Data Platform Troubleshooting, Scaling, and Operational Reference Guide

This document serves as a comprehensive reference for identifying and resolving common issues within your enterprise data platform, along with practical guidelines for scaling its various components. It integrates insights from the "Deep-Dive Addendum: Testing & Observability Patterns," "Deep-Dive Addendum: DR & Runbooks," and other related documentation to provide actionable advice for maintaining a robust and performant data ecosystem.

1. General Troubleshooting Principles

Before diving into component-specific issues, consider these general troubleshooting steps:

- **Check Logs First:** The most critical step. Examine the logs of the affected service and any upstream/downstream dependencies. Look for error messages, warnings, and unusual patterns.
- **Verify Service Health:** Use docker compose ps (local) or AWS console/CLI (cloud) to ensure all related services are running and healthy. Check their healthcheck status.
- **Network Connectivity:** Ensure services can communicate with each other (e.g., Kafka to Spark, FastAPI to Kafka). Use ping or telnet from within containers if necessary.
- **Resource Utilization:** Monitor CPU, memory, and disk I/O using Grafana/CloudWatch dashboards (cAdvisor locally, CloudWatch Container Insights in AWS). High utilization can indicate a bottleneck.
- **Dependencies:** Confirm all prerequisite services are up and functioning correctly.
- **Idempotency:** For jobs or processes, ensure they can be re-run safely without causing data duplication or corruption.

2. Component-Specific Troubleshooting & Scaling

This section addresses common problems and scaling strategies for each core component of your data platform.

2.1. FastAPI Ingestor (Local) / AWS Lambda + API Gateway (Cloud)

Common Problems & Resolutions:

- **Q: Data not appearing in Kafka/MSK, but API returns 200 OK.**
    - **A:** Check FastAPI ingestor logs for internal errors (e.g., Kafka producer issues, serialization errors). Verify Kafka/MSK broker connectivity from the ingestor.
- **Q: High latency or timeouts on API calls.**
    - **A:** Monitor FastAPI/Lambda concurrency and resource usage. Optimize API logic, especially any synchronous blocking operations. For Lambda, increase memory allocation. Check network path to Kafka/MSK.

- **Q: Invalid data being ingested despite client-side validation.**
    - **A:** Implement robust server-side input validation (e.g., Pydantic models in FastAPI, schema validation in Lambda). Return appropriate HTTP 4xx error codes.
- **Q: Too many open connections/file handles.**
    - **A:** Ensure proper connection pooling and resource management within the FastAPI application.

Scaling Guidelines:

- **Local (FastAPI):**
    - **Vertical Scaling:** Increase cpus and mem_limit in docker-compose.yml.
    - **Horizontal Scaling:** Run multiple fastapi-ingestor replicas, potentially fronted by a load balancer (e.g., Nginx, or Docker Swarm's built-in load balancing if deploying there).
- **Cloud (AWS Lambda + API Gateway):**
    - **Automatic Scaling:** Lambda automatically scales concurrency based on demand. Optimize Lambda memory/CPU and duration.
    - **API Gateway:** Handles scaling of API endpoints. Throttling and caching can be configured to protect backend services.
    - **Refinement:** Monitor Lambda cold starts; optimize dependencies to minimize package size.

2.2. Apache Kafka (Local) / Amazon MSK (Cloud)

Common Problems & Resolutions:

- **Q: High Kafka consumer lag (messages are piling up in Kafka).**
    - **A:** This is a critical indicator of downstream processing bottlenecks.
        1. **Identify the lagging consumer:** Check consumer group offsets (e.g., kafka-consumer-groups.sh locally, CloudWatch metrics for MSK).
        2. **Troubleshoot the consumer:** Examine consumer application logs (e.g., Spark Structured Streaming job logs) for errors, pauses, or slow processing.
        3. **Scale the consumer:** Increase the number of consumer instances or partitions if the consumer is under-provisioned.
- **Q: Producer throughput issues (messages not being published fast enough).**
    - **A:** Check producer application logs. Monitor broker CPU, network I/O, and disk I/O. Consider increasing Kafka topic partitions.
- **Q: Broker unreachability or frequent restarts.**
    - **A:** Check Zookeeper/Kafka broker logs. Verify network connectivity between brokers and Zookeeper. Ensure sufficient resources (CPU, memory, disk). For MSK, check service health in the AWS console.

