubleshoot Azure Solutions

Protect secrets and configuration with Key Vault and App Configuration, then trace and diagnose production issues with OpenTelemetry and KQL.

Securing secrets with Azure Key Vault

Key Vault is the standard place to keep anything an AI application must never hard-code: API keys for a model endpoint, database connection strings, certificates, and encryption keys.

Storing and retrieving secrets

- Secrets, keys, and certificates are distinct object types in Key Vault with different intended uses — a *secret* is an arbitrary string (a connection string, an API key), a *key* is used for cryptographic operations without ever leaving the vault in plaintext, and a *certificate* combines a key and its public certificate for TLS scenarios.
- Applications should authenticate to Key Vault with a managed identity rather than a client secret — the whole point of Key Vault is removing hard-coded credentials, and authenticating to it *with* a hard-coded credential undermines that.
- Once authenticated, the SDK (`SecretClient` and equivalents) retrieves a secret's current value by name; App Service, Functions, and Container Apps can also reference a Key Vault secret directly from an app setting so application code never has to call the Key Vault SDK explicitly.

Rotation

- A secret in Key Vault can have an expiration date and, for supported secret types, an automatic rotation policy that generates a new version on a schedule before the old one expires.
- Rotation only actually protects you if consumers pick up the new version — an app that cached a secret's value at startup and never re-reads it won't benefit from rotation until it restarts, which is why some architectures poll for the latest version or subscribe to Key Vault's change notifications instead of reading once and holding on indefinitely.

> **Common confusion**
> Enabling soft-delete and purge protection on a vault protects against accidental permanent deletion of secrets — it's a data-protection setting, unrelated to rotation, which is about periodically replacing a secret's *value*, not protecting the vault's contents from deletion.

Azure App Configuration: centralizing application settings

App Configuration is a dedicated service for the non-secret configuration values an application needs — feature flags, connection endpoints, retry policies, service URLs — separate from Key Vault, which is reserved for values that are actually sensitive.

Storing and retrieving configuration

- Configuration is stored as key-value pairs, optionally organized with a *label* (for example, separating `Development` and `Production` values for the same key) so one App Configuration store can serve multiple environments.
- Applications read configuration through the App Configuration provider/SDK, typically at startup, and can be configured to poll for changes on an interval — letting a running application pick up a configuration change (like a feature flag flip) without a redeploy or restart.
- A key can hold a *Key Vault reference* rather than a literal value — the application resolves it through App Configuration, but the actual secret is still fetched from and protected by Key Vault, giving you one place to browse all configuration while secrets stay properly isolated.

Why not just use app settings everywhere

- App Configuration adds a management layer app settings alone don't have: point-in-time snapshots, configuration change history, feature flag management with targeting rules, and a single store shared across multiple services or environments instead of duplicating settings per App Service/Function app/Container App.

> **Common confusion**
> App Configuration and Key Vault are complementary, not competing — the rule of thumb is App Configuration for values that are fine to see in a config blade, and Key Vault (referenced from App Configuration or directly) for anything that would be a security incident if it leaked.

Distributed tracing with OpenTelemetry

A single user-facing AI request often fans out across several services — a Function, a Container App, a Cosmos DB query, a call to a model endpoint — and distributed tracing is what lets you reconstruct that whole path as one coherent timeline instead of a pile of disconnected logs.

How a trace is structured

- A *trace* represents one end-to-end operation; it's made up of *spans*, each representing one unit of work (an HTTP request, a database call, a function execution) with a start time, duration, and its own attributes.
- Spans are linked in a parent-child hierarchy — a span for an incoming HTTP request might have a child span for the Cosmos DB query it triggers, which shows up nested under the request in the trace view, making it obvious which downstream call is responsible for the overall latency.
- OpenTelemetry is the vendor-neutral standard for producing this data: language SDKs instrument common frameworks and libraries automatically (auto-instrumentation) and let you add custom spans by hand for anything auto-instrumentation doesn't cover.

Getting traces into Azure Monitor

- The Azure Monitor OpenTelemetry Distro (or an exporter package like `azure-monitor-opentelemetry-exporter`) sends span data to an Application Insights resource via its connection string, where it powers the *transaction diagnostics* view (one request's full call tree) and the *application map* (how services interact across many requests).
- Many Azure SDKs — including the Cosmos DB SDK — emit their own OpenTelemetry spans for operations like queries, which show up automatically in a trace once the exporter is configured, without you writing any manual instrumentation for that call.
- Every telemetry item carries an operation id shared across the whole distributed operation, which is what lets Application Insights stitch together spans emitted by completely different processes into one trace.

Common confusion
Enabling an OpenTelemetry SDK and pointing it at *some* backend (like a local Jaeger instance for development) is not the same as it reaching Application Insights — the exporter and connection string must specifically target Azure Monitor for traces to appear there, and forgetting to initialize the OpenTelemetry SDK before other instrumented libraries load is a common reason traces come out incomplete.

KQL for analyzing logs and metrics

Once diagnostic data — logs, traces, and metrics — lands in a Log Analytics workspace, Kusto Query Language (KQL) is how you actually ask questions of it, whether that's during an active incident or while writing an alert rule.

