Enterprise Data Migration Architecture: MS SQL Server to Snowflake on AWS

Comprehensive Implementation of Secure, Automated Batch Ingestion using Python, Active Directory, and HashiCorp Vault

Executive Overview

The modernization of enterprise data ecosystems increasingly demands the migration of high-value transactional data from legacy on-premises systems, such as Microsoft SQL Server, to cloud-native platforms like Snowflake on AWS. This transition is rarely a simple "lift and shift"; it requires a robust, secure, and scalable data pipeline capable of handling high throughput while adhering to stringent enterprise security policies. This report details the technical architecture and implementation strategies for a Python-based middleware solution designed to facilitate this movement.
The architecture strictly enforces security best practices by eliminating hard-coded credentials through HashiCorp Vault integration and leveraging Active Directory (AD) for authentication across both source and destination systems. Specifically, it addresses the complexity of headless (non-interactive) authentication: utilizing Kerberos for MS SQL Server on Linux/Unix-based runners and the OAuth 2.0 Client Credentials flow for Snowflake via Microsoft Entra ID (formerly Azure AD). This document serves as an exhaustive implementation manual for data architects and senior engineers, covering protocol details, library selection, memory management for large datasets, and step-by-step code implementation.

---

1. Strategic Architecture and Design Principles

The proposed solution utilizes a "Pull-Push" architectural pattern orchestrated by a Python-based middleware service. Unlike "Push-Push" models where the source database exports files to an intermediate storage layer independently, or "Pull-Pull" models where the destination warehouses scrape the source, the middleware approach offers a balance of control, security, and transformational capability. This service acts as the ephemeral compute layer, extracting data in optimized batches from the source and loading it into the destination.

1.1 Architectural Components and Roles

The system is composed of four distinct layers: the Data Source, the Secure Enclave, the Compute/Orchestration Layer, and the Data Destination.

Component	Technology	Role	Critical Configuration
Source System	Microsoft SQL Server	Hosts legacy transactional data.	Configured for Active Directory Integrated Authentication (Kerberos). [1]
Destination System	Snowflake (AWS Region)	Cloud Data Warehouse target.	Uses virtual warehouses for compute. Configured with Security Integrations for External OAuth. [2]
Orchestrator	Python 3.9+ (Docker)	Runs the extraction and loading logic.	Hosted on ECS, Kubernetes, or EC2. Must have network visibility to both on-premise and AWS VPCs. [3]
Secrets Engine	HashiCorp Vault	Stores sensitive configuration.	Uses AppRole for machine authentication. Stores OAuth Client Secrets and Keytabs (or paths). [4]
Identity Provider	Microsoft Active Directory	Central authority for identity.	Provides Kerberos tickets for SQL Server and OAuth tokens for Snowflake via Entra ID. [5]
Staging Area	Snowflake Internal Stage	Temporary storage for Parquet files.	Utilizes Snowflake's internal storage layer to facilitate the COPY INTO command. [6]

1.2 Data Flow and Security Protocol

The lifecycle of a single data migration batch is governed by strict security protocols. The flow is designed to ensure that no long-lived credentials reside in the application code or configuration files.

1. **Bootstrapping and Identity Verification:** The Python application initializes and authenticates to HashiCorp Vault using a machine identity (AppRole). It presents a Role ID (embedded in the environment) and a Secret ID (injected at runtime). Vault verifies these credentials and issues a short-lived Vault Token. [7]

2. **Secret Retrieval:** Using the Vault Token, the application retrieves the necessary connection secrets from the KV Version 2 engine: the MS SQL Service Principal Name (SPN), the binary content of the Kerberos Keytab, and the Azure AD OAuth Client ID and Secret. [8]
3. **Source Authentication (Kerberos Handshake):** The application writes the Keytab to a secure, memory-backed temporary file system. It then executes the kinit command to request a Ticket Granting Ticket (TGT) from the Active Directory Domain Controller (KDC). This ticket enables the ODBC driver to authenticate to MS SQL Server without an interactive password. [9]
4. **Destination Authentication (OAuth 2.0 Flow):** The application contacts the Microsoft Entra ID token endpoint, presenting its Client ID and Client Secret. Upon validation, Entra ID issues a JSON Web Token (JWT) scoped for Snowflake access. This token acts as the bearer credential. [11]
5. **Extraction (Chunked Streaming):** Data is read from MS SQL Server using pyodbc and pandas. Crucially, data is fetched in memory-controlled chunks (e.g., 50,000 rows) to prevent OOM (Out of Memory) errors on the container. [12]
6. **Ingestion (Batch Write):** Each chunk is serialized into Apache Parquet format and uploaded to a Snowflake internal stage using write_pandas. The COPY INTO command is implicitly executed to commit the data to the target table. [13]
7. **Cleanup and Rotation:** Temporary files (Keytabs, Parquet buffers) are securely wiped. Connections are closed. If the process runs longer than the token lifetime, renewal logic is triggered.