Scaling Guidelines:

- **Local (Kafka):**
    - **Horizontal Scaling:** Add more Kafka broker services (kafka-2, kafka-3) and

    update KAFKA_BROKER_ID and KAFKA_ADVERTISED_LISTENERS accordingly.
  - **Partitioning:** Increase the number of partitions for your Kafka topics (kafka-topics.sh). More partitions allow for greater parallelism in consumers.
- **Cloud (Amazon MSK):**
  - **Cluster Scaling:** MSK allows scaling the number and type of brokers in your cluster without downtime.
  - **Topic Partitions:** Dynamically increase topic partitions to handle higher throughput.
  - **Auto-Scaling (Broker Type/Number):** MSK offers auto-scaling features based on cluster utilization.

2.3. Apache Spark (Local) / AWS Glue ETL / Amazon EMR (Cloud)

Common Problems & Resolutions:

- **Q: Spark job failures (e.g., TaskFailed exceptions, OOM errors).**
  - **A:**
    1. **Examine Spark UI (18080 locally) / CloudWatch Logs for Glue/EMR:** Look at the "Stages" and "Tasks" tabs to identify where the failure occurred.
    2. **Review driver/executor logs:** Search for specific exception messages (e.g., java.lang.OutOfMemoryError).
    3. **Optimize code:** Address data skew, large shuffles, or inefficient transformations. Use repartition() carefully.
    4. **Increase resources:** Allocate more memory (spark.executor.memory, spark.driver.memory) or cores (spark.executor.cores) to executors/driver.
- **Q: Slow Spark job performance.**
  - **A:**
    1. **Data Skew:** Identify if data is unevenly distributed across partitions. Use techniques like salting or broadcast joins.
    2. **Shuffle Spills:** High shuffle read/write indicates inefficient data movement. Optimize joins and aggregations.
    3. **Small Files:** Many small files in Delta Lake can lead to overhead. Use OPTIMIZE and ZORDER for compaction.
    4. **Resource Bottleneck:** Monitor CPU and memory utilization of Spark workers/executors. Increase resources if consistently high.
- **Q: Schema evolution issues with Delta Lake.**
  - **A:** Ensure your Spark write operations handle schema evolution appropriately (e.g., spark.write.format("delta").mode("append").option("mergeSchema", "true").save(...)). Understand when overwriteSchema is appropriate vs. mergeSchema.

Scaling Guidelines:

- **Local (Spark):**

    - **Vertical Scaling:** Increase SPARK_WORKER_CORES and SPARK_WORKER_MEMORY for spark-worker-1 (and any additional workers).
    - **Horizontal Scaling:** Add more spark-worker services in docker-compose.yml.
- **Cloud (AWS Glue ETL / Amazon EMR):**
    - **AWS Glue:** Scales automatically with DPUs (Data Processing Units). Adjust NumberOfWorkers and WorkerType (e.g., G.1X, G.2X) based on workload. Glue is serverless, so no cluster management.
    - **Amazon EMR:** Scale out by adding more EC2 instances (nodes) to your EMR cluster. Choose appropriate instance types for worker and master nodes. EMR allows dynamic scaling and step-based execution.

2.4. PostgreSQL / MongoDB (Local) / Amazon RDS / DocumentDB (Cloud)

Common Problems & Resolutions:

- **Q: Database connection issues or timeouts.**
    - **A:** Verify database service health. Check network connectivity from application. Ensure correct credentials and hostnames.
- **Q: Slow query performance.**
    - **A:**
        1. **Identify slow queries:** Enable slow query logs.
        2. **Indexing:** Create appropriate indexes on frequently queried columns.
        3. **Query Optimization:** Rewrite inefficient queries, avoid N+1 queries.
        4. **Resource Contention:** Check database CPU, memory, and I/O utilization.
- **Q: Disk space running low.**
    - **A:** Monitor disk usage. Implement data retention policies. Archive old data.
- **Q: Data corruption (rare for managed services).**
    - **A:** Restore from a recent backup. For MongoDB, ensure replica set integrity.
    - *Reference:* Deep-Dive Addendum: DR & Runbooks.pdf for database recovery.

Scaling Guidelines:

- **Local (PostgreSQL, MongoDB):**
    - **Vertical Scaling:** Increase cpus and mem_limit in docker-compose.yml.
    - **MongoDB Replica Set:** Already configured in your docker-compose.yml for basic high availability. For true scale-out, you'd need to set up sharding across multiple instances.
- **Cloud (Amazon RDS / DocumentDB):**
    - **Vertical Scaling:** Easily scale up instance types (CPU, memory) for RDS/DocumentDB.
    - **Read Replicas:** Create read replicas to offload read traffic from the primary instance.
    - **Multi-AZ Deployments:** For high availability and automatic failover.

