Complete Implementation Guide: Git-Style Versioned Lakehouse

Apache Iceberg + Project Nessie + Polars

📋 TABLE OF CONTENTS

🎯 PROJECT OVERVIEW {#overview}

This guide implements a production-ready data lakehouse using:

- **Apache Iceberg**: Open table format with ACID transactions
- **Project Nessie**: Git-like version control for data
- **Polars**: Fast data transformation engine
- **MinIO**: S3-compatible object storage

- **Medallion Architecture**: Bronze → Silver → Gold layers

Key Features:

- Git-like branching for data
- Write-Audit-Publish pattern
- Time-travel queries
- Data quality gates
- Complete test coverage

✅ PREREQUISITES {#prerequisites}

Required Software

```bash
# 1. Docker Desktop (latest)
https://www.docker.com/products/docker-desktop

# 2. Python 3.9+
python --version

# 3. Git
git --version

# 4. Code Editor (VS Code recommended)
```

System Requirements

- 8GB+ RAM (16GB recommended)
- 50GB free disk space
- Multi-core CPU (4+ cores)

🚀 PHASE 1: INFRASTRUCTURE SETUP {#phase1}

Step 1.1: Create Project Structure

```bash
```

```
# Create main directory
mkdir lakehouse-project
cd lakehouse-project

# Create subdirectories
mkdir -p {config,data/{raw,bronze,silver,gold},scripts/{bronze,silver,gold,utils},logs,tests,notebooks,orchestration/dags}
```

Step 1.2: Create docker-compose.yml

yaml

```yaml
version: '3.8'

services:
  minio:
    image: minio/minio:latest
    container_name: lakehouse-minio
    ports:
      - "9000:9000"
      - "9001:9001"
    environment:
      MINIO_ROOT_USER: admin
      MINIO_ROOT_PASSWORD: password123
    command: server /data --console-address ":9001"
    volumes:
      - minio-data:/data
    networks:
      - lakehouse-network
    healthcheck:
      test: ["CMD", "curl", "-f", "http://localhost:9000/minio/health/live"]
      interval: 30s
      timeout: 20s
      retries: 3

  nessie:
    image: projectnessie/nessie:latest
    container_name: lakehouse-nessie
    ports:
      - "19120:19120"
    environment:
      QUARKUS_HTTP_PORT: 19120
      NESSIE_VERSION_STORE_TYPE: INMEMORY
    networks:
      - lakehouse-network
    healthcheck:
      test: ["CMD", "curl", "-f", "http://localhost:19120/api/v2/config"]
      interval: 30s
      timeout: 10s
      retries: 3

volumes:
  minio-data:

networks:
```

```
  lakehouse-network:
    driver: bridge
```

Step 1.3: Start Infrastructure

```bash
# Start services
docker-compose up -d

# Verify running
docker-compose ps

# Check logs
docker-compose logs nessie
docker-compose logs minio

# Test connections
curl http://localhost:19120/api/v2/config
# Open browser: http://localhost:9001 (login: admin/password123)
```

⚙️ PHASE 2: STORAGE CONFIGURATION {#phase2}

Step 2.1: Create config/iceberg_config.py

```python
```

```
# MinIO/S3 Configuration
S3_ENDPOINT = "http://localhost:9000"
S3_ACCESS_KEY = "admin"
S3_SECRET_KEY = "password123"
S3_BUCKET = "lakehouse"

# Nessie Configuration
NESSIE_URI = "http://localhost:19120/api/v2"

# Iceberg Catalog
CATALOG_CONFIG = {
    "type": "rest",
    "uri": NESSIE_URI,
    "warehouse": f"s3://{S3_BUCKET}/warehouse",
    "s3.endpoint": S3_ENDPOINT,
    "s3.access-key-id": S3_ACCESS_KEY,
    "s3.secret-access-key": S3_SECRET_KEY,
    "s3.path-style-access": "true",
}

# Configuration
NAMESPACE = "ecommerce"
BRONZE_BRANCH = "bronze"
SILVER_BRANCH = "silver"
GOLD_BRANCH = "gold"
```

