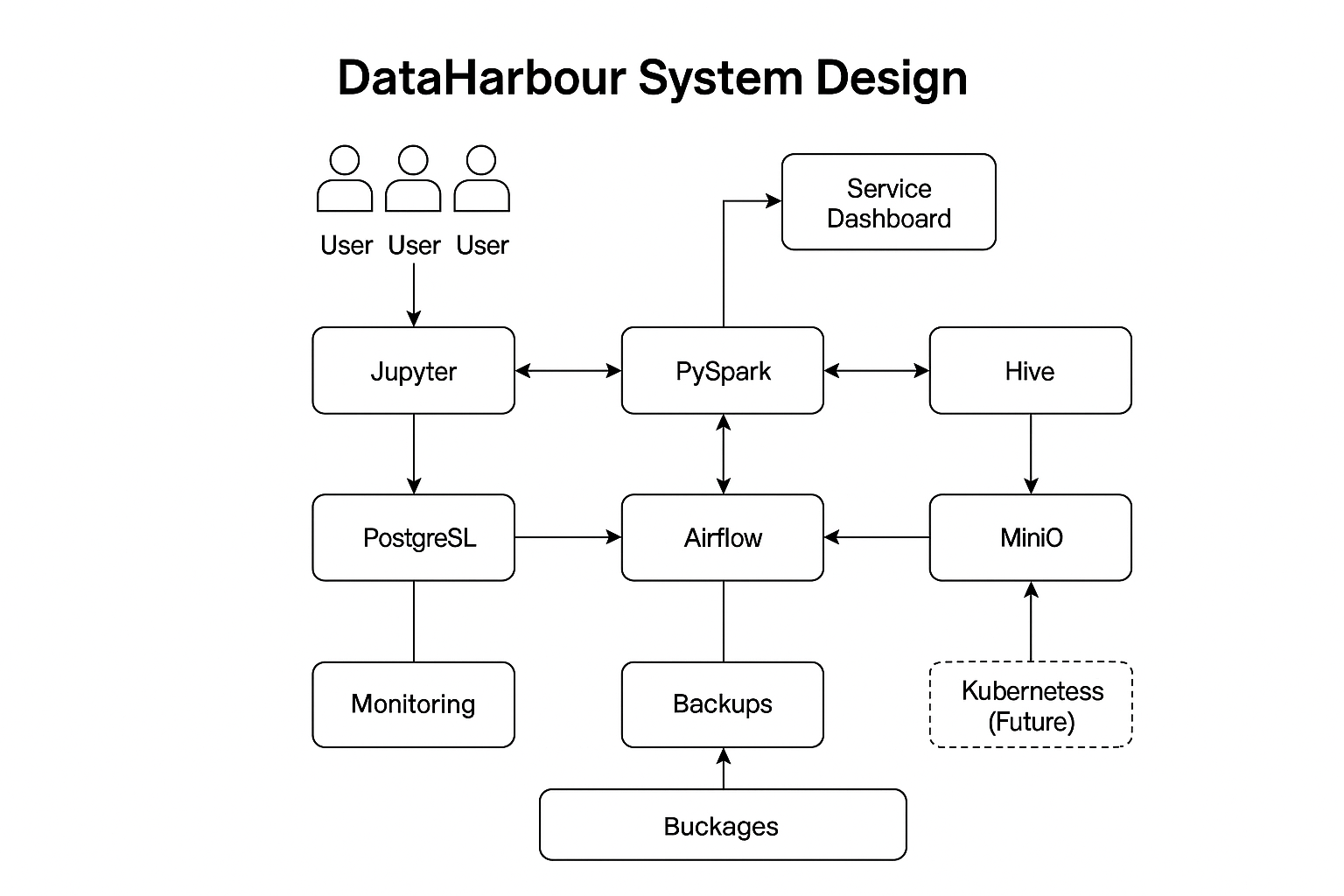
1. **Project Introduction – DataHarbour**

DataHarbour is a comprehensive, Docker-based data engineering platform designed for students, educators, and small businesses to process, analyze, and manage data efficiently. It integrates key tools like PySpark, Jupyter, Airflow, MinIO, PostgreSQL, and Hive into a unified system for data workflows. The platform supports real-time orchestration, interactive development, and S3-compatible storage, all accessible through a user-friendly dashboard. With built-in monitoring, backup, and scalability features, DataHarbour enables on-premises deployment for secure and flexible data infrastructure.



1. **🔹 Functional Requirements**

These describe what the system should do.

1. User Authentication & Authorization
   * Admins and users (students, engineers) should be able to log in securely.
   * Role-based access control (RBAC) for services like Airflow, Jupyter, MinIO.
2. Service Deployment via Docker
   * Launch and manage containers for Spark, Airflow, MinIO, PostgreSQL, Jupyter, and Dashboard.
3. Data Processing and Analysis
   * Enable users to run Spark jobs through Jupyter notebooks.
   * Use PySpark to process datasets stored in MinIO.
   * Store and retrieve Delta Lake tables.
4. Job Scheduling
   * Admins can define DAGs in Apache Airflow for ETL or ML workflows.
   * Schedule and monitor job execution and logs.
5. Storage Management
   * Upload/download data via MinIO (S3-compatible).
   * Persistent volume support for Jupyter notebooks, Spark outputs, and PostgreSQL metadata.
6. Service Dashboard
   * Visual monitoring for CPU/memory usage, service status, and logs.
   * Ability to start/stop services from the web interface.
7. Monitoring & Logging
   * Centralized logging with Loki + Filebeat.
   * Metrics collection via Prometheus + Grafana dashboards.
8. Backup and Restore
   * Periodic backup of PostgreSQL and MinIO using cron jobs.
   * Restore support for disaster recovery.
9. **🔸 Non-Functional Requirements**

These define how the system should behave.

