What is Azure Cosmos DB for MongoDB?

Azure Cosmos DB is a fully managed NoSQL, relational, and vector database for modern app development. It offers single-digit millisecond response times, automatic and instant scalability, and guaranteed speed at any scale.

It is the database that ChatGPT relies on to dynamically scale with high reliability and low maintenance.

Azure Cosmos DB for MongoDB makes it easy to use Azure Cosmos DB as if it were a MongoDB database. You can use your existing MongoDB skills and continue to use your favorite MongoDB drivers, SDKs, and tools by pointing your application to the connection string for your account using the API for MongoDB.

Azure Cosmos DB for MongoDB provides an SLA that covers the full stack: the database and the underlying infrastructure. Unlike third-party MongoDB services such as MongoDB Atlas, which only cover the database and exclude services, hardware, or software provided by the cloud platform.

vCore architecture (recommended)

A fully managed MongoDB-compatible service with dedicated instances for new and existing MongoDB apps. This architecture offers a familiar vCore architecture for MongoDB users, instantaneous scaling, and seamless native integration with Azure services.

Native Vector Search: Seamlessly integrate your AI-based applications with your data that's stored in Azure Cosmos DB for MongoDB vCore. This integration is an all-in-one solution, unlike other vector search solutions that send your data between service integrations.

Instantaneous scalability: With Autoscale, your database scales instantaneously with zero warmup period. Other MongoDB offerings such as MongoDB Atlas can take hours to scale up and up to days to scale down.

Flat pricing with Low total cost of ownership: Enjoy a familiar pricing model for Azure Cosmos DB for MongoDB vCore, based on compute (vCores & RAM) and storage (disks).

Elevate querying with Text Indexes: Enhance your data querying efficiency with our text indexing feature. Seamlessly navigate full-text searches across MongoDB collections, simplifying the process of extracting valuable insights from your documents.

Scale with no shard key required: Simplify your development process with high-capacity vertical scaling, all without the need for a shard key. Sharding and scaling horizontally is simple once collections are into the TBs.

Free 35 day Backups with point in time restore (PITR): Azure Cosmos DB for MongoDB vCore offers free 35 day backups for any amount of data.

Choose vCore-based if

- You're migrating (lift & shift) an existing MongoDB workload or building a new MongoDB application.
- Your workload has more long-running queries, complex aggregation pipelines, distributed transactions, joins, etc.
- You prefer high-capacity vertical and horizontal scaling with familiar vCore-based cluster tiers such as M30, M40, M50 and more.
- You're running applications requiring 99.995% availability.
- You need native support for storing and searching vector embeddings.

Request Unit (RU) architecture

A fully managed MongoDB-compatible service with flexible scaling using Request Units (RUs). Designed for cloud-native applications.

Instantaneous scalability: With the Autoscale feature, your database scales instantaneously with zero warmup period. Other MongoDB offerings such as MongoDB Atlas can take hours to scale up and up to days to scale down.

Automatic and transparent sharding: The API for MongoDB manages all of the infrastructure for you. This management includes sharding and optimizing the number of shards. Other MongoDB offerings such as MongoDB Atlas, require you to specify and manage sharding to horizontally scale. This automation gives you more time to focus on developing applications for your users.

Five 9's of availability: 99.999% availability is easily configurable to ensure your data is always there for you.

Active-active database: Unlike MongoDB Atlas, Cosmos DB for MongoDB supports active-active across multiple regions. Databases can span multiple regions, with no single point of failure for writes and reads for the same data. MongoDB Atlas global clusters only support active-passive deployments for writes for the same data.

Cost efficient, granular, unlimited scalability: Sharded collections can scale to any size, unlike other MongoDB service offerings. The Azure Cosmos DB platform can scale in increments as small as 1/100th of a VM due to its architecture. This scalability means that you can scale your database to the exact size you need, without paying for unused resources.

Real time analytics (HTAP) at any scale: Run analytics workloads against your transactional MongoDB data in real time with no effect on your database. This analysis is fast and inexpensive, due to the cloud native analytical columnar store being utilized, with no ETL pipelines. Easily create Power BI dashboards, integrate with Azure Machine Learning and Azure AI services, and bring all your data from your MongoDB workloads into a single data warehousing solution. Learn more about the Azure Synapse Link.

Serverless deployments: Cosmos DB for MongoDB offers a serverless capacity mode. With Serverless, you're only charged per operation, and don't pay for the database when you don't use it.