Step 2.2: Create scripts/utils/storage_utils.py

python

```
from pyiceberg.catalog import load_catalog
from pyiceberg.schema import Schema
from pyiceberg.types import NestedField, StringType, IntegerType, TimestampType, DoubleType, BooleanType
import sys
import os

sys.path.append(os.path.join(os.path.dirname(__file__), '..', '..'))
from config.iceberg_config import CATALOG_CONFIG, NAMESPACE

def get_catalog(branch="main"):
    config = CATALOG_CONFIG.copy()
    config["ref"] = branch
    return load_catalog("nessie", **config)

def create_namespace(catalog, namespace=NAMESPACE):
    try:
        catalog.create_namespace(namespace)
        print(f"✓ Created namespace: {namespace}")
    except Exception as e:
        if "already exists" in str(e).lower():
            print(f"✓ Namespace exists: {namespace}")
        else:
            raise

def table_exists(catalog, namespace, table_name):
    try:
        catalog.load_table(f"{namespace}.{table_name}")
        return True
    except:
        return False

def create_table_if_not_exists(catalog, namespace, table_name, schema):
    full_name = f"{namespace}.{table_name}"
    if table_exists(catalog, namespace, table_name):
        print(f"✓ Table exists: {full_name}")
        return catalog.load_table(full_name)

    table = catalog.create_table(identifier=full_name, schema=schema)
    print(f"✓ Created table: {full_name}")
    return table

# Schema Definitions
ORDERS_SCHEMA = Schema(
```

```
    NestedField(1, "order_id", StringType(), required=True),
    NestedField(2, "customer_id", StringType(), required=True),
    NestedField(3, "order_date", TimestampType(), required=True),
    NestedField(4, "total_amount", DoubleType(), required=True),
    NestedField(5, "status", StringType(), required=True),
    NestedField(6, "created_at", TimestampType(), required=True),
)

CUSTOMERS_SCHEMA = Schema(
    NestedField(1, "customer_id", StringType(), required=True),
    NestedField(2, "name", StringType(), required=True),
    NestedField(3, "email", StringType(), required=True),
    NestedField(4, "signup_date", TimestampType(), required=True),
    NestedField(5, "is_active", BooleanType(), required=True),
)
```

Step 2.3: Setup MinIO and Nessie

Create `scripts/utils/setup_minio.py`:

```
python

from minio import Minio

client = Minio("localhost:9000", access_key="admin", secret_key="password123", secure=False)

bucket_name = "lakehouse"
if not client.bucket_exists(bucket_name):
    client.make_bucket(bucket_name)
    print(f"✓ Created bucket: {bucket_name}")
else:
    print(f"✓ Bucket exists: {bucket_name}")
```

Create `scripts/utils/setup_nessie.py`:

```
python
```

```
import requests

NESSIE_URL = "http://localhost:19120/api/v2"

def create_branch(branch_name):
    payload = {"name": branch_name, "type": "BRANCH"}
    response = requests.post(f"{NESSIE_URL}/trees", json=payload, headers={"Content-Type": "application/json"})

    if response.status_code in [200, 201]:
        print(f"✓ Created branch: {branch_name}")
    elif response.status_code == 409:
        print(f"✓ Branch exists: {branch_name}")
    else:
        print(f"✗ Failed: {response.text}")

for branch in ["bronze", "silver", "gold"]:
    create_branch(branch)

response = requests.get(f"{NESSIE_URL}/trees")
if response.status_code == 200:
    branches = response.json().get("references", [])
    print("\nBranches:")
    for b in branches:
        print(f"  - {b['name']}")
```

PHASE 3: PYTHON ENVIRONMENT {#phase3}

Step 3.1: Create requirements.txt

txt

```
pyiceberg==0.5.1
requests==2.31.0
polars==0.20.0
pandas==2.1.4
pyarrow==14.0.1
minio==7.2.0
boto3==1.34.0
python-dotenv==1.0.0
pyyaml==6.0.1
pytest==7.4.3
```

Step 3.2: Setup Virtual Environment

```bash
# Create virtual environment
python -m venv venv

# Activate (Linux/Mac)
source venv/bin/activate

# Activate (Windows)
venv\Scripts\activate

# Install dependencies
pip install --upgrade pip
pip install -r requirements.txt
```

Step 3.3: Initialize System

```bash
# Install minio package first
pip install minio requests

# Setup MinIO bucket
python scripts/utils/setup_minio.py

# Setup Nessie branches
python scripts/utils/setup_nessie.py
```

Step 3.4: Generate Sample Data

Create `scripts/utils/generate_sample_data.py`:

```python
```

```
import polars as pl
from datetime import datetime, timedelta
import random
import os

def generate_orders_data(num_records=1000):
    start_date = datetime(2024, 1, 1)
    data = {
        "order_id": [f"ORD{i:06d}" for i in range(1, num_records + 1)],
        "customer_id": [f"CUST{random.randint(1, 200):04d}" for _ in range(num_records)],
        "order_date": [start_date + timedelta(days=random.randint(0, 365)) for _ in range(num_records)],
        "total_amount": [round(random.uniform(10, 1000), 2) for _ in range(num_records)],
        "status": [random.choice(["pending", "completed", "cancelled", "refunded"]) for _ in range(num_records)],
        "created_at": [datetime.now() for _ in range(num_records)],
    }
    return pl.DataFrame(data)

def generate_customers_data(num_records=200):
    data = {
        "customer_id": [f"CUST{i:04d}" for i in range(1, num_records + 1)],
        "name": [f"Customer {i}" for i in range(1, num_records + 1)],
        "email": [f"customer{i}@example.com" for i in range(1, num_records + 1)],
        "signup_date": [datetime(2023, 1, 1) + timedelta(days=random.randint(0, 500)) for _ in range(num_records)],
        "is_active": [random.choice([True, False]) for _ in range(num_records)],
    }
    return pl.DataFrame(data)

os.makedirs("data/raw", exist_ok=True)
orders_df = generate_orders_data(1000)
orders_df.write_csv("data/raw/orders.csv")
print(f"✓ Generated orders.csv: {len(orders_df)} records")

customers_df = generate_customers_data(200)
customers_df.write_csv("data/raw/customers.csv")
print(f"✓ Generated customers.csv: {len(customers_df)} records")
```

