AWS Migration Readiness and Action Plan for Enterprise Data Platform

This document outlines a strategic plan for migrating your enterprise-ready data platform from its local Docker Compose environment to Amazon Web Services (AWS). It builds upon the architectural principles and component comparisons detailed in the "Deep Dive: Cloud Component Comparison (AWS, Azure, GCP)" and other addendums, providing a structured approach to achieve a scalable, production-ready cloud environment.

1. Introduction: From Local to Cloud-Native

Your robust local data platform, built with Docker Compose, has served as an invaluable environment for rapid iteration, testing, and skill development. However, achieving enterprise-grade scalability, reliability, and reduced operational overhead necessitates a transition to managed cloud services. This plan focuses specifically on AWS, mapping your existing components to their cloud-native equivalents and outlining the phases for a successful migration.

2. Migration Principles and Objectives

Before diving into the action plan, it's crucial to establish core principles and objectives for the migration:

- **Phased Approach:** Minimize risk by migrating components or pipelines incrementally rather than a "big bang" approach.
- **Infrastructure as Code (IaC):** Automate provisioning and management of all AWS resources using Terraform for consistency, repeatability, and version control.
- **Managed Services First:** Prioritize AWS managed services (PaaS/SaaS) to reduce operational burden, unless specific requirements necessitate self-managed solutions.
- **Cost Optimization:** Design for cost-efficiency from the outset, leveraging serverless options where appropriate and implementing robust monitoring for spend.
- **Security & Compliance:** Embed security best practices (IAM, network segmentation, encryption) and ensure compliance from day one.
- **Observability:** Implement comprehensive monitoring, logging, and tracing using AWS-native tools to gain deep operational insights.
- **Disaster Recovery (DR):** Define RPO/RTO and build robust DR capabilities using AWS's inherent high-availability features.

3. Core Component Mapping (Local to AWS)

Based on the "Deep Dive: Cloud Component Comparison (AWS, Azure, GCP).pdf", here's a recap of the primary AWS service equivalents for your local platform components:

Local Component (Docker	Primary AWS	Role in AWS

Compose)	Replacement(s)	
FastAPI Ingestor	AWS Lambda + Amazon API Gateway	Scalable, serverless API for real-time ingestion.
Apache Kafka	Amazon MSK (Managed Streaming for Kafka)	Fully managed, highly available Kafka cluster.
MinIO (S3-compatible)	Amazon S3 (Simple Storage Service)	Highly durable, scalable object storage for data lake.
Delta Lake (on MinIO)	AWS Lake Formation (governance over S3), Databricks on AWS, EMR with Delta Lake	Data lakehouse format on S3 with ACID properties.
Apache Spark (PySpark Jobs)	AWS Glue (Serverless ETL), Amazon EMR (Managed Spark)	Distributed processing for batch/streaming ETL/ELT.
PostgreSQL	Amazon RDS for PostgreSQL, Amazon Aurora PostgreSQL	Managed relational database service.
MongoDB	Amazon DocumentDB (MongoDB-compatible), DynamoDB (Key-Value/Document)	Managed NoSQL document database.
Apache Airflow	Amazon MWAA (Managed Workflows for Apache Airflow)	Fully managed Apache Airflow service for orchestration.
Grafana Alloy (OTLP Collector)	AWS Distro for OpenTelemetry (ADOT)	Standardized telemetry collection and forwarding.
Grafana	Amazon Managed Grafana, CloudWatch Dashboards	Interactive data visualization and monitoring.
Spline	AWS Glue Data Catalog (basic lineage), External Tools	Spark data lineage tracking.
OpenMetadata	AWS Glue Data Catalog, AWS DataZone, External Tools	Centralized metadata management and data cataloging.
cAdvisor	AWS CloudWatch Container Insights	Container resource usage and performance analysis.
Locust (Load Testing)	AWS Fargate/ECS (for distributed Locust), AWS Batch	Scalable load testing for cloud endpoints.

4. Migration Readiness Checklist (Pre-Migration)

Before initiating the migration, ensure the following foundational aspects are addressed:

- **4.1. AWS Account Setup:**
    - Create and configure an AWS Organization for multi-account strategy (production, staging, dev, sandbox).
    - Set up AWS IAM (Identity and Access Management) for granular user and role

