Data in Motion — Intelligent Multi-Cloud Data Management

Project Presentation (Hackathon Prototype)

Author: Team GigaBITS
Date: 2025-11-08
Version: 0.1

1) Problem Understanding

Organizations operate across on-prem, private cloud, and public clouds. They need to:

- Optimize where data lives (hot/warm/cold) to balance performance and cost
- Migrate data across clouds with minimal disruption
- Stream, analyze, and act on real-time access patterns
- Predict usage trends and proactively move or reclassify data
- Maintain availability and consistency during failures
- Offer operators a unified view of data placement, costs, and migration activity

This prototype showcases an event-driven data management platform that uses access patterns and policies to automate placement decisions while ensuring integrity and availability.

2) Objectives

- Optimize Data Placement based on access frequency, latency SLOs, cost, and predicted trends
- Enable Multi-Cloud Data Migration with integrity checks and minimal downtime
- Integrate Real-Time Streaming (Kafka) for events and orchestration
- Provide insights and automated recommendations based on recent access and policy constraints
- Ensure Consistency & Availability across replicas
- Ship a Unified Interface (REST + optional CLI) for observability and control
- Containerized deployment for straightforward local and cloud simulation

3) Architecture Overview

Core Components:

- Python FastAPI Service (Policy + Analytics)
  - Ingests access events
  - Aggregates usage and applies rules for tier recommendations
  - Enqueues migration jobs and tracks status
  - Exposes REST APIs for datasets, policies, recommendations, and jobs
- Go Migration Mover
  - Consumes migration jobs (Kafka)
  - Executes copy → verify → switch → complete cycle
  - Publishes migration status updates
  - Exposes REST APIs for job status and health
- SQLite Metadata Store
  - Tracks datasets, jobs, and aggregates
- Kafka-compatible Broker (Redpanda for local dev)
  - Topics: access_events, migration_jobs, migration_status, recommendations
- Storage Abstraction
  - Local FS (file://) for on-prem simulation
  - Optional GCS (gs://) and MinIO (s3://) integrations
- Observability
  - Health endpoints, structured logs, and metrics

High-Level Flow:

1. Clients send access events → Python API → Kafka (access_events)
2. Python aggregates usage and applies rules → emits migration_jobs
3. Go mover consumes jobs → runs migration lifecycle → emits migration_status
4. Python updates DB → recommendations and dashboards reflect current state

---

4) Architecture Diagram (ASCII)

```
           +----------------------------+
           |        Clients / Apps      |
           |  (producers of access)     |
           +--------------+-------------+
                          |
                          v
                   +------+------+
                   |   Kafka     |  access_events
                   | (Redpanda)  |  migration_jobs
                   +---+-----+---+  migration_status
                       |     |
+----------------------+     +--------------------+
|                                              |
v                                              v
```

```
+------+------+                              +--------+--------+
| Python API   |                              |   Go Mover     |
| (FastAPI)    |                              | (Orchestrator) |
| - Policy     | <-- recommendations          | - Copy/Verify  |
| - REST / DB  | --> migration_jobs           | - Status Emit  |
+------+-------+                              +--------+-------+
       |                                                |
       v                                                |
+------+-------------------------+                      |
|           SQLite DB           |                      |
| datasets, jobs, aggregates... |<---------------------+
+-------------------------------+    migration_status updates
```

---

5) Components & Responsibilities

- Python FastAPI:
  - Endpoints: /datasets, /access-events, /recommendations, /plan-migration, /jobs, /train
  - Produces: access_events, migration_jobs
  - Consumes: migration_status (updates SQLite)
  - Tiering logic: recency-based heuristic for recommendations
- Go Mover:
  - Consumes: migration_jobs
  - Produces: migration_status
  - Lifecycles: queued → copying → verifying → switching → completed/failed
  - Persistent state: jobs mirrored in SQLite
- Kafka (Redpanda):
  - Simple single-node broker for dev
- SQLite:
  - Lightweight metadata store for rapid iteration

---

6) Data Model (SQLite, Minimal Viable)

Tables (current):

- datasets
  - id, name, path_uri, current_tier(hot|warm|cold), latency_slo_ms, size_bytes, owner, last_access_ts, created_at, updated_at
- migration_jobs
  - job_id (UUID), job_key (unique), dataset_id, source_uri, dest_uri, dest_storage_class
  - status (queued|copying|verifying|switching|completed|failed|enqueue_failed)
  - error, submitted_at, updated_at

Roadmap tables (extensible in Go storage.go + future ORM):

- replicas, policies, access_aggregates, recommendations, models

---

7) Streaming Topics

- access_events: dataset access telemetry (read/write, size, client latency, timestamp)
- migration_jobs: commands for Go mover (source/dest, class)
- migration_status: lifecycle updates for jobs (copying, verifying, switching, completed, failed)
- recommendations (optional): human-readable recommendations from Python
- metrics (optional): counters/latency from services

---

8) Placement Strategy

- Signals:
  - read_count, write_count, bytes_read, bytes_written
  - last_access_age_s, size_bytes
  - policy constraints (latency SLOs, cost targets)
- Decision rules:
  - Recency threshold
    - <=60s → hot
    - <=10m → warm
    - > 10m → cold
  - Policy checks may adjust the final tier
- Metadata:
  - Decisions and reasons are stored in SQLite
- Operation:
  - Recommendations can be computed on demand via the API and the Web UI

