**Project Title : CDM Modenization**

**Project Overview**

This project aims to **migrate approximately 2000 Informatica PowerCenter workflow mappings** to scalable, maintainable **PySpark-based ETL pipelines**, and deploy them securely and efficiently within the **AWS Cloud ecosystem**. The migration addresses technical debt, cloud modernization, and operational efficiency by shifting from a legacy ETL platform to an open, cloud-native, big data processing framework.

**Project Goal:** Migrate ~2,000 existing Informatica PowerCenter ETL mappings into modern, **scalable PySpark-based data pipelines** deployed on AWS cloud infrastructure.

**Major Objectives:**

* Eliminate **technical debt** associated with the legacy PowerCenter platform (outdated code, high maintenance overhead).
* Enable a **cloud-native architecture** on AWS for improved scalability, agility, and easier infrastructure management.
* **Automate and streamline ETL workflows** to reduce manual effort, minimize errors, and improve data processing efficiency.

**Project Phases:**

* **Discovery:**

Inventory all existing PowerCenter mappings and workflows, assess their complexity, and identify dependencies. This phase establishes a clear migration roadmap and success criteria.

* **Conversion:**

Convert the 2,000 Informatica mappings into PySpark code equivalents. Uses an automated approach to translate mapping logic into PySpark, ensuring the new pipelines replicate the old workflows’ functionality with minimal manual recoding.

* **Validation:**

Rigorously **test and validate** each PySpark pipeline against the current PowerCenter output. Ensure data accuracy, performance benchmarks, and edge-case handling match or exceed the legacy system. Any discrepancies are fixed to guarantee the new system’s reliability before going live.

* **Framework Development:**

Develop a reusable **PySpark ETL framework** and coding standards to support the converted mappings. This includes building common modules for logging, error handling, and configuration management, providing a consistent structure for all new pipelines.

* **Cloud Modernization:**

Set up the AWS environment and deploy the PySpark pipelines in the cloud. Leverage AWS services for scheduling, orchestration, and scaling (for example, using AWS EMR or AWS Glue for Spark execution and AWS Step Functions or CloudWatch for workflow orchestration) to run the data pipelines efficiently.

* **Cutover:**

Gradually switch over (or “cut over”) from the old Informatica jobs to the new AWS-based PySpark pipelines in production. This phase involves running parallel jobs if needed, final data reconciliation, and then decommissioning the legacy environment once the new pipelines are confirmed stable. A careful cutover plan minimizes downtime and business disruption.

**Approaches for Converting Informatica Mappings to PySpark:**

Migrating more than 2,000 Informatica workflow mappings to PySpark is a complex but essential step toward modernizing data infrastructure. Several viable methods exist—ranging from manual rewrites to fully automated, no-code tools—each offering different levels of automation, development effort, and technical flexibility. The following summary outlines the most practical and widely adopted approaches for this transformation.