Choose RU-based if

- You're building new cloud-native MongoDB apps or refactoring existing apps for cloud-native benefits.
- Your workload has more point reads (fetching a single item by its _id and shard key value) and few long-running queries and complex aggregation pipeline operations.
- You want limitless horizontal scalability, instantaneous scale up, and granular throughput control.
- You're running mission-critical applications requiring industry-leading 99.999% availability.

1.2 Azure Cosmos DB for NoSQL

Azure Cosmos DB for NoSQL library for .NET

1. Authenticate to the Azure Developer CLI using azd auth login. Follow the steps specified by the tool to authenticate to the CLI using your preferred Azure credentials.

```
azd auth login
```

2. Use azd init to initialize the project.

```
azd init --template cosmos-db-nosql-dotnet-quickstart
```

3. Deploy the Azure Cosmos DB account using azd up. The Bicep templates also deploy a sample web application.

```
azd up
```

4. Install the client library

```
dotnet add package Microsoft.Azure.Cosmos --version 3.*
dotnet add package Azure.Identity --version 1.12.*
```

Object model

Name	Description
CosmosClient	This class is the primary client class and is used to manage account-wide metadata or databases.
Database	This class represents a database within the account.
Container	This class is primarily used to perform read, update, and delete operations on either the container or the items stored within the container.
PartitionKey	This class represents a logical partition key. This class is required for many common operations and queries.

Databases, containers, and items in Azure Cosmos DB

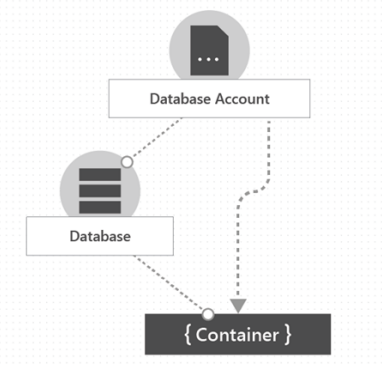
Azure Cosmos DB is a fully managed platform as a service (PaaS).
To begin using Azure Cosmos DB, create an Azure Cosmos DB account in an Azure resource group in your subscription. Then, create databases and containers within the account.
Your Azure Cosmos DB account contains a unique Domain Name System (DNS) name. You can manage the DNS name by using many tools, including:

- Azure portal
- Azure Resource Manager templates
- Bicep templates
- Azure PowerShell
- Azure CLI
- Azure Management SDKs
- Azure REST API

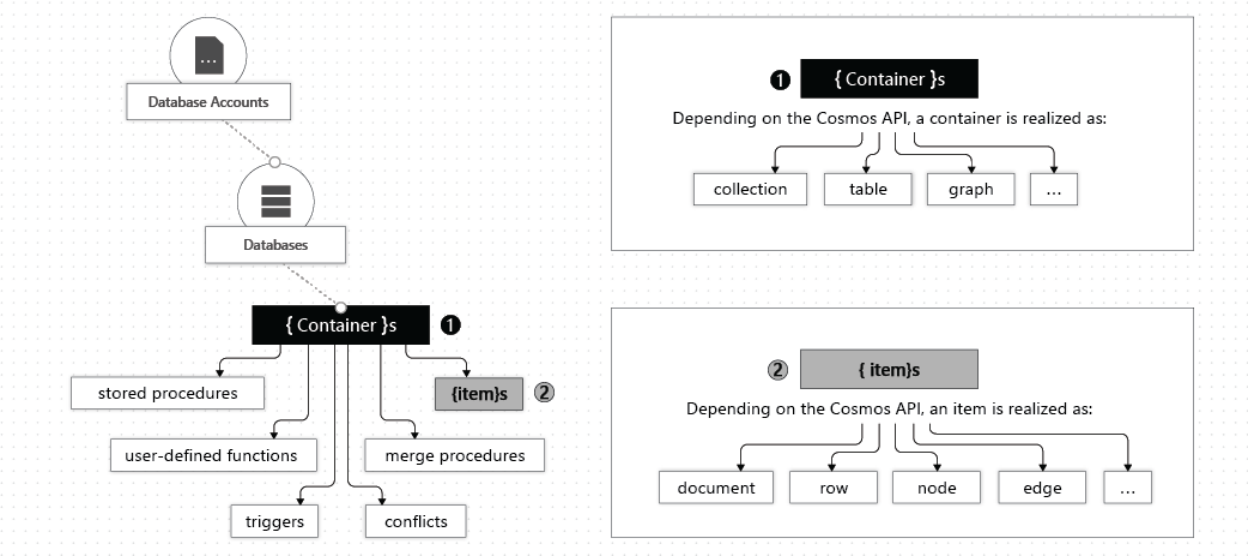
For replicating your data and throughput across multiple Azure regions, you can add and remove Azure regions to your account at any time. You can configure your account to have either a single region or multiple write regions.

