Deep Dive: Applying Platform Concepts to Snowflake

This document explores how the architectural principles and component functionalities of your locally deployed enterprise data platform translate to a cloud data warehouse environment, specifically **Snowflake**. While your local setup uses open-source components like Spark, Kafka, and Delta Lake, Snowflake offers managed, highly scalable, and performant services that fulfill similar roles, often with a serverless or "data warehouse as a service" paradigm.

Understanding these equivalences is crucial for migrating your platform to a cloud-native solution and leveraging the specific strengths of Snowflake for modern data analytics.

1. Core Platform Components: Snowflake Equivalents

Let's map the key components of your local data platform to their corresponding functionalities and services within the Snowflake ecosystem:

Local Platform Component	Role in Local Platform	Snowflake Equivalent / Approach	Snowflake Benefits
MinIO (S3-compatible) + Delta Lake	Object Storage & Data Lakehouse	**Snowflake Internal/External Stages**	Unified storage for structured, semi-structured, and unstructured data. Separation of storage and compute. Optimized for analytics.
	ACID Transactions, Schema Enforcement, Time Travel	**Snowflake Tables + Streams + Time Travel**	ACID compliance, schema evolution (variant), historical data access.
Apache Kafka	Distributed Streaming Platform	**Snowpipe, Snowpipe Streaming, Kafka Connector**	Managed ingestion of streaming data into Snowflake tables with low latency.
FastAPI Ingestor	Real-time Ingestion API	**Snowflake Snowpipe REST API, AWS API Gateway + Lambda**	Serverless ingestion API, direct HTTP endpoint for loading data into Snowflake.
Apache Spark (PySpark Jobs)	Distributed Processing Engine (ETL/ELT)	**Snowpark, SQL, Stored Procedures, Tasks**	Powerful API for data processing (Python/Scala/Java),

			SQL-based transformations, scheduled batch jobs.
PostgreSQL	Relational Database (Metastore, Reference Data)	**Snowflake Database/Schemas/Tables**	Relational capabilities within Snowflake; can store reference data, configuration.
MongoDB	NoSQL Document Database	**Snowflake VARIANT Data Type, External Tables, or integrate with DocumentDB/CosmosDB**	Handles semi-structured data directly in tables, or connects to external NoSQL stores.
Apache Airflow	Workflow Orchestration	**Snowflake Tasks, External Orchestrators (e.g., Airflow with Snowflake Operator)**	Native scheduling within Snowflake, or integration with existing Airflow instances for broader orchestration.
OpenMetadata	Data Catalog & Governance	**Snowflake Information Schema, External Data Catalogs (e.g., Alation, Collibra, or OpenMetadata with Snowflake Connector)**	Rich metadata available natively; integrates with external governance platforms.
Spline	Spark Data Lineage	**Snowflake Access History, Query Tagging, External Lineage Tools**	Native lineage tracking for SQL queries; integrates with external lineage solutions.
Grafana Alloy	Telemetry Collector	**Snowflake Account Usage, Information Schema (for logs/metrics), Cloud-native monitoring agents**	Built-in monitoring of Snowflake usage; logs sent to cloud-native logging (CloudWatch, Azure Monitor, Cloud Logging).
Grafana	Interactive Visualization & Monitoring	**Snowflake Partner Connect (Tableau, Power BI), Native Dashboards**	Rich visualization ecosystem, direct integration with BI tools, custom

		(Streamlit in Snowflake), Grafana with Snowflake Data Source	dashboards.
AWS SAM CLI (Local Serverless Dev)	Local Lambda/API Gateway Simulation	**No direct Snowflake equivalent for local dev, but cloud-native dev tools (e.g., Snowflake Snowpark Local Testing)**	Develop and test Snowpark code locally.

2. Interactive How-Tos: Applying Concepts to Snowflake

Let's walk through key data platform scenarios and how they are implemented using Snowflake.

Basic Use Case: Ingesting Semi-Structured Data and Querying in Snowflake

Objective: To demonstrate how Snowflake efficiently ingests semi-structured data (like your Kafka messages) into a table using its VARIANT data type and COPY INTO command, and then queries it using SQL.
Role in Platform: Act as the central repository for raw and structured data, leveraging Snowflake's native capabilities for semi-structured data.
Setup/Configuration (Conceptual Snowflake Environment):

1. **Snowflake Account:** Access to a Snowflake account and a database/schema.
2. **Snowflake Stage:** An internal or external stage (e.g., pointing to an S3 bucket or MinIO if accessible from Snowflake) where your raw JSON data would land. For this example, we'll assume files are landed on an internal stage.