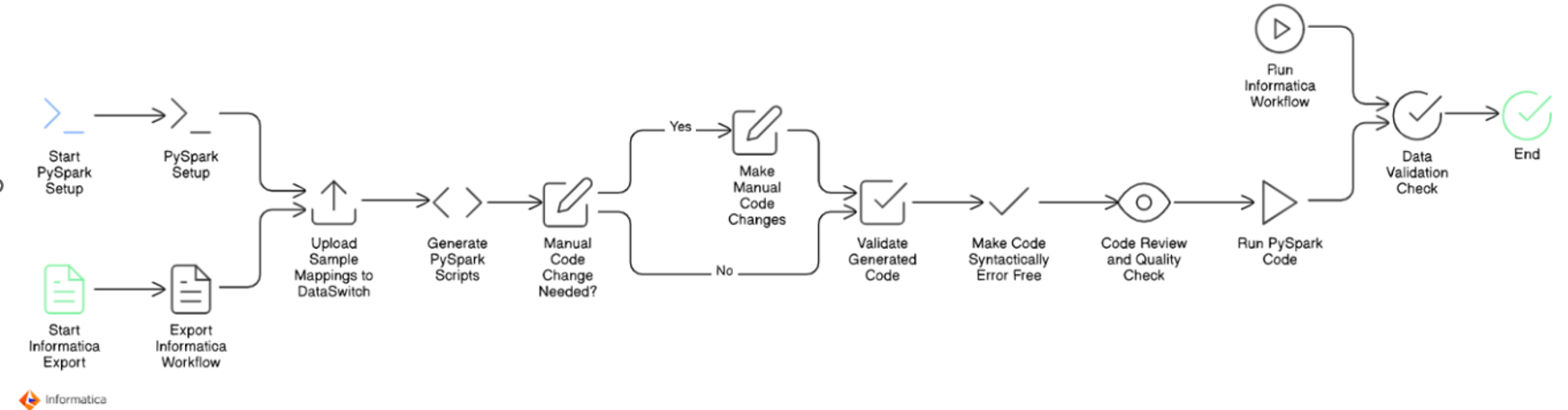
|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Option Name | Description | Best For | Automation Level (Est.) | Pros | Cons |
| **Manual Coding** | Fully manual rewrite of each Informatica mapping in PySpark by a developer. | Small volumes, one-off critical jobs | ~0% | Full control, precise logic replication | Not scalable, time-consuming, high cost |
| **Custom XML-to-PySpark Parser** | Internal tool reads Informatica XML and generates PySpark scripts using business rules. | Orgs with common mapping patterns and in-house engineering | ~50% | Tailored to your needs, reusable | Medium effort to build and maintain |
| **DSL-Based Code Generator** | Convert Informatica logic into JSON/YAML and use templates to generate PySpark. | Teams with strong metadata discipline and repeatable transformations | ~60% | Decouples logic from code, reusable templates | Requires upfront design of DSL and mappings |
| **LLM/AI-Assisted Code Generation** | Use LLMs like GPT to convert mappings from XML to PySpark using predefined patterns. | Teams open to AI experimentation and rapid prototyping | ~80-90% | High automation potential, fast output | Needs human QA, may miss complex edge cases |
| **No-Code Converter Tool (DataSwitch)** | Commercial no-code tool to convert Informatica XML to PySpark with configurable logic. | Enterprises needing speed with moderate customization | ~50-60% (DataSwitch specific) | High automation, minimal coding | usage cost |
| **Excel-Based Mapping Templates** | Use Excel to document mapping logic and auto-generate PySpark via Python scripts. | Data teams comfortable with low-code interfaces | ~40% | Business-user friendly, simple to implement | Not ideal for complex transformations |
| **Internal GUI Mapping Tool** | Build a web UI (e.g., Streamlit) where users define mappings, and backend generates PySpark. | Orgs needing user-friendly control over transformation logic | ~70% | Semi-automated, UI-driven | High upfront development effort |
| **No-Code ETL Tools with Export** | Rebuild workflows in tools like Talend or NiFi that support PySpark export. | Teams already using these tools for new pipelines | ~60% | Quick visual mapping, good for simple jobs | Limited for complex/nested logic |
| **Metadata-Driven Code Generation** | Use metadata lineage tools to extract logic and auto-generate PySpark code. | Enterprises with mature data governance | ~80% | Centralized control, scalable | High implementation complexity |

**Approach and Tool Selection for Converting Informatica Mappings:**

*We have chosen* ***DataSwitch****, a no-code automated ETL conversion platform, to facilitate the mapping conversion.* This strategic business decision aims to accelerate the migration by automatically converting PowerCenter mappings to PySpark, reducing manual coding effort and project risk.

**Informatica Workflow to PySpark Code Conversion – Step-by-Step Process using DataSwitch**

This diagram outlines the end-to-end workflow for converting Informatica PowerCenter mappings (XML exports) into PySpark scripts using a semi-automated approach via DataSwitch, a no-code conversion tool. It visualizes how technical teams can efficiently transition legacy ETL workflows to a modern PySpark-based architecture while ensuring validation, quality, and functional equivalence with the original mappings.



**Step-by-Step Explanation**

1. **Start Informatica Export**

* Begin by identifying the Informatica PowerCenter mappings intended for migration.
* Export these mappings in XML format. This serves as the raw input for the conversion process.

2. **Start PySpark Setup**

* In parallel, initiate the setup for the PySpark environment.
* This includes setting up Spark clusters (local or cloud), libraries, configuration files, and dev/test environments where the generated PySpark code will be executed and validated.

