azure-archive-solution

A ready-to-upload GitHub repo for the **Cost Optimization — Azure serverless + Cosmos DB** assignment. Drop the files below into a repository named `azure-archive-solution` and push to GitHub.

Repo structure

```
azure-archive-solution/
├─ README.md
├─ diagrams/
│  └─ architecture.mmd
├─ src/
│  ├─ archive_function/
│  │  ├─ function_app.py
│  │  └─ requirements.txt
│  └─ api/
│     └─ read_handler.py
├─ infra/
│  └─ azurecli-commands.sh
├─ docs/
│  ├─ failure_modes.md
│  └─ cost_strategy.md
├─ sample_data/
│  └─ sample-record.json
└─ LICENSE
```

Files (copy each file content into that path)

README.md

```
# Azure Archive Solution

This repository contains a working solution for the **Cost Optimization:
Managing Billing Records in Azure Serverless Architecture** assignment.

## Overview

- Move cold data (>90 days) from Cosmos DB into compressed blobs in Azure Blob
Storage.
```

```
- Keep a small stub in Cosmos DB so existing API contracts remain unchanged.
- Use an Azure Function to copy-verify-replace documents.
- API reads Cosmos; if stub found, fetches the full record from Blob Storage and
caches in Redis.

## What to upload

This repo contains the implementation for:
- Archival Azure Function (Python)
- API read handler (Python)
- Infra scripts (Azure CLI)
- Architecture diagram (Mermaid)
- Docs: failure modes and cost strategy

## Deploy (high-level)

1. Create resources (see `infra/azurecli-commands.sh`).
2. Configure environment variables / Managed Identity for the Function App.
3. Deploy the Function App and API.
4. Run the archival job manually once to bootstrap, then enable Timer trigger.

## Environment variables (example)

Set these in Function App configuration or use Managed Identity + Key Vault:

- `COSMOS_ENDPOINT`
- `COSMOS_KEY` or use AAD/Managed Identity
- `COSMOS_DB`
- `COSMOS_CONTAINER`
- `BLOB_CONNECTION_STRING` or use Managed Identity
- `BLOB_CONTAINER`
- `REDIS_CONNECTION_STRING` (optional)

## Files of interest

- `src/archive_function/function_app.py` — the archival worker
- `src/api/read_handler.py` — read-time handler with blob fallback
- `infra/azurecli-commands.sh` — quick infra bootstrap script

## Notes

- **Do not** commit secrets to the repo. Use GitHub Secrets / Azure Key Vault.
- This repo uses gzip JSON for archive objects. You can switch to Parquet for
analytics.

---
```

~~~
## diagrams/architecture.mmd

```mermaid
flowchart LR
  Client[Client / API] --> API[API]
  API --> Cosmos[Cosmos DB]
  Cosmos -->|archival job reads old docs| ArchFn[Azure Function (Archive)]
  ArchFn --> Blob[Azure Blob Storage]
  ArchFn --> Cosmos[Update doc -> stub]
  API -->|if stub| BlobFetch[Blob fetch + Cache]
  BlobFetch --> Blob
  BlobFetch --> Redis[Redis Cache]
~~~

src/archive_function/function_app.py

```python
import os
import json
import gzip
import hashlib
import logging
from datetime import datetime, timedelta, timezone

import azure.functions as func
from azure.storage.blob import BlobServiceClient
from azure.cosmos import CosmosClient, PartitionKey

# Environment configuration (use Function App settings or Key Vault)
COSMOS_ENDPOINT = os.environ.get("COSMOS_ENDPOINT")
COSMOS_KEY = os.environ.get("COSMOS_KEY")
COSMOS_DB = os.environ.get("COSMOS_DB", "billing")
COSMOS_CONTAINER = os.environ.get("COSMOS_CONTAINER", "records")
BLOB_CONN_STR = os.environ.get("BLOB_CONNECTION_STRING")
BLOB_CONTAINER = os.environ.get("BLOB_CONTAINER", "billing-archive")