Currently, you can create a maximum of 50 Azure Cosmos DB accounts under an Azure subscription. You can increase this limit by making a support request.

You can manage a virtually unlimited amount of data and provisioned throughput by using a single Azure Cosmos DB account. To manage your data and provisioned throughput, you create one or more databases within your account and then create one or more containers to store your data.The following image shows the hierarchy of elements in an Azure Cosmos DB account.



The following image shows the hierarchy of entities in an Azure Cosmos DB account.



In Azure Cosmos DB, a database is similar to a namespace. A database is simply a group of containers. The following table shows how a database is mapped to various API-specific entities:
Expand table

Azure Cosmos DB entity	API for NoSQL	API for Apache Cassandra	API for MongoDB	API for Apache Gremlin	API for Table
Azure Cosmos DB database	Database	Keyspace	Database	Database	Not applicable

With API for Table accounts, tables in Azure Cosmos DB are created at the account level to maintain compatibility with Azure Table Storage.

Azure Cosmos DB containers

An Azure Cosmos DB container is where data is stored. Unlike most relational databases, which scale up with larger sizes of virtual machines, Azure Cosmos DB scales out.

Data is stored on one or more server's called *partitions*. To increase partitions, you increase throughput, or they grow automatically as storage increases. This relationship provides a virtually unlimited amount of throughput and storage for a container.

When you create a container, you need to supply a partition key.

The partition key is a property that you select from your items to help Azure Cosmos DB distribute the data efficiently across partitions. Azure Cosmos DB uses the value of this property to route data to the appropriate partition to be written, updated, or deleted

The underlying storage mechanism for data in Azure Cosmos DB is called a *physical partition*. Physical partitions can have a throughput amount up to 10,000 Request Units per second, and they can store up

to 50 GB of data. Azure Cosmos DB abstracts this partitioning concept with a logical partition, which can store up to 20 GB of data.

Logical partitions allow the service to provide greater elasticity and better management of data on the underlying physical partitions as you add more partitions.

When you create a container, you configure throughput in one of the following modes:

- Dedicated throughput: The throughput on a container is exclusively reserved for that container. There are two types of dedicated throughput: standard and autoscale. To learn more, see Provision standard (manual) throughput on an Azure Cosmos DB container.

- Shared throughput: Throughput is specified at the database level and then shared with up to 25 containers within the database. Sharing of throughput excludes containers that are configured with their own dedicated throughput.

Shared throughput can be a good option when all of the containers in the database have similar requests and storage needs, or when you don't need predictable performance on the data.

To avoid affecting performance, you can set a time to live (TTL) on selected items in a container or on the entire container to delete those items automatically in the background with unused throughput.

Azure Cosmos DB provides a built-in capability for changing data capture called change feed. You can use it to subscribe to all the changes to data within your container.

You can register stored procedures, triggers, user-defined functions (UDFs), and merge procedures for your container.
A container is specialized into API-specific entities, as shown in the following table:
Expand table

Azure Cosmos DB entity	API for NoSQL	API for Cassandra	API for MongoDB	API for Gremlin	API for Table
Azure Cosmos DB container	Container	Table	Collection	Graph	Table

Create a synthetic partition key

Concatenate multiple properties of an item

You can form a partition key by concatenating multiple property values into a single artificial partitionKey property.

```
{
"deviceId": "abc-123",
"date": 2018
}
```

For the previous document, one option is to set /deviceId or /date as the partition key. Use this option, if you want to partition your container based on either device ID or date.

Another option is to concatenate these two values into a synthetic partitionKey property that's used as the partition key.

```
{
"deviceId": "abc-123",
"date": 2018,
"partitionKey": "abc-123-2018"
}
```

Use a partition key with a random suffix

Another possible strategy to distribute the workload more evenly is to append a random number at the end of the partition key value.
When you distribute items in this way, you can perform parallel write operations across partitions.