1. Scalability
   * Support scaling services for larger workloads or concurrent users (Kubernetes-ready in future).
2. Performance
   * Efficient execution of Spark jobs and low-latency service access.
   * Dashboard and logs must refresh in near real-time.
3. Availability
   * 99.9% uptime for all critical services like Jupyter, Airflow, MinIO, PostgreSQL.
   * Self-healing via Docker restart policies.
4. Security
   * HTTPS and BasicAuth with NGINX reverse proxy.
   * Secure storage of credentials.
   * Limited container access (Principle of Least Privilege).
5. Portability
   * Entire environment must be reproducible via docker-compose up.
6. Usability
   * Easy onboarding for students and small teams with minimal technical setup.
   * UI/UX consistency across services.
7. Maintainability
   * Modular architecture to add/remove services easily.
   * Central config file for port bindings and credentials.
8. Data Integrity
   * ACID support via Delta Lake and PostgreSQL.
   * Consistent data across pipeline executions
9. ***📊* Capacity Estimation – DataHarbour**

|  |  |  |  |
| --- | --- | --- | --- |
| Component | Estimation Basis | Expected Load | Notes |
| Users | Concurrent users (students, engineers, admins) | 10–50 active users | Can scale to 100+ with Kubernetes. |
| Notebooks | Avg. notebook size + data | ~50MB per user | Stored in mounted volumes; persistent across sessions. |
| MinIO Storage | Uploaded datasets, processed files, logs | ~500GB – 1TB for small teams | Easily extendable via MinIO's scaling. |
| Airflow DAGs | ETL/ML job orchestration | 20–50 DAGs | Avg. 1–2 DAGs per user for recurring pipelines. |
| Spark Cluster | PySpark job processing | 2–3 worker containers with 2–4 cores, 8GB RAM each | Suitable for medium workloads; can scale horizontally. |
| PostgreSQL | Metadata and Airflow backend | ~2–5GB | Requires backup & indexing for performance. |
| Jupyter | Notebook runtime per user | 512MB–1GB RAM per session | Controlled via Docker resource limits. |
| Logging | Logs via Filebeat → Loki | 1–2GB per week | Rotated weekly; old logs offloaded to MinIO. |
| Monitoring | Prometheus metrics, Grafana dashboards | ~1GB/month | Lightweight metric collection. |

📌 Key Considerations

* Storage: Use SSDs with minimum 1TB capacity for high I/O performance.
* Memory: At least 16–32GB RAM on host for smooth multi-service operations.
* CPU: 8+ vCPUs recommended for concurrent usage and Spark workloads.
* Network: Internal bandwidth of 1Gbps+ for fast inter-container communication.

1. **🧩 Low-Level Design (LLD) – DataHarbour**

1. Architecture Overview

All components run in isolated Docker containers and communicate over a Docker network. Services are orchestrated using docker-compose with shared volumes for persistence.

2. Service Breakdown

🔹 Jupyter Notebook

* Runs interactive Python/PySpark notebooks
* Mounts ./data/jupyter for persistence
* Connects to Spark via internal Docker hostname (spark://spark-master:7077)

🔹 PySpark Cluster

* Spark master and worker containers
* Reads/writes data from MinIO using Hadoop S3 connector
* Communicates with Jupyter and Airflow

🔹 Airflow

* Webserver + scheduler + worker + metadata DB (PostgreSQL)
* DAGs mounted from ./data/airflow/dags
* Uses PostgreSQL as backend and Spark for ETL

🔹 PostgreSQL

* Stores Airflow metadata
* Persistent volume ./data/postgres
* Accessed by Airflow, pgAdmin

🔹 MinIO

* S3-compatible object storage
* Used for storing datasets, logs, backup
* Mounts ./data/minio/data
* Interacts with Spark and Jupyter

🔹 NGINX

* Reverse proxy for Jupyter, Airflow, MinIO, and pgAdmin
* Configurable for HTTPS and basic auth

🔹 Monitoring Stack

* Prometheus scrapes metrics from services
* Grafana displays dashboards
* Loki + Filebeat aggregate logs

🔹 Dashboard (Optional)

* Web UI for container health, metrics, logs (accesses Docker socket)

3. Network Design

* All services share a bridge network (dataharbour\_net)
* Internal DNS allows services to communicate using container names
* External ports are mapped for user access:
  + Jupyter: 8888
  + Airflow: 8081
  + MinIO: 9000/9001
  + pgAdmin: 5050

4. Security Design

* NGINX with basic auth in front of services
* MinIO and PostgreSQL use access keys and credentials
* Docker volumes are permission-controlled
* Optionally use HTTPS with self-signed or Let's Encrypt certs

5. Data Flow

1. User uploads data via Jupyter or MinIO.
2. Data is stored in MinIO buckets.
3. Spark jobs read data from MinIO and process it.
4. Results are stored back in MinIO or PostgreSQL.
5. Airflow orchestrates ETL jobs using Spark operators.
6. Monitoring stack captures metrics and logs.
7. **📐 High-Level Design (HLD) – DataHarbour**

🔰 1. System Overview

DataHarbour is a containerized, modular data engineering platform built on Docker. It integrates various data tools (e.g., PySpark, Jupyter, MinIO, Airflow, PostgreSQL) to facilitate data processing, storage, orchestration, and visualization. It is tailored for students, educators, and small businesses seeking an on-premises or lightweight cloud deployment.

🧱 2. Core Components

|  |  |
| --- | --- |
| Component | Purpose |
| Jupyter | Interactive notebooks for data exploration and PySpark job submission. |
| PySpark | Distributed data processing engine. |
| Airflow | Job scheduling and orchestration (ETL/ML pipelines). |
| MinIO | S3-compatible object storage for datasets, logs, and results. |
| PostgreSQL | Metadata storage and backend for Airflow. |
| pgAdmin | Web-based database management tool for PostgreSQL. |
| NGINX | Reverse proxy with security and access control. |
| Prometheus + Grafana | System monitoring and metric visualization. |
| Loki + Filebeat | Centralized log aggregation and viewing. |
| Dashboard (Optional) | Web interface for container/service health, metrics, and logs. |

🕸️ 3. Component Interaction Diagram (Textual View)

A screenshot of a computer program

AI-generated content may be incorrect.

🔌 4. Service Communication & Data Flow

1. User Interaction
   * Users access Jupyter, Airflow, MinIO UI, and pgAdmin via NGINX reverse proxy using HTTPS and BasicAuth.
   * They submit Spark jobs via Jupyter notebooks.
2. Data Processing Flow
   * Datasets are uploaded to MinIO.
   * Spark jobs (triggered from Jupyter or Airflow) read data from MinIO.
   * Processing results are written back to MinIO or stored in PostgreSQL.
3. Orchestration
   * Admins define DAGs in Airflow to automate Spark-based ETL or ML pipelines.
   * Airflow stores metadata in PostgreSQL.
4. Monitoring
   * Prometheus scrapes metrics from all services.
   * Grafana visualizes service metrics.
   * Logs are collected via Filebeat, sent to Loki, and made searchable.