# Archival threshold in days
ARCHIVE_DAYS = int(os.environ.get("ARCHIVE_DAYS", "90"))
BATCH_SIZE = int(os.environ.get("ARCHIVE_BATCH_SIZE", "100"))

# Initialize clients
cosmos_client = CosmosClient(COSMOS_ENDPOINT, COSMOS_KEY)
db_client = cosmos_client.get_database_client(COSMOS_DB)
container = db_client.get_container_client(COSMOS_CONTAINER)

blob_service = BlobServiceClient.from_connection_string(BLOB_CONN_STR)
archive_container = blob_service.get_container_client(BLOB_CONTAINER)
```

```
def md5_bytes(b: bytes) -> str:
    return hashlib.md5(b).hexdigest()


def make_blob_path(doc: dict) -> str:
    created = doc.get("createdAt", datetime.now(timezone.utc).isoformat())
    # Expect ISO format, fallback safe-parsing simple slicing
    year = created[:4]
    month = created[5:7] if len(created) >= 7 else "01"
    return f"year={year}/month={month}/{doc['id']}.json.gz"


def upload_blob_if_missing(path: str, data: bytes):
    blob = archive_container.get_blob_client(path)
    if not blob.exists():
        blob.upload_blob(data, overwrite=False)
    return blob


def archive_document(doc: dict):
    # 1) serialize and compress
    payload = json.dumps(doc, default=str).encode("utf-8")
    gz = gzip.compress(payload)
    blob_path = make_blob_path(doc)

    # 2) upload
    blob = upload_blob_if_missing(blob_path, gz)

    # 3) verify checksum by re-downloading (simple verification step)
    downloaded = blob.download_blob().readall()
    if md5_bytes(downloaded) != md5_bytes(gz):
        raise RuntimeError("Checksum mismatch after upload")

    # 4) replace doc with stub (preserve id and partition key)
    stub = {
        "id": doc["id"],
        "partitionKey": doc.get("partitionKey", doc.get("pk", "default")),
        "archived": True,
        "archiveLocation": blob_path,
        "archiveSize": len(gz),
        "archivedAt": datetime.now(timezone.utc).isoformat(),
        # keep a subset of searchable fields to avoid rehydrating for simple
queries
        "searchable": {
            "invoiceNo": doc.get("invoiceNo"),
            "amount": doc.get("amount"),
            "createdAt": doc.get("createdAt")
        }
    }

    # Upsert stub (this will overwrite the existing doc)
    container.upsert_item(stub)


def main(mytimer: func.TimerRequest) -> None:
    logging.info("Archive function started")
    cutoff = (datetime.now(timezone.utc) - timedelta(days=ARCHIVE_DAYS)).isoformat()

    query = (
        "SELECT * FROM c WHERE c.createdAt < @cutoff AND (NOT IS_DEFINED(c.archived) OR c.archived = false)"
    )
    parameters = [{"name": "@cutoff", "value": cutoff}]

    # Query with pagination and batching
    items_iterable = container.query_items(
        query=query,
        parameters=parameters,
        enable_cross_partition_query=True,
    )

    count = 0
    for item in items_iterable:
        try:
            archive_document(item)
            count += 1
            if count >= BATCH_SIZE:
                logging.info(f"Processed batch of {count}, stopping to avoid RU spikes")
                break
        except Exception as e:
            logging.exception(f"Failed to archive {item.get('id')}: {e}")
            # leave the item for retry

    logging.info(f"Archive run complete. Archived {count} documents.")
```

src/archive_function/requirements.txt

```
azure-functions
azure-storage-blob
azure-cosmos
```

src/api/read_handler.py

```
import os
import json
import gzip

from azure.cosmos import CosmosClient
from azure.storage.blob import BlobServiceClient

