Comprehensive Research: Databricks Apps, Databricks Lakehouse, and Databricks Asset Bundles

Research Date: October 20, 2025
Sources: Official Databricks Documentation, Databricks Community, GitHub Open Source Repositories

---

Table of Contents

Executive Summary

This comprehensive research document provides an in-depth analysis of three critical components of the Databricks Data Intelligence Platform: **Databricks Apps**, **Databricks Lakehouse**, and **Databricks Asset Bundles**. Each technology represents a fundamental capability for building modern data and AI applications on Databricks.

Databricks Apps enables developers to build and deploy production-ready data and AI applications directly within the Databricks platform, supporting popular frameworks like Streamlit, Dash, Flask, Gradio, and React. **Databricks Lakehouse** provides a unified architecture combining the best of data lakes and data warehouses, implementing the medallion architecture pattern for progressive data quality improvement. **Databricks Asset Bundles** offers infrastructure-as-code capabilities for managing Databricks resources, enabling robust CI/CD workflows and deployment automation.

This research synthesizes information from official Databricks documentation, community resources, and open-source GitHub repositories to provide actionable guidance for developers, data engineers, and architects.

Databricks Apps

Overview and Architecture

Databricks Apps is a production-ready platform for building, deploying, and hosting data and AI applications directly within the Databricks ecosystem. Released in October 2024, it provides a new modality for serving interactive applications that leverage the full power of the Databricks Data Intelligence Platform.

Core Value Proposition

Databricks Apps democratizes data intelligence by enabling even non-technical business analysts to access organizational data through intuitive, application-based interfaces. It eliminates the complexity of managing separate infrastructure for applications while maintaining enterprise-grade security, governance, and scalability.

Architecture Components

Databricks Apps operates as a **containerized service model** within the Databricks platform:

1. **Compute Layer**: Apps run on dedicated Databricks compute resources with configurable CPU and memory
2. **Runtime Environment**: Containerized execution environment supporting Python and Node.js runtimes
3. **Integration Layer**: Native integration with Databricks services (Unity Catalog, SQL Warehouses, Feature Store, Model Serving)
4. **Security Layer**: OAuth 2.0 authentication with dual identity model (app identity and user identity)
5. **Deployment Layer**: Automated deployment pipeline with version control and rollback capabilities

Key Concepts

App Structure

Every Databricks App consists of:

- **Source Code**: Application logic written in supported frameworks
- **Configuration File** (`app.yaml`): Defines runtime behavior, environment variables, and resource requirements
- **Dependencies**: Managed through `requirements.txt` (Python) or `package.json` (Node.js)
- **Static Assets**: Images, CSS, JavaScript files stored in the app directory
- **Compute Resources**: Configurable CPU/memory allocation

Dual Identity Model

Databricks Apps implements a sophisticated authentication model:

1. **App Identity**: Service principal or user account under which the app runs
2. **User Identity**: Individual user accessing the app, used for personalized data access

This enables apps to: - Access shared resources using the app identity - Enforce row-level security based on user identity - Maintain audit trails for compliance

Supported Frameworks

Python Frameworks

Framework	Type	Best For	Key Features
Streamlit	Data Apps	Rapid prototyping, dashboards	Simple API, reactive programming, built-in widgets
Dash	Analytics Apps	Complex dashboards, callbacks	Plotly integration, enterprise features
Gradio	ML Interfaces	Model demos, ML workflows	Auto-generated UI, easy sharing
Flask	Web Apps	Custom backends, APIs	Full control, lightweight, extensible
FastAPI	APIs	High-performance APIs	Async support, automatic documentation

Node.js Frameworks

Framework	Type	Best For	Key Features
React	Frontend	Interactive UIs	Component-based, virtual DOM, rich ecosystem
Angular	Frontend	Enterprise apps	Full framework, TypeScript, dependency injection
Svelte	Frontend	Lightweight apps	Compile-time optimization, minimal runtime
Express	Backend	APIs, middleware	Minimalist, flexible routing

