Hackathon Plan — Data in Motion: Intelligent Cloud Storage

Goal: Build a winning prototype that dynamically analyzes, tiers, and moves data across hybrid/multi-cloud environments while processing real-time streams and providing predictive, automated data placement.

---

1) Executive summary

Deliver a compact, demo-ready system that:
- Accepts simulated streaming data (Kafka) and ingests it into a multi-storage environment (local filesystem / SQLite as on-premise, AWS S3 mock or real S3 as public cloud, and a local MinIO or Azure Blob emulator as private cloud).
- Implements a rules + ML hybrid placement engine that decides *hot/warm/cold* placement based on access frequency, latency requirement, storage cost, and forecasts.
- Provides a migration/synchronization prototype to move objects between storages with minimal downtime and secure transfer.
- Visualizes data distribution, streaming throughput, migrations, predicted moves, and cost/latency KPIs in a unified dashboard (React or simple Flask web UI).

Deliverables for judges: working demo (web UI + CLI), short slide deck (10 slides), short video walkthrough (2–3 minutes), README with how to run locally (Docker compose), and test data generator.

---

2) MVP (must-have) vs Stretch (bonus)

MVP (complete during hackathon) - Stream ingestion using Kafka (producer simulating events) and a simple consumer that writes files/objects to the chosen storage. - Three storage tiers simulated: On-premise (local FS or SQLite), Private cloud (MinIO), Public cloud (S3 or S3 mock).
- Rule-based placement engine (hot/warm/cold rules): move files after thresholds.
- A lightweight ML predictor (scikit-learn) that forecasts access count for each object and recommends moves.
- Migration/sync prototype: copy & verify object with checksums; show progress in UI.
- Dashboard showing objects, tier, last access, predicted access, migration activity, cost and latency estimates.
- Basic resilience: retry on network failure, idempotent moves, and a simple conflict resolution (last-write-wins or versioned objects).

Stretch / Bonus - Use real cloud (AWS S3) integration + IAM and server-side encryption.
- Containerized deploy (Docker Compose or Kubernetes) and Helm chart.
- Adaptive access control policies per storage location.
- Advanced ML (LSTM or Prophet) for better temporal forecasting.
- Automated policy triggers/alerts (Slack/Email) for cost/latency anomalies.

3) Suggested architecture (component map)

1. **Data Producers** — Simulated sensors or apps producing events to Kafka topics (JSON with object id, payload, metadata).
2. **Streaming Layer** — Apache Kafka cluster (single node for hackathon) with topics: `ingest`, `access_events`, `migration_commands`.
3. **Ingestion/Consumer Service** — Python consumer (aio or threaded) that receives events and persists objects to the storage layer according to current placement decisions.
4. **Storage Adapters** — Small adapter layer with unified API for operations: `put(object)`, `get(object)`, `delete(object)`, `list(prefix)`, `metadata(object)`. Implement adapters for: local FS (on-prem), MinIO (private), S3 or s3mock (public).
5. **Placement Engine** — Hybrid engine with two parts:
6. *Rules Engine* (immediate decisions): simple rules (e.g., if object size < X and access frequency > Y → hot).
7. *Predictive Engine* (ML): periodically runs to forecast next-N access counts and suggests pre-moves.
8. **Orchestrator / Migration Service** — Executes migrations (copy + checksum + delete) and writes `migration_commands` events to Kafka so the dashboard and logs can track progress.
9. **Metadata DB** — Single source of truth (MongoDB / SQLite for hackathon). Stores object metadata: id, current_tier, creation_ts, last_access_ts, access_count, version, checksum, predicted_access, cost_estimate.
10. **Consistency & Conflict Manager** — Lightweight store that logs operations; uses optimistic versioning (version numbers) to detect conflicts, and resolves with a policy (e.g., latest version or mergeable metadata).
11. **Dashboard UI** — React (or Flask + simple templates) for visualizations: topology, list of objects by tier, migration activity timeline, predicted moves, cost & latency heatmaps.
12. **Auth & Security** — Basic API key or JWT for services; encryption at rest option and TLS for data movement if integrating cloud services.

---

4) Tech stack (recommended)

- **Backend / Services:** Python 3.10+ (FastAPI or Flask). Python has excellent Kafka clients, scikit-learn and many cloud SDKs.
- **Streaming:** Apache Kafka (single-node) or Kafka Docker image; alternatively MQTT if simpler.
- **Storage adapters:** MinIO (Docker), local filesystem, AWS S3 SDK (boto3) — can be mocked.
- **Metadata DB:** SQLite (lightweight) or MongoDB (document model) — SQLite is easier for hackathon.
- **ML:** scikit-learn (RandomForestRegressor or simple linear/Poisson regression) for access forecasting.
- **Dashboard:** React (Vite) or Flask + Bootstrap for speed.
- **Containerization:** Docker Compose with services: kafka, zookeeper, minio, backend, db, frontend.
- **CI / Testing:** simple pytest + scripts.