---

2. Identity & Access Management (IAM) Framework

The most critical and complex aspect of this pipeline is securing the "handshake" between the automation script and the database systems without human intervention. This requires moving beyond simple username/password combinations to enterprise-grade protocols.

2.1 Microsoft SQL Server: The Kerberos Protocol

While connecting to MS SQL Server using Active Directory from a Windows machine is trivial via SSPI (Security Support Provider Interface), data pipelines typically run on Linux (e.g., Docker containers, Kubernetes pods). This necessitates the use of **Kerberos**, the underlying protocol that Active Directory uses for authentication.

2.1.1 The Mechanics of Linux-to-SQL Auth

In a Linux environment, the pyodbc driver cannot directly "pass through" Windows credentials. Instead, it relies on the GSSAPI (Generic Security Services Application Program Interface) library to handle the negotiation.

- **The Principal:** The Python script runs as a specific AD user, known as a Service Account (e.g., svc_datapipeline@CORP.EXAMPLE.COM).

- **The Keytab:** A keytab (Key Table) file is a cryptographic file containing the service account's password hash. It is functionally equivalent to a password but designed for machine use. This file allows the kinit command to authenticate to the KDC without manual input. [14]
- **The Ticket Cache:** When kinit successfully authenticates using the keytab, it obtains a Ticket Granting Ticket (TGT) and stores it in a file cache (typically /tmp/krb5cc_<uid>).
- **The ODBC Driver:** When pyodbc initiates a connection with Trusted_Connection=yes, the Microsoft ODBC Driver 18 detects the presence of the ticket cache and uses the TGT to request a Service Ticket for the SQL Server SPN. [10]

2.2 Snowflake: OAuth 2.0 Client Credentials

For a headless service, browser-based SSO (Okta/SAML) is prone to failure due to MFA prompts and UI changes. The robust standard for machine-to-machine (M2M) authentication in modern cloud environments is the **OAuth 2.0 Client Credentials Flow**.
In this model, the Python script acts as a confidential client. It does not impersonate a human user; rather, it authenticates as itself (a Service Principal).

- **Resource:** Snowflake is registered as an Enterprise Application in Azure AD (Entra ID).
- **Client:** The Python script is registered as a separate App Registration in Azure AD.
- **Trust Relationship:** Snowflake is configured with a SECURITY INTEGRATION object that explicitly trusts tokens issued by the specific Azure AD tenant.
- **Token Exchange:** The Python script requests a token from the Azure AD token endpoint (https://login.microsoftonline.com/<tenant>/oauth2/v2.0/token), presenting its Client ID and Secret. Azure returns a JWT access token.
- **Connection:** The script passes this JWT to the Snowflake Connector using the authenticator='oauth' parameter. [11]

---

3. Secrets Management Strategy: HashiCorp Vault

Hard-coding credentials in source code or environment variables is a significant security vulnerability. This architecture utilizes HashiCorp Vault as the central source of truth for all secrets.

3.1 AppRole Authentication

We utilize Vault's **AppRole** authentication method, which is specifically designed for machines and services. It creates a separation of concerns between the identity of the machine and the secrets it accesses.

- **Role ID:** This is the public identifier for the role. It is typically embedded in the application image or passed as a non-sensitive environment variable.
- **Secret ID:** This is the "password" for the role. It is sensitive and should be injected at runtime (e.g., via a Kubernetes sidecar, AWS Secrets Manager, or a CI/CD pipeline).

The Python script uses the hvac library to exchange these two credentials for a temporary Vault Token. This token has a limited Time-To-Live (TTL) and specific policies attached to it. [7]

3.2 KV Version 2 Secrets Engine

The architecture mandates the use of the **KV (Key-Value) Version 2** secrets engine. Unlike Version 1, Version 2 supports secret versioning and soft deletion, which are critical for operational safety.