Run it:

```
bash

python scripts/utils/generate_sample_data.py
```

🥉 PHASE 4: BRONZE LAYER {#phase4}

Step 4.1: Create scripts/bronze/ingest_orders.py

python

```python
import polars as pl
import sys
import os
from datetime import datetime

sys.path.append(os.path.join(os.path.dirname(__file__), '..', '..'))
from scripts.utils.storage_utils import get_catalog, create_namespace, create_table_if_not_exists, ORDERS_SCHEMA
from config.iceberg_config import NAMESPACE, BRONZE_BRANCH

def ingest_orders_to_bronze():
    print("Starting Bronze Orders Ingestion...")

    # Read raw data
    df = pl.read_csv("data/raw/orders.csv")
    print(f"✓ Loaded {len(df)} records")

    # Connect to Nessie
    catalog = get_catalog(branch=BRONZE_BRANCH)
    create_namespace(catalog, NAMESPACE)

    # Create table
    table = create_table_if_not_exists(catalog, NAMESPACE, "orders_bronze", ORDERS_SCHEMA)

    # Write data
    arrow_table = df.to_arrow()
    table.append(arrow_table)
    print(f"✓ Wrote {len(df)} records to Bronze")

if __name__ == "__main__":
    try:
        ingest_orders_to_bronze()
    except Exception as e:
        print(f"✗ Error: {e}")
        import traceback
        traceback.print_exc()
        sys.exit(1)
```

Step 4.2: Create scripts/bronze/ingest_customers.py

python

```python
import polars as pl
import sys
import os

sys.path.append(os.path.join(os.path.dirname(__file__), '..', '..'))
from scripts.utils.storage_utils import get_catalog, create_namespace, create_table_if_not_exists, CUSTOMERS_SCHEMA
from config.iceberg_config import NAMESPACE, BRONZE_BRANCH

def ingest_customers_to_bronze():
    print("Starting Bronze Customers Ingestion...")
    df = pl.read_csv("data/raw/customers.csv")
    print(f"✓ Loaded {len(df)} records")

    catalog = get_catalog(branch=BRONZE_BRANCH)
    create_namespace(catalog, NAMESPACE)
    table = create_table_if_not_exists(catalog, NAMESPACE, "customers_bronze", CUSTOMERS_SCHEMA)

    arrow_table = df.to_arrow()
    table.append(arrow_table)
    print(f"✓ Wrote {len(df)} records to Bronze")

if __name__ == "__main__":
    try:
        ingest_customers_to_bronze()
    except Exception as e:
        print(f"✗ Error: {e}")
        sys.exit(1)
```

Step 4.3: Test Bronze Ingestion

bash

```bash
python scripts/bronze/ingest_orders.py
python scripts/bronze/ingest_customers.py
```

PHASE 5: SILVER LAYER {#phase5}

Step 5.1: Create scripts/silver/transform_orders.py

```
python
```

```
import polars as pl
import sys
import os
from datetime import datetime
from pyiceberg.schema import Schema
from pyiceberg.types import NestedField, StringType, TimestampType, DoubleType, IntegerType

sys.path.append(os.path.join(os.path.dirname(__file__), '..', '..'))
from scripts.utils.storage_utils import get_catalog, create_namespace, create_table_if_not_exists
from config.iceberg_config import NAMESPACE, BRONZE_BRANCH, SILVER_BRANCH

ORDERS_SILVER_SCHEMA = Schema(
    NestedField(1, "order_id", StringType(), required=True),
    NestedField(2, "customer_id", StringType(), required=True),
    NestedField(3, "order_date", TimestampType(), required=True),
    NestedField(4, "total_amount", DoubleType(), required=True),
    NestedField(5, "status", StringType(), required=True),
    NestedField(6, "year", IntegerType(), required=True),
    NestedField(7, "month", IntegerType(), required=True),
    NestedField(8, "quarter", IntegerType(), required=True),
    NestedField(9, "is_completed", IntegerType(), required=True),
    NestedField(10, "processed_at", TimestampType(), required=True),
)

def transform_orders_to_silver():
    print("Starting Silver Orders Transformation...")

    # Read from Bronze
    bronze_catalog = get_catalog(branch=BRONZE_BRANCH)
    bronze_table = bronze_catalog.load_table(f"{NAMESPACE}.orders_bronze")
    df = pl.from_arrow(bronze_table.scan().to_arrow())
    print(f"✓ Loaded {len(df)} records from Bronze")

    # Clean data
    initial_count = len(df)
    df = df.unique(subset=["order_id"])
    df = df.filter(pl.col("total_amount") > 0)
    df = df.with_columns(pl.col("status").str.to_lowercase().str.strip_chars())
    print(f"✓ Cleaned data: {initial_count} → {len(df)} records")

    # Enrich data
    df = df.with_columns([
        pl.col("order_date").dt.year().alias("year"),
```

```
        pl.col("order_date").dt.month().alias("month"),
        pl.col("order_date").dt.quarter().alias("quarter"),
        pl.when(pl.col("status") == "completed").then(1).otherwise(0).alias("is_completed"),
        pl.lit(datetime.now()).alias("processed_at"),
    ])
    print(f"✓ Enriched data with derived columns")

    # Write to Silver
    silver_catalog = get_catalog(branch=SILVER_BRANCH)
    create_namespace(silver_catalog, NAMESPACE)
    silver_table = create_table_if_not_exists(silver_catalog, NAMESPACE, "orders_silver", ORDERS_SILVER_SCHEMA)

    arrow_table = df.to_arrow()
    silver_table.append(arrow_table)
    print(f"✓ Wrote {len(df)} records to Silver")

if __name__ == "__main__":
    try:
        transform_orders_to_silver()
    except Exception as e:
        print(f"✗ Error: {e}")
        import traceback
        traceback.print_exc()
        sys.exit(1)
```