Development Workflow

Step-by-Step Development Process

1. Environment Setup

```
# Install required dependencies
pip install gradio pandas databricks-sdk

# Create project directory
mkdir my-databricks-app
cd my-databricks-app
```

2. Create Application Code

Example Streamlit app (`app.py`):

```python
import streamlit as st
import pandas as pd
from databricks import sql
import os

# Get environment variables set by Databricks
warehouse_id = os.getenv("DATABRICKS_WAREHOUSE_ID")
host = os.getenv("DATABRICKS_HOST")

st.title("Sales Analytics Dashboard")

# Connect to Databricks SQL Warehouse
@st.cache_resource
def get_connection():
    return sql.connect(
        server_hostname=host,
        http_path=f"/sql/1.0/warehouses/{warehouse_id}",
        credentials_provider=lambda: {}  # Uses app identity
    )

# Query data
def load_data(query):
    conn = get_connection()
    cursor = conn.cursor()
    cursor.execute(query)
    return cursor.fetchall_arrow().to_pandas()

# UI Components
date_range = st.date_input("Select Date Range", [])
region = st.selectbox("Region", ["North", "South", "East", "West"])

if st.button("Load Data"):
    query = f"""
        SELECT date, region, SUM(sales) as total_sales
        FROM sales_data
        WHERE region = '{region}'
        GROUP BY date, region
        ORDER BY date
    """
    df = load_data(query)
    st.line_chart(df.set_index('date')['total_sales'])
    st.dataframe(df)
```

3. Configure App Runtime (`app.yaml`)

```yaml
command: ['streamlit', 'run', 'app.py', '--server.port=8080']

env:
  - name: 'DATABRICKS_WAREHOUSE_ID'
    value: 'abc123def456'
  - name: 'STREAMLIT_GATHER_USAGE_STATS'
    value: 'false'
  - name: 'CATALOG_NAME'
    value: 'production'
  - name: 'SCHEMA_NAME'
    value: 'sales'
```

4. Define Dependencies (`requirements.txt`)

```
streamlit==1.28.0
pandas==2.1.0
databricks-sql-connector==3.0.0
plotly==5.17.0
```

5. Local Development and Testing

```
# Run locally
python app.py

# Or use Databricks CLI for local debugging
databricks apps run-local --prepare-environment --debug
```

6. Deploy to Databricks

```
# Deploy using Databricks CLI
databricks apps deploy my-app \
  --source-path . \
  --compute-size SMALL

# Or deploy via UI
# Navigate to Workspace → Apps → Create App
```

Configuration and Deployment

App.yaml Configuration Reference

The `app.yaml` file controls app execution behavior:

```yaml
# Command to start the application
command:
  - gunicorn
  - app:app
  - -w
  - 4
  - --bind
  - 0.0.0.0:8080

# Environment variables
env:
  # Hardcoded values
  - name: 'APP_ENV'
    value: 'production'

  # Reference secrets from Databricks Secrets
  - name: 'API_KEY'
    valueFrom:
      secretKeyRef:
        scope: 'my-scope'
        key: 'api-key'

  # Reference Unity Catalog volumes
  - name: 'DATA_PATH'
    value: '/Volumes/catalog/schema/volume'

  # SQL Warehouse configuration
  - name: 'WAREHOUSE_ID'
    value: '${var.warehouse_id}'
```

Compute Size Configuration

Size	vCPUs	Memory	Best For
SMALL	2	8 GB	Development, low-traffic apps
MEDIUM	4	16 GB	Production apps, moderate traffic
LARGE	8	32 GB	High-traffic apps, complex processing
XLARGE	16	64 GB	Enterprise apps, heavy workloads

Deployment Logic

Databricks Apps uses intelligent deployment logic:

Default Behavior: - **Python apps**: Executes `python <first_py_file>` - **Node.js apps**: Executes `npm run start`

