

**I have a serverless application deployed in Azure where billing records are stored in Azure Cosmos DB. The application is read-heavy, but access to records older than 3 months is rare.**

Here are the system constraints and requirements:  
Current Setup:  
Each billing record can be up to 300 KB  
Over 2 million records in the Cosmos DB collection  
Need to retain historical records, but access latency of a few seconds is acceptable  
Optimization Goals:  
Reduce Cosmos DB costs, especially from storage and provisioned throughput  
No downtime or data loss during the transition  
No changes to existing API contracts  
Simple and easy-to-maintain architecture  
What I Need:  
Step-by-step solution for cost optimization, including data archival strategies and technologies involved  
How to seamlessly migrate older records (older than 3 months) without affecting the live system  
Strategy to serve archived data on request, while keeping the current API unchanged  
A simple architecture diagram showing components and data flow  
Pseudocode or scripts (e.g., using Azure Functions, Data Factory, Logic Apps, or CLI) for:  
Archiving old records  
Accessing archived records when requested  
Monitoring and managing costs  
Any best practices around data tiering in Cosmos DB or integrating with Azure Blob Storage, Azure Table Storage, or other services  
Bonus:  
If possible, recommend serverless patterns that match this scenario  
Show how to implement a transparent cache or proxy layer to abstract archive retrieval without modifying the current API

Here is a complete, step-by-step solution to optimize costs for your read-heavy Azure serverless application with Cosmos DB, implementing efficient data archival, seamless access, and simple architecture—all without changing existing API contracts.

**1. Cost Optimization & Data Archival: Approach**

**a. Archive Old Data to Azure Blob Storage**

* **Why?** Rarely read data (>3 months old) drives up Cosmos DB storage and RU costs. Blob Storage is much cheaper for archival and tolerates higher latency[[1]](#fn1)[[2]](#fn2)[[3]](#fn3).
* **How?**
  + Move records older than 3 months from Cosmos DB to Azure Blob Storage (preferably in JSON or Parquet format for easy retrieval).
  + Consider using Blob Storage “Cool” or “Archive” access tier for further savings, since you accept seconds-level latency[[4]](#fn4)[[5]](#fn5).

**b. Retrieval Strategy**

* When a request for old data arrives, the system first checks Cosmos DB:
  + **If found:** Return immediately.
  + **If not found:** Fetch from Blob Storage, rehydrate if needed, and serve (possibly caching back in Cosmos DB or a cache for subsequent requests).

**2. Zero-Downtime Migration Strategy**

1. **Start with full read/write access** in Cosmos DB.
2. **Archival Job**: Use Azure Data Factory or an Azure Function scheduled job to:
   * Identify records older than 3 months.
   * Copy them to Azure Blob Storage (in batch jobs).
   * After successful copy, *flag* or *delete* in Cosmos DB to free up storage[[2]](#fn2)[[3]](#fn3).
3. **No API contract changes:** The archival/retrieval logic should be implemented behind the scenes in your backend or in a middleware Azure Function/Logic App triggered by API calls or proxies, thus preserving the existing API.

* **Important:** Test thoroughly in a staging environment to prevent data loss and ensure zero downtime.

**3. Serving Archived Data Seamlessly**

* Use a “proxy” or transparent cache layer (e.g., API facade with Azure Functions, or a reverse proxy with logic) that intercepts requests:
  + If data not present in Cosmos DB, tries to fetch from Blob.
  + Optionally, re-inserts frequently accessed archived records back to Cosmos DB or an in-memory cache (e.g., Azure Cache for Redis) to reduce future latency.
* This keeps your API contract unchanged[[5]](#fn5)[[6]](#fn6).

**4. Architecture Diagram Description**

[Client App/API]  
 |  
 v  
 [Serverless App Logic]  
 |  
 +--[Check Cosmos DB]---(Found?)---Yes---> Return Result  
 | |  
 | No  
 v |  
[Check Blob Storage (Azure Function/Logic App/ADF pipeline)]  
 |  
 v  
[Return Data]----(Optional: cache back in Cosmos DB/Redis for hotness)

* **Archival Pipeline** (runs periodic batch job): Cosmos DB ➔ Blob Storage
* **API Facade/Proxy**: Handles reads transparently (Cosmos DB first, then Blob Storage)