---

9) Migration Lifecycle

- queued
- copying: stream source → dest; compute checksum
- verifying: compare checksum/ETag; ensure object count
- switching: atomically update primary replica and dataset pointer
- completed: source retired if policy allows
- failed: supports retry; idempotent job_key prevents duplication

Consistency & Idempotency:

- Source remains primary until verification passes
- Job keys provide dedupe; retries safe
- Transactions on DB updates

---

10) Security & Compliance (MVP → Production)

- In-flight: TLS (future); local dev simple HTTP
- At-rest: GCS/S3 native encryption; optional local envelope encryption
- Access control: Add API keys or OAuth proxy in front of FastAPI/Go services
- Auditing: Persist decisions (reason, model_version) in recommendations table
- Secrets: Mount cloud credentials as read-only; never commit to repo

---

11) Cloud Simulation (GCP Free Tier)

- Use GCP Cloud Storage (Always Free) for "warm/cold" simulation
- Local filesystem for "hot/on-prem"
- Environment:
  - GOOGLE_APPLICATION_CREDENTIALS=/secrets/gcp-sa.json
  - Optional bucket: gs:// in eligible region
- For local multi-cloud:
  - MinIO (S3-compatible) as private cloud simulation

---

12) Demo Plan (Local, Docker Compose)

1. Start stack:

   - docker compose up —build -d
   - Health: curl http://localhost:8080/health and http://localhost:8090/health

2. Create dataset:

   - POST /datasets with path_uri=file:///shared_storage/ds1

3. Generate access:

   - POST /access-events (multiple times with reads/writes)

4. Get recommendations:

   - GET /recommendations?dataset_id=1

5. Approve migration:

    - POST /plan-migration (dest=file:///shared_storage/migrated/ds1)
    - Observe Go mover transitions via GET /jobs and logs

6. Verify:

    - Check dataset's current_tier and job completion status

---

13) Performance & Insights (Prototype-Level)

- SQLite with WAL is sufficient for serialized metadata ops in dev
- Kafka-based eventing enables decoupled, asynchronous orchestration
- Go mover can scale out horizontally with multiple consumers and partitioned topics
- Tiering logic is heuristic and can evolve over time

---

14) Scalability & Roadmap

Near-term:

- Finish full DB wiring for all endpoints; persist recommendations
- Implement real Kafka consumer in Go (segmentio/kafka-go or sarama)
- File-based migration: chunked copy + checksum verification

Mid-term:

- Switch metadata to Postgres/Cloud SQL
- Add Prometheus metrics and Grafana dashboards
- Add policy-as-code and cost modeling engine
- Kubernetes deployment (HPA for mover replicas)

Long-term:

- Storage drivers for GCS/S3 with optimized transfers (multi-part, retries)
- Advanced feature sets (seasonality, client clusters) and richer rule-based scoring
- Multi-region replication with conflict resolution strategy

---

15) Risks & Mitigations

- SQLite contention → WAL + move to Postgres
- Kafka outages → buffered retries; NullKafka fallback for dev
- Partial migrations → idempotent job processing; resumable copy design
- Credential security → secret mounts; least-privilege service accounts
- Model drift → scheduled evaluation & retraining; fallback to heuristic

---

16) API Summary (Draft)

Python API:

- GET /health
- POST /datasets; GET /datasets; GET /datasets/{id}; PATCH /datasets/{id}
- POST /access-events
- GET /recommendations[?dataset_id=...]
- POST /plan-migration
- POST /train
- GET /jobs; GET /jobs/{job_id}

Go API:

- GET /health
- GET /jobs; GET /jobs/{id}
- POST /jobs/retry/{id}

---

17) Quickstart (Local)

- docker compose -f deploy/docker/docker-compose.yml up —build -d
- Create dataset:
  - curl -X POST http://localhost:8080/datasets -H 'Content-Type: application/json' -d '{"name":"sample","path_uri":"file:///shared_storage/sample","size_bytes":1048576}'
- Access event:
  - curl -X POST http://localhost:8080/access-events -H 'Content-Type: application/json' -d '{"dataset_id":1,"op":"read","size_bytes":4096,"client_lat_ms":12.3}'
- Recommendation:
  - curl http://localhost:8080/recommendations?dataset_id=1
- Plan migration:
  - curl -X POST http://localhost:8080/plan-migration -H 'Content-Type: application/json' -d '{"dataset_id":1,"target_location":"file:///shared_storage/migrated/sample","storage_class":"standard"}'
- Observe jobs:
  - curl http://localhost:8080/jobs
  - curl http://localhost:8090/jobs

---

18) Closing

This prototype demonstrates an intelligent, event-driven foundation for multi-cloud data management:

- Streaming-first architecture with decoupled services
- Data-driven, policy-aware placement decisions

- Safe, idempotent migration lifecycle
- Extensible schema and storage drivers
- Containerized for rapid local and cloud simulations

We’re ready to extend toward real cloud storage operations (GCS/S3) and robust observability to meet production needs.

Thank you!