- **Versioning:** Allows the operations team to rotate the Azure Client Secret or the SQL Keytab without breaking the application immediately, provided the application logic handles version retrieval correctly.
- **Structure:** Secrets in KV2 are stored as JSON objects. For this pipeline, a single secret path (e.g., secret/data/pipelines/mssql_to_snowflake) will contain multiple keys: mssql_spn, keytab_base64, client_id, client_secret, and tenant_id. [4]

---

4. Source System Analysis: Microsoft SQL Server

Optimizing the extraction from MS SQL Server is often the bottleneck in migration pipelines.

4.1 Connectivity Libraries: pyodbc

While pure Python libraries like pymssql exist, pyodbc is the industry standard for production workloads because it wraps the official Microsoft ODBC Driver. This ensures full support for enterprise features like Always Encrypted, Active Directory Integrated Authentication, and transparent failover in Availability Groups. [16]

4.2 Extraction Methodology: Chunking vs. Streaming

Attempting to load a massive table into a single pandas DataFrame will inevitably cause an Out-Of-Memory (OOM) crash. The solution is **chunked extraction**.

Using pandas.read_sql with the chunksize parameter returns an iterator rather than a single DataFrame. This allows the application to process N rows (e.g., 50,000) at a time. The memory footprint remains constant regardless of the total table size. [12]

Performance Note: To avoid "Row-By-Agonizing-Row" (RBAR) performance, the arraysize of the underlying cursor must be tuned. By default, pyodbc might fetch one row at a time from the network buffer. Setting cursor.fast_executemany = True (for writes) and tuning fetch buffer sizes (for reads) can improve throughput by orders of magnitude. [17]

---

5. Destination System Analysis: Snowflake

Snowflake's architecture separates storage from compute, allowing for unique ingestion strategies.

5.1 The write_pandas Abstraction

The snowflake-connector-python library includes a specialized module: snowflake.connector.pandas_tools. The write_pandas function within this module is the

preferred method for DataFrame ingestion.

Mechanism:

1. **Serialization:** It converts the pandas DataFrame chunks into Apache Parquet files on the local client (the Python container). Parquet is a columnar binary format that is highly compressible and optimized for Snowflake's micro-partitions.
2. **Staging:** It utilizes the Snowflake PUT command to upload these Parquet files to a temporary internal stage (e.g., a temporary table stage or user stage). This leverages Snowflake's internal file transfer protocols which optimize for network throughput and encryption.
3. **Loading:** It executes the COPY INTO command. This instructs the Snowflake compute warehouse to read the staged Parquet files and insert them into the target table. [13]

5.2 Comparison with S3 External Stages

An alternative approach is to write files to an AWS S3 bucket (External Stage) and then trigger a copy.

- **Pros of S3:** Creates a persistent Data Lake; allows other tools to access the raw data; decouples the extract from the load.
- **Cons of S3:** Requires managing AWS IAM credentials (AWS Access Keys or IAM Roles) within the Python script; higher latency due to S3 API overhead; increased cost (S3 storage + PUT requests).
- **Verdict:** For a direct migration project where the goal is simply moving data to Snowflake, write_pandas (using internal stages) is superior due to lower complexity and tighter security integration (no extra AWS keys needed). [18]

---

6. Technical Implementation: The Python Solution

This section provides the hands-on code and configuration required to build the solution.

6.1 System Prerequisites and ODBC Configuration

Before the Python code can execute, the underlying Linux OS must be configured to support Kerberos and ODBC.

Step 1: Install System Dependencies (Ubuntu/Debian Example)

Bash

```
# Update and install Kerberos and ODBC libraries
apt-get update && apt-get install -y \
  krb5-user \
  unixodbc \
  unixodbc-dev \
  libgssapi-krb5-2 \
```

```
    curl \
    gnupg

# Install Microsoft ODBC Driver 18 for SQL Server
curl https://packages.microsoft.com/keys/microsoft.asc | apt-key add -
curl https://packages.microsoft.com/config/ubuntu/22.04/prod.list >
/etc/apt/sources.list.d/mssql-release.list
apt-get update
ACCEPT_EULA=Y apt-get install -y msodbcsql18
```

Step 2: Configure Kerberos (/etc/krb5.conf) This file directs the Kerberos client to the correct Active Directory Domain Controllers. [9]

Ini, TOML

```
[libdefaults]
    default_realm = CORP.EXAMPLE.COM
    dns_lookup_realm = false
    dns_lookup_kdc = true
    ticket_lifetime = 24h
    renew_lifetime = 7d
    forwardable = true

[realms]
    CORP.EXAMPLE.COM = {
        kdc = ad-dc-01.corp.example.com
        admin_server = ad-dc-01.corp.example.com
        default_domain = corp.example.com
    }

[domain_realm]
   .corp.example.com = CORP.EXAMPLE.COM
    corp.example.com = CORP.EXAMPLE.COM
```

6.2 Module 1: HashiCorp Vault Integration (vault_service.py)

This module handles authentication and secret retrieval. It abstracts the complexity of the KV2 API.