🗂️ 5. Functional Groupings

| Functional Area | Services Involved |
| --- | --- |
| Data Exploration | Jupyter + PySpark |
| Data Storage | MinIO (object storage), PostgreSQL |
| Workflow Orchestration | Airflow + PostgreSQL |
| Monitoring & Logging | Prometheus, Grafana, Loki, Filebeat |
| Security & Access | NGINX, Docker volume permissions |
| Admin & Database UI | pgAdmin |

🔐 6. Security Architecture

* NGINX acts as a gatekeeper for all web services, using:
  + HTTPS for encrypted traffic.
  + Basic Authentication for user login.
* Role-Based Access Control (RBAC) in services like Jupyter and Airflow.
* MinIO and PostgreSQL use credentials for access.
* Docker Volumes enforce file access boundaries.

🛠️ 7. Deployment Strategy

* Containerized with Docker Compose
  + Each component is isolated and can be individually restarted.
  + Shared volumes enable data persistence across sessions.
* Single Host Setup
  + Ideal for small teams; all services run on one machine.
* Scalable Design
  + Kubernetes-ready architecture for scaling to 100+ users.
  + Can scale horizontally (e.g., more Spark workers, multiple Jupyter containers).

📊 8. Capacity and Performance

| Resource | Requirement |
| --- | --- |
| Disk | SSD, ~1TB+ for MinIO, logs, notebooks |
| Memory | 16–32GB RAM recommended |
| CPU | 8+ vCPUs for concurrent Spark jobs |
| Network | 1Gbps internal Docker network |

🔄 9. Data Lifecycle Overview

1. Input: Data is uploaded via Jupyter or MinIO.
2. Processing: Spark jobs perform ETL or ML tasks.
3. Storage: Results go to MinIO (files) or PostgreSQL (structured data).
4. Orchestration: Scheduled or triggered via Airflow DAGs.
5. Monitoring: Logs and metrics captured in real time.
6. Backup: Periodic backups of MinIO and PostgreSQL.
7. ***📘 Data Model Design – DataHarbour***

***🧩 1. Conceptual Data Model***

*At a high level, the platform handles:*

* ***Users & Roles***
* ***Notebooks & Jobs***
* ***Datasets & Storage Objects***
* ***Workflow Orchestration (DAGs)***
* ***Processing Results***
* ***Logs & Metrics***

***Entities and Relationships***

***A computer screen shot of words

AI-generated content may be incorrect.***

***🧮 2. Logical Data Model (Tables & Buckets)***

*This section breaks down how data is stored in* ***PostgreSQL*** *and* ***MinIO****, along with how* ***Spark jobs*** *read/write.*

***📦 A. PostgreSQL (Relational – Metadata & Workflow)***

***users***

| ***Column*** | ***Type*** | ***Description*** |
| --- | --- | --- |
| *user\_id (PK)* | *UUID* | *Unique identifier* |
| *username* | *TEXT* | *Login name* |
| *password\_hash* | *TEXT* | *Encrypted password* |
| *role* | *TEXT* | *e.g., admin, student, engineer* |
| *created\_at* | *TIMESTAMP* | *Registration time* |

***notebooks***

| ***Column*** | ***Type*** | ***Description*** |
| --- | --- | --- |
| *notebook\_id (PK)* | *UUID* | *Unique notebook ID* |
| *user\_id (FK)* | *UUID* | *Owner* |
| *path* | *TEXT* | *File path in Docker volume* |
| *created\_at* | *TIMESTAMP* | *Upload or creation time* |

***spark\_jobs***

| ***Column*** | ***Type*** | ***Description*** |
| --- | --- | --- |
| *job\_id (PK)* | *UUID* | *Unique job ID* |
| *notebook\_id (FK)* | *UUID* | *Originating notebook* |
| *status* | *TEXT* | *Queued, Running, Failed, Done* |
| *start\_time* | *TIMESTAMP* | *When job started* |
| *end\_time* | *TIMESTAMP* | *When job ended* |
| *output\_path* | *TEXT* | *Result path (MinIO or PostgreSQL)* |

***dag\_runs (Airflow metadata)***

| ***Column*** | ***Type*** | ***Description*** |
| --- | --- | --- |
| *dag\_run\_id (PK)* | *UUID* | *Unique DAG run* |
| *dag\_id* | *TEXT* | *DAG name* |
| *status* | *TEXT* | *Running, Success, Failed* |
| *triggered\_by* | *UUID (user)* | *Who triggered it* |
| *execution\_time* | *TIMESTAMP* | *Scheduled execution time* |

*Airflow also uses many metadata tables like task\_instance, dag, job, etc., as part of its own schema.*

***☁️ B. MinIO (Object Storage – Datasets, Results, Logs)***

*MinIO uses* ***S3-like buckets****. Each bucket stores files (objects), which Spark and Jupyter access via path URIs.*

***Suggested Buckets***

| ***Bucket Name*** | ***Contents*** |
| --- | --- |
| *datasets/* | *Raw uploaded data files* |
| *notebooks/* | *Backup of .ipynb files* |
| *spark-outputs/* | *Processed results from Spark jobs* |
| *logs/* | *Airflow + Spark + Jupyter logs* |
| *backups/* | *PostgreSQL & MinIO backups* |

*Each object in MinIO has metadata like:*

| ***Metadata*** | ***Description*** |
| --- | --- |
| *object\_key* | *e.g., datasets/sales.csv* |
| *size* | *File size in bytes* |
| *last\_modified* | *Timestamp of last update* |
| *user\_metadata* | *Custom tags, e.g., uploaded\_by* |

***⚙️ 3. Delta Lake (Optional for Spark Tables)***

*If you use* ***Delta Lake*** *on MinIO, Spark can maintain* ***ACID-compliant tables****. These tables are stored as* ***parquet + transaction logs****.*

*Example table paths:*

* *s3a://spark-outputs/sales\_data\_delta/*
* *s3a://datasets/user\_profiles\_delta/*

***🧪 4. Job Data Lifecycle Example***

1. ***Upload****: User uploads orders.csv to datasets/orders.csv via Jupyter or MinIO.*
2. ***Execution****: User runs a notebook that starts a Spark job to aggregate orders.*
3. ***Storage****: Spark job writes s3a://spark-outputs/order\_summary.parquet.*
4. ***Metadata****: Job metadata stored in spark\_jobs, and output path is saved.*
5. ***Orchestration****: Airflow DAGs can automate these steps daily.*

***🔐 5. Security & Access Control (RBAC Concept)***

*In PostgreSQL and MinIO, associate users with permissions:*

|  |  |  |  |
| --- | --- | --- | --- |
| ***Role*** | ***Notebook Access*** | ***DAG Creation*** | ***MinIO Buckets*** |
| *Admin* | *All* | *Yes* | *Full access* |
| *Engineer* | *Own + Shared* | *Yes* | *Own + shared* |
| *Student* | *Own only* | *No* | *Own only* |

1. ✅ **Testing Suggestions for DataHarbour**

🧪 1. Authentication & Authorization

🔹 Functional Tests

* Verify login/logout functionality for all user roles (admin, student, engineer).
* Validate role-based access (RBAC) for services (e.g., Jupyter, Airflow, MinIO).
* Test invalid credentials and brute-force prevention.