Steps to Exercise (Conceptual Snowflake Operations):

1. Prepare Sample Data (JSON Lines):
   Imagine this JSON is in files in a Snowflake Internal Stage (e.g., @my_internal_stage/financial_transactions/).
   # financial_transaction_1.json
   {"transaction_id": "FT-001", "timestamp": "2024-01-01T10:00:00Z", "account_id": "ACC-001", "amount": 100.50, "currency": "USD"}
   {"transaction_id": "FT-002", "timestamp": "2024-01-01T10:05:00Z", "account_id": "ACC-002", "amount": 200.75, "currency": "EUR", "merchant": "ShopCo"}

   Note the merchant field in FT-002 is optional, demonstrating semi-structured nature.
2. **Create a Target Table with VARIANT Column:**

```sql
-- Connect to your Snowflake worksheet
USE DATABASE YOUR_DATABASE;
USE SCHEMA YOUR_SCHEMA;

CREATE TABLE IF NOT EXISTS RAW_FINANCIAL_TRANSACTIONS (
  RAW_DATA VARIANT,
  LOAD_TIMESTAMP TIMESTAMP_NTZ DEFAULT CURRENT_TIMESTAMP()
);
```

3. Load Data using COPY INTO:
   In a real scenario, you would have files in an external stage (e.g., S3) or use Snowpipe. For this basic example, we simulate copying from an internal stage where files were uploaded.

```sql
-- Assume files are already in @my_internal_stage/financial_transactions/
-- You can upload files to an internal stage using Snowflake UI or SnowSQL PUT command.
-- Example PUT command (from your local machine assuming file exists):
-- PUT file://<local_path>/financial_transaction_1.json @my_internal_stage/financial_transactions/ AUTO_COMPRESS=TRUE;

COPY INTO RAW_FINANCIAL_TRANSACTIONS (RAW_DATA)
FROM @my_internal_stage/financial_transactions/
FILE_FORMAT = (TYPE = JSON);

-- Or, if loading from external S3 bucket directly:
-- COPY INTO RAW_FINANCIAL_TRANSACTIONS (RAW_DATA)
-- FROM 's3://your-s3-bucket/path/to/json/'
-- CREDENTIALS = (AWS_KEY_ID = 'your_key_id' AWS_SECRET_KEY = 'your_secret_key')
-- FILE_FORMAT = (TYPE = JSON);
```

4. **Query the Semi-Structured Data:**

```sql
SELECT
  RAW_DATA:transaction_id::VARCHAR AS transaction_id,
  RAW_DATA:timestamp::TIMESTAMP_NTZ AS transaction_timestamp,
  RAW_DATA:account_id::VARCHAR AS account_id,
  RAW_DATA:amount::FLOAT AS amount,
  RAW_DATA:currency::VARCHAR AS currency,
  RAW_DATA:merchant::VARCHAR AS merchant_name, -- Accessing an optional field
  LOAD_TIMESTAMP
FROM
  RAW_FINANCIAL_TRANSACTIONS
LIMIT 10;
```

Observe: The query extracts specific fields from the VARIANT column using dot notation and type casting. The merchant_name will be NULL for FT-001 and populated for FT-002.

Verification:

- **Snowflake Worksheet Output:** The COPY INTO command reports successful rows loaded. The SELECT query correctly parses and extracts fields from the VARIANT JSON, demonstrating Snowflake's ability to handle schema flexibility.

Advanced Use Case 1: Streamlining Data Ingestion with Snowpipe

Objective: To demonstrate how Snowpipe (or Snowpipe Streaming for even lower latency) can automatically ingest new data files arriving in an external stage (like S3) into a Snowflake table, providing a continuous ingestion pipeline similar to Kafka + Spark streaming.

Role in Platform: Automate continuous, low-latency data ingestion from cloud storage into Snowflake, replacing the need for a manually managed streaming consumer.

Setup/Configuration (Conceptual Snowflake Environment):

1. **Snowflake Account & External Stage:** An external stage configured to an S3 bucket.
2. **AWS S3 Bucket & Event Notifications:** An S3 bucket where new files will land, with S3 Event Notifications configured to publish messages to an SQS queue.
3. **Snowflake Integration:** A Snowflake STORAGE INTEGRATION and NOTIFICATION INTEGRATION to securely connect to S3 and SQS.