Python

```
import hvac
import os
import sys

class VaultService:
    def __init__(self, vault_url, role_id, secret_id):
        self.client = hvac.Client(url=vault_url)
        self.role_id = role_id
        self.secret_id = secret_id
        self.is_authenticated = False

    def authenticate(self):
        """
        Authenticates using AppRole to obtain a client token.
        : AppRole login returns a token with specific policies attached.
        """
        try:
            self.client.auth.approle.login(
                role_id=self.role_id,
                secret_id=self.secret_id
            )
            self.is_authenticated = self.client.is_authenticated()
            if not self.is_authenticated:
                raise PermissionError("Vault authentication failed.")
            print("Successfully authenticated to Vault.")
        except Exception as e:
            print(f"Error authenticating to Vault: {str(e)}")
            sys.exit(1)

    def get_secret(self, path, mount_point='secret'):
        """
        Retrieves a secret from KV Version 2 engine.
        : The read_secret_version method is required for KV2.
        """
        if not self.is_authenticated:
            self.authenticate()

        try:
            # The actual secret data is nested deep in the response structure
            # response['data']['data'] contains the key-value pairs
            response = self.client.secrets.kv.v2.read_secret_version(
                path=path,
```

```
            mount_point=mount_point
        )
        return response['data']['data']
    except Exception as e:
        print(f"Failed to read secret at {path}: {str(e)}")
        raise
```

6.3 Module 2: MS SQL Connector with Kerberos (mssql_connector.py)

This module manages the kinit lifecycle and establishes the ODBC connection.

Python

```
import subprocess
import pyodbc
import pandas as pd
import time
from contextlib import contextmanager

class MSSQLConnector:
    def __init__(self, server, database, keytab_path, principal):
        self.server = server
        self.database = database
        self.keytab_path = keytab_path
        self.principal = principal
        self.driver = '{ODBC Driver 18 for SQL Server}'
        self.last_kinit_time = 0
        self.kinit_interval = 3600  # Refresh ticket every hour

    def _refresh_kerberos_ticket(self):
        """
        Invokes kinit via subprocess.
        [19]: This is necessary because pyodbc does not natively manage the ticket cache.
        """
        current_time = time.time()
        if current_time - self.last_kinit_time > self.kinit_interval:
            print(f"Refreshing Kerberos ticket for {self.principal}...")
            try:
                # Command: kinit -k -t /path/to/keytab user@REALM
                cmd = ['kinit', '-k', '-t', self.keytab_path, self.principal]
                subprocess.check_call(cmd)
                self.last_kinit_time = current_time
```

```
            print("Kerberos ticket refreshed successfully.")
        except subprocess.CalledProcessError as e:
            print(f"kinit failed with error code {e.returncode}")
            raise RuntimeError("Could not obtain Kerberos Ticket.")

    def get_connection_string(self):
        # : Trusted_Connection=yes triggers GSSAPI/Kerberos
        return (
            f"DRIVER={self.driver};"
            f"SERVER={self.server};"
            f"DATABASE={self.database};"
            f"Trusted_Connection=yes;"
            f"Encrypt=yes;"
            f"TrustServerCertificate=yes;" # Adjust based on internal CA trust
        )

    def get_data_iterator(self, query, chunksize=50000):
        """
        Returns a pandas DataFrame iterator to manage memory.
        """
        self._refresh_kerberos_ticket()
        conn_str = self.get_connection_string()

        # : Establish connection
        conn = pyodbc.connect(conn_str)
        try:
            # : chunksize parameter enables the iterator mode
            return pd.read_sql(query, conn, chunksize=chunksize), conn
        except Exception as e:
            conn.close()
            raise e
```

6.4 Module 3: Snowflake Connector with OAuth (snowflake_connector.py)

This module handles token acquisition from Azure AD and data loading.