🔹 Security Tests

* Ensure HTTPS is enforced (test NGINX TLS configuration).
* Test password storage (should be hashed, not plain text).
* Check session timeout and cookie security.

🧪 2. Service Availability & Container Integration

🔹 Functional Tests

* Confirm all containers (Jupyter, Airflow, Spark, PostgreSQL, MinIO) launch correctly via docker-compose.
* Test inter-container communication (e.g., Spark can read from MinIO).
* Restart each container and verify data persistence.

🔹 Health & Smoke Tests

* API health checks (e.g., Airflow /health, MinIO /minio/health/live).
* Dashboard service status indicators reflect actual container states.

🧪 3. Notebook & Spark Job Testing

🔹 Functional Tests

* Create, save, and reopen Jupyter notebooks.
* Submit PySpark jobs via notebooks – verify outputs in MinIO or PostgreSQL.
* Handle job failures gracefully (e.g., invalid input file, syntax errors).

🔹 Performance Tests

* Test large dataset processing (~1GB+) and observe resource usage.
* Monitor Spark worker scaling and memory management.

🧪 4. Data Storage: MinIO and PostgreSQL

🔹 MinIO (S3-compatible)

* Upload and download files via UI and programmatic access.
* Test access with and without credentials.
* Simulate file corruption and test recovery.

🔹 PostgreSQL

* Validate schema integrity (users, jobs, etc.).
* Perform read/write/load tests using Airflow and pgAdmin.
* Check indexing and query performance.

🧪 5. Airflow DAG Execution

🔹 Functional Tests

* Add, enable, trigger DAGs; verify task run and logs.
* Test Spark-based DAGs with data pipelines.
* Retry failed DAGs and check alerting behavior (if configured).

🔹 Regression Tests

* Ensure existing DAGs run after upgrades or config changes.

🧪 6. Monitoring & Logging

🔹 Logging

* Check that Filebeat captures logs from all containers.
* Validate Loki receives and stores logs (search via Grafana).
* Rotate logs weekly and verify MinIO offloading.

🔹 Metrics

* Test Prometheus scraping for all services (e.g., Docker, Node Exporter).
* Verify Grafana dashboards show real-time CPU/RAM/network metrics.

🧪 7. Backup & Restore

* Simulate cron backup of PostgreSQL and MinIO.
* Test restoring from backups – PostgreSQL schema and MinIO files.
* Perform disaster recovery test (delete container volumes and recover).

🧪 8. Scalability & Load Testing

* Run concurrent Jupyter sessions (10–50 users).
* Schedule overlapping Spark jobs and monitor CPU/memory.
* Use tools like Locust or JMeter to simulate load on Airflow and MinIO.

🧪 9. Portability & Deployment Testing

* Run docker-compose up on a fresh system – validate startup.
* Change ports and credentials in the central config and verify propagation.
* Test on multiple operating systems (Linux, macOS, Windows with WSL).

🧪 10. Usability Testing

* Validate UI consistency across Jupyter, MinIO, Airflow, pgAdmin.
* Check onboarding documentation and first-time user experience.
* Test service dashboard for accurate resource and log display.

📋 Suggested Tools

| Testing Area | Tool Suggestions |
| --- | --- |
| API/Service Testing | Postman, curl |
| Load Testing | Locust, JMeter, Apache Bench |
| Unit Testing | pytest, unittest (for scripts) |
| UI Testing | Selenium, Playwright |
| Monitoring Validation | Prometheus UI, Grafana |
| Backup Testing | Cron logs, manual restore steps |

1. ***🔐 Secrets Management Using. env in DataHarbour***

*✅ 1. Why Use. env Files?*

*.env files help manage sensitive configurations like:*

* *Database passwords*
* *MinIO access keys*
* *JWT tokens or BasicAuth credentials*
* *Spark or Airflow environment variables*

*They allow you to:*

* *Avoid hardcoding secrets in docker-compose.yml*
* *Easily change secrets without editing code*
* *Version-control templates (without actual secrets)*