3**. Export Informatica Workflow**

* Export the full workflow related to the mappings, including all transformation logic, connections, and dependencies.
* This ensures accurate replication during the conversion to PySpark.

4. **Upload Sample Mappings to DataSwitch**

* Load the exported XML files into the DataSwitch platform.
* DataSwitch analyzes the XML files and interprets the mapping logic using its built-in engine.

5. **Generate PySpark Scripts**

* DataSwitch generates base PySpark code by automatically translating the Informatica transformation logic into Python scripts compatible with Apache Spark.
* This step significantly reduces manual coding effort.

6. **Manual Code Change Needed?**

* Perform an initial review of the auto-generated code.
* If the code is not fully accurate or misses complex transformation logic, determine whether manual intervention is needed.

7. **Make Manual Code Changes (if required)**

* Developers manually adjust and refine the PySpark scripts where automation falls short.
* This could involve custom joins, filters, lookups, or handling edge cases that are not fully captured by automated tooling.

8. **Validate Generated Code**

* Validate the PySpark code by comparing logic flow and transformation steps against the original Informatica mapping.
* This helps catch any discrepancies early in the migration cycle.

9**. Make Code Syntactically Error-Free**

* Ensure the generated and manually modified code is syntactically correct.
* Fix missing imports, indentation, variable mismatches, or API misuses to make the code runnable.

10**. Code Review and Quality Check**

* Conduct peer reviews to ensure the code meets performance, security, and maintainability standards.
* Apply linting, formatting, and static analysis if applicable.

11. **Run PySpark Code**

* Execute the PySpark job in a controlled test environment.
* Monitor output logs and check if the job runs successfully end-to-end.

12. **Run Informatica Workflow (Parallel Testing)**

* In parallel, run the original Informatica workflow using the same input data.
* This helps in benchmarking and functional validation.

13. **Data Validation Check**

* Compare the outputs from the Informatica job and the PySpark job.
* Perform record-level and field-level comparisons to ensure both jobs produce identical results.

14. **End**

* Once validation passes and functional equivalence is confirmed, the migration is considered complete for that mapping.
* The code is then ready for production deployment or inclusion in the broader migration batch.

**Cloud Modernization Plan Overview**

In this project, we are addressing two major components:

1. Converting Informatica Workflow Mappings (XML) to PySpark code
2. Deploying the Converted Pipelines to AWS – Cloud Modernization

We have already covered the process of converting Informatica mappings into PySpark. Now, let’s shift focus to the second part: Cloud Modernization, where we explore how to deploy and orchestrate these PySpark pipelines in the AWS environment.

**Current State Analysis**

**Before selecting the right cloud deployment strategy, it is important to understand the current landscape and metrics:**

* Job Volume: Approximately 2,000 Informatica workflows (XML) need to be migrated.
* Data Volume: Each job typically processes up to 1 million records.
* Delta Loads: Incremental load jobs are implemented separately.
* **Execution Frequency:**
  + Most workflows do not run daily.
  + On average, 300–400 jobs run per day.
  + At peak, up to 1,200 jobs may run daily.
* **Database:**
  + Source/target is Microsoft SQL Server hosted on Azure.
  + Connection properties have already been configured.
* **Current Setup:**
  + Jobs are triggered using AutoSys.
  + Workflows run on on-premises Informatica servers.
  + Shell scripts are used for notifications, logging, and job metrics—all handled within the on-prem infrastructure.

**Cloud Deployment Approaches Considered**

To modernize our existing Informatica workflows by deploying PySpark pipelines on AWS, we carefully evaluated several cloud-based deployment strategies. Each approach was assessed based on criteria like scalability, maintainability, cost-effectiveness, and compatibility with our current infrastructure.

Below is a detailed overview of the evaluated options and our preliminary recommendations:

**❌ Option 1: Databricks (Rejected)**

* **Description:** Databricks provides a unified, managed platform for Spark workloads with extensive collaboration and analytics capabilities.
* **Pros:**
  + Optimized performance and built-in analytics.
  + Strong integration with Spark ecosystem and notebooks.
  + Dataswitch pyspark code fully compatible here
  + Provides granular control over Spark clusters and compute resources.
  + Inbuild schedule system or integrate with third party schedular
  + **Databricks ecosystem** is a unified platform designed for **data, analytics, and AI** and data Lakehouse
* **Cons:**
  + High licensing and operational costs.
  + Our workloads (typically processing only 5–7 rows per job across 2,000 jobs) do not justify the expense.