Python

```
import snowflake.connector
from snowflake.connector.pandas_tools import write_pandas
```

```
import requests
import time

class SnowflakeConnector:
    def __init__(self, account, user, warehouse, database, schema,
                 tenant_id, client_id, client_secret):
        self.account = account
        self.user = user
        self.warehouse = warehouse
        self.database = database
        self.schema = schema

        # Azure AD Config
        self.tenant_id = tenant_id
        self.client_id = client_id
        self.client_secret = client_secret
        self.scope = "https://database.windows.net/.default"
        # Note: The scope might need to be specific to the Snowflake App Registration URI
        # e.g., 'api://<snowflake-app-id>/.default'

        self.token_url = f"https://login.microsoftonline.com/{tenant_id}/oauth2/v2.0/token"
        self.access_token = None
        self.token_expiry = 0

    def _get_oauth_token(self):
        """
        Fetches or refreshes the OAuth JWT from Azure AD.
        : Implements Client Credentials Flow.
        """
        if self.access_token and time.time() < self.token_expiry:
            return self.access_token

        print("Requesting new OAuth token from Azure AD...")
        payload = {
            'grant_type': 'client_credentials',
            'client_id': self.client_id,
            'client_secret': self.client_secret,
            'scope': self.scope
        }

        response = requests.post(self.token_url, data=payload)
        response.raise_for_status()
```

```
        data = response.json()
        self.access_token = data['access_token']
        # Set expiry with a safety buffer (e.g., 60 seconds)
        self.token_expiry = time.time() + int(data.get('expires_in', 3600)) - 60
        return self.access_token

    def load_data(self, dataframe, table_name):
        """
        Uses write_pandas to bulk load data via internal stage.
        """
        token = self._get_oauth_token()

        # [15]: Authenticator must be set to 'oauth'
        conn = snowflake.connector.connect(
            user=self.user,
            account=self.account,
            authenticator='oauth',
            token=token,
            warehouse=self.warehouse,
            database=self.database,
            schema=self.schema
        )

        try:
            # : write_pandas handles the PUT and COPY INTO logic
            # quote_identifiers=False assumes DB columns are upper-case or case-insensitive
            success, nchunks, nrows, _ = write_pandas(
                conn=conn,
                df=dataframe,
                table_name=table_name,
                quote_identifiers=False,
                chunk_size=100000 # Controls Parquet file size
            )
            return success, nrows
        finally:
            conn.close()
```

6.5 Orchestration Logic (main.py)

The controller script ties the modules together, managing the secure temporary file for the keytab and the iteration loop.

Python

```
import os
import tempfile
import base64
from modules.vault_service import VaultService
from modules.mssql_connector import MSSQLConnector
from modules.snowflake_connector import SnowflakeConnector

def main():
    # 1. Initialize Vault Connection
    # Role ID/Secret ID should be injected via CI/CD or Orchestrator
    vault = VaultService(
        vault_url=os.getenv('VAULT_ADDR'),
        role_id=os.getenv('VAULT_ROLE_ID'),
        secret_id=os.getenv('VAULT_SECRET_ID')
    )

    # 2. Retrieve Secrets from Vault KV2
    print("Fetching pipeline configuration from Vault...")
    secrets = vault.get_secret('data-pipelines/mssql-to-snowflake')

    # 3. Securely Materialize Keytab
    # The keytab is stored as a base64 encoded string in Vault to survive JSON serialization
    keytab_content = base64.b64decode(secrets['mssql_keytab_base64'])

    # Use NamedTemporaryFile to ensure cleanup on exit
    with tempfile.NamedTemporaryFile(delete=False, suffix='.keytab') as kt_file:
        kt_file.write(keytab_content)
        keytab_path = kt_file.name

    try:
        # 4. Configure Connectors
        mssql = MSSQLConnector(
            server=secrets['mssql_server_fqdn'],
            database='SalesDB',
            keytab_path=keytab_path,
            principal=secrets['mssql_spn'] # e.g., svc_loader@CORP.COM
        )

        sf = SnowflakeConnector(
            account=secrets['sf_account'],
            user=secrets['sf_user_email'], # Identity mapped in Snowflake
```

```
        warehouse='LOAD_WH',
        database='RAW_ZONE',
        schema='SALES',
        tenant_id=secrets['azure_tenant_id'],
        client_id=secrets['azure_client_id'],
        client_secret=secrets['azure_client_secret']
    )

    # 5. Define Extraction Strategy
    table_name = "TRANSACTIONS_HISTORY"
    # Use simple SELECT; logic handled by chunking.
    # For incremental loads, add WHERE clause based on high-watermark.
    query = "SELECT * FROM dbo.Transactions WHERE ModifiedDate >= '2023-01-01'"

    print(f"Starting migration for {table_name}...")

    # Get the iterator
    # chunksize=50000 balances memory usage vs network round-trips
    df_iterator, mssql_conn = mssql.get_data_iterator(query, chunksize=50000)

    total_rows = 0
    batch_num = 0

    for df_chunk in df_iterator:
        batch_num += 1
        print(f"Processing Batch {batch_num} ({len(df_chunk)} rows)...")

        # Optional: Data Transformation
        # df_chunk = pd.to_datetime(df_chunk)

        # Load to Snowflake
        success, rows_loaded = sf.load_data(df_chunk, table_name)

        if success:
            total_rows += rows_loaded
            print(f"Batch {batch_num} committed. Total rows: {total_rows}")
        else:
            print(f"Critical Error: Batch {batch_num} failed to load.")
            break

    print(f"Migration successfully completed. Total rows: {total_rows}")

finally:
```

```
        # 6. Critical Security Cleanup
        if os.path.exists(keytab_path):
            os.remove(keytab_path)
            print("Security Cleanup: Temporary keytab deleted.")

if __name__ == "__main__":
    main()
```