***📁 2. Basic Structure of .env***

*Create a .env file in your root project folder (not committed to Git):*

*# PostgreSQL*

*POSTGRES\_USER=airflow*

*POSTGRES\_PASSWORD=airflow\_secure\_pwd*

*POSTGRES\_DB=airflow\_metadata*

*# MinIO*

*MINIO\_ROOT\_USER=minioadmin*

*MINIO\_ROOT\_PASSWORD=minio\_secure\_pwd*

*# Jupyter*

*JUPYTER\_TOKEN=mytoken123*

*# Airflow*

*AIRFLOW\_\_CORE\_\_FERNET\_KEY=your\_fernet\_key\_here*

*AIRFLOW\_\_CORE\_\_EXECUTOR=LocalExecutor*

*# Basic Auth (NGINX)*

*BASIC\_AUTH\_USER=admin*

*BASIC\_AUTH\_PASS=secure\_nginx\_pass*

***🐳 3. Referencing in docker-compose.yml***

*You can now use variables in your docker-compose.yml:*

*version: '3.8'*

*services:*

*postgres:*

*image: postgres:14*

*environment:*

*- POSTGRES\_USER=${POSTGRES\_USER}*

*- POSTGRES\_PASSWORD=${POSTGRES\_PASSWORD}*

*- POSTGRES\_DB=${POSTGRES\_DB}*

*volumes:*

*- ./data/postgres:/var/lib/postgresql/data*

*minio:*

*image: minio/minio*

*environment:*

*- MINIO\_ROOT\_USER=${MINIO\_ROOT\_USER}*

*- MINIO\_ROOT\_PASSWORD=${MINIO\_ROOT\_PASSWORD}*

*command: server /data*

*ports:*

*- "9000:9000"*

*- "9001:9001"*

*jupyter:*

*image: jupyter/pyspark-notebook*

*environment:*

*- JUPYTER\_TOKEN=${JUPYTER\_TOKEN}*

*ports:*

*- "8888:8888"*

*And load .env automatically:*

*bash*

*CopyEdit*

*docker-compose --env-file .env up -d*

*Most recent versions of Docker Compose automatically read .env by default.*

***🚫 4. Security Best Practices***

| ***Practice*** | ***Description*** |
| --- | --- |
| *.gitignore* | *Add .env to .gitignore to prevent leaks.* |
| *.env.template* | *Create a sample file for teammates, without secrets.* |
| *File Permissions* | *Run chmod 600 .env to restrict read/write access.* |
| *Docker Secrets (Optional Upgrade)* | *For higher security, switch to Docker secrets in swarm/K8s.* |

***🔁 5. Using in Airflow, Jupyter, and Spark***

*Make sure internal services like Airflow and Spark pick up env variables:*

* ***Airflow****:*
  + *Store config in .env or mount .env as a file and source it inside entrypoint.sh.*
* ***Jupyter****:*
  + *Use JUPYTER\_TOKEN to control notebook access securely.*
* ***Spark****:*
  + *Pass credentials via Hadoop config for S3 access:*

*spark.hadoop.fs.s3a.access.key=${MINIO\_ROOT\_USER}*

*spark.hadoop.fs.s3a.secret.key=${MINIO\_ROOT\_PASSWORD}*

***🧪 6. Testing the Setup***

* *Run docker-compose up and verify services use your .env secrets.*
* *Log into MinIO and check credentials.*
* *Try Jupyter notebook token login.*
* *Check Airflow UI and metadata database connection.*

**🔐 RBAC Roles per Service – DataHarbour**

| Role | Description |
| --- | --- |
| Admin | Full system access, service control, user management |
| Engineer | Technical role for developing DAGs, notebooks, running Spark jobs |
| Analyst | Primarily reads data, runs analysis in notebooks, no deployment or config |

🧱 1. RBAC Matrix (Service vs. Role)

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Service | Action | Admin ✅ | Engineer ✅ | Analyst ✅ |
| Jupyter | Create/Edit/Delete Notebooks | ✅ | ✅ | ✅ |
|  | Run PySpark jobs | ✅ | ✅ | ✅ |
|  | Access Other Users' Notebooks | ✅ | Optional 🔄 | ❌ |
| Airflow | View DAGs | ✅ | ✅ | ✅ (Read) |
|  | Create/Edit/Delete DAGs | ✅ | ✅ | ❌ |
|  | Trigger/Stop DAGs | ✅ | ✅ | ❌ |
|  | View Logs & Task Status | ✅ | ✅ | ✅ |
| MinIO | Upload/Download Datasets | ✅ | ✅ | ✅ |
|  | Manage Buckets | ✅ | ❌ | ❌ |
|  | Delete Files | ✅ | ✅ | ❌ |
| PostgreSQL | Query Data (via pgAdmin) | ✅ | ✅ | ✅ (limited) |
|  | Create/Drop Tables | ✅ | ✅ | ❌ |
|  | Manage DB Users | ✅ | ❌ | ❌ |
| NGINX Auth | Change Credentials | ✅ | ❌ | ❌ |
| Monitoring (Grafana) | View Dashboards | ✅ | ✅ | ✅ |
|  | Create/Edit Dashboards | ✅ | ✅ | ❌ |
| Logs (Loki) | View Logs | ✅ | ✅ | ✅ |
|  | Search Logs | ✅ | ✅ | ✅ |
| System Dashboard | Start/Stop Containers | ✅ | ❌ | ❌ |
|  | View Container Status & Health | ✅ | ✅ | ✅ (view only) |

🔄 "Optional" means configurable by admin depending on whether engineers should collaborate across notebooks or not.

🛡️ 2. Implementation Tips per Service

Jupyter

* Use token authentication.
* Restrict container mounts to user-specific volumes.

Airflow

* Use [RBAC UI roles](https://airflow.apache.org/docs/apache-airflow/stable/security/access-control.html):
  + Admin → Admin
  + Engineer → Op
  + Analyst → Viewer

MinIO

* Use access keys per user.
* Define per-bucket or per-prefix permissions with MinIO policies.

PostgreSQL

* Create roles and grant them schema/table-level access.
* Example:

sql

CopyEdit

CREATE ROLE analyst LOGIN PASSWORD 'xyz';

GRANT SELECT ON ALL TABLES IN SCHEMA public TO analyst;

Grafana

* Use built-in roles:
  + Admin: full access
  + Editor (Engineer): create dashboards
  + Viewer (Analyst): read-only access

📦 3. RBAC Storage Location Options

* Central .env/config.yaml defining user-role mapping
* Use LDAP or OAuth2 if integrating with enterprise auth in the future
* RBAC rules can be enforced via:
  + Application-level (e.g., Airflow UI)
  + Reverse proxy (e.g., NGINX per-path restriction)
  + Volume-level permission (for notebooks)

1. ***🚨 Monitoring Alerts – DataHarbour***

***📡 1. Alerting System Architecture***

***Tools Involved****:*

| ***Tool*** | ***Role*** |
| --- | --- |
| ***Prometheus*** | *Collects time-series metrics* |
| ***Alertmanager*** | *Sends notifications (email, Slack, etc.)* |
| ***Grafana*** | *Visualizes metrics, defines alert rules* |
| ***Loki*** *+* ***Filebeat*** | *Collects and indexes logs for failure detection* |

***🔔 2. Key Alert Categories***

| ***Category*** | ***Description*** |
| --- | --- |
| ***Service Health*** | *Service up/down, container restarts* |
| ***Resource Usage*** | *CPU, memory, disk, network thresholds* |
| ***Spark Failures*** | *Job failures, task retries, long execution* |
| ***Airflow Failures*** | *DAG/task failures, missed runs* |
| ***MinIO Errors*** | *Access failures, storage full, high latency* |
| ***PostgreSQL*** | *Connection failures, slow queries, DB growth* |
| ***Backup Status*** | *Missed or failed scheduled backups* |
| ***Log-based*** | *Regex match for ERROR, FATAL, OOM, etc.* |