---

5) Data model (example schema)

ObjectMetadata (SQLite table) - `object_id` TEXT PRIMARY KEY - `path` TEXT (storage key) - `current_tier` TEXT (`onprem` | `private` | `public`) - `size_bytes` INTEGER - `created_at` TIMESTAMP - `last_accessed_at` TIMESTAMP - `access_count` INTEGER - `version` INTEGER - `checksum` TEXT - `predicted_next_24h` FLOAT - `cost_per_gb_month` FLOAT - `latency_ms_est` FLOAT

AccessEvent (Kafka message) - `{ object_id, user, timestamp, op: 'read'|'write'|'delete', latency_ms }`

---

6) Placement logic & ML design

Rule engine (fast path) - If `access_count` in last 1 hour > HOT_THRESHOLD => place on `onprem` or `private` depending on latency requirement.
- If `last_accessed_at` older than 30 days => `cold` → move to `public` (cheapest).
- If object flagged `low_latency_required` => prefer `onprem` / `private`.

Predictive engine (periodic) - Input features per object: past N access counts (time series), object_size, creation_age, user_flags, day_of_week, hour_of_day.
- Label: next-window access count (e.g., next 24 hours).
- Model: RandomForestRegressor or GradientBoosting for structured features; or a lightweight AR model.
- Output: predicted accesses → convert to expected cost vs latency tradeoff and generate recommendation (`move_to_tier`, `keep`, `replicate`).

Decision combining - Score each tier for an object: `score = w_freq * predicted_freq_norm - w_cost * cost_norm - w_latency * latency_norm`
- Choose tier with highest score subject to constraints (minimum replication, legal constraints, etc.).
- Weights can be tuned and exposed via UI sliders for demo.

---

7) Migration & synchronization strategy

Prototype flow 1. Orchestrator receives `move` command from placement engine.
2. It calls target storage adapter `put(object)` and streams bytes from source adapter to target adapter (avoid writing to disk if possible).
3. After target write returns, compute checksums on both sides (or preserve checksum) and verify.
4. Update metadata DB atomically: set `current_tier`, increment `version`.
5. Delete source copy if policy says so (for move vs replicate).
6. Broadcast migration completion event to Kafka.

Resilience - Use idempotent operations (copy with object version + checksum) to retry safely.
- Keep migration state in DB so migration can resume after crash.
- For partial failures, mark object as `in_migration` and expose to UI for manual resolution.

Consistency options - For most file/object workloads eventual consistency is acceptable.
- For stronger requirements: implement optimistic locking with version numbers and conflict resolution rules (e.g., keep latest write, or merge metadata).

---

8) Real-time streaming integration

- **Kafka topics**: `ingest` (payload -> new object), `access_events` (reads/writes), `migration_commands`, `alerts`.
- **Use case flow**: Producers write JSON events to `ingest`, the consumer persists object and metadata, and the access event generator simulates reads (to change hotness).
- **Reactive behavior**: Consumers can react to `access_events` to immediately duplicate/move objects (e.g., replicate a hot object to private cloud when access spiking).

---

9) Dashboard / UX

Pages / Widgets - Overview: total objects, distribution by tier (pie), total estimated monthly cost, average latency.
- Object list: sortable table showing id, tier, size, last access, predicted access, actions (force move/replicate).
- Streaming monitor: recent events per second, topic lag, producer rates.
- Migration timeline: recent migrations with status.
- Policy & weights: sliders to tweak w_freq, w_cost, w_latency and re-run recommendations.

Demo actions - A button to simulate a traffic spike for object X (producer sends many `access_events`) so judges can see the system react and move object to a lower-latency tier.

---

10) Hackathon execution plan (48 hours timeline — adapt to your slot)

Pre-hack (before event) - Prep environment templates: Docker Compose that spins Kafka, Zookeeper, MinIO, SQLite, backend.
- Skeleton repo with README, basic adapters, data generator script.

Day 1 — Morning (3–4 hours) - Set up repo, Docker Compose, basic Kafka and MinIO.
- Implement Storage adapters and Metadata DB.
- Implement a simple Kafka producer (data generator) and consumer that writes incoming payloads to `public` (S3/MinIO) bucket.

Day 1 — Afternoon (3–4 hours) - Implement rule-based placement engine and orchestrator for move commands.
- Implement migration logic (copy + checksum + update DB).
- Build minimal frontend to show object list and migration activity.