**Conclusion:** **Rejected** because the cost outweighs the benefits for our specific scenario.

Between **Databricks** and **Informatica**, typically **Informatica's enterprise licensing costs are** lesser compared to Databricks, particularly for large-scale, enterprise-grade solutions.

**Databricks**:

* Pay-as-you-go pricing model based on actual compute usage.
* Addition cost of Cloud Infrastructure Expenses (Amazon S3, Azure Blob Storage), networking, data transfer
* Flexible scaling, with costs varying by cloud provider, instance type, and features.
* Additional Licensing Databricks SQL, Delta Lake optimization, advanced security features
* Typically, more cost-effective for moderate workloads or variable usage.
* Administrative and Operational Overhead

**Informatica**:

* Subscription-based, tiered pricing that escalates quickly with increased data volume, users, or advanced features (e.g., MDM, data governance).
* Often higher fixed licensing costs at enterprise scale.

**Conclusion**:  
Informatica generally incurs higher costs at enterprise scale, especially as data volumes and complexity increase, making Databricks potentially more economical depending on usage patterns and needs.

**✅ Option 2: AWS Glue + Apache Airflow (Recommended)**

* **Description:** This approach leverages AWS Glue for serverless, scalable ETL execution paired with Apache Airflow for flexible, robust workflow orchestration.
* **Pros:**
  + Fully managed serverless ETL service, reducing operational overhead.
  + Airflow provides advanced scheduling, monitoring, error handling, and retry mechanisms.
  + Cost-effective, cloud-native architecture aligned with AWS best practices.
* **Cons:**
  + Requires setup and maintenance of Airflow (can be managed via AWS MWAA or self-hosted).
  + Dataswitch pyspark code need convert or make compatible as per AWS Glue
* **Conclusion:** **Recommended** as the best practice solution due to superior scalability, cost-effectiveness, and robust orchestration capabilities.

**⚙️ Option 3: EMR + AWS Step Functions (Considered)**

* **Description:** Amazon EMR is utilized to run Spark workloads, orchestrated through AWS Step Functions, which offer managed workflows.
* **Pros:**
  + Provides granular control over Spark clusters and compute resources.
  + Good fit for large-scale, compute-intensive jobs with specialized tuning.
  + Dataswitch pyspark code more compatible here
* **Cons:**
  + Higher complexity in managing cluster resources.
  + Setup and ongoing operational effort are substantial compared to serverless solutions.
* **Conclusion:** Technically viable but **less attractive** due to higher operational complexity for our typical workloads.

**✅ Option 4: AWS Glue + AutoSys (Feasible)**

* **Description:** Utilize existing AutoSys scheduler to orchestrate jobs while leveraging AWS Glue for execution of PySpark scripts.
* **Pros:**
  + Minimal changes required in job orchestration logic.
  + Enables a gradual transition by maintaining familiarity with AutoSys scheduling.
  + Immediate compatibility with current scheduling processes and existing infrastructure.
* **Cons:**
  + Continues dependence on on-premise legacy scheduling solutions.
  + Limited integration with AWS-native monitoring, governance, and management tools.
  + Potentially creates long-term complexity in maintaining a hybrid orchestration environment.
  + Dataswitch pyspark code need convert or make compatible as per AWS Glue
* **Conclusion:** **Feasible as an interim solution** to ease transition, but less ideal from a cloud-native perspective.

**❌ Option 5: EC2 with Spark Cluster (Rejected)**

* **Description:** Manually managing Spark clusters on Amazon EC2 infrastructure.
* **Pros:**
  + Maximum flexibility and control over cluster configuration and Spark tuning.
  + Dataswitch pyspark code fully compatible here
* **Cons:**
  + High operational burden and infrastructure management overhead.
  + Lacks scalability, elasticity, and operational simplicity compared to managed serverless offerings.
* **Conclusion:** **Rejected** due to operational complexity, maintenance overhead, and limited scalability advantages.

NOTE:  
The following diagrams not defined detailed definitions for:

* Network: Connectivity, firewall, VPC settings.
* Security: IAM roles, encryption, compliance.
* Error Handling: Job retries and monitoring.