***🧪 3. Sample Alert Rules (Prometheus)***

*Here are some* ***Prometheus alert rules*** *you can define in alert.rules.yml:*

*yaml*

*CopyEdit*

*groups:*

*- name: dataharbour-alerts*

*rules:*

*- alert: HighCPUUsage*

*expr: rate(container\_cpu\_usage\_seconds\_total{container!="",image!=""}[2m]) > 0.9*

*for: 2m*

*labels:*

*severity: warning*

*annotations:*

*summary: "High CPU Usage detected"*

*description: "Container {{ $labels.container }} is using >90% CPU"*

*- alert: ContainerDown*

*expr: up{job="docker"} == 0*

*for: 30s*

*labels:*

*severity: critical*

*annotations:*

*summary: "Container {{ $labels.instance }} is down"*

*description: "Check if service crashed or was stopped"*

*- alert: MinIOStorageFull*

*expr: minio\_disk\_storage\_used\_bytes / minio\_disk\_storage\_total\_bytes > 0.9*

*for: 1m*

*labels:*

*severity: warning*

*annotations:*

*summary: "MinIO storage almost full"*

*description: "Over 90% usage on MinIO disk volume"*

***📊 4. Grafana Alerts***

*Grafana 8+ has* ***unified alerting****. Use this for:*

| ***Dashboard*** | ***Alert Examples*** |
| --- | --- |
| *Spark Job Monitor* | *Job duration > 10 min; error count > 3* |
| *Airflow Dashboard* | *DAG failed more than 3 times in a row* |
| *System Metrics* | *RAM > 80%, Disk > 90%, Network drops* |
| *PostgreSQL* | *Connections > threshold, slow queries* |

*Create alerts directly from the Grafana panel UI → Alert tab → Add rule.*

***🔎 5. Log-Based Alerts (Loki + Filebeat)***

*Set up* ***log regex match rules*** *via Grafana Loki alerts:*

***Example Log Alert (Loki)***

*logql*

*CopyEdit*

*count\_over\_time({container="spark-worker"} |= "ERROR" [5m]) > 5*

*Triggers if more than 5 errors in Spark logs over 5 minutes.*

*Other patterns:*

* *OOMKilled, exit code, connection refused, SSL error*

***✉️ 6. Alert Routing (Alertmanager)***

*Define* ***routes and receivers*** *in alertmanager.yml:*

*yaml*

*CopyEdit*

*receivers:*

*- name: 'email-admins'*

*email\_configs:*

*- to: 'ops@datateam.com'*

*from: 'alert@datateam.com'*

*smarthost: 'smtp.mailserver.com:587'*

*auth\_username: 'alert@datateam.com'*

*auth\_password: 'password'*

*route:*

*receiver: 'email-admins'*

*group\_by: ['alertname']*

*repeat\_interval: 1h*

*Optional receivers:*

* *Slack*
* *Microsoft Teams*
* *PagerDuty*
* *Telegram bots*

***🧾 7. Alert Testing Checklist***

| ***Task*** | ***Status*** |
| --- | --- |
| *Test CPU/RAM alert with load* | *✅* |
| *Simulate DAG failure in Airflow* | *✅* |
| *Fill MinIO disk with dummy data* | *✅* |
| *Stop a container manually* | *✅* |
| *Trigger log-based alert via Spark error* | *✅* |
|  |  |

1. ***🖥️ Suggested User Interface (Dashboard) for DataHarbour***

***🎯 Goal of the Dashboard***

*Provide an easy-to-use interface for:*

* ***Monitoring system health***
* ***Viewing service status/logs***
* ***Starting/stopping services***
* ***Accessing metrics***
* ***Tracking Spark jobs & Airflow DAGs***

***🔧 1. Suggested Tech Stack***

| ***Layer*** | ***Technology*** | ***Justification*** |
| --- | --- | --- |
| ***Frontend*** | *React + Tailwind CSS* | *Fast, responsive, customizable UI* |
| ***Backend*** | *Node.js or Python (Flask/FastAPI)* | *REST APIs to access Docker status, logs, metrics* |
| ***Metrics Source*** | *Prometheus (JSON API)* | *Existing integration* |
| ***Logs Source*** | *Loki (via Grafana API or direct queries)* | *Already used in stack* |
| ***Docker Interface*** | *Docker SDK (Python or Node)* | *For container controls* |
| ***Authentication*** | *Reuse NGINX BasicAuth or JWT-based UI login* | *To avoid duplicating user stores* |

***📊 2. Dashboard UI Components***

| ***Section*** | ***Features*** |
| --- | --- |
| ***Header*** | *System name, active user, refresh toggle* |
| ***Service Overview*** | *Grid of all services (Jupyter, Airflow, Spark, etc.) 🔹 Status: Running/Stopped 🔹 Controls: Start/Stop/Restart 🔹 CPU/RAM Usage (via Prometheus)* |
| ***Logs Viewer*** | *Dropdown to select service + log search field Live tail of logs from Loki* |
| ***Metrics Dashboard*** | *Embed Grafana if needed, or show inline: – System CPU/RAM/Network – Per-container stats – Spark job performance* |
| ***Airflow DAG Tracker*** | *List DAGs: status, last run, duration Trigger DAGs manually* |
| ***Spark Job Monitor*** | *Table of recent jobs: status, runtime, output link* |
| ***MinIO Browser (Optional)*** | *Embed MinIO browser UI or expose basic file navigator for datasets/outputs* |
| ***Backup Status*** | *Last successful backup, next scheduled, restore button* |
| ***Alerts & Notifications*** | *Show active Prometheus alerts with severity levels* |

***🧩 3. Data Sources & Integration***

| ***Data*** | ***Source*** | ***Access Method*** |
| --- | --- | --- |
| *Service Health* | *Docker API* | *Docker SDK* |
| *Logs* | *Loki* | *REST or Grafana API* |
| *Metrics* | *Prometheus* | *HTTP API: /api/v1/query* |
| *DAGs* | *Airflow REST API* | */dags, /dagRuns* |
| *Spark Jobs* | *PostgreSQL or parsing logs* | *Via spark\_jobs table* |
| *Backups* | *Cron log files or status flag in MinIO* | *Script output or custom REST API* |

***🔐 4. Security Considerations***

* *Integrate with existing NGINX BasicAuth.*
* *Optionally allow token-based login (JWT) for the dashboard only.*
* *Limit Docker control actions to admins.*
* *Hide sensitive logs or env data from non-admins.*