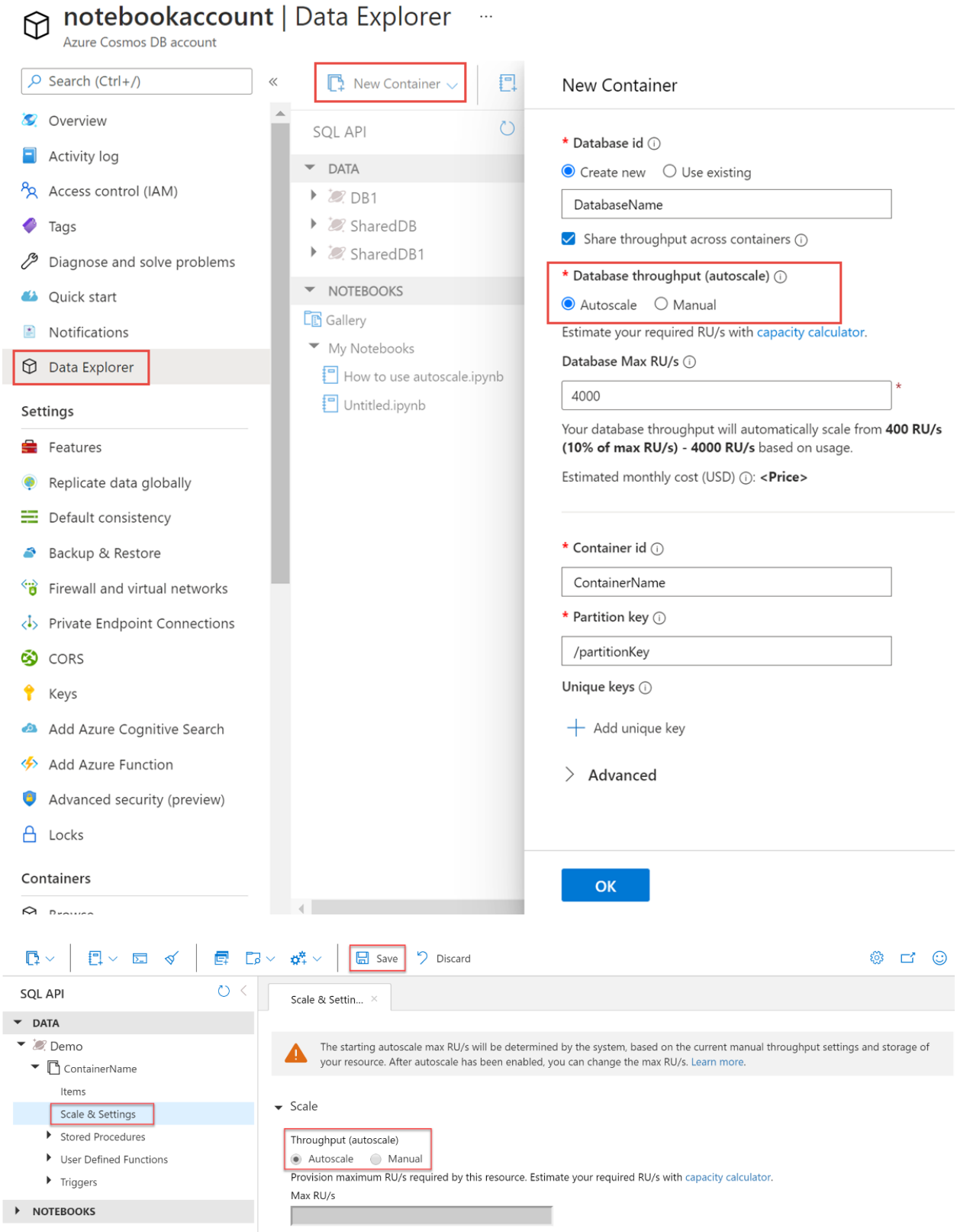
An example is if a partition key represents a date. You might choose a random number between 1 and 400 and concatenate it as a suffix to the date. This method results in partition key values like 2018-08-09.1,2018-08-09.2, and so on, through 2018-08-09.400

Use a partition key with pre-calculated suffixes

Instead of using a random number to distribute the items among the partitions, use a number that is calculated based on something that you want to query.

Provision autoscale throughput on database or container in Azure Cosmos DB - API for NoSQL

You can enable autoscale on a single container, or provision autoscale throughput on a database and share it among all the containers in the database.