---

7. Performance Optimization and Memory Management

Building a pipeline for high-volume data requires careful consideration of compute resources.

7.1 Memory Profiling

Python's pandas library is notoriously memory-hungry. A general rule of thumb is that pandas requires 5x to 10x the RAM of the raw dataset size on disk.

- **Chunk Sizing:** With a chunksize of 50,000 rows, assuming an average row width of 500 bytes, the raw data is ~25MB. Pandas may allocate 250MB. This fits comfortably within a standard 2GB or 4GB container limit.
- **Garbage Collection:** In the main loop, the df_chunk variable is overwritten in each iteration. Python's reference counting usually handles this well, but for extremely tight loops, explicitly calling del df_chunk and gc.collect() at the end of the loop can prevent fragmentation.

7.2 Throughput Calculations

Throughput is a function of network bandwidth and serialization speed.

- **Network:** The write_pandas function uploads Parquet files. If the container runs in AWS (EC2/ECS), the bandwidth to Snowflake (also AWS) is extremely high (up to 25 Gbps). The bottleneck is usually the extraction from the on-premise MS SQL Server over VPN/DirectConnect.
- **Serialization:** The write_pandas function uses pyarrow for serialization. Setting parallel=N in the write_pandas call (where N is the number of vCPUs) allows the connector to upload multiple Parquet chunks simultaneously, saturating the available network bandwidth. [17]

7.3 Data Type Optimization

MS SQL DATETIME2 and MONEY types can cause friction.

- **Dates:** Ensure pandas parses dates as datetime64[ns] and not object (strings). String parsing in Parquet is slow and results in larger files.
- **Decimals:** Snowflake handles high-precision decimals well, but pandas often converts

them to floating-point numbers, risking precision loss. Using the use_logical_type=True parameter in write_pandas helps preserve decimal precision during the Parquet conversion.

---

8. Infrastructure and Deployment

8.1 Docker Containerization

To ensure consistency across development and production, the application should be containerized.
Dockerfile Optimization:

Dockerfile

```
FROM python:3.9-slim-bullseye

# Install System Dependencies (Kerberos, ODBC)
RUN apt-get update && apt-get install -y \
    krb5-user unixodbc unixodbc-dev libgssapi-krb5-2 curl gnupg \
    && curl https://packages.microsoft.com/keys/microsoft.asc | apt-key add - \
    && curl https://packages.microsoft.com/config/debian/11/prod.list >
/etc/apt/sources.list.d/mssql-release.list \
    && apt-get update \
    && ACCEPT_EULA=Y apt-get install -y msodbcsql18 \
    && rm -rf /var/lib/apt/lists/*

WORKDIR /app

# Install Python Dependencies
COPY requirements.txt.
RUN pip install --no-cache-dir -r requirements.txt

COPY..

# Entrypoint
CMD ["python", "main.py"]
```

8.2 Network Security

- **PrivateLink:** For maximum security, configure Snowflake to use AWS PrivateLink. This

ensures that data traffic between the Python orchestrator (in AWS VPC) and Snowflake never traverses the public internet.
- **VPC Peering/VPN:** The connection back to the on-premise MS SQL Server typically rides over a Site-to-Site VPN or AWS Direct Connect. Ensure the Security Groups allow outbound traffic on port 1433 (SQL Server) and 443 (Snowflake/Azure AD/Vault).