permissions. Implement MFA for all users.
    - Configure AWS Budgets and Cost Explorer to monitor and control spending.
    - Establish AWS VPCs (Virtual Private Clouds) with public and private subnets, NAT Gateways, and VPC Endpoints for secure communication.
    - Define Security Groups and Network ACLs for network isolation.
- **4.2. Codebase Preparation:**
    - **Modularization:** Ensure your FastAPI application, PySpark jobs, and Airflow DAGs are modular and decoupled from the local environment specifics.
    - **Configuration Externalization:** All connection strings, secrets, and environment-specific parameters should be externalized (e.g., using environment variables, AWS Systems Manager Parameter Store, AWS Secrets Manager).
    - **Containerization (if not serverless):** Finalize Dockerfiles for any services that will run on ECS/EKS/Fargate, ensuring they are optimized for production.
    - **Dependency Management:** Clean up requirements.txt or pyproject.toml to include only necessary libraries.
- **4.3. Data Preparation:**
    - **Data Inventory:** Catalog all data sources, volumes, growth rates, and access patterns.
    - **Data Cleansing & Quality:** Ensure data quality checks are robust and integrated into your pipelines.
    - **Historical Data Migration Strategy:** Plan how to migrate existing data from local volumes to S3 (e.g., aws s3 sync, AWS DataSync, database migration services).
- **4.4. CI/CD Pipeline Readiness:**
    - Familiarize with "Deep-Dive Addendum: IaC & CI/CD Recipes.pdf".
    - Set up a Git repository (e.g., AWS CodeCommit, GitHub) for all IaC and application code.
    - Configure CI/CD pipelines (e.g., AWS CodePipeline, GitHub Actions) for automated testing, building Docker images, and deploying Terraform.

5. Phased AWS Migration Action Plan

This section outlines the migration process in distinct phases, allowing for iterative deployment and validation.

Phase 1: Foundational Infrastructure (IaC First)

Objective: Establish the secure, scalable networking and core data lake storage in AWS using Terraform.

- **5.1. Networking (VPC, Subnets, Security Groups):**
    - Define production and non-production VPCs.
    - Create public and private subnets across multiple Availability Zones (AZs) for high availability.
    - Configure NAT Gateways for outbound internet access from private subnets.
    - Set up granular Security Groups and Network ACLs for ingress/egress control.

    - *Terraform Reference:* terraform_infra/modules/vpc (from Deep-Dive Addendum: Cloud Migration + Terraform Snippets.pdf)
- **5.2. S3 Data Lake Setup:**
    - Provision S3 buckets for raw, processed, curated data zones.
    - Implement S3 bucket policies for access control.
    - Enable S3 versioning, lifecycle policies (e.g., transition to infrequent access, glacier), and server-side encryption (SSE-S3, SSE-KMS).
    - *Terraform Reference:* terraform_infra/modules/s3
- **5.3. IAM Roles & Policies:**
    - Define minimal-privilege IAM roles for all AWS services (Lambda, Glue, MWAA) that will interact with S3, MSK, RDS, etc.
    - Implement IAM policies for fine-grained access control to data buckets and services.
    - *Terraform Reference:* Deep-Dive Addendum: Cloud Migration + Terraform Snippets.pdf (IAM Setup section)

Phase 2: Data Ingestion & Streaming

Objective: Migrate the real-time data ingestion pipeline from local FastAPI/Kafka to AWS Lambda/API Gateway and MSK.

- **5.4. Amazon MSK Cluster Provisioning:**
    - Create a multi-AZ Amazon MSK cluster within your private subnets.
    - Configure Kafka topics (e.g., raw_financial_transactions, raw_insurance_claims) with appropriate partitions and replication factors.
    - *Terraform Reference:* terraform_infra/modules/msk
- **5.5. FastAPI Ingestor to AWS Lambda + API Gateway:**
    - Refactor FastAPI endpoints into AWS Lambda functions (e.g., one Lambda for financial, one for insurance ingestion).
    - Deploy an Amazon API Gateway to expose these Lambda functions as public HTTP/S endpoints.
    - Configure Lambda permissions to write to MSK topics and CloudWatch Logs.
    - Implement input validation and error handling within Lambdas.
    - *Terraform Reference:* terraform_infra/modules/lambda_api_gateway
- **5.6. Initial Data Flow Validation:**
    - Use simulate_data.py (modified to hit API Gateway endpoint) to send data.
    - Verify data lands in MSK topics.
    - Monitor Lambda invocations, errors, and duration in CloudWatch.

Phase 3: Data Processing & Transformation

Objective: Migrate Spark batch/streaming jobs from local Spark Cluster to AWS Glue or Amazon EMR.

- **5.7. Spark Jobs to AWS Glue ETL or Amazon EMR:**
    - **Option A: AWS Glue (Serverless ETL):**
        - Convert PySpark jobs into AWS Glue ETL jobs.