notebookaccount | Data Explorer
Azure Cosmos DB account
Search (Ctrl+/)
Overview
Activity log
Access control (IAM)
Tags
Diagnose and solve problems
Quick start
Notifications
Data Explorer
Settings
Features
Replicate data globally
Default consistency
Backup & Restore
Firewall and virtual networks
Private Endpoint Connections
CORS
Keys
Add Azure Cognitive Search
Add Azure Function
Advanced security (preview)
Locks
Containers
New Container
SQL API
DATA
DB1
SharedDB
SharedDB1
NOTEBOOKS
Gallery
My Notebooks
How to use autoscale.ipynb
Untitled.ipynb
New Container
* Database id
Create new
Use existing
DatabaseName
Share throughput across containers
* Database throughput (autoscale)
Autoscale
Manual
Estimate your required RU/s with capacity calculator.
Database Max RU/s
4000
Your database throughput will automatically scale from 400 RU/s (10% of max RU/s) - 4000 RU/s based on usage.
Estimated monthly cost (USD): <Price>
* Container id
ContainerName
* Partition key
/partitionKey
Unique keys
Add unique key
Advanced
OK
Save
Discard
SQL API
DATA
Demo
ContainerName
Items
Scale & Settings
Stored Procedures
User Defined Functions
Triggers
NOTEBOOKS
Scale & Settin...
The starting autoscale max RU/s will be determined by the system, based on the current manual throughput settings and storage of your resource. After autoscale has been enabled, you can change the max RU/s. Learn more.
Scale
Throughput (autoscale)
Autoscale
Manual
Provision maximum RU/s required by this resource. Estimate your required RU/s with capacity calculator.
Max RU/s

Consistency levels in Azure Cosmos DB

Azure Cosmos DB offers five well-defined levels. From strongest to weakest, the levels are:

- Strong
- Bounded staleness
- Session
- Consistent prefix
- Eventual

Strong consistency

Strong consistency offers a linearizability guarantee. Linearizability refers to serving requests concurrently. The reads are guaranteed to return the most recent committed version of an item.

A client never sees an uncommitted or partial write. Users are always guaranteed to read the latest committed write.

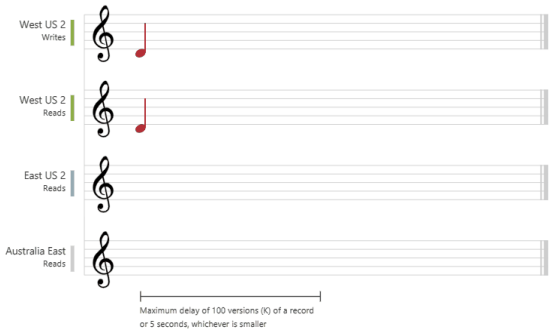
Bounded staleness consistency

In bounded staleness consistency, the lag of data between any two regions is always less than a specified amount. The amount can be "K" versions (that is, "updates") of an item or by "T" time intervals, whichever is reached first. In other words, when you choose bounded staleness, the maximum "staleness" of the data in any region can be configured in two ways:

- The number of versions (*K*) of the item
- The time interval (*T*) reads might lag behind the writes

Bounded Staleness is beneficial primarily to single-region write accounts with two or more regions. If the data lag in a region (determined per physical partition) exceeds the configured staleness value, writes for that partition are throttled until staleness is back within the configured upper bound.

For a single-region account, Bounded Staleness provides the same write consistency guarantees as Session and Eventual Consistency.
With Bounded Staleness, data is replicated to a local majority (three replicas in a four replica set) in the single region.



Session consistency

In session consistency, within a single client session, reads are guaranteed to honor the read-your-writes, and write-follows-reads guarantees.
This guarantee assumes a single “writer” session or sharing the session token for multiple writers.

Writes are replicated to a minimum of three replicas (in a four replica set) in the local region, with asynchronous replication to all other regions.

Session Tokens in Azure Cosmos DB are partition-bound, meaning they are exclusively associated with one partition. In order to ensure you can read your writes, use the session token that was last generated for the relevant item(s).

If the replica doesn't contain data for that session, the client retries the request against another replica in the region. If necessary, the client retries the read against extra available regions until data for the specified session token is retrieved.

Session consistency is the most widely used consistency level for both single region and globally distributed applications. It provides write latencies, availability, and read throughput comparable to that of eventual consistency.

Consistent prefix consistency

Like all consistency levels weaker than Strong, writes are replicated to a minimum of three replicas (in a four-replica set) in the local region, with asynchronous replication to all other regions.

In consistent prefix, updates made as single document writes see eventual consistency.

Updates made as a batch within a transaction, are returned consistent to the transaction in which they were committed.
Write operations within a transaction of multiple documents are always visible together.

Eventual consistency

Like all consistency levels weaker than Strong, writes are replicated to a minimum of three replicas (in a four replica set) in the local region, with asynchronous replication to all other regions.

In Eventual consistency, the client issues read requests against any one of the four replicas in the specified region. This replica may be lagging and could return stale or no data.

Eventual consistency is the weakest form of consistency because a client may read the values that are older than the ones it read in the past. Eventual consistency is ideal where the application doesn't *require any ordering guarantees*. Examples include count of Retweets, Likes, or non-threaded comments.