---

9. Conclusion

This report has outlined a comprehensive architecture for securely migrating data from MS SQL Server to Snowflake. By integrating HashiCorp Vault and utilizing Active Directory for both Kerberos (Source) and OAuth (Destination) authentication, the solution achieves a "Zero Trust" security posture suitable for highly regulated enterprise environments. The modular Python design, leveraging pandas chunking and write_pandas, ensures that the pipeline is both scalable and maintainable, capable of handling large-scale data loads without compromising on security or stability. This architecture represents the current state-of-the-art for batch data ingestion in hybrid cloud ecosystems.

Works cited

1. Tutorial: Use Active Directory Authentication for SQL Server on Linux ..., accessed February 9, 2026, https://learn.microsoft.com/en-us/sql/linux/sql-server-linux-active-directory-authentication?view=sql-server-ver17
2. Introduction to OAuth | Snowflake Documentation, accessed February 9, 2026, https://docs.snowflake.com/en/user-guide/oauth-intro
3. A Comprehensive Guide: Ingesting Data into Snowflake, accessed February 9, 2026, https://www.snowflake.com/en/developers/guides/a-comprehensive-guide-to-ingesting-data-into-snowflake/
4. Database — hvac 2.4.0 documentation, accessed February 9, 2026, https://python-hvac.org/en/stable/usage/secrets_engines/database.html
5. Overview of federated authentication and SSO - Snowflake Documentation, accessed February 9, 2026, https://docs.snowflake.com/en/user-guide/admin-security-fed-auth-overview
6. Data loading / unloading DDL - Snowflake Documentation, accessed February 9, 2026, https://docs.snowflake.com/en/sql-reference/ddl-stage
7. Approle — hvac 2.4.0 documentation, accessed February 9, 2026, https://python-hvac.org/en/stable/usage/auth_methods/approle.html
8. Custom Vault Integrations: Python - Shadow-Soft, accessed February 9, 2026, https://shadow-soft.com/content/custom-vault-integrations-python
9. BEN CHEN's Homepage - SQL Server windows authentication linux python - Google, accessed February 9, 2026, https://sites.google.com/site/hellobenchen/home/wiki/python/sql-server-windows

-authentication-linux-python
10. Using Integrated Authentication - ODBC Driver for SQL Server | Microsoft Learn, accessed February 9, 2026, https://learn.microsoft.com/en-us/sql/connect/odbc/linux-mac/using-integrated-authentication?view=sql-server-ver17
11. Connecting to Snowflake with the Python Connector, accessed February 9, 2026, https://docs.snowflake.com/en/developer-guide/python-connector/python-connector-connect
12. pandas.read_sql — pandas 3.0.0 documentation - PyData |, accessed February 9, 2026, https://pandas.pydata.org/docs/reference/api/pandas.read_sql.html
13. Python Connector API | Snowflake Documentation, accessed February 9, 2026, https://docs.snowflake.com/en/developer-guide/python-connector/python-connector-api.html#write_pandas
14. Access SQL Server with Trusted Connection in Linux with Python Using Kerberos Keytab - tech jogging, accessed February 9, 2026, https://techjogging.com/access-sqlserver-trusted-connection-linux-python-kerberos-keytab.html
15. Authenticating connections - Snowflake Documentation, accessed February 9, 2026, https://docs.snowflake.com/en/developer-guide/node-js/nodejs-driver-authenticate
16. Python Byte: Loading Data to Snowflake from SQL Server - Eric Heilman, accessed February 9, 2026, https://ericheilman.com/2024/01/29/python-byte-loading-data-to-snowflake-from-sql-server/
17. python - How to speed up reading SQL Server data into dataframe ..., accessed February 9, 2026, https://stackoverflow.com/questions/79625781/how-to-speed-up-reading-sql-server-data-into-dataframe
18. Load Data into Snowflake Using Python with Pandas - Simple Talk - Redgate Software, accessed February 9, 2026, https://www.red-gate.com/simple-talk/databases/snowflake/load-data-into-snowflake-using-python-with-pandas/