Deep-Dive Addendum: DR & Runbooks

This addendum focuses on Disaster Recovery (DR) strategies and the creation of detailed runbooks, critical components for ensuring the resilience and rapid recovery of your enterprise data platform. These practices minimize downtime and data loss in the face of unforeseen incidents.

6.1. RPO and RTO in Context

Understanding and defining your Recovery Point Objective (RPO) and Recovery Time Objective (RTO) are fundamental to a robust disaster recovery strategy.

- **RPO (Recovery Point Objective):** The maximum tolerable amount of data (measured in time) that can be lost after a disaster. It determines the frequency of your data backups or replication.
    - *Example:* If your RPO is 1 hour, you can afford to lose at most 1 hour of data. This means your backups or replication must occur at least hourly.
    - **Implications for Data Platform:**
        - **Streaming Data (Kafka/MSK):** For critical real-time data, RPO should ideally be near zero (minutes or seconds). This is achieved through continuous replication and durable message storage.
        - **Batch Data (Delta Lake/S3):** RPO can be longer (hours or days) depending on criticality, often achieved through frequent snapshots or replication of the object storage.
        - **Databases (PostgreSQL/MongoDB/RDS/DocumentDB):** RPO depends on the business impact of data loss, typically ranging from seconds (through transaction logs/replication) to hours (through daily backups).
- **RTO (Recovery Time Objective):** The maximum tolerable duration of time within which a business process must be restored after a disaster to avoid unacceptable consequences.
    - *Example:* If your RTO is 4 hours, your entire data platform (or critical components) must be fully operational within 4 hours of a disaster being declared.
    - **Implications for Data Platform:**
        - **Automation:** Achieving low RTO requires significant automation in infrastructure provisioning (IaC), application deployment (CI/CD), and data recovery processes.
        - **Testing:** Regular DR drills and testing are essential to validate that the RTO can indeed be met.
        - **Compute Provisioning:** Rapid provisioning of compute resources (e.g., EC2, EMR, Glue) for data processing in a DR scenario.
        - **Database Spin-up:** Quick restoration or failover of databases.

6.2. Backup & Restore Verification

A backup strategy is only as good as its restore capabilities. Regular verification is non-negotiable.

- **Strategy Components:**
  - **Database Backups:**
    - **PostgreSQL/RDS:** Automated snapshots, point-in-time recovery (PITR), or logical backups (pg_dump).
    - **MongoDB/DocumentDB:** Automated backups, replica set snapshots, or oplog-based recovery.
  - **Object Storage (MinIO/S3):** Versioning, cross-region replication (for S3), or daily/hourly snapshots of critical data lake paths.
  - **Kafka/MSK:** While Kafka itself stores messages durably for a configured retention period, for long-term archival or disaster recovery, external backup mechanisms (e.g., Kafka Connect to S3, or replication to another cluster) might be needed. Consumer offsets also need to be managed.
  - **Application Code & Configuration:** Version-controlled in Git, automatically deployed via CI/CD. Docker images should be stored in registries (e.g., ECR, Docker Hub).
- **Verification Procedures:**
  - **Automated Validation:** Implement automated jobs (e.g., Airflow DAGs) that periodically:
    - Restore a recent backup to a temporary, isolated environment.
    - Run data quality checks (using Great Expectations) on the restored data.
    - Perform smoke tests on critical data pipelines using the restored data.
    - Compare a sample of restored data with the source data (if feasible).
  - **Manual Spot Checks:** Regularly restore a small, critical dataset manually to ensure the process is well-documented and human-executable.
  - **DR Drills:** Conduct full-scale DR drills at least annually, simulating a disaster scenario (e.g., region outage) and executing the entire recovery playbook to validate RTO and RPO. Document any discrepancies and refine the plan.

6.3. Runbook Templates for Critical Systems

Runbooks are step-by-step guides for executing operational procedures, especially crucial during incidents or disaster recovery. They ensure consistency, reduce cognitive load under stress, and enable efficient handovers.

Purpose of a Runbook:

- **Standardization:** Provides a consistent approach to common operational tasks and incident response.
- **Efficiency:** Reduces the time to resolve incidents by providing clear, pre-defined steps.
- **Knowledge Transfer:** Documents tribal knowledge, making operations less reliant on specific individuals.
- **Reduced Stress:** Offers a calm, structured guide during high-pressure situations.