Step 5.2: Create scripts/silver/transform_customers.py

python

```
import polars as pl
import sys
import os
from datetime import datetime
from pyiceberg.schema import Schema
from pyiceberg.types import NestedField, StringType, TimestampType, BooleanType

sys.path.append(os.path.join(os.path.dirname(__file__), '..', '..'))
from scripts.utils.storage_utils import get_catalog, create_namespace, create_table_if_not_exists
from config.iceberg_config import NAMESPACE, BRONZE_BRANCH, SILVER_BRANCH

CUSTOMERS_SILVER_SCHEMA = Schema(
    NestedField(1, "customer_id", StringType(), required=True),
    NestedField(2, "name", StringType(), required=True),
    NestedField(3, "email", StringType(), required=True),
    NestedField(4, "signup_date", TimestampType(), required=True),
    NestedField(5, "is_active", BooleanType(), required=True),
    NestedField(6, "email_domain", StringType(), required=True),
    NestedField(7, "processed_at", TimestampType(), required=True),
)

def transform_customers_to_silver():
    print("Starting Silver Customers Transformation...")

    bronze_catalog = get_catalog(branch=BRONZE_BRANCH)
    bronze_table = bronze_catalog.load_table(f"{NAMESPACE}.customers_bronze")
    df = pl.from_arrow(bronze_table.scan().to_arrow())
    print(f"✓ Loaded {len(df)} records")

    df = df.unique(subset=["customer_id"])
    df = df.filter(pl.col("email").is_not_null())
    df = df.with_columns([
        pl.col("email").str.to_lowercase(),
        pl.col("email").str.extract(r"@(.+)$", 1).alias("email_domain"),
        pl.lit(datetime.now()).alias("processed_at")
    ])
    print(f"✓ Cleaned and enriched data")

    silver_catalog = get_catalog(branch=SILVER_BRANCH)
    create_namespace(silver_catalog, NAMESPACE)
    silver_table = create_table_if_not_exists(silver_catalog, NAMESPACE, "customers_silver", CUSTOMERS_SILVER_SCH

    arrow_table = df.to_arrow()
```

```
    silver_table.append(arrow_table)
    print(f"✓ Wrote {len(df)} records to Silver")

if __name__ == "__main__":
    try:
        transform_customers_to_silver()
    except Exception as e:
        print(f"✗ Error: {e}")
        sys.exit(1)
```

Step 5.3: Test Silver Transformations

bash

```
python scripts/silver/transform_orders.py
python scripts/silver/transform_customers.py
```