Steps to Exercise (Conceptual Snowflake & AWS Operations):

1. **Create File Format (if not exists):**
```
CREATE FILE FORMAT IF NOT EXISTS JSON_FORMAT
TYPE = JSON
STRIP_OUTER_ARRAY = FALSE; -- Important if your JSON is an array of objects per file
```

2. **Create Stage (External):**
```
CREATE OR REPLACE STAGE RAW_FINANCIAL_TRANSASACTIONS_STAGE
  URL = 's3://your-s3-raw-bucket/financial_transactions/'
  STORAGE_INTEGRATION = s3_storage_integration_for_data_platform; -- Your
pre-configured storage integration
```

3. **Create Target Table:**
```
CREATE TABLE IF NOT EXISTS RAW_FINANCIAL_TRANSACTIONS_SNOWPIPE (
    RAW_DATA VARIANT,
    FILE_NAME VARCHAR,
    LOAD_TIMESTAMP TIMESTAMP_NTZ DEFAULT CURRENT_TIMESTAMP()
);
```

4. **Create Snowpipe:**
```
CREATE OR REPLACE PIPE FINANCIAL_TRANSACTIONS_PIPE
  AUTO_INGEST = TRUE
  AS
```

```
  COPY INTO RAW_FINANCIAL_TRANSACTIONS_SNOWPIPE (RAW_DATA, FILE_NAME)
  FROM (SELECT $1, METADATA$FILENAME FROM
@RAW_FINANCIAL_TRANSACTIONS_STAGE)
  FILE_FORMAT = (TYPE = JSON);
```

This creates a pipe that listens to SQS notifications from S3 and automatically loads new files.

5. **Get SQS Queue ARN from Snowpipe:**
```
SHOW PIPES LIKE 'FINANCIAL_TRANSACTIONS_PIPE';
-- Look for 'notification_channel' column in the output, this is the SQS Queue ARN.
```

   You would take this SQS ARN and configure your S3 bucket's event notification to publish s3:ObjectCreated: events to this SQS queue.*
6. Simulate New File Arrival (Manual Upload or Programmatic from FastAPI/Lambda):
   In your local environment, you would run a script that uploads a new JSON file to your S3 bucket (which the external stage points to).
   From your local FastAPI (now configured to upload to S3 instead of MinIO or Kafka in a cloud-migrated scenario), trigger new financial transactions.
7. **Monitor Snowpipe Progress:**
```
SELECT *
FROM TABLE(INFORMATION_SCHEMA.PIPE_USAGE_HISTORY(
   DATE_RANGE_START=>DATEADD('hour', -1, CURRENT_TIMESTAMP()),
   PIPE_NAME=>'FINANCIAL_TRANSACTIONS_PIPE'));
```

   This query shows if files were processed and any errors.
8. **Query Data in Snowflake:**
```
SELECT COUNT(*) FROM RAW_FINANCIAL_TRANSACTIONS_SNOWPIPE;
```

   Wait a few moments after uploading a file, then run this query. The count should increase.

Verification:

- **Snowflake Pipe History:** The PIPE_USAGE_HISTORY shows new files being detected and loaded.
- **Snowflake Table Count:** The RAW_FINANCIAL_TRANSACTIONS_SNOWPIPE table accumulates new records automatically after files are dropped into S3, demonstrating Snowpipe's automated, continuous ingestion.

Advanced Use Case 2: Feature Engineering and Transformations with Snowpark

Objective: To demonstrate how Snowpark (Snowflake's developer experience for Python/Java/Scala) can be used to perform complex data transformations and feature engineering directly within Snowflake's compute engine, similar to how PySpark jobs run on

Spark.
Role in Platform: Perform scalable data transformations, aggregations, and feature engineering on data stored in Snowflake, leveraging its elastic compute without moving data out of the warehouse.
Setup/Configuration (Conceptual Snowflake Environment + Python):

1. **Snowflake Account & Warehouse:** Access to a Snowflake account and an active warehouse.
2. **Snowflake Table with Raw Data:** Ensure RAW_FINANCIAL_TRANSACTIONS_SNOWPIPE (from previous use case) has data.
3. **Snowflake Client (Python):** Install snowflake-snowpark-python on your local machine.