Core Runbook Sections (Template):

Runbook: [Descriptive Title, e.g., "Restore Financial Transactions Delta Lake Table"]

1. Overview

* **Purpose:** [Briefly explain the goal of this runbook, e.g., "To restore the financial_transactions Delta Lake table from a daily backup due to data corruption or accidental deletion."]
* **Affected System(s):** [List systems, e.g., "financial_transactions Delta Lake table (Raw and Curated zones), downstream BI reports, data analysts."]
* **Trigger Condition:** [When should this runbook be used? e.g., "Detection of data integrity issues in financial_transactions table; accidental `DROP TABLE` or `DELETE`."]
* **RTO/RPO Impact:** [How does following this runbook affect RTO/RPO? e.g., "Expected RTO: 2 hours. Expected RPO: 24 hours (based on last daily snapshot)."]
* **Severity:** [e.g., "High (P1-P2) - Immediate business impact, data unavailability."]
* **Owner:** [Team/Individual responsible for this system, e.g., "Data Platform Team - Core Data Engineering"]

2. Pre-requisites & Preparation

* **Access:**
  * AWS Console access (Administrator/PowerUser IAM Role).
  * AWS CLI configured with appropriate credentials.
  * Access to Terraform state for `terraform_infra/environments/prod`.
  * SSH access to EMR/EC2 instances (if applicable).
  * Grafana/CloudWatch access for monitoring.
* **Tools:**
  * `terraform` CLI installed.
  * `aws cli` installed.
  * `kubectl` or `ecs-cli` (if containerized).
  * Python environment with `boto3`, `pyspark`, `delta-spark` installed.
* **Information Needed:**
  * Last known good S3 bucket path/prefix for the Delta table backup.
  * AWS Region.
  * Specific `transaction_id` or timestamp of corrupted data (if known).
  * Relevant Jira/Incident ticket number.

3. Execution Steps

IMPORTANT: Follow steps sequentially. Document all commands and outputs in the incident ticket.

3.1. Incident Declaration & Communication

1. **Declare Incident:** Notify relevant stakeholders via [Communication Channel, e.g., Slack channel #data-incidents, PagerDuty].
2. **Create Incident Ticket:** Log a new incident in Jira/ServiceNow with title "[INCIDENT] - [Brief Title]".
3. **Establish Bridge/War Room:** Create a dedicated communication channel (e.g., Zoom/Slack Huddle) for the incident team.

3.2. Isolate & Confirm Scope

1. **Stop Downstream Consumers:** Temporarily halt all Spark jobs and other consumers reading from the `financial_transactions` curated Delta table.
   * *Command (Airflow):* Pause relevant DAGs (`data_transformation_dag`).
   * *Command (EMR/Glue):* Terminate active Spark jobs or stop Glue triggers.
2. **Verify Data Corruption/Loss:**
   * Connect to Spark environment (EMR/Glue/local).
   * Attempt to read the corrupted Delta table:
     ```python
     # PySpark snippet to read the table

spark.read.format("delta").load("s3://your-curated-bucket/financial_transactions/").show()
     ```
   * Confirm corruption/missing data (e.g., `SELECT COUNT(*)` vs. expected, check for nulls, garbled data).
3. **Identify Last Known Good State:**
   * Review S3 bucket versions/backups for `financial_transactions` Delta table.
   * Consult data quality reports or audit logs for the last successful data load.

3.3. Recovery Procedure

1. **Navigate to Backup Location:**
   * Identify the S3 path of the last healthy backup or a specific version to restore.
   * Example: `s3://your-backup-bucket/daily_snapshots/financial_transactions_2023-10-25/`
2. **Restore Data:**
   * **Option A: Delta Lake Time Travel (if applicable & sufficient):** If the corruption is recent and within your Delta Lake retention, simply query a previous version.
     ```python
     # PySpark: Read a specific version
     df = spark.read.format("delta").option("versionAsOf", <version_number>).load("s3://your-curated-bucket/financial_transactions/")
     # Then, if needed, overwrite the current table with the restored data
     ```

df.write.format("delta").mode("overwrite").save("s3://your-curated-bucket/financial_transactions/")
      ```
   * **Option B: Copy from Backup to Active Path:** If time travel is not feasible or the backup is external.
      * **Important:** Consider renaming/moving the current corrupted table first (e.g., `s3 mv s3://current-table/ s3://corrupted-backup/`).
      * Copy the good data:
        ```bash
        # AWS CLI: Copy entire directory
        aws s3 cp s3://your-backup-bucket/daily_snapshots/financial_transactions_2023-10-25/ s3://your-curated-bucket/financial_transactions/ --recursive
        ```
3. **Run Post-Restore Data Quality Checks:**
   * Execute targeted Great Expectations suites or custom validation scripts on the restored `financial_transactions` table.
   * Verify key metrics: record count, sum of amounts, absence of nulls in critical columns.
4. **Resync Dependent Systems:**
   * If any downstream systems consume directly from this Delta table, trigger a refresh or full reload for them.
   * Restart all downstream Spark jobs/Airflow DAGs that were paused in step 3.1.
   * Monitor their catch-up process.