# Optional: Redis caching (not implemented here) — placeholder for integration

COSMOS_ENDPOINT = os.environ.get("COSMOS_ENDPOINT")
COSMOS_KEY = os.environ.get("COSMOS_KEY")
COSMOS_DB = os.environ.get("COSMOS_DB", "billing")
COSMOS_CONTAINER = os.environ.get("COSMOS_CONTAINER", "records")
BLOB_CONN_STR = os.environ.get("BLOB_CONNECTION_STRING")
BLOB_CONTAINER = os.environ.get("BLOB_CONTAINER", "billing-archive")

cosmos_client = CosmosClient(COSMOS_ENDPOINT, COSMOS_KEY)
container =
cosmos_client.get_database_client(COSMOS_DB).get_container_client(COSMOS_CONTAINER)
blob_service = BlobServiceClient.from_connection_string(BLOB_CONN_STR)
archive_container = blob_service.get_container_client(BLOB_CONTAINER)


def get_record(id: str, partition_key: str) -> dict:
    doc = container.read_item(item=id, partition_key=partition_key)
    if doc.get("archived"):
        # Fetch blob
        blob_path = doc["archiveLocation"]
        blob = archive_container.get_blob_client(blob_path)
        data = blob.download_blob().readall()
        payload = gzip.decompress(data)
        record = json.loads(payload)
        return record
    return doc


# Example usage (for local testing)
```

```
if __name__ == "__main__":
    import sys
    id = sys.argv[1]
    pk = sys.argv[2]
    print(get_record(id, pk))
```

infra/azurecli-commands.sh

```
#!/bin/bash
set -e

RG="rg-billing-$(date +%s)"
LOCATION="eastus"
STORAGE_ACCOUNT="sbillingarchive$RANDOM"
COSMOS_ACCOUNT="cosmosbilling$RANDOM"
FUNCTION_APP="fn-archive-$RANDOM"

# 1) Resource group
az group create -n $RG -l $LOCATION

# 2) Storage account for Function & Blobs
az storage account create -n $STORAGE_ACCOUNT -g $RG -l $LOCATION --sku Standard_LRS

# Create blob container
az storage container create --name billing-archive --account-name $STORAGE_ACCOUNT

# 3) Cosmos DB account (serverless)
az cosmosdb create -n $COSMOS_ACCOUNT -g $RG --capabilities EnableServerless

# 4) Function App (Consumption plan) - requires a storage account
az functionapp create --resource-group $RG --consumption-plan-location $LOCATION \
  --name $FUNCTION_APP --storage-account $STORAGE_ACCOUNT --runtime python --runtime-version 3.11

# Print outputs — remember to set these as config for your function
az cosmosdb keys list -n $COSMOS_ACCOUNT -g $RG --type keys

echo "Resource group: $RG"
echo "Storage account: $STORAGE_ACCOUNT"
echo "Cosmos account: $COSMOS_ACCOUNT"
echo "Function App: $FUNCTION_APP"
```

docs/failure_modes.md

```
# Failure Modes & Mitigations

1. Partial upload / corrupted blob
   - Verify checksum after upload. If mismatch, abort and retry.

2. Function crashes mid-run
   - Make steps idempotent. Use blob existence and upsert semantics to resume.

3. Read while migrating
   - Use copy-then-replace. Reads will see original doc or a stub that points to
a valid blob.

4. RU throttling on Cosmos
   - Batch archival, add sleeps, and use continuation tokens. Monitor RU and
back off.

5. Large documents rehydration latency
   - Add Redis cache and/or return lightweight metadata while streaming the
payload.
```

docs/cost_strategy.md

```
# Cost Optimization Strategy

- Move cold objects (>90 days) to Blob Storage (Cool / Archive tiers) to save on
storage costs.
- Keep only minimal searchable metadata in Cosmos to reduce RU consumption.
- Use serverless or autoscale Cosmos to avoid fixed RU costs.
- Batch archive operations to reduce transaction overhead.
- Consider Parquet for analytics and smaller read IO when doing large-scale
analytics.
```

sample_data/sample-record.json

```
{
  "id": "sample-123",
  "partiti
```