Steps to Exercise (Conceptual Python Script using Snowpark):

1. **Create a Python Script (snowpark_feature_engineering.py):**

```
# snowpark_feature_engineering.py
from snowflake.snowpark import Session
from snowflake.snowpark.functions import col, count, sum, avg, to_date, lit,
current_timestamp
import json
import os

# --- Snowflake Connection Configuration ---
# Replace with your actual Snowflake connection details
connection_parameters = {
   "account": os.getenv("SNOWFLAKE_ACCOUNT", "your_account_id"),
   "user": os.getenv("SNOWFLAKE_USER", "your_user"),
   "password": os.getenv("SNOWFLAKE_PASSWORD", "your_password"),
   "role": os.getenv("SNOWFLAKE_ROLE", "SYSADMIN"),
   "warehouse": os.getenv("SNOWFLAKE_WAREHOUSE", "COMPUTE_WH"),
   "database": os.getenv("SNOWFLAKE_DATABASE", "YOUR_DATABASE"),
   "schema": os.getenv("SNOWFLAKE_SCHEMA", "YOUR_SCHEMA")
}

if __name__ == "__main__":
   session = None
   try:
      print("Creating Snowpark session...")
      session = Session.builder.configs(connection_parameters).create()
      print(f"Snowpark session created successfully. Current database:
{session.get_current_database()}, schema: {session.get_current_schema()}")

      input_table_name = "RAW_FINANCIAL_TRANSACTIONS_SNOWPIPE"
      output_table_name = "CURATED_FINANCIAL_FEATURES_DAILY"

      print(f"Reading raw data from: {input_table_name}")
```

```
    # Read from the raw table, parsing the VARIANT column
    df_raw = session.table(input_table_name).select(
        col("RAW_DATA"):("transaction_id").as_("transaction_id"),
        col("RAW_DATA"):("timestamp").as_("timestamp"),
        col("RAW_DATA"):("account_id").as_("account_id"),
        col("RAW_DATA"):("amount").as_("amount").cast("float"),
        col("RAW_DATA"):("currency").as_("currency")
        # Add other fields as needed
    )
    df_raw.show(5)

    print("Performing feature engineering: daily aggregates per account...")
    # Convert timestamp to date, then group and aggregate
    df_features = df_raw.withColumn("transaction_date", to_date(col("timestamp"))) \
                    .groupBy("account_id", "transaction_date") \
                    .agg(
                        count(col("transaction_id")).alias("daily_transaction_count"),
                        sum(col("amount")).alias("daily_total_amount"),
                        avg(col("amount")).alias("daily_average_amount")
                    ) \
                    .withColumn("feature_created_at", current_timestamp())

    print("Schema of engineered features:")
    df_features.show(5)

    # Write the engineered features to a new curated table in Snowflake
    print(f"Writing engineered features to: {output_table_name}")
    df_features.write.mode("overwrite").save_as_table(output_table_name)
    print(f"Feature engineering job completed. Data written to {output_table_name}.")

  except Exception as e:
    print(f"An error occurred: {e}")
    import traceback
    traceback.print_exc()
  finally:
    if session:
        session.close()
        print("Snowpark session closed.")
```

2. **Run the Python Script:**
   python3 snowpark_feature_engineering.py

   Ensure SNOWFLAKE_ACCOUNT, SNOWFLAKE_USER, SNOWFLAKE_PASSWORD (or *other auth methods) are set as environment variables or directly in the script.*

3. **Verify Data in Snowflake:**
   - In your Snowflake worksheet, query the new table:

```
SELECT * FROM CURATED_FINANCIAL_FEATURES_DAILY LIMIT 10;
```

Verification:

- **Script Output:** The Python script prints messages indicating successful session creation, data reading, transformation, and writing.
- **Snowflake Worksheet:** The CURATED_FINANCIAL_FEATURES_DAILY table is created and populated with the aggregated features, demonstrating Snowpark's capability to perform complex ETL/feature engineering directly in Snowflake.

Advanced Use Case 3: Data Lineage and Governance with Snowflake

Objective: To conceptually explain how Snowflake's native features (ACCESS_HISTORY, QUERY_HISTORY) provide rich data lineage, and how this integrates with external data catalog tools (like OpenMetadata).
Role in Platform: Provide robust, automated data lineage within the warehouse, enabling comprehensive data governance, impact analysis, and compliance.
Setup/Configuration (Conceptual Snowflake & OpenMetadata Integration):

1. **Snowflake Account:** Ensure query history and access history are enabled (they are by default for most accounts).
2. **OpenMetadata with Snowflake Connector:** An OpenMetadata instance configured with a Snowflake connector.