Custom Commands: Override defaults in `app.yaml`:

```
# Flask with Gunicorn
command: ['gunicorn', 'app:app', '-w', '4', '--bind', '0.0.0.0:8000']

# Streamlit with custom port
command: ['streamlit', 'run', 'app.py', '--server.port=8501']

# FastAPI with Uvicorn
command: ['uvicorn', 'main:app', '--host', '0.0.0.0', '--port', '8000']
```

Authentication and Security

OAuth 2.0 Integration

Databricks Apps implements enterprise-grade OAuth 2.0 authentication:

App-Level Authentication:

```
from databricks.sdk import WorkspaceClient

# Automatic authentication using app identity
w = WorkspaceClient()

# Access Unity Catalog
tables = w.tables.list(catalog_name="main", schema_name="default")
```

User-Level Authentication:

```
import os
from databricks.sdk.core import Config, oauth_service_principal

# Get current user context
user_email = os.getenv("DATABRICKS_USER_EMAIL")

# Enforce row-level security
query = f"""
    SELECT * FROM sensitive_data
    WHERE authorized_user = '{user_email}'
"""
```

Security Best Practices

1. **Secret Management**: Store API keys and credentials in Databricks Secrets ```yaml env:
    - name: 'OPENAI_API_KEY' valueFrom: secretKeyRef: scope: 'ml-secrets' key: 'openai-key' ```

2. **Unity Catalog Integration**: Leverage Unity Catalog for data governance `python # Access governed data spark.sql("USE CATALOG production") df = spark.table("customers") # Automatically enforces ACLs`

3. **Network Security**: Apps run in isolated containers with controlled network access

4. **Audit Logging**: All app access and data queries are logged for compliance

Best Practices

Development Best Practices

1. **Modular Architecture** `my-app/ ├── app.py # Main entry point ├── components/ # Reusable UI components │ ├── header.py │ ├── sidebar.py │ └── charts.py ├── utils/ # Utility functions │ ├── data_loader.py │ └── auth.py ├── assets/ # Static files │ ├── logo.png │ └── styles.css ├── app.yaml # App configuration └── requirements.txt # Dependencies`

2. **Environment-Specific Configuration** ```python import os

ENV = os.getenv("APP_ENV", "development")

if ENV == "production": DEBUG = False WAREHOUSE_ID = os.getenv("PROD_WAREHOUSE_ID") else: DEBUG = True WAREHOUSE_ID = os.getenv("DEV_WAREHOUSE_ID") ```

1. **Caching and Performance** ```python import streamlit as st

@st.cache_data(ttl=3600) # Cache for 1 hour def load_large_dataset(): return spark.table("large_table").toPandas()

@st.cache_resource def get_db_connection(): return sql.connect(...) ```

1. **Error Handling** `python try: data = load_data(query) except Exception as e: st.error(f"Failed to load data: {str(e)}") st.stop()`

Deployment Best Practices

1. **Use Databricks Asset Bundles** for managing apps as code

2. **Implement CI/CD pipelines** for automated testing and deployment
3. **Version Control**: Store app code in Git repositories
4. **Monitor Performance**: Use Databricks monitoring tools to track app metrics
5. **Implement Health Checks**: Add endpoints for monitoring app status