🏅 PHASE 6: GOLD LAYER {#phase6}

Step 6.1: Create scripts/gold/create_order_analytics.py

python

```
import polars as pl
import sys
import os
from datetime import datetime
from pyiceberg.schema import Schema
from pyiceberg.types import NestedField, StringType, IntegerType, DoubleType, TimestampType

sys.path.append(os.path.join(os.path.dirname(__file__), '..', '..'))
from scripts.utils.storage_utils import get_catalog, create_namespace, create_table_if_not_exists
from config.iceberg_config import NAMESPACE, SILVER_BRANCH, GOLD_BRANCH

ORDER_ANALYTICS_SCHEMA = Schema(
    NestedField(1, "year", IntegerType(), required=True),
    NestedField(2, "month", IntegerType(), required=True),
    NestedField(3, "quarter", IntegerType(), required=True),
    NestedField(4, "status", StringType(), required=True),
    NestedField(5, "total_orders", IntegerType(), required=True),
    NestedField(6, "total_revenue", DoubleType(), required=True),
    NestedField(7, "avg_order_value", DoubleType(), required=True),
    NestedField(8, "created_at", TimestampType(), required=True),
)

def create_order_analytics():
    print("Starting Gold Order Analytics...")

    silver_catalog = get_catalog(branch=SILVER_BRANCH)
    silver_table = silver_catalog.load_table(f"{NAMESPACE}.orders_silver")
    df = pl.from_arrow(silver_table.scan().to_arrow())
    print(f"✓ Loaded {len(df)} records from Silver")

    analytics_df = df.group_by(["year", "month", "quarter", "status"]).agg([
        pl.count().alias("total_orders"),
        pl.sum("total_amount").alias("total_revenue"),
        pl.mean("total_amount").alias("avg_order_value"),
    ])

    analytics_df = analytics_df.with_columns([pl.lit(datetime.now()).alias("created_at")])
    print(f"✓ Created {len(analytics_df)} aggregated rows")

    gold_catalog = get_catalog(branch=GOLD_BRANCH)
    create_namespace(gold_catalog, NAMESPACE)
    gold_table = create_table_if_not_exists(gold_catalog, NAMESPACE, "order_analytics", ORDER_ANALYTICS_SCHEM
```

```
    arrow_table = analytics_df.to_arrow()
    gold_table.append(arrow_table)
    print(f"✓ Wrote {len(analytics_df)} analytics to Gold")

if __name__ == "__main__":
    try:
        create_order_analytics()
    except Exception as e:
        print(f"✗ Error: {e}")
        sys.exit(1)
```

Step 6.2: Create scripts/gold/create_customer_analytics.py

python

```
import polars as pl
import sys
import os
from datetime import datetime
from pyiceberg.schema import Schema
from pyiceberg.types import NestedField, StringType, IntegerType, DoubleType, TimestampType

sys.path.append(os.path.join(os.path.dirname(__file__), '..', '..'))
from scripts.utils.storage_utils import get_catalog, create_namespace, create_table_if_not_exists
from config.iceberg_config import NAMESPACE, SILVER_BRANCH, GOLD_BRANCH

CUSTOMER_ANALYTICS_SCHEMA = Schema(
    NestedField(1, "customer_id", StringType(), required=True),
    NestedField(2, "customer_name", StringType(), required=True),
    NestedField(3, "total_orders", IntegerType(), required=True),
    NestedField(4, "total_spent", DoubleType(), required=True),
    NestedField(5, "avg_order_value", DoubleType(), required=True),
    NestedField(6, "created_at", TimestampType(), required=True),
)

def create_customer_analytics():
    print("Starting Gold Customer Analytics...")

    silver_catalog = get_catalog(branch=SILVER_BRANCH)
    customers_table = silver_catalog.load_table(f"{NAMESPACE}.customers_silver")
    customers_df = pl.from_arrow(customers_table.scan().to_arrow())

    orders_table = silver_catalog.load_table(f"{NAMESPACE}.orders_silver")
    orders_df = pl.from_arrow(orders_table.scan().to_arrow())
    print(f"✓ Loaded customers and orders")

    order_metrics = orders_df.group_by("customer_id").agg([
        pl.count().alias("total_orders"),
        pl.sum("total_amount").alias("total_spent"),
        pl.mean("total_amount").alias("avg_order_value"),
    ])

    analytics_df = customers_df.join(order_metrics, on="customer_id", how="inner")
    analytics_df = analytics_df.select([
        pl.col("customer_id"),
        pl.col("name").alias("customer_name"),
        pl.col("total_orders"),
        pl.col("total_spent"),
```

```
        pl.col("avg_order_value"),
        pl.lit(datetime.now()).alias("created_at"),
    ])
    print(f"✓ Created analytics for {len(analytics_df)} customers")

    gold_catalog = get_catalog(branch=GOLD_BRANCH)
    create_namespace(gold_catalog, NAMESPACE)
    gold_table = create_table_if_not_exists(gold_catalog, NAMESPACE, "customer_analytics", CUSTOMER_ANALYTICS_

    arrow_table = analytics_df.to_arrow()
    gold_table.append(arrow_table)
    print(f"✓ Wrote {len(analytics_df)} customer analytics to Gold")

if __name__ == "__main__":
    try:
        create_customer_analytics()
    except Exception as e:
        print(f"✗ Error: {e}")
        sys.exit(1)
```

Step 6.3: Test Gold Layer

bash

```
python scripts/gold/create_order_analytics.py
python scripts/gold/create_customer_analytics.py
```