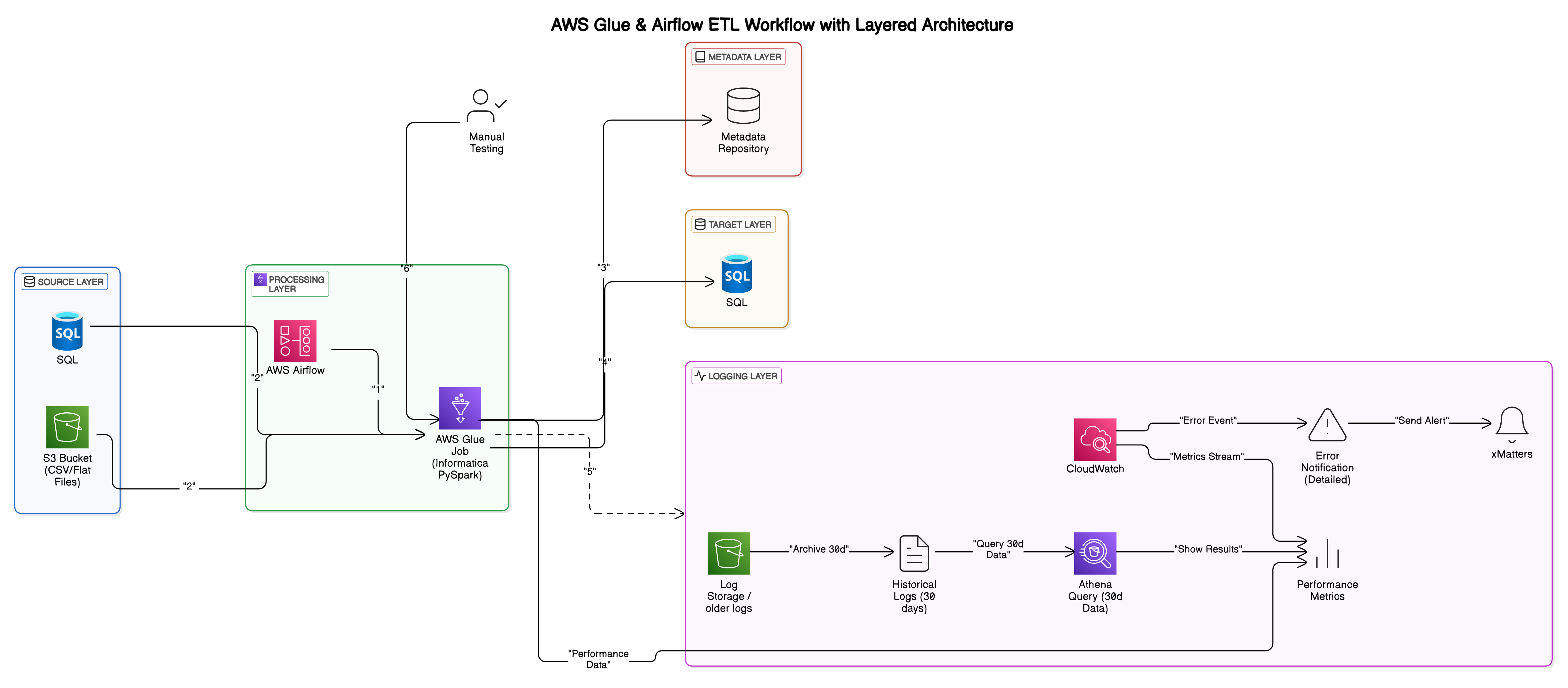
**Detailed Architecture Discussion**

Given the analysis above, we plan to focus further discussions specifically on:

* **Option 2 / Approach 1: AWS Glue + Apache Airflow** *(Recommended, best practice)*
* **Option 4/ Approach 2: AWS Glue + AutoSys** *(Feasible, transitional approach)*

We will provide detailed **architectural diagrams**, cost estimations, technical considerations, and implementation guidelines for both options. These discussions will help us make a well-informed final decision tailored to our short-term migration goals and long-term cloud modernization strategy.