Query fundamentals

- A KQL query starts from a table (`AppTraces`, `AppRequests`, `AppDependencies`, `ContainerAppConsoleLogs`, and so on) and pipes (`|`) data through a sequence of operators, each transforming the result of the previous one — similar in spirit to a Unix pipeline.
- Common operators: `where` filters rows, `summarize` aggregates (counts, averages, percentiles) often combined with `by` to group per dimension, `project` selects and renames columns, and `order by` sorts results.
- A troubleshooting query typically looks like: start from requests or traces, filter to a time window and a specific operation or severity, then summarize failure count or latency percentiles by dependency name or status code to find the actual bottleneck or error source.

Logs vs. metrics for diagnosis

- Metrics are lightweight, pre-aggregated numeric time series (CPU percentage, request count) — fast to query and well suited to real-time alerting, but limited in the questions they can answer.
- Logs are full structured records queried with KQL, which can express arbitrarily complex conditions (correlate a specific customer's failed requests with the exact downstream dependency call that failed) that a metric alone can't represent.
- A production troubleshooting workflow commonly starts with a metric-based alert firing (something is wrong, cheaply detected), then pivots to a KQL query against logs to find out specifically what and why.

Common confusion
`AppExceptions` (unhandled application exceptions) and `AppTraces` (explicit log statements, at whatever severity the code logged them) are different tables answering different questions — a failing operation that was caught and logged as a warning shows up in `AppTraces`, not `AppExceptions`, so searching only one table can miss real failures the application handled gracefully but still needs attention.

CHAPTER 5

Quick-reference: common confusions

Every “Common confusion” callout from this guide, gathered here as a last-day revision sheet.

Develop Containerized Solutions on Azure

ACR Tasks is a build and automation service; it doesn't run your production workload continuously. Once an image lands in ACR, orchestration and hosting are separate concerns handled by App Service, Container Apps, or AKS.

Changing an app setting on a container-based App Service app restarts the container (since environment variables are read at process startup) — there's no way to hot-reload a changed setting into a running container process.

KEDA scale rules control how many replicas run; they don't change compute size per replica (CPU/memory), which is set separately in the container app's resource configuration.

A Deployment ensures pods exist and restarts them on failure, but a Service is what makes them reachable at a stable address — deleting a Deployment's Service doesn't stop the pods, and creating pods without a Service leaves them unreachable from outside the cluster.

A pod stuck in `Pending` is a scheduling problem (no node has room, or a required resource isn't available); a pod stuck in `CrashLoopBackOff` is a runtime problem (the container starts and then exits) — the two point troubleshooting in very different directions.

Develop AI Solutions Using Azure Data Management Services

RU consumption is billed the same whether a query returns zero results or many — a query's *cost* is driven by how much data it has to scan and process, not by how much it returns, which is why a well-chosen partition key and indexing policy matter even for queries with small result sets.

The change feed only reflects changes going forward from when a processor's lease container was first initialized (or a specific configured start time) — it isn't a way to replay history from before that unless you explicitly enable an all-versions-and-deletes read mode or start from the beginning of the container's lifetime.

Adding more compute (vCores/memory) to a PostgreSQL server doesn't fix a connection-exhaustion problem by itself — the number of usable connections is bounded by the `max_connections` setting and by memory per connection, so pooling is often a bigger lever than scaling up.

IVFFlat's `lists` parameter is chosen relative to the number of rows *at index build time* — adding a large amount of new data afterward without rebuilding the index degrades recall, since the list boundaries no longer reflect the data's actual distribution. HNSW doesn't have this problem since it has no training phase.

Redis vector search trades some capability for speed — features like Cosmos DB's DiskANN-scale approximate search over huge datasets aren't the point of Redis; it's optimized for low-latency lookups over data that comfortably fits the tier's memory profile, not for being the system of record for a large embedding corpus.

Connect to and Consume Azure Services

A message that keeps failing and is redelivered isn't lost — it's still in the queue being retried — until it's either dead-lettered after too many attempts or its TTL expires. Monitoring dead-letter queue depth is a standard health signal precisely because a growing DLQ means something downstream is systematically broken, not just

occasionally slow.

Event Grid notifies you that something happened; it doesn't guarantee a consumer will process it exactly once, and it isn't a work queue you pull batches from — Service Bus (or Event Grid delivering *into* Service Bus) is the better fit when you need ordered, exactly-once-processed units of work rather than fan-out notifications.

A trigger and an input binding sound similar but are different: a trigger starts the function's execution and a function has exactly one; input/output bindings are optional, can be multiple, and only supply or receive data during a run that's already been triggered by something else.

Secure, Monitor, and Troubleshoot Azure Solutions

Enabling soft-delete and purge protection on a vault protects against accidental permanent deletion of secrets — it's a data-protection setting, unrelated to rotation, which is about periodically replacing a secret's *value*, not protecting the vault's contents from deletion.

App Configuration and Key Vault are complementary, not competing — the rule of thumb is App Configuration for values that are fine to see in a config blade, and Key Vault (referenced from App Configuration or directly) for anything that would be a security incident if it leaked.

Enabling an OpenTelemetry SDK and pointing it at *some* backend (like a local Jaeger instance for development) is not the same as it reaching Application Insights — the exporter and connection string must specifically target Azure Monitor for traces to appear there, and forgetting to initialize the OpenTelemetry SDK before other instrumented libraries load is a common reason traces come out incomplete.

`AppExceptions` (unhandled application exceptions) and `AppTraces` (explicit log statements, at whatever severity the code logged them) are different tables answering different questions — a failing operation that was caught and logged as a warning shows up in `AppTraces`, not `AppExceptions`, so searching only one table can miss real failures the application handled gracefully but still needs attention.