✅ PHASE 7: QUALITY CHECKS {#phase7}

Step 7.1: Create scripts/utils/quality_checks.py

python

```python
import polars as pl
from typing import List

class DataQualityCheck:
    def __init__(self, table_name: str):
        self.table_name = table_name
        self.checks_passed = []
        self.checks_failed = []

    def check_row_count(self, df: pl.DataFrame, min_rows: int = 1) -> bool:
        row_count = len(df)
        if row_count >= min_rows:
            self.checks_passed.append(f"✓ Row count: {row_count} >= {min_rows}")
            return True
        else:
            self.checks_failed.append(f"✗ Row count: {row_count} < {min_rows}")
            return False

    def check_no_nulls(self, df: pl.DataFrame, columns: List[str]) -> bool:
        all_passed = True
        for col in columns:
            null_count = df[col].null_count()
            if null_count == 0:
                self.checks_passed.append(f"✓ No nulls in {col}")
            else:
                self.checks_failed.append(f"✗ {null_count} nulls in {col}")
                all_passed = False
        return all_passed

    def check_unique(self, df: pl.DataFrame, column: str) -> bool:
        total = len(df)
        unique = df[column].n_unique()
        if total == unique:
            self.checks_passed.append(f"✓ All unique in {column}")
            return True
        else:
            self.checks_failed.append(f"✗ {total - unique} duplicates in {column}")
            return False

    def print_report(self) -> bool:
        print(f"\n{'='*60}")
        print(f'Quality Report: {self.table_name}')
        print(f'{'='*60}')
```

```
        print(f"\nPASSED ({len(self.checks_passed)}):")
        for check in self.checks_passed:
            print(f"  {check}")

        if self.checks_failed:
            print(f"\nFAILED ({len(self.checks_failed)}):")
            for check in self.checks_failed:
                print(f"  {check}")

        print(f"\n{'='*60}\n")
        return len(self.checks_failed) == 0

def run_bronze_quality_checks(df: pl.DataFrame) -> bool:
    qc = DataQualityCheck("orders_bronze")
    qc.check_row_count(df, min_rows=1)
    qc.check_no_nulls(df, ["order_id", "customer_id", "total_amount"])
    qc.check_unique(df, "order_id")
    return qc.print_report()
```

🔄 PHASE 8: ORCHESTRATION {#phase8}

Step 8.1: Create Master Pipeline

Create `scripts/run_full_pipeline.py`:

```
python
```

```python
import sys
import os
from datetime import datetime

sys.path.append(os.path.dirname(__file__))

from bronze.ingest_orders import ingest_orders_to_bronze
from bronze.ingest_customers import ingest_customers_to_bronze
from silver.transform_orders import transform_orders_to_silver
from silver.transform_customers import transform_customers_to_silver
from gold.create_order_analytics import create_order_analytics
from gold.create_customer_analytics import create_customer_analytics

def run_full_pipeline():
    print("\n" + "="*70)
    print(" FULL LAKEHOUSE PIPELINE")
    print("="*70)
    print(f"Start: {datetime.now()}\n")

    stages = [
        ("Bronze - Orders", ingest_orders_to_bronze),
        ("Bronze - Customers", ingest_customers_to_bronze),
        ("Silver - Orders", transform_orders_to_silver),
        ("Silver - Customers", transform_customers_to_silver),
        ("Gold - Order Analytics", create_order_analytics),
        ("Gold - Customer Analytics", create_customer_analytics),
    ]

    for stage_name, stage_func in stages:
        print(f"\n{'='*70}")
        print(f"STAGE: {stage_name}")
        print(f"{'='*70}\n")

        try:
            stage_func()
            print(f"\n✓ {stage_name} completed")
        except Exception as e:
            print(f"\n✗ {stage_name} failed: {e}")
            sys.exit(1)

    print(f"\n{'='*70}")
    print("✓ PIPELINE COMPLETED SUCCESSFULLY")
    print(f"End: {datetime.now()}")
```

```
    print(f"{'='*70}\n")

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

Step 8.2: Test Full Pipeline

```
bash

python scripts/run_full_pipeline.py
```

🧪 PHASE 9: TESTING {#phase9}

Step 9.1: Create Test Suite

Create `tests/test_pipeline.py`:

```
python
```

```python
import unittest
import polars as pl
import sys
import os

sys.path.append(os.path.join(os.path.dirname(__file__), '..'))
from scripts.utils.storage_utils import get_catalog, table_exists
from config.iceberg_config import NAMESPACE, BRONZE_BRANCH, SILVER_BRANCH, GOLD_BRANCH

class TestPipeline(unittest.TestCase):

    def test_bronze_tables_exist(self):
        catalog = get_catalog(branch=BRONZE_BRANCH)
        self.assertTrue(table_exists(catalog, NAMESPACE, "orders_bronze"))
        self.assertTrue(table_exists(catalog, NAMESPACE, "customers_bronze"))

    def test_bronze_has_data(self):
        catalog = get_catalog(branch=BRONZE_BRANCH)
        table = catalog.load_table(f"{NAMESPACE}.orders_bronze")
        df = pl.from_arrow(table.scan().to_arrow())
        self.assertGreater(len(df), 0)