        - Configure Glue jobs to read from MSK/S3 raw zone and write to S3 processed/curated zones using Delta Lake.
        - Set up Glue Data Catalog as a metadata store for your Delta Lake tables.
        - *Terraform Reference:* terraform_infra/modules/glue (for Glue jobs)
    - **Option B: Amazon EMR (Managed Spark):**
        - Provision Amazon EMR clusters.
        - Submit Spark jobs as steps on EMR clusters.
        - Configure EMR to use S3 for data and logging, and interact with MSK.
        - *Terraform Reference:* terraform_infra/modules/emr (for EMR clusters)
- **5.8. Database Migration (PostgreSQL & MongoDB):**
    - Migrate postgres to Amazon RDS for PostgreSQL or Aurora PostgreSQL.
    - Migrate mongodb to Amazon DocumentDB or set up MongoDB Atlas on AWS.
    - Update connection strings in Glue/EMR jobs to point to managed AWS databases.
- **5.9. Processed Data Validation:**
    - Verify transformed data lands in the correct S3 curated buckets.
    - Perform data quality checks on curated data.
    - Monitor Glue/EMR job run times, DPU utilization, and error logs.

Phase 4: Orchestration, Observability & Governance

Objective: Migrate Airflow DAGs to MWAA, set up comprehensive AWS-native observability, and integrate cloud data cataloging.

- **5.10. Apache Airflow to Amazon MWAA:**
    - Provision an Amazon MWAA environment.
    - Migrate your Airflow DAGs (.py files) to an S3 bucket connected to MWAA.
    - Update DAGs to use AWS operators (e.g., GlueOperator, EcsRunTaskOperator, S3Sensor) to interact with AWS services.
    - *Terraform Reference:* terraform_infra/modules/mwaa
- **5.11. Observability (CloudWatch, ADOT, Managed Grafana):**
    - Configure AWS CloudWatch Logs for all services (Lambda, MSK, Glue, RDS, DocumentDB, MWAA).
    - Deploy AWS Distro for OpenTelemetry (ADOT) collectors for metrics and traces, similar to your local Grafana Alloy.
    - Set up Amazon Managed Grafana dashboards to visualize metrics from CloudWatch and ADOT.
    - Configure CloudWatch Alarms for critical metrics (e.g., MSK consumer lag, Lambda errors, Glue job failures).
    - *Terraform Reference:* Deep-Dive Addendum: Cloud Migration + Terraform Snippets.pdf (Monitoring & Logging section)
- **5.12. Data Governance (AWS Glue Data Catalog, DataZone, OpenMetadata Integration):**
    - Leverage AWS Glue Data Catalog for automatic schema discovery on S3 data.
    - Evaluate AWS DataZone for unified data discovery and access control.
    - If OpenMetadata is still desired as a central layer, plan for its deployment (e.g., on

ECS/Fargate) and configuration to ingest metadata from Glue Data Catalog and other AWS services.
    - Configure automated metadata ingestion (via MWAA DAGs) to OpenMetadata.
- **5.13. End-to-End System Testing & Optimization:**
    - Perform comprehensive integration testing across all migrated components.
    - Conduct performance and load testing using AWS Batch-provisioned Locust.
    - Monitor costs closely and optimize resource allocation (e.g., Glue DPU counts, MSK instance types, Lambda memory).

Phase 5: Operational Excellence & Future State

Objective: Establish robust operational procedures and plan for continuous improvement.

- **5.14. Disaster Recovery Planning & Runbooks:**
    - Formalize RPO and RTO for each data component.
    - Develop and test DR runbooks for critical scenarios (e.g., MSK region failover, RDS point-in-time recovery).
    - Leverage Multi-AZ deployments, S3 cross-region replication, and automated backups.
    - *Reference:* Deep-Dive Addendum: DR & Runbooks.pdf
- **5.15. Security Hardening:**
    - Regular security audits and vulnerability assessments.
    - Data encryption at rest (S3 SSE, RDS/DocumentDB encryption) and in transit (TLS for MSK, API Gateway).
    - Secrets management with AWS Secrets Manager.
    - Network segmentation and least-privilege IAM policies.
- **5.16. Cost Management:**
    - Continuous monitoring using AWS Cost Explorer and Reserved Instances/Savings Plans.
    - Rightsizing resources based on actual usage patterns.
    - Automated shutdown/startup for non-production environments.
- **5.17. Ongoing Maintenance & Improvements:**
    - Regular updates to IaC (Terraform).
    - Refinement of CI/CD pipelines.
    - Exploration of new AWS services and features.

6. Key Deliverables at Each Phase

- **Phase 1:** Deployed VPC, S3 Buckets, foundational IAM roles via Terraform.
- **Phase 2:** Functional Lambda/API Gateway ingestion, data flowing to MSK.
- **Phase 3:** Working Glue/EMR jobs transforming data to S3 curated, databases migrated.
- **Phase 4:** MWAA orchestrating pipelines, comprehensive CloudWatch/Grafana dashboards, initial metadata catalog in Glue/OpenMetadata.
- **Phase 5:** Documented DR plan, security hardening report, cost optimization report.

This action plan provides a structured roadmap for your data platform's journey to AWS,

ensuring a controlled, secure, and efficient migration to a production-ready cloud environment.