***📌 5. Optional Enhancements***

* ***Dark/light mode toggle***
* ***Service uptime graphs***
* ***Email/slack notification toggle per alert***
* ***User activity logs***

***📁 Directory Structure (React + Flask Example)***

*bash*

*CopyEdit*

*/dashboard*

*/frontend*

*/components*

*/pages*

*App.jsx*

*tailwind.config.js*

*/backend*

*app.py (Flask server)*

*docker\_api.py*

*prometheus\_api.py*

*airflow\_api.py*

*.env*

*docker-compose.yml*

***🧪 Want a Starter UI?***

*I can generate:*

* *A* ***React + Tailwind UI mockup***
* *Backend Flask/Node endpoints to connect to Prometheus, Docker, Airflow*
* *A simple dashboard layout to extend*

1. ***🛡️ Goal*** *Provide* ***centralized authentication and authorization*** *across:*

* *Jupyter*
* *Airflow*
* *MinIO*
* *pgAdmin*
* *Dashboard (optional)*

***🔐 1. Why Move Beyond BasicAuth?***

| ***Limitation (BasicAuth)*** | ***Benefit with OAuth2 / SSO*** |
| --- | --- |
| *No user provisioning* | *Supports user roles, groups, claims* |
| *Hardcoded credentials* | *Central user management* |
| *No password reset or MFA* | *Can enforce MFA, SSO policies* |
| *Not scalable* | *Works with LDAP, Google, Microsoft, GitHub, etc.* |
| *No audit trail* | *Identity providers log access and auth events* |

***🧩 2. Recommended Integration Architecture***

***Option A: Keycloak (Self-hosted)***

* *Open-source Identity and Access Management (IAM)*
* *Integrates well with Dockerized environments*
* *Offers:*
  + *OAuth2, OIDC, SAML*
  + *Role-based access*
  + *User federation (LDAP, GitHub, Google)*

***Flow:***

1. *Host Keycloak in a Docker container*
2. *Register each service (Jupyter, Airflow, etc.) as a Keycloak client*
3. *Configure reverse proxy (NGINX or Traefik) to enforce token validation*
4. *Use Keycloak login UI as the central portal*

***Option B: Auth0 (Managed Cloud IAM)***

* *Quick to set up, enterprise-grade*
* *Easy GitHub/Google/LDAP login*
* *Good for MVPs or external user onboarding*
* *Pricing required beyond free tier*

***🔧 3. How to Implement (Step-by-Step)***

***✅ Phase 1 – Setup IAM***

* *Deploy* ***Keycloak*** *via Docker:*

*bash*

*CopyEdit*

*docker run -d --name keycloak \*

*-e KEYCLOAK\_ADMIN=admin -e KEYCLOAK\_ADMIN\_PASSWORD=secret \*

*-p 8080:8080 quay.io/keycloak/keycloak:latest start-dev*

* *Access at http://localhost:8080*

***✅ Phase 2 – Configure Clients***

*In Keycloak:*

* *Create clients for each service:*
  + *dataharbour-jupyter*
  + *dataharbour-airflow*
  + *dataharbour-minio*
  + *dataharbour-dashboard*
* *Set redirect URIs and enable PKCE (for browser apps)*

***✅ Phase 3 – Protect Services***

* *For each service, configure OIDC middleware:*
  + ***Jupyter****: Use* [*jupyterhub with OAuthenticator*](https://github.com/jupyterhub/oauthenticator)
  + ***Airflow****: Use* [*OIDC auth backend*](https://airflow.apache.org/docs/apache-airflow/stable/security/auth.html#open-id-connect)
  + ***MinIO****: Currently supports LDAP/OIDC via enterprise edition only*
  + ***pgAdmin****: Limited support; recommend access via VPN or proxy*

***✅ Phase 4 – NGINX as Auth Gateway (Optional)***

*Replace BasicAuth with OpenID Connect middleware (e.g.* [*lua-resty-openidc*](https://github.com/zmartzone/lua-resty-openidc) *or oauth2-proxy).*

*Example:*

*nginx*

*CopyEdit*

*location /jupyter {*

*auth\_request /oauth2/auth;*

*proxy\_pass http://jupyter:8888;*

*}*

***🗂️ 4. RBAC with OAuth2***

* *Define roles/groups in Keycloak: admin, engineer, analyst*
* *Assign claims (scopes) to tokens:*

*json*

*CopyEdit*

*{*

*"roles": ["engineer"],*

*"preferred\_username": "jdoe"*

*}*

* *Jupyter/Airflow should map roles to access levels*

***📦 5. Containerize It***

*Add to docker-compose.yml:*

*yaml*

*CopyEdit*

*keycloak:*

*image: quay.io/keycloak/keycloak:latest*

*command: start-dev*

*environment:*

*KEYCLOAK\_ADMIN: admin*

*KEYCLOAK\_ADMIN\_PASSWORD: secret*

*ports:*

*- "8080:8080"*

***✨ Bonus: Optional Enhancements***

| ***Feature*** | ***How to Enable*** |
| --- | --- |
| *MFA (2FA)* | *Built-in in Keycloak* |
| *SSO with Google/GitHub* | *Add "Identity Provider" in Keycloak* |
| *Rate-limiting* | *Add fail2ban or limit tokens per IP* |
| *Session Timeout / Revocation* | *Set token lifetimes and revocation policies* |

***✅ Summary Recommendations***

| ***For*** | ***Suggestion*** |
| --- | --- |
| *Internal teams* | *Start with* ***Keycloak*** |
| *External-facing / SaaS* | *Use* ***Auth0*** *or* ***Okta*** |
| *Lightweight use cases* | *oauth2-proxy + GitHub login via NGINX* |

***✅ Why Log Enrichment Matters***

| ***Without Enrichment*** | ***With Enrichment*** |
| --- | --- |
| *“Spark job failed.”* | *{"job\_id":"abc123", "user":"jane", "dag":"etl\_daily", "error":"file not found"}* |
| *“Notebook disconnected.”* | *{"notebook":"sales\_analysis.ipynb", "user":"john", "time":"12:00"}* |
| *“Airflow task failed.”* | *{"dag":"daily\_metrics", "task":"transform", "status":"failed", "user":"admin"}* |

***🔧 How to Enrich Logs in Each Component***

***1. Jupyter (Notebook Logs)***

* *Customize the notebook Docker image to:*
  + *Override start-notebook.sh*
  + *Inject user\_id into logs via environment or notebook metadata*
* *Enable logging in JSON format:*

