**I have a cost optimization challenge in my Azure serverless architecture. We use Azure Cosmos DB to store billing records, and the system is read-heavy. Each billing record is up to 300 KB in size, and we currently have over 2 million records. Records older than 3 months are rarely accessed, but when they are, we still need to serve them within a few seconds of latency.**

Constraints:

• No data loss and no downtime

• No changes to the existing API contracts (read/write)

• The solution should be simple to implement and maintain

Requirements:

• I want a detailed and practical solution to reduce the cost of storing these records while maintaining data availability and acceptable access latency for older data.

Please include:

• The architectural changes or patterns you suggest (e.g., tiered storage, archival, etc.)

• Azure services involved and why they are chosen

• A step-by-step migration plan that avoids downtime or breaking APIs

• An estimation of expected cost savings or billing model comparison (if possible)

**Assume this is a production environment with real users and strict availability and performance SLAs.**

**Here’s a practical and production-grade cost optimization solution for your Azure serverless architecture using Cosmos DB. We'll implement hot-cold data tiering to move rarely accessed records (older than 3 months) to a cheaper storage tier, while keeping your existing APIs, data availability, and SLAs intact.**

**✅ High-Level Solution Overview**

We’ll implement a **two-tiered architecture**:

| **Data Tier** | **Storage Type** | **Use Case** | **Latency** | **Cost** |
| --- | --- | --- | --- | --- |
| **Hot** | Azure Cosmos DB | Records < 3 months | Milliseconds | High |
| **Cold** | Azure Blob Storage (Cool or Archive Tier) | Records > 3 months | Seconds | Very Low |

We'll **extend your data access layer** to handle both tiers **transparently**, ensuring no changes to your external APIs.

**🔧 Azure Services Involved & Why**

| **Azure Service** | **Role** | **Justification** |
| --- | --- | --- |
| **Azure Cosmos DB** | Primary (hot) data store | For fast, read-heavy access to recent records |
| **Azure Blob Storage** | Secondary (cold) store | For cost-effective, large object archival |
| **Azure Functions** | Scheduled archival + On-demand cold read handler | Serverless, low-maintenance processing |
| **Azure Queue Storage / Event Grid** | Optional for decoupling archival logic | For reliable, async movement |
| **Azure App Configuration / Key Vault** | Store TTL, tier flags, etc. | Centralized configuration & secrets |
| **Application Insights / Log Analytics** | Monitoring + Alerts | Ensure SLA tracking & health |

**🏗️ Architectural Design**

plaintext

CopyEdit

+------------------------+

| External Clients |

+-----------+------------+

|

v

+-------------------+ +----------------------+

| Azure API / Logic | --> | Data Access Layer |

| Apps / Functions | | (Smart Router) |

+-------------------+ +----------------------+

| |

+---------------------+ +------------------+

| |

v v

+---------------------+ +----------------------------+

| Azure Cosmos DB | | Azure Blob Storage (Cold) |

| (Billing Records <3m)| | (Billing Records >3m) |

+---------------------+ +----------------------------+

* **Data Access Layer** will check the record's timestamp and redirect to Cosmos DB (hot) or Blob (cold).
* **Archival Function** will move older records daily to Blob Storage.
* Read requests for cold data will be handled via **Function proxy** or lazy cache.

**📋 Step-by-Step Migration Plan (Zero Downtime)**

**Phase 1 – Preparation (No production changes)**

1. **Update Data Access Layer (DAL)**:
   * Add logic to:
     + Check record age
     + Route reads to Cosmos DB (<3 months) or Blob Storage (>3 months)
   * DAL will remain backward-compatible with existing API.
2. **Create Blob Container**:
   * Use lifecycle-optimized **Cool** or **Archive** tier
   * Use structured folder naming: billing-records/yyyy/mm/
3. **Build Archival Azure Function**:
   * Daily job to:
     + Query Cosmos DB for records > 90 days
     + Serialize records to JSON
     + Write to Blob Storage
     + Mark original Cosmos record with “archived: true” (optional)
     + Optionally, delete after a delay (or soft delete with TTL)

**Phase 2 – Shadow Archival (Dry Run)**

1. **Run archival job in shadow mode**:
   * Move records to Blob, but do **not delete** from Cosmos DB
   * Track performance, logs, and correctness
2. **Verify cold reads** via testing:
   * Manually simulate a cold read
   * Monitor latency (goal: < 2–5 seconds)

**Phase 3 – Live Archival Rollout**

1. **Enable cold reads in DAL**
   * For records older than 3 months, Cosmos read fallback → Blob
   * Use async retry or caching if necessary
2. **Start Cosmos data cleanup**
   * Begin deleting archived records after 15–30 days of dual presence
   * Use TTL or manual deletion with Function

**Phase 4 – Monitoring and Optimization**

1. **Monitor access logs**:
   * Use Application Insights to track cold read frequency
2. **Auto-Tiering Option** (Optional):
   * Move older blobs from Cool → Archive tier automatically (after 6+ months)

**💰 Cost Comparison & Savings Estimate**

| **Metric** | **Cosmos DB** | **Azure Blob (Cool Tier)** |
| --- | --- | --- |
| Storage Cost (approx.) | ~$0.25/GB/month | ~$0.01–$0.02/GB/month |
| 2M records x 300 KB = 600 GB | $150/month | ~$6–$12/month |
| **Savings per month (est.)** | Up to **90–95%** on cold storage |  |

If 75% of data is cold (1.5M records), expected **monthly savings = ~$108–$135+**

**⚠️ Notes on Performance & SLAs**

* **Cold reads** from Blob may take 1–3 seconds.
* Use **Azure CDN** or **read cache** for common cold reads.
* Blob access can be optimized with **Index blobs**, e.g., records/{yyyy}/{mm}/{customerId}.json

**📁 Bonus: Blob Storage Format Example**

**File Path:**  
billing-records/2024/04/invoice\_456789.json

**Content:**

json

CopyEdit

{

"recordId": "456789",

"customerId": "CUST101",

"timestamp": "2024-04-15T10:22:00Z",

"amount": 3500.00,

"status": "PAID",

"details": { ... }

}

**✅ Benefits Recap**

* ✅ **No change to existing APIs**
* ✅ **Zero downtime migration**
* ✅ **90%+ cost savings on storage**
* ✅ **Easy to implement and maintain**
* ✅ **Scalable and observable**