Databases

Built-in role	Description	ID
Azure Connected SQL Server Onboarding	Allows for read and write access to Azure resources for SQL Server on Arc-enabled servers.	e8113dce-c529-4d33-91fa-e9b972617508
Cosmos DB Account Reader Role	Can read Azure Cosmos DB account data. See DocumentDB Account Contributor for managing Azure Cosmos DB accounts.	fbdf93bf-df7d-467e-a4d2-9458aa1360c8
Cosmos DB Operator	Lets you manage Azure Cosmos DB accounts, but not access data in them. Prevents access to account keys and connection strings.	230815da-be43-4aae-9cb4-875f7bd000aa
CosmosBackupOperator	Can submit restore request for a Cosmos DB database or a container for an account	db7b14f2-5adf-42da-9f96-f2ee17bab5cb
CosmosRestoreOperator	Can perform restore action for Cosmos DB database account with continuous backup mode	5432c526-bc82-444a-b7ba-57c5b0b5b34f
DocumentDB Account Contributor	Can manage Azure Cosmos DB accounts. Azure Cosmos DB is formerly known as DocumentDB.	5bd9cd88-fe45-4216-938b-f97437e15450

PostgreSQL Flexible Server Long Term Retention Backup Role	Role to allow backup vault to access PostgreSQL Flexible Server Resource APIs for Long Term Retention Backup.	c088a766-074b-43ba-90d4-1fb21feae531
Redis Cache Contributor	Lets you manage Redis caches, but not access to them.	e0f68234-74aa-48ed-b826-c38b57376e17
SQL DB Contributor	Lets you manage SQL databases, but not access to them. Also, you can't manage their security-related policies or their parent SQL servers.	9b7fa17d-e63e-47b0-bb0a-15c516ac86ec
SQL Managed Instance Contributor	Lets you manage SQL Managed Instances and required network configuration, but can't give access to others.	4939a1f6-9ae0-4e48-a1e0-f2cbe897382d
SQL Security Manager	Lets you manage the security-related policies of SQL servers and databases, but not access to them.	056cd41c-7e88-42e1-933e-88ba6a50c9c3
SQL Server Contributor	Lets you manage SQL servers and databases, but not access to them, and not their security-related policies.	6d8ee4ec-f05a-4a1d-8b00-a9b17e38b437

Configure multi-region writes in your applications that use Azure Cosmos DB

After you enable your account for multiple write regions, you must make two changes in your application to the ConnectionPolicy.

Within the ConnectionPolicy, set UseMultipleWriteLocations to true and pass the name of the region where the application is deployed to ApplicationRegion.

This action populates the PreferredLocations property based on the geo-proximity from location passed in. If a new region is later added to the account, the application doesn't have to be updated or redeployed. It automatically detects the closer region and auto-homes on to it should a regional event occur.

```
ConnectionPolicy policy = new ConnectionPolicy
  {
     ConnectionMode = ConnectionMode.Direct,
     ConnectionProtocol = Protocol.Tcp,
```