3.4. Verification & Handover

1. **Monitor System Health:**
   * Verify all relevant Grafana dashboards (e.g., Delta Lake health, Spark job health, API throughput) show normal operation.
   * Check for new errors or warnings in CloudWatch/Grafana logs.
2. **Confirm Data Flow:**
   * Send a few test transactions through the FastAPI API.
   * Verify they flow through Kafka and Spark and appear correctly in the Delta table.
3. **Communicate Resolution:**
   * Update incident ticket with resolution details.
   * Notify stakeholders of service restoration.
4. **Clean Up:** Remove any temporary files or resources used during recovery.

4. Post-Incident Analysis

* **Schedule Post-Mortem (within 24-48 hours):** Conduct a blameless post-mortem meeting to identify root causes, contributing factors, and action items. (See Section 5.6.3 for template).
        ```

* **Update Documentation:** Update this runbook, relevant architectural diagrams, and data quality checks based on lessons learned.
* **Improve Monitoring/Alerting:** Adjust SLIs/SLOs or add new alerts to prevent recurrence or improve detection.
* **Automate More:** Identify manual steps in the recovery process that can be automated in future iterations.

Appendix G: Disaster Recovery (DR) Runbook Examples

This appendix provides more detailed examples of common DR runbooks.

DR Runbook Example: Kafka Cluster Failover (Conceptual)

* **Purpose:** To failover to a secondary Kafka cluster (e.g., in a different AWS region or a mirrored cluster) in case of a primary cluster outage.
* **Key Steps:**
    1. **Confirm Primary Cluster Outage:** Verify using Kafka tools (`kafka-topics.sh --describe`) and monitoring dashboards.
    2. **Update DNS/Client Configurations:** Change application configurations (FastAPI, Spark) to point to the secondary Kafka cluster's bootstrap servers.
    3. **Verify New Message Production:** Ensure FastAPI (or other producers) is successfully publishing to the secondary cluster.
    4. **Verify Consumer Offset Migration/Reset:** Decide if consumer offsets need to be migrated or if consumers should start from the earliest available offset on the secondary cluster (this implies data loss up to that point).
    5. **Restart Consumers:** Restart Spark streaming jobs and other Kafka consumers to connect to the secondary cluster.
    6. **Monitor Lag and Data Flow:** Ensure consumers are catching up and data is flowing as expected.
    7. **Failback (Optional):** Plan and execute a controlled failback to the primary cluster once it's restored and stable.

DR Runbook Example: Critical Database Restoration (Conceptual)

* **Purpose:** To restore a critical database (e.g., PostgreSQL for application metadata or Airflow) from a backup.
* **Key Steps:**
    1. **Identify Recovery Point:** Determine the desired point in time for restoration based on RPO and known good state.
    2. **Spin Up New Database Instance:** Provision a new RDS instance or Docker container for PostgreSQL. *Do NOT restore over the corrupted production instance directly.*
    3. **Perform Restore:** Use AWS RDS snapshot restore, PITR, or `pg_restore` from a logical backup file.

4. **Validate Data Integrity:** Run schema checks and sample data queries to confirm successful restoration.
5. **Update Application Configuration:** Modify application (FastAPI, Airflow) database connection strings to point to the newly restored instance.
6. **Restart Applications:** Restart services dependent on the database.
7. **Monitor:** Check application logs and database metrics for healthy operation.

DR Runbook Example: Airflow Metastore Database Recovery (Conceptual)

* **Purpose:** Recover the Airflow metastore database (PostgreSQL in this case) from corruption or loss. This is critical as Airflow DAGs and their states are stored here.
* **Key Steps:**
1. **Stop Airflow Components:** Halt all Airflow scheduler, webserver, and worker instances.
2. **Restore Metastore Database:** Follow the "Critical Database Restoration" runbook (above) specifically for the Airflow metastore.
3. **Verify Database Integrity:** Run `airflow db check` or manually inspect tables for data.
4. **Start Airflow Metastore First:** Bring up *only* the PostgreSQL service for Airflow.
5. **Run `airflow db upgrade`:** If necessary, run this command to ensure schema is up-to-date with Airflow version.
6. **Start Airflow Scheduler and Webserver:** Bring up the core Airflow components.
7. **Inspect DAGs:** Verify all DAGs are visible, and their last run states are consistent with the restored point. DAGs might need to be re-run or marked as successful if the state is lost.
8. **Start Airflow Workers:** Bring up the worker processes.
9. **Monitor:** Check Airflow UI and logs for any issues.