    - **MongoDB Sharding (DocumentDB/Atlas):** For very large datasets and high write throughput, implement sharding.

2.5. Apache Airflow (Local) / Amazon MWAA (Cloud)

Common Problems & Resolutions:

- **Q: DAGs not appearing in UI.**
    - **A:** Check Airflow scheduler and webserver logs. Ensure DAG files are in the correct dags volume. Verify permissions.
- **Q: DAGs stuck in running state or tasks failing unexpectedly.**
    - **A:**
        1. **Check task logs:** The most important step. Airflow UI provides direct access to logs.
        2. **Dependencies:** Ensure all external dependencies (Spark, Kafka, databases, MinIO) are healthy and reachable.
        3. **Airflow Metastore:** Check PostgreSQL database for any issues.
        4. **Worker Health:** Verify airflow-worker service is healthy and connected to Redis/PostgreSQL.
- **Q: Airflow UI unresponsive or slow.**
    - **A:** Monitor airflow-webserver resources. Check network latency to the webserver. Ensure PostgreSQL is not bottlenecked.
- **Q: airflow-init failing.**
    - **A:** Ensure PostgreSQL and Redis are fully healthy *before* running airflow-init. Check the airflow-init container logs for specific errors during DB migration or user creation.

Scaling Guidelines:

- **Local (Airflow):**
    - **Workers:** Increase the number of airflow-worker replicas. Celery Executor automatically distributes tasks.
    - **Scheduler Concurrency:** Adjust AIRFLOW__CORE__MAX_ACTIVE_TASKS_PER_DAG and AIRFLOW__CORE__MAX_ACTIVE_RUNS_PER_DAG to control parallelism.
    - **Metastore:** Ensure PostgreSQL has sufficient resources.
- **Cloud (Amazon MWAA):**
    - **Environment Scaling:** MWAA handles scaling of the webserver, scheduler, and workers automatically based on demand. You can configure minimum and maximum worker counts.
    - **MWAA Environment Size:** Choose an appropriate environment size (e.g., Small, Medium, Large) based on your workload.
    - **Database:** MWAA's metastore database is fully managed.

2.6. MinIO (Local) / Amazon S3 (Cloud)

Common Problems & Resolutions:

- **Q: Access denied errors when reading/writing to MinIO/S3.**
  - **A:** Verify MINIO_ROOT_USER/MINIO_ROOT_PASSWORD (local) or IAM roles/policies (cloud) are correctly configured and have necessary permissions. Check S3 bucket policies.
- **Q: Slow read/write performance.**
  - **A:**
    1. **Many Small Files:** Combine small files into larger ones (e.g., during Spark transformations or using OPTIMIZE for Delta Lake) to reduce object overhead.
    2. **Network Bandwidth:** Check network bandwidth between your compute and storage.
    3. **Concurrent Access:** Ensure read/write patterns are optimized for object storage (e.g., using S3's strong consistency for Delta Lake).
- **Q: MinIO console/API unresponsive.**
  - **A:** Check MinIO container logs, ensure sufficient resources on the host machine.

Scaling Guidelines:

- **Local (MinIO):**
  - **Vertical Scaling:** Ensure the host machine has enough disk I/O, CPU, and memory. MinIO itself can scale, but in a docker-compose setup, it's typically a single instance relying on host resources.
- **Cloud (Amazon S3):**
  - **Automatic Scaling:** S3 is inherently designed for massive scale and handles throughput and storage automatically.
  - **Performance Optimization:** Use S3 Transfer Acceleration for faster uploads, choose appropriate storage classes, and optimize read/write patterns for analytical queries (e.g., Parquet/ORC with columnar queries via Athena/Spectrum).

2.7. Observability Stack (Grafana, Grafana Alloy/ADOT, cAdvisor/CloudWatch)

Common Problems & Resolutions:

- **Q: Metrics not appearing in Grafana dashboards.**
  - **A:**
    1. **Verify Data Source:** In Grafana, check if the Prometheus/CloudWatch data source is correctly configured and reachable.
    2. **Collector Health:** Ensure Grafana Alloy/ADOT agent is running and configured to scrape/collect metrics from the correct targets. Check its logs.

        3. **Scraping Targets:** For Prometheus/Alloy, verify that prometheus.yml/grafana-alloy.river correctly lists the job_name/targets for each service (FastAPI, Kafka, Spark, etc.).
        4. **Service Exporter:** Confirm that the service itself is exposing metrics on the expected endpoint and port (e.g., /metrics for FastAPI, 8080/metrics for cAdvisor).
- **Q: Dashboard loading slowly or unresponsive.**
    - **A:** Optimize dashboard queries in Grafana. Ensure Prometheus/CloudWatch is healthy and has sufficient resources to process queries.
- **Q: Too many alerts (alert fatigue).**
    - **A:** Refine alert rules, adjust thresholds, and implement notification throttling. Ensure alerts are actionable and contextual.