**AWS Glue + Apache Airflow** *(Recommended, best practice)*



**AWS Glue & Airflow ETL Workflow – Layered Architecture Overview**

This architecture replaces legacy Informatica ETL workflows with a scalable AWS-based solution using AWS Glue and Apache Airflow. It is divided into clear, purpose-specific layers:

**1. Source Layer**

* **SQL Database (Azure-hosted)**: Structured data source.
* **Amazon S3**: Storage for flat-file inputs (CSV).

**2. Processing Layer (AWS Glue + Apache Airflow)**

* **AWS Airflow**:
  + Manages scheduling, dependencies, and workflow orchestration.
  + Sends alerts on workflow failures.
* **AWS Glue (PySpark)**:
  + Executes data transformation and loads processed data.
  + Serverless, auto-scalable ETL jobs.

**3. Target Layer**

* **MS SQL Database (Azure-hosted or on frame )**: Receives processed data for analytics.

**4. Metadata Layer**

* Tracks job executions, dependencies, and data lineage.
* Supports operational monitoring and troubleshooting.

**5. Logging Layer**

* **AWS CloudWatch**: Real-time monitoring and alerts.
* **Amazon S3**: Archival of historical logs.

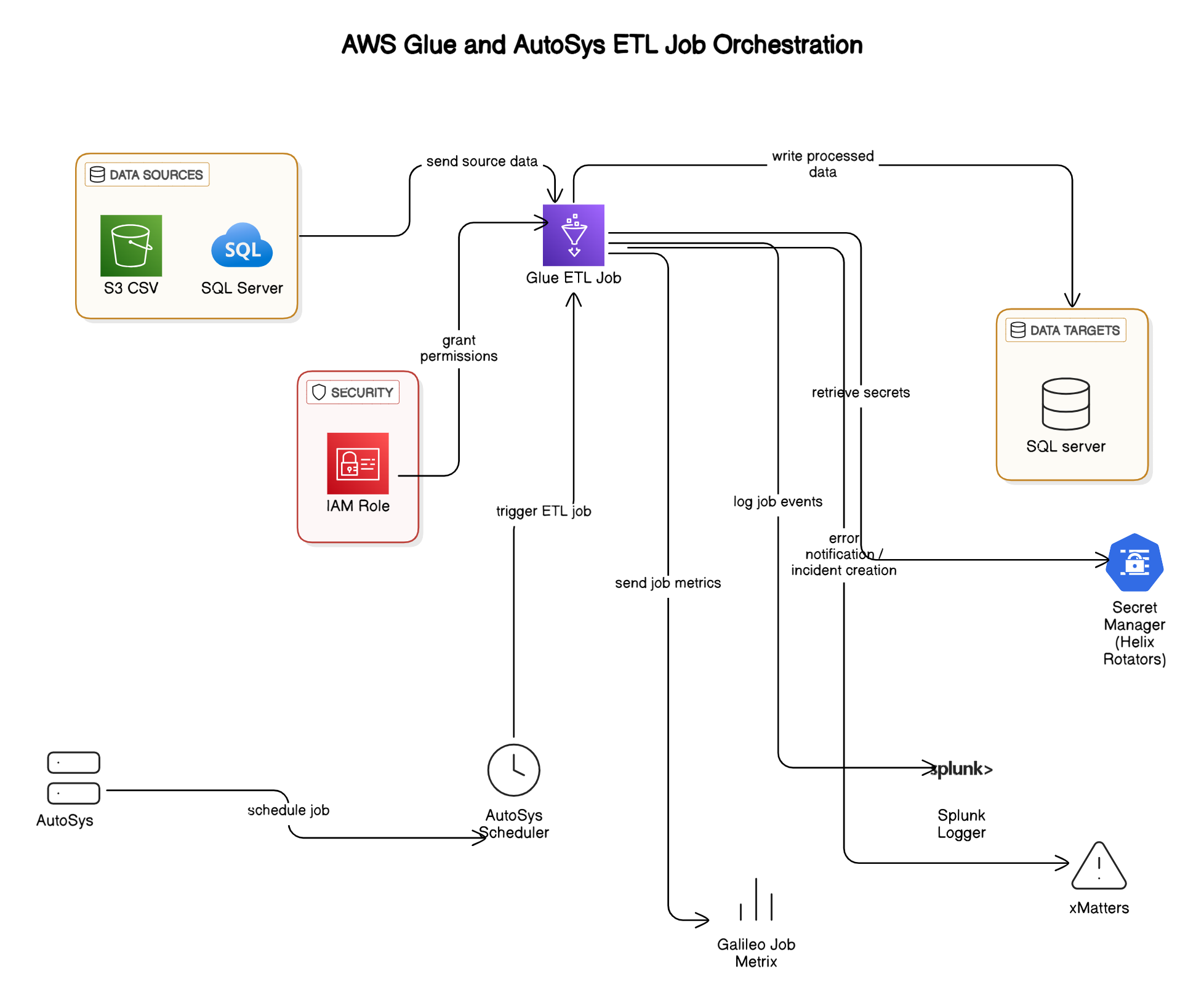
**6. Manual Testing**

* Periodic manual validation for quality assurance.

**Typical Workflow**

1. Airflow initiates Glue jobs.
2. Glue extracts data from sources.
3. Glue transforms and loads data to the target database.
4. Job metadata is recorded.
5. Logs captured in CloudWatch, archived to S3.
6. Manual quality checks performed as needed.

**AWS Glue + AutoSys** *(transitional approach)*



**AWS Glue and AutoSys ETL Job Orchestration**

This architecture illustrates how converted Informatica workflows (now PySpark ETL jobs) can be orchestrated using AWS Glue for processing, while leveraging the existing on-premises AutoSys scheduler for job scheduling. It represents a hybrid architecture effectively bridging cloud-native and on-premises systems.

**Detailed Architecture Overview**

The architecture consists of the following major components:

**1. Data Sources**

* **SQL Server (On-Premise)**:
  + Provides structured data inputs for the ETL process.
* **Amazon S3 (CSV files)**:
  + Serves as a storage source for flat-file data, accessible directly by AWS Glue.

**2. AWS Glue ETL Job**

* Centralized ETL processing using PySpark scripts converted from original Informatica workflows.
* Retrieves data from sources, performs transformations, enrichments, and loads processed data to the target.