Day 2 — Morning (3–4 hours) - Implement access event simulator & hooks so reads update access_count in metadata.
- Implement ML predictor (quick model training using simulated history) and recommendation endpoint.
- Hook predictor results to placement engine to trigger preemptive moves.

Day 2 — Afternoon (3–4 hours) - Polish UI, add graphs, implement demo scripts (traffic spike to show automatic move).
- Add resilience features: migration state store, retry logic.
- Prepare slides and 2–3 minute demo video.
- Test end-to-end and write README.

Final hour - Rehearse demo steps, keep one “showstopper” scenario (e.g., sudden spike, show migration & cost tradeoff) and one fallback (manual move) in case of bugs.

---

11) Demo script (short & crisp)

1. Start services with `docker compose up --build` (1 minute).
2. Open dashboard: show object distribution and baseline costs.
3. Trigger a simulated spike for object `X` (button or script).
4. Show Kafka traffic, show placement engine decision, watch orchestrator begin migration and object appear in lower-latency tier.
5. Show predicted access graph and explain ML recommendation.
6. Show resilience: kill orchestrator container then restart — migration resumes from state.
7. Close with summary of cost/latency tradeoffs and next steps.

---

12) Scoring alignment & pitch pointers

Innovation & NetApp domain: Emphasize hybrid placement engine, streaming integration, and real-time decisioning — highlight how it reduces operating cost and increases performance.
Technical depth: Explain architecture decisions (why Kafka, why MinIO, why hybrid ML+rules), scaling points, and bottlenecks.
Scalability & efficiency: Explain how metadata indexes, partitioning of topics, and parallel migrators will scale.
UX: Keep dashboard minimal and interactive; show the cause→effect flow.
Presentation: 10 slides max: problem, architecture, demo flow, technical highlights, results (simulated), roadmap.

---

13) Quick implementation snippets (conceptual)

Storage adapter interface (Python)

```python
class StorageAdapter:
    def put(self, key: str, stream) -> None: ...
    def get(self, key: str) -> bytes: ...
    def delete(self, key: str) -> None: ...
    def metadata(self, key: str) -> dict: ...
```

Simplified placement scoring

```
score = w_freq * normalize(predicted_freq) - w_cost * normalize(cost_per_gb) -
w_latency * normalize(latency_ms)
```

Kafka consumer pattern

```python
from confluent_kafka import Consumer
# subscribe to ingest; on message -> persist object via chosen adapter
```

---

14) Tests & evaluation

- Unit tests for adapters (mock storages).
- Integration tests: simulate ingestion and spikes, assert object migrated within threshold.
- Performance basics: measure migration throughput (MB/s) and Kafka consumer lag.
- Demonstrate cost calculation: sum(size_gb * cost_per_gb_month / 30 * days_held) for objects moved vs not.

---

15) Risks & mitigations

- **Time constraints:** Focus on rule engine + UI first, then add ML as a recommendation rather than enforcement.
- **Cloud access/quotas:** Use MinIO or local mock S3 to avoid cloud limits.
- **Kafka complexity:** Use a single-node Kafka (Docker) or even replace with simple in-process queue if needed.

---

16) Extra polish ideas (if time permits)

- Add cost/latency slider presets and a scenario comparison tool.
- Add a small policy DSL (YAML) for easy judge customization.
- Show a short cost saved calculation comparing naive always-public storage vs your smart tiering.

---

17) What to build first (priority checklist)

1. Docker Compose skeleton with Kafka + MinIO + metadata DB.
2. Storage adapters and simple consumer to persist objects.
3. Metadata DB & basic dashboard object listing.
4. Rule-based placement & orchestrator (move flow).
5. Demo script to show automated move on spike.
6. ML predictor as a recommendation service.
7. Resilience and polish.

---

18) Deliverables & repo layout suggestion

```
/backend
  /adapters
  /services
  main.py
/frontend
  react-app
/docker-compose.yml
/scripts
  simulate_spike.py
/docs
  presentation.pdf
  demo_video.mp4
README.md
```

Final notes (presentation talking points)

- Start with the user pain: data scattered across clouds leads to cost and latency issues.
- Demonstrate a concrete before/after scenario where a single object becomes hot and your system moves it to low-latency tier automatically.
- Emphasize modularity — rules + ML, pluggable storage adapters, and streaming based reactive behavior.
- Finish with roadmap: multi-region replication, stronger consistency options, policy DSL, and enterprise security.

If you want, I can generate: - A minimal Docker Compose + starter code scaffold for the MVP (Python + Kafka + MinIO + SQLite).
- The slide deck (10 slides) with speaker notes. - A short demo script file and README with exact commands.

Tell me which one you'd like me to produce first and I will create it.