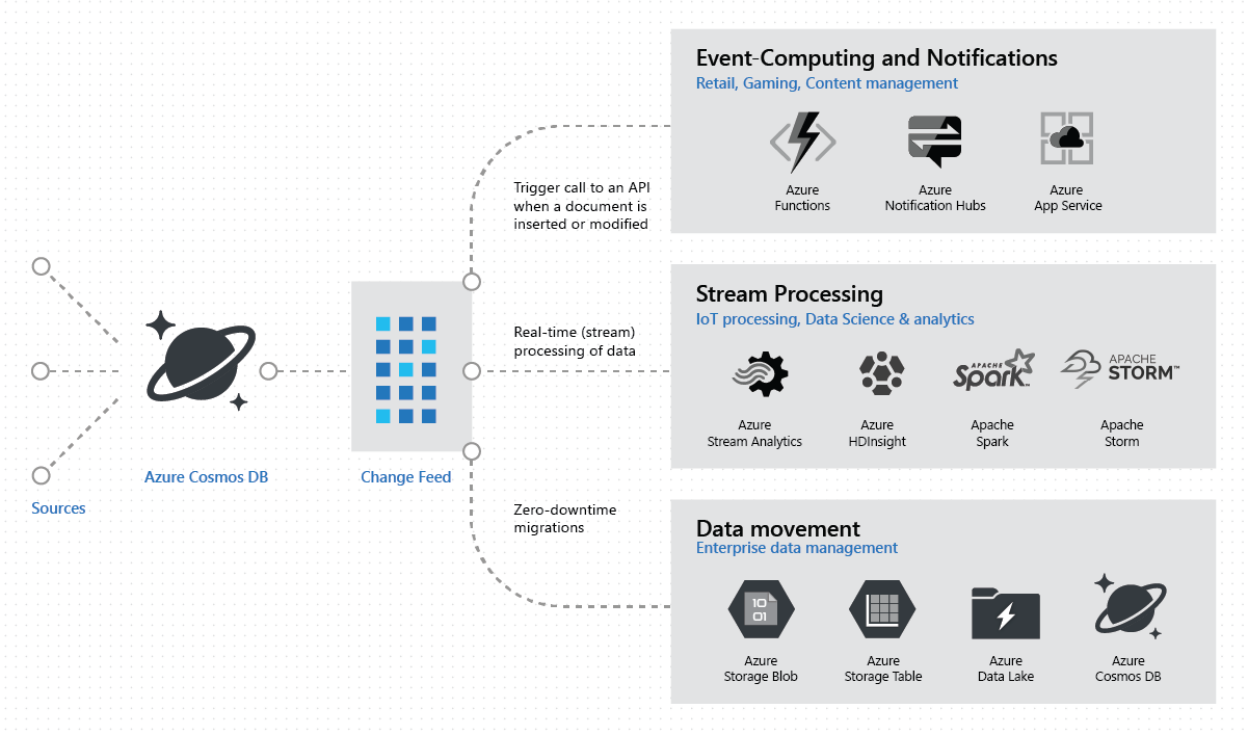
```
    UseMultipleWriteLocations = true
  };
policy.SetCurrentLocation("West US 2");
```

Change feed design patterns in Azure Cosmos DB

The Azure Cosmos DB change feed enables efficient processing of large datasets that have a high volume of writes. Change feed also offers an alternative to querying an entire dataset to identify what has changed.

Azure Cosmos DB is well-suited for IoT, gaming, retail, and operational logging applications. A common design pattern in these applications is to use changes to the data to trigger other actions. Examples of these actions include:

- Triggering a notification or a call to an API when an item is inserted, updated, or deleted.
- Real-time stream processing for IoT or real-time analytics processing on operational data.
- Data movement such as synchronizing with a cache, a search engine, a data warehouse, or cold storage.

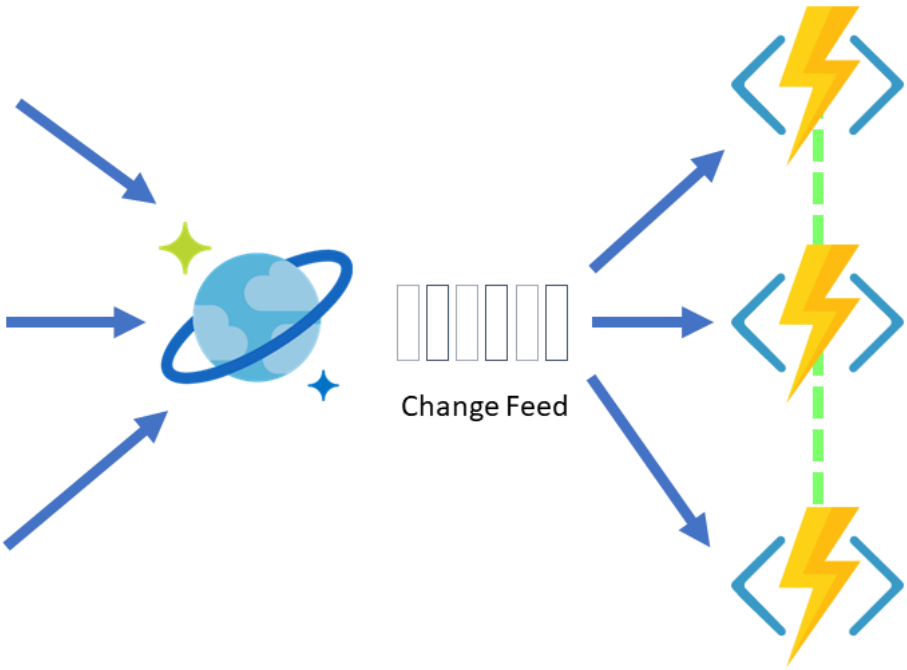


Serverless event-based architecture with Azure Cosmos DB and Azure Functions

Azure Functions provides the simplest way to connect to the change feed. You can create small reactive Azure Functions that will be automatically triggered on each new event in your Azure Cosmos DB container's change feed.