    def test_silver_tables_exist(self):
        catalog = get_catalog(branch=SILVER_BRANCH)
        self.assertTrue(table_exists(catalog, NAMESPACE, "orders_silver"))
        self.assertTrue(table_exists(catalog, NAMESPACE, "customers_silver"))

    def test_gold_tables_exist(self):
        catalog = get_catalog(branch=GOLD_BRANCH)
        self.assertTrue(table_exists(catalog, NAMESPACE, "order_analytics"))
        self.assertTrue(table_exists(catalog, NAMESPACE, "customer_analytics"))

    def test_data_quality(self):
        catalog = get_catalog(branch=SILVER_BRANCH)
        table = catalog.load_table(f"{NAMESPACE}.orders_silver")
        df = pl.from_arrow(table.scan().to_arrow())

        # Check required columns exist
        self.assertIn("order_id", df.columns)
        self.assertIn("year", df.columns)
        self.assertIn("month", df.columns)

        # Check no nulls in key columns
```

```
        self.assertEqual(df["order_id"].null_count(), 0)

        # Check value ranges
        self.assertTrue((df["month"] >= 1).all() and (df["month"] <= 12).all())

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

Step 9.2: Run Tests

```
bash

python tests/test_pipeline.py
```

🚀 PHASE 10: PRODUCTION DEPLOYMENT {#phase10}

Step 10.1: Query Utilities

Create `scripts/utils/query_tables.py`:

```
python
```

```python
import sys
import os
import polars as pl

sys.path.append(os.path.join(os.path.dirname(__file__), '..', '..'))
from scripts.utils.storage_utils import get_catalog
from config.iceberg_config import NAMESPACE

def query_table(branch: str, table_name: str, limit: int = 10):
    print(f"\nQuerying {NAMESPACE}.{table_name} on branch '{branch}'")
    print(f"{'='*60}\n")

    try:
        catalog = get_catalog(branch=branch)
        table = catalog.load_table(f"{NAMESPACE}.{table_name}")

        data = table.scan().to_arrow()
        df = pl.from_arrow(data)

        print(f"Total Records: {len(df)}")
        print(f"\nFirst {min(limit, len(df))} rows:")
        print(df.head(limit))

        return df
    except Exception as e:
        print(f"✗ Error: {e}")
        return None

if __name__ == "__main__":
    print("=== BRONZE LAYER ===")
    query_table("bronze", "orders_bronze", limit=5)

    print("\n=== SILVER LAYER ===")
    query_table("silver", "orders_silver", limit=5)

    print("\n=== GOLD LAYER ===")
    query_table("gold", "order_analytics", limit=10)
```

Step 10.2: Monitoring Dashboard

Create `scripts/utils/monitoring.py`:

```python
```

```
import requests
import sys
import os

sys.path.append(os.path.join(os.path.dirname(__file__), '..', '..'))
from scripts.utils.storage_utils import get_catalog
from config.iceberg_config import NAMESPACE

def check_infrastructure():
    print("\n" + "="*60)
    print(" INFRASTRUCTURE HEALTH")
    print("="*60 + "\n")

    # Check Nessie
    try:
        response = requests.get("http://localhost:19120/api/v2/config", timeout=5)
        if response.status_code == 200:
            print("✓ Nessie: Running")
        else:
            print(f"✗ Nessie: Status {response.status_code}")
    except:
        print("✗ Nessie: Not accessible")

    # Check MinIO
    try:
        response = requests.get("http://localhost:9000/minio/health/live", timeout=5)
        if response.status_code == 200:
            print("✓ MinIO: Running")
        else:
            print(f"✗ MinIO: Status {response.status_code}")
    except:
        print("✗ MinIO: Not accessible")

def check_tables():
    print("\n" + "="*60)
    print(" TABLE STATISTICS")
    print("="*60 + "\n")

    tables = [
        ("bronze", "orders_bronze"),
        ("bronze", "customers_bronze"),
        ("silver", "orders_silver"),
        ("silver", "customers_silver"),
```

```
        ("gold", "order_analytics"),
        ("gold", "customer_analytics"),
    ]

    for branch, table_name in tables:
        try:
            catalog = get_catalog(branch=branch)
            table = catalog.load_table(f"{NAMESPACE}.{table_name}")
            data = table.scan().to_arrow()
            print(f"{branch}/{table_name}: {len(data):,} records")
        except Exception as e:
            print(f"{branch}/{table_name}: ✗ Error - {e}")

if __name__ == "__main__":
    check_infrastructure()
    check_tables()
    print()
```

⚡ QUICK START COMMANDS {#quickstart}

Initial Setup (One Time)

```
bash

# 1. Start infrastructure
docker-compose up -d

# 2. Setup Python environment
python -m venv venv
source venv/bin/activate  # or venv\Scripts\activate on Windows
pip install -r requirements.txt

# 3. Initialize system
python scripts/utils/setup_minio.py
python scripts/utils/setup_nessie.py

# 4. Generate sample data
python scripts/utils/generate_sample_data.py
```