* Leverages AWS Glue’s serverless infrastructure for scalability and cost efficiency.

**3. Data Targets**

* **SQL Server Database**:
  + Acts as the final destination for the transformed and cleansed data.
  + Supports analytical queries, reporting, or downstream business applications.

**4. Security Layer**

* **AWS IAM Roles**:
  + Securely grant AWS Glue necessary permissions to access resources (S3, SQL Server, Secrets Manager).
  + Ensure fine-grained access control and secure execution of ETL jobs.
* **AWS Secrets Manager (Helix Rotators)**:
  + Provides secure and managed storage of sensitive credentials required by Glue ETL jobs to access source and target systems.

**5. Job Scheduling & Orchestration**

* **AutoSys Scheduler (On-Prem)**:
  + Existing on-premises scheduler used to trigger AWS Glue jobs.
  + Continues to schedule ETL workloads, ensuring minimal disruption to existing job orchestration practices.
  + Allows a phased migration approach by maintaining familiar job scheduling processes.

**6. Logging and Monitoring**

* **Splunk Logger**:
  + Captures and logs job events, providing centralized logging and monitoring capability.
  + Facilitates troubleshooting, auditing, and compliance reporting.
* **Galileo Job Metrics**:
  + Collects and analyzes job execution metrics, such as performance and operational efficiency data.

**Typical Workflow Execution**

1. **AutoSys (On-Prem)** schedules and triggers the AWS Glue job.
2. **AWS Glue ETL Job** begins execution, authorized via an **IAM role** for secure access.
3. Glue retrieves source data from **S3** and **SQL Server** databases.
4. Glue retrieves credentials securely from **AWS Secrets Manager**.
5. Glue performs transformations and writes processed data to the target **SQL Server**.
6. Glue sends execution logs and metrics to **Splunk Logger** and **Galileo** for monitoring.

**Open Questions for AWS Glue and AutoSys ETL**

Below are critical questions that need clarification and discussion as we proceed with the Informatica-to-PySpark ETL migration using the above architecture:

* **AutoSys Details:** Should we set up a new AutoSys cluster in the cloud, or continue using the existing on-premises setup?
* **Splunk API and Job Metrics:** If a new AutoSys cluster is created, do we need to set up new Splunk APIs and job metrics, or can we reuse the existing configurations?