With the Azure Functions trigger for Azure Cosmos DB, you can leverage the Change Feed Processor's scaling and reliable event detection functionality without the need to maintain any worker infrastructure. Just focus on your Azure Function's logic without worrying about the rest of the event-sourcing pipeline. You can even mix the Trigger with any other Azure Functions bindings.

Azure Functions SendGrid bindings



Send email by using SendGrid bindings in Azure Functions. Azure Functions supports an output binding for SendGrid.

Event sourcing

The event sourcing pattern involves using an append-only store to record the full series of actions on that data. The Azure Cosmos DB change feed is a great choice as a central data store in event sourcing architectures in which all data ingestion is modeled as writes (no updates or deletes).

In this case, each write to Azure Cosmos DB is an "event," so there's a full record of past events in the change feed. Typical uses of the events published by the central event store are to maintain materialized views or to integrate with external systems. Because there's no time limit for retention in the change feed latest version mode, you can replay all past events by reading from the beginning of your Azure

Cosmos DB container's change feed. You can even have multiple change feed consumers subscribe to the same container's change feed.

Azure Cosmos DB is a great central append-only persistent data store in the event sourcing pattern because of its strengths in horizontal scalability and high availability. In addition, the change feed processor offers an "at least once" guarantee, ensuring that you don't miss processing any events.

How to write stored procedures

Stored procedures are written using JavaScript, and they can create, update, read, query, and delete items inside an Azure Cosmos DB container. Stored procedures are registered per collection, and can operate on any document or an attachment present in that collection.

Note

Azure Cosmos DB has a different charging policy for stored procedures. Because stored procedures can execute code and consume any number of request units (RUs), each execution requires an upfront charge. This ensures that stored procedure scripts don't impact backend services. The amount charged upfront equals the average charge consumed by the script in previous invocations. The average RUs per operation is reserved before execution. If the invocations have a lot of variance in RUs, your budget utilization might be affected. As an alternative, you should use batch or bulk requests instead of stored procedures to avoid variance around RU charges.

Here's a simple stored procedure that returns a "Hello World" response.

JavaScriptCopy

```
var helloWorldStoredProc = {
   id: "helloWorld",
   serverScript: function () {
      var context = getContext();
      var response = context.getResponse();

      response.setBody("Hello, World");
   }
}
```

Pre-triggers

The following example shows how a pre-trigger is used to validate the properties of an Azure Cosmos DB item that's being created. This example uses the ToDoList sample from the Quickstart .NET API for NoSQL to add a timestamp property to a newly added item if it doesn't contain one.

JavaScriptCopy

```
function validateToDoItemTimestamp() {
   var context = getContext();
   var request = context.getRequest();
```

```
    // item to be created in the current operation
    var itemToCreate = request.getBody();

    // validate properties
    if (!("timestamp" in itemToCreate)) {
        var ts = new Date();
        itemToCreate["timestamp"] = ts.getTime();
    }

    // update the item that will be created
    request.setBody(itemToCreate);
}
```

Pre-triggers can't have any input parameters. The request object in the trigger is used to manipulate the request message associated with the operation. In the previous example, the pre-trigger is run when creating an Azure Cosmos DB item, and the request message body contains the item to be created in JSON format.

When triggers are registered, you can specify the operations that it can run with. This trigger should be created with a TriggerOperation value of TriggerOperation.Create, which means that using the trigger in a replace operation isn't permitted.

For examples of how to register and call a pre-trigger, see pre-triggers and post-triggers.

Post-triggers

The following example shows a post-trigger. This trigger queries for the metadata item and updates it with details about the newly created item.

JavaScriptCopy

```
function updateMetadata() {
    var context = getContext();
    var container = context.getCollection();
    var response = context.getResponse();

    // item that was created
    var createdItem = response.getBody();

    // query for metadata document
    var filterQuery = 'SELECT * FROM root r WHERE r.id = "_metadata"';
    var accept = container.queryDocuments(container.getSelfLink(), filterQuery,
        updateMetadataCallback);
    if(!accept) throw "Unable to update metadata, abort";

    function updateMetadataCallback(err, items, responseOptions) {
        if(err) throw new Error("Error" + err.message);

        if(items.length != 1) throw 'Unable to find metadata document';

        var metadataItem = items[0];

        // update metadata
```

```
        metadataItem.createdItems += 1;
        metadataItem.createdNames += " " + createdItem.id;
        var accept = container.replaceDocument(metadataItem._self,
           metadataItem, function(err, itemReplaced) {
                if(err) throw "Unable to update metadata, abort";
           });

        if(!accept) throw "Unable to update metadata, abort";
        return;
    }
}
```