Run Pipeline

```
bash
```

```
# Run full pipeline
python scripts/run_full_pipeline.py

# Or run individual stages
python scripts/bronze/ingest_orders.py
python scripts/silver/transform_orders.py
python scripts/gold/create_order_analytics.py
```

Query and Monitor

```
bash

# Query tables
python scripts/utils/query_tables.py

# Check system health
python scripts/utils/monitoring.py

# Run tests
python tests/test_pipeline.py
```

Stop Services

```
bash

docker-compose down
```

🔧 TROUBLESHOOTING {#troubleshooting}

Problem: Docker containers won't start

```
bash

# Check Docker is running
docker info

# Check for port conflicts
docker-compose ps

# Remove old containers and restart
docker-compose down -v
docker-compose up -d
```

Problem: Cannot connect to Nessie

```bash
# Test Nessie API
curl http://localhost:19120/api/v2/config

# Check logs
docker logs lakehouse-nessie

# Restart Nessie
docker-compose restart nessie
```

Problem: MinIO access denied

```bash
# Verify credentials match in:
# - docker-compose.yml
# - config/iceberg_config.py

# Access MinIO console
# http://localhost:9001
# Login: admin / password123
```

Problem: Module not found errors

```bash
# Ensure virtual environment is activated
source venv/bin/activate  # or venv\Scripts\activate

# Reinstall requirements
pip install --upgrade pip
pip install -r requirements.txt

# Verify installation
python -c "import pyiceberg; print('OK')"
```

Problem: Table not found

```bash
```

```
# Check branch exists
curl http://localhost:19120/api/v2/trees

# Verify you're on correct branch in your script
# catalog = get_catalog(branch="bronze")

# Re-run ingestion if needed
python scripts/bronze/ingest_orders.py
```

Problem: Quality checks failing

```
python

# Debug data issues
import polars as pl

df = pl.read_csv("data/raw/orders.csv")
print(df.schema)
print(df.describe())
print(df.null_count())
```

Problem: Out of memory

```
bash

# Increase Docker memory in Docker Desktop settings
# Or process data in smaller batches
```

Problem: Slow queries

```
python

# Use predicate pushdown
scan = table.scan(row_filter="year = 2024")

# Select only needed columns
scan = table.scan(selected_fields=("order_id", "total_amount"))
```

📊 PROJECT STRUCTURE

```
lakehouse-project/
├── config/
```

```
│   └── iceberg_config.py         # Configuration
├── data/
│   ├── raw/                      # Raw CSV files
│   ├── bronze/                   # (MinIO handles storage)
│   ├── silver/
│   └── gold/
├── scripts/
│   ├── bronze/
│   │   ├── ingest_orders.py
│   │   └── ingest_customers.py
│   ├── silver/
│   │   ├── transform_orders.py
│   │   └── transform_customers.py
│   ├── gold/
│   │   ├── create_order_analytics.py
│   │   └── create_customer_analytics.py
│   ├── utils/
│   │   ├── storage_utils.py
│   │   ├── quality_checks.py
│   │   ├── generate_sample_data.py
│   │   ├── setup_minio.py
│   │   ├── setup_nessie.py
│   │   ├── query_tables.py
│   │   └── monitoring.py
│   └── run_full_pipeline.py
├── tests/
│   └── test_pipeline.py
├── logs/
├── docker-compose.yml
└── requirements.txt
```

📝 EXECUTION CHECKLIST

- [ ] Docker Desktop installed and running
- [ ] Python 3.9+ installed
- [ ] Project directory created
- [ ] docker-compose.yml created
- [ ] Containers started (`docker-compose up -d`)
- [ ] MinIO accessible (http://localhost:9001)
- [ ] Nessie accessible (http://localhost:19120)
- [ ] Virtual environment created

- [ ] Requirements installed
- [ ] MinIO bucket created
- [ ] Nessie branches created
- [ ] Sample data generated
- [ ] Bronze ingestion tested
- [ ] Silver transformation tested
- [ ] Gold analytics tested
- [ ] Full pipeline runs successfully
- [ ] Tests pass
- [ ] Monitoring dashboard works

🎉 SUCCESS!

You now have a complete Git-style versioned lakehouse with:

✅ **Medallion Architecture** (Bronze → Silver → Gold) ✅ **Version Control** for data (Nessie branches) ✅ **Quality Checks** (Write-Audit-Publish) ✅ **Production Ready** code ✅ **Complete Testing** suite ✅ **Monitoring** capabilities

Next Steps:

1. **Customize** for your data sources
2. **Add** more transformations
3. **Implement** Airflow orchestration
4. **Deploy** to production
5. **Scale** as needed

Resources:

- Apache Iceberg: https://iceberg.apache.org/
- Project Nessie: https://projectnessie.org/
- Polars: https://pola-rs.github.io/polars/

Happy Data Engineering! 🚀