Steps to Exercise (Conceptual Discussion):

1. **Snowflake Native Lineage (ACCESS_HISTORY & QUERY_HISTORY):**
   - **ACCOUNT_USAGE.ACCESS_HISTORY View:** This view captures comprehensive lineage information, including which objects (tables, views) were read and written by which queries.

```
-- Example query to see access history for a table
SELECT
   QUERY_ID,
   QUERY_TEXT,
   BASE_OBJECTS_ACCESSED, -- Objects read
   DIRECT_OBJECTS_MODIFIED -- Objects written
FROM
   SNOWFLAKE.ACCOUNT_USAGE.ACCESS_HISTORY
WHERE
   QUERY_TYPE = 'INSERT' OR QUERY_TYPE = 'CREATE_TABLE_AS_SELECT'
   AND QUERY_START_TIME >= DATEADD('day', -7, CURRENT_TIMESTAMP())
ORDER BY
   QUERY_START_TIME DESC
LIMIT 10;
```

- **ACCOUNT_USAGE.QUERY_HISTORY View:** Provides details about every query executed, including user, warehouse, duration, and associated tags. This can be joined with ACCESS_HISTORY for a full picture.
```
-- Example: Find queries that read from
RAW_FINANCIAL_TRANSACTIONS_SNOWPIPE
SELECT
   qh.QUERY_ID,
   qh.QUERY_TEXT,
   qh.USER_NAME,
   qh.WAREHOUSE_NAME,
   ah.BASE_OBJECTS_ACCESSED,
   ah.DIRECT_OBJECTS_MODIFIED
FROM
   SNOWFLAKE.ACCOUNT_USAGE.QUERY_HISTORY qh
JOIN
   SNOWFLAKE.ACCOUNT_USAGE.ACCESS_HISTORY ah ON qh.QUERY_ID =
ah.QUERY_ID
WHERE
   ARRAY_CONTAINS(

'YOUR_DATABASE.YOUR_SCHEMA.RAW_FINANCIAL_TRANSACTIONS_SNOWPIPE'::
VARIANT, -- Replace with full object path
       ah.BASE_OBJECTS_ACCESSED
   )
ORDER BY qh.START_TIME DESC
LIMIT 10;
```

2. **OpenMetadata with Snowflake Connector:**
   - **Configure Connector:** In OpenMetadata UI, add a new "Service" of type "Database" and choose "Snowflake." Provide connection details (account, role, warehouse, database, schema).
   - **Ingestion Workflows:** Configure and run ingestion workflows in OpenMetadata to:
     - **Metadata Ingestion:** Pull table/view schemas, descriptions, and column details.
     - **Profiler Ingestion:** Run data profiling to get statistics (row counts, min/max, nulls) for tables.
     - **Usage Ingestion:** This is where the magic happens for lineage. The OpenMetadata Snowflake connector can parse QUERY_HISTORY to infer lineage relationships (e.g., if SELECT * FROM A JOIN B creates C, it infers A and B feed C).
   - **View Lineage in OpenMetadata:** After successful ingestion, navigate to a Snowflake table in OpenMetadata (e.g., CURATED_FINANCIAL_FEATURES_DAILY).

Go to its "Lineage" tab.

- **Expected:** You should see a graphical representation showing the RAW_FINANCIAL_TRANSACTIONS_SNOWPIPE table (source), a conceptual "transformation" node (representing the Snowpark job or SQL transformation), and the CURATED_FINANCIAL_FEATURES_DAILY table (destination).

Verification (Conceptual):

- **Snowflake Query History:** Successfully query ACCESS_HISTORY and QUERY_HISTORY to manually trace data flow.
- **OpenMetadata UI:** The data catalog accurately reflects Snowflake schemas, and the "Lineage" tab displays end-to-end data flow for tables processed within Snowflake, demonstrating effective data governance and lineage tracking. This highlights how Snowflake's native capabilities, combined with tools like OpenMetadata, provide a robust solution for data transparency.

This concludes the deep dive into applying your platform concepts to Snowflake.