**5. Pseudocode/Scripts**

**a. Archiving Old Records**

**Azure Function or Data Factory Pipeline Pseudocode:**

# Run daily/weekly, triggered as Timer Function or ADF schedule  
for record in cosmosdb.query("SELECT \* FROM records WHERE timestamp < (NOW() - 90 days)"):  
 blob\_path = f"archive/{record.id}.json"  
 blob\_storage.write(blob\_path, record)  
 # Optional: Delete or update record in Cosmos DB  
 cosmosdb.delete(record.id)

**Using Azure CLI (Data Factory) to Export and Archive**

az datafactory pipeline create-run --factory-name <factory> --resource-group <rg> --name ArchiveOldRecordsPipeline

Pipeline would:

* Query Cosmos DB via Data Factory connector
* Filter for records older than 3 months
* Write to Azure Blob Storage sink

**b. Accessing Archived Records (Azure Function example)**

def get\_billing\_record(record\_id):  
 record = cosmosdb.get(record\_id)  
 if record:  
 return record  
 else:  
 # If not found, try Blob Storage  
 blob\_path = f"archive/{record\_id}.json"  
 record = blob\_storage.read(blob\_path)  
 if record:  
 # Optional: cache in Cosmos DB or Redis  
 # cosmosdb.write(record)  
 return record  
 else:  
 return None

**c. Cost Monitoring/Management**

* Enable alerts and budgets in Azure Cost Management[[7]](#fn7)[[8]](#fn8).
* Use Azure Monitor Workbooks to track Blob Storage and Cosmos DB metrics and costs.
* Example: Set up Azure Alert when storage usage approaches certain thresholds.

**6. Best Practices**

* **Indexing:** Fine-tune Cosmos DB indexing policies—index only fields you need for queries; keep archived data indexing to a minimum before export[[1]](#fn1).
* **Partitioning:** Regularly check hot/cold partition usage for balance[[9]](#fn9).
* **Security:** Ensure both Cosmos DB and Blob Storage have proper access control and encryption[[10]](#fn10).
* **Backup:** Make backups before running archival/deletion scripts.

**7. Serverless/Data Tiering Patterns**

* **Event-Driven Archival:** Use Cosmos DB Change Feed to trigger archiving (for future data). For historical, use batch migration with ADF/Azure Functions[[2]](#fn2)[[3]](#fn3).
* **Function-as-a-Proxy:** An Azure Function or Logic App that fronts all reads and manages Cosmos DB/Blob Storage routing seamlessly[[5]](#fn5)[[6]](#fn6).
* **Cache-aside:** Integrate Azure Redis Cache if latency for “cold” Blob requests needs further optimization.

**8. Bonus: Transparent Proxy/Cache Layer**

* Deploy an Azure API Management Gateway or implement an application-level reverse proxy (e.g., with Azure Functions or App Gateway) that intercepts calls, checks Cosmos DB, then falls back to Blob Storage if needed (using the logic above).
* Optionally: use Redis or another in-memory cache to speed up access of recently requested archived data.

**Key Technologies:**

* Azure Cosmos DB (hot, recent data)
* Azure Blob Storage (cool/archive, old data)
* Azure Data Factory or Azure Functions (for ETL archiving jobs)
* Azure Functions or Logic Apps (for API proxy or “archive fetch” logic)
* Azure Monitor + Cost Management (for tracking spend)

This solution ensures cost savings in storage/compute, no downtime, no API contract changes, low maintenance, and a scalable serverless architecture tailored for your application scenario[[1]](#fn1)[[7]](#fn7)[[2]](#fn2).

⁂

1. <https://learn.microsoft.com/en-us/azure/cosmos-db/optimize-cost-storage>

1. <https://stackoverflow.com/questions/59746790/best-practice-of-archive-data-in-cosmos-db>

1. <https://learn.microsoft.com/en-us/answers/questions/1281134/customer-plan-to-use-cosmos-db-and-retain-operatio>

1. <https://azure.microsoft.com/en-au/products/storage>

1. <https://github.com/olafwrieden/azure-data-archiving-practices>

1. <https://estuary.dev/integrations/azure-blob-storage-to-cosmosdb-mongodb/>

1. <https://turbo360.com/blog/azure-cosmos-db-cost-optimization>

1. <https://learn.microsoft.com/en-us/azure/cosmos-db/optimize-costs>

1. <https://learn.microsoft.com/en-us/azure/cosmos-db/optimize-cost-throughput>

1. <https://imperiumdynamics.com/blog/Difference-between-Azure-blob-security-and-Cosmos-DB.html>