Scaling Guidelines:

- **Local (Grafana, Alloy, Prometheus, cAdvisor):**
    - **Vertical Scaling:** Increase cpus and mem_limit for Prometheus and Grafana, especially if collecting many metrics or running complex queries.
    - **Alloy/cAdvisor:** These are agents and typically run per host/container. Ensure they have enough resources to collect and forward data.
- **Cloud (Amazon Managed Grafana, CloudWatch, ADOT):**
    - **Automatic Scaling:** Most AWS monitoring services (CloudWatch, Managed Grafana) scale automatically to handle metric ingest and querying.
    - **ADOT Collectors:** Deploy ADOT collectors as DaemonSets in Kubernetes or on EC2 instances to scale with your application instances.

2.8. OpenMetadata / Spline (Local) / AWS Glue Data Catalog / DataZone (Cloud)

Common Problems & Resolutions:

- **Q: Metadata not ingesting into OpenMetadata.**
    - **A:**
        1. **Check Ingestion Logs:** Review logs of the openmetadata-ingestion service (or the Airflow DAG running ingestion). Look for connection errors to data sources (Kafka, Delta Lake) or to the OpenMetadata server/OpenSearch.
        2. **OpenMetadata Server/OpenSearch Health:** Ensure openmetadata-server, openmetadata-mysql, and openmetadata-opensearch are all healthy and reachable.
        3. **Ingestion Configuration:** Verify the JSON configuration for the ingestion pipeline is correct (credentials, hostnames, service names).
- **Q: Spline lineage not appearing for Spark jobs.**
    - **A:**
        1. **Spark Integration:** Ensure the Spline agent JAR is correctly configured in

your Spark job's --packages or --jars (if self-managed Spark). Verify spark.spline.enable is true.
2. **Spline REST Server:** Ensure spline-rest is healthy and reachable from the Spark cluster.
3. **Spark Job Completion:** Lineage is typically published after Spark job completion.

- **Q: UI issues (OpenMetadata, Spline UI).**
  - **A:** Check the respective UI container logs. Ensure the backend REST APIs are healthy and reachable.

Scaling Guidelines:

- **Local (OpenMetadata, Spline):**
  - **Vertical Scaling:** Increase cpus and mem_limit for openmetadata-server, openmetadata-opensearch, and spline-rest if they become resource-constrained.
  - **Database:** Ensure openmetadata-mysql has adequate resources.
- **Cloud (AWS Glue Data Catalog, DataZone, OpenMetadata Integration):**
  - **Automatic Scaling:** AWS Glue Data Catalog and DataZone are managed services that scale automatically.
  - **OpenMetadata Deployment:** If self-hosting OpenMetadata in AWS, consider deploying it on Amazon ECS or Kubernetes (EKS) for scalable and resilient deployment. Ensure the underlying database (RDS for MySQL) and search engine (OpenSearch Service) are also scaled appropriately.

3. General Scaling Considerations

- **Horizontal vs. Vertical Scaling:**
  - **Vertical (Scale Up):** Increase resources (CPU, memory) of an existing instance. Easier to implement but has limits. Good for initial optimization.
  - **Horizontal (Scale Out):** Add more instances of a service. Provides greater fault tolerance and often better long-term scalability. More complex to manage state and coordination.
- **Bottleneck Identification:** Always identify the actual bottleneck before scaling. Scaling the wrong component won't improve performance. Use your observability tools (Grafana/CloudWatch) to pinpoint the bottleneck (e.g., high CPU on Spark workers, high Kafka consumer lag).
- **Cost Implications:** Understand the cost impact of scaling decisions in the cloud (e.g., more Glue DPUs, larger EC2 instances, more MSK brokers).
- **Automated Scaling:** In cloud environments, leverage auto-scaling groups for EC2, auto-scaling for MSK/RDS/DocumentDB, and serverless scaling for Lambda/Glue to dynamically adjust resources based on demand.
- **Network Bandwidth:** Ensure sufficient network bandwidth between services, especially for high-throughput data pipelines (e.g., Kafka to Spark, Spark to S3).

This guide is intended as a living document. Continuous monitoring, regular reviews of performance metrics, and learning from incidents will further refine your operational practices and scaling strategies.