*bash*

*CopyEdit*

*exec jupyter notebook \*

*--log-level=INFO \*

*--log-format='{"user":"${JUPYTER\_USER}", "notebook":"${NOTEBOOK\_NAME}", "message":"%(message)s"}'*

* *You can use papermill to programmatically tag notebook runs.*

***2. PySpark***

* *Add custom loggers to your Spark job scripts:*

*python*

*CopyEdit*

*import logging*

*logger = logging.getLogger("spark\_job\_logger")*

*job\_context = {*

*"user": "alice",*

*"job\_id": "job-123",*

*"input\_file": "sales.csv"*

*}*

*logger.info(f"Starting job", extra=job\_context)*

* *Or configure log4j.properties to include job/user tags:*

*properties*

*CopyEdit*

*log4j.appender.console.layout.ConversionPattern=%d{yyyy-MM-dd HH:mm:ss} [%X{user}] [%X{job\_id}] - %m%n*

* *Use MDC (Mapped Diagnostic Context) to inject metadata dynamically.*

***3. Airflow***

* *Customize airflow.cfg logging section:*

*ini*

*CopyEdit*

*[logging]*

*log\_format = {"dag":"%(dag\_id)s", "task":"%(task\_id)s", "user":"%(username)s", "msg":"%(message)s"}*

* *Enrich logs in DAGs:*

*python*

*CopyEdit*

*from airflow.utils.log.logging\_mixin import LoggingMixin*

*logger = LoggingMixin().log*

*logger.info("Task started", extra={"user": "bob", "dag\_id": "daily\_summary"})*

***4. Loki + Filebeat Config***

*Make sure Filebeat parses logs into structured JSON and sends labels to Loki:*

*yaml*

*CopyEdit*

*filebeat.inputs:*

*- type: container*

*paths:*

*- /var/lib/docker/containers/\*/\*.log*

*json.keys\_under\_root: true*

*json.add\_error\_key: true*

*processors:*

*- add\_docker\_metadata: ~*

*output.loki:*

*hosts: ["http://loki:3100"]*

*labels:*

*job: "dataharbour"*

*user: "${json.user}"*

*job\_id: "${json.job\_id}"*

*dag: "${json.dag}"*

***🧪 Filter Examples in Grafana Loki***

* *See all logs from user “alice”:*

*logql*

*CopyEdit*

*{user="alice"}*

* *Spark job errors with job ID:*

*logql*

*CopyEdit*

*{job="dataharbour", job\_id="job-456"} |= "ERROR"*

* *DAG etl\_daily logs in the last 30 minutes:*

*logql*

*CopyEdit*

*{dag="etl\_daily"} |~ ".\*"*

***✅ Next Steps for Implementation***

| ***Task*** | ***Action*** |
| --- | --- |
| *Define common log format* | *JSON format with user, service, job/task ID* |
| *Update all services* | *Inject contextual metadata via env or runtime* |
| *Modify Filebeat config* | *Enable JSON parsing and label extraction* |
| *Review Grafana dashboards* | *Add filters/panels by user, job, dag, etc.* |

***🎯 Why Add a Data Catalog?***

| ***Problem Without Catalog*** | ***Benefit With Catalog*** |
| --- | --- |
| *Users don’t know what datasets exist* | *Browse/search datasets, notebooks, outputs* |
| *No ownership or schema visibility* | *See who uploaded data and what schema it has* |
| *Poor reuse of jobs or DAGs* | *Document DAGs, jobs, results, parameters* |
| *No lineage or impact tracking* | *Trace how data was created and used* |

***✅ Recommended Tools (Lightweight Integration)***

***1. Amundsen (by Lyft)***

* *Focus: Dataset discovery + lineage*
* *Stack: Neo4j + Flask + Elasticsearch*
* *Features:*
  + *Dataset, table, and column metadata*
  + *Ownership, popularity metrics*
  + *Search by name, column, tags*

***2. DataHub (by LinkedIn)***

* *Focus: Metadata graph across datasets, jobs, dashboards*
* *Stack: Kafka + Elasticsearch + GraphQL + React UI*
* *Features:*
  + *Lineage visualization (from Spark or Airflow)*
  + *Programmatic metadata ingestion*
  + *Schema versioning + RBAC*

***3. OpenMetadata (modern & extensible)***

* *Fast-growing open-source option*
* *Good UI/UX*
* *Native support for Airflow, Spark, MinIO, PostgreSQL*

***🔧 Suggested Minimal Integration Plan (Phase-wise)***

***🔹 Phase 1: Metadata Collection (Script-Based)***

*Create a script to extract metadata from:*

* ***MinIO****:*
  + *Bucket name, object name, size, timestamp, uploader (via user\_metadata)*
* ***PostgreSQL****:*
  + *Table names, row counts, schema*
* ***Airflow DAGs****:*
  + *DAG name, owner, last run, tags*

*Store results in a central metadata table (metadata.catalog) in PostgreSQL or as structured JSON in a catalog/ MinIO bucket.*

***🔹 Phase 2: Add Search UI (Lightweight)***

*Build a simple search page or notebook that:*

* *Lists datasets and notebooks by name*
* *Filters by owner, tag, or update time*
* *Links to files in MinIO or DAGs in Airflow*

***Optional Tools****:*

* *Use React + Flask or Streamlit for a UI*
* *Use SQLite if not ready for a full search engine*

***🔹 Phase 3: Integrate with Amundsen or DataHub***

1. *Install Amundsen or DataHub with Docker*
2. *Use* ***Airflow metadata plugin*** *and custom extractors for MinIO*
3. *Ingest metadata:*
   * *From PostgreSQL tables*
   * *From MinIO (via S3 APIs or custom scripts)*
4. *Use UI to search datasets, track lineage, and assign ownership*

*📘 Reference:*

* *Amundsen Setup*
* *DataHub Quickstart*

***🧩 Example Metadata Entry***

| ***Field*** | ***Value*** |
| --- | --- |
| *Dataset Name* | *sales\_2024.csv* |
| *Location* | *s3a://datasets/sales\_2024.csv* |
| *Uploaded By* | *analyst1* |
| *Schema* | *customer\_id: int, amount: float* |
| *Created* | *2024-12-01 10:23* |
| *Lineage* | *source: order\_raw → sales\_2024 → dashboard\_metric\_dag* |
| *Tags* | *sales, monthly, sensitive* |