Code Examples

Example 1: Dash Analytics App

```
 import dash
from dash import dcc, html, Input, Output
import plotly.express as px
from databricks import sql
import pandas as pd
import os

app = dash.Dash(__name__)

# Databricks connection
def get_data(query):
    connection = sql.connect(
        server_hostname=os.getenv("DATABRICKS_HOST"),
        http_path=f"/sql/1.0/warehouses/{os.getenv('WAREHOUSE_ID')}"
    )
    cursor = connection.cursor()
    cursor.execute(query)
    return cursor.fetchall_arrow().to_pandas()

# Layout
app.layout = html.Div([
    html.H1("Sales Performance Dashboard"),

    dcc.Dropdown(
        id='region-dropdown',
        options=[
            {'label': 'North America', 'value': 'NA'},
            {'label': 'Europe', 'value': 'EU'},
            {'label': 'Asia Pacific', 'value': 'APAC'}
        ],
        value='NA'
    ),

    dcc.Graph(id='sales-graph'),
    dcc.Graph(id='trend-graph')
])

# Callbacks
@app.callback(
    [Output('sales-graph', 'figure'),
     Output('trend-graph', 'figure')],
    [Input('region-dropdown', 'value')]
)
def update_graphs(region):
    query = f"""
        SELECT product, SUM(revenue) as total_revenue
        FROM sales
        WHERE region = '{region}'
        GROUP BY product
    """
    df = get_data(query)

    fig1 = px.bar(df, x='product', y='total_revenue',
                  title=f'Sales by Product - {region}')

    trend_query = f"""
        SELECT date, SUM(revenue) as daily_revenue
        FROM sales
        WHERE region = '{region}'
```

```
        GROUP BY date
        ORDER BY date
    ```

    df_trend = get_data(trend_query)
    fig2 = px.line(df_trend, x='date', y='daily_revenue',
                   title='Revenue Trend')

    return fig1, fig2

if __name__ == '__main__':
    app.run_server(host='0.0.0.0', port=8080)
```
```

Example 2: Flask API with ML Model

```python
from flask import Flask, request, jsonify
from databricks.sdk import WorkspaceClient
import mlflow
import os

app = Flask(__name__)

# Load ML model from MLflow
w = WorkspaceClient()
model_name = os.getenv("MODEL_NAME")
model = mlflow.pyfunc.load_model(f"models:/{model_name}/Production")

@app.route('/health', methods=['GET'])
def health():
    return jsonify({"status": "healthy"}), 200

@app.route('/predict', methods=['POST'])
def predict():
    try:
        data = request.get_json()
        features = data.get('features')

        # Make prediction
        prediction = model.predict([features])

        return jsonify({
            "prediction": prediction[0],
            "model_version": os.getenv("MODEL_VERSION")
        }), 200

    except Exception as e:
        return jsonify({"error": str(e)}), 500

@app.route('/batch-predict', methods=['POST'])
def batch_predict():
    try:
        data = request.get_json()
        table_name = data.get('table_name')

        # Load data from Unity Catalog
        df = spark.table(table_name)

        # Batch prediction
        predictions = model.predict(df)

        # Save results
        result_table = f"{table_name}_predictions"
        predictions_df = df.withColumn("prediction", predictions)
        predictions_df.write.mode("overwrite").saveAsTable(result_table)

        return jsonify({
            "status": "success",
            "result_table": result_table,
            "rows_processed": df.count()
        }), 200

    except Exception as e:
        return jsonify({"error": str(e)}), 500

if __name__ == '__main__':
    app.run(host='0.0.0.0', port=8000)
```

Example 3: Gradio ML Interface

```python
import gradio as gr
from databricks.sdk import WorkspaceClient
import mlflow
import pandas as pd

# Load model
model = mlflow.pyfunc.load_model("models:/customer-churn/Production")

def predict_churn(age, tenure, monthly_charges, total_charges):
    """Predict customer churn probability"""
    features = pd.DataFrame({
        'age': [age],
        'tenure': [tenure],
        'monthly_charges': [monthly_charges],
        'total_charges': [total_charges]
    })

    prediction = model.predict(features)[0]
    probability = model.predict_proba(features)[0][1]

    return {
        "Churn Prediction": "Yes" if prediction == 1 else "No",
        "Churn Probability": f"{probability:.2%}",
        "Retention Recommendation": get_recommendation(probability)
    }

def get_recommendation(probability):
    if probability > 0.7:
        return "High Risk - Immediate intervention required"
    elif probability > 0.4:
        return "Medium Risk - Monitor and engage"
    else:
        return "Low Risk - Standard retention program"

# Create Gradio interface
interface = gr.Interface(
    fn=predict_churn,
    inputs=[
        gr.Number(label="Customer Age"),
        gr.Number(label="Tenure (months)"),
        gr.Number(label="Monthly Charges ($)"),
        gr.Number(label="Total Charges ($)")
    ],
    outputs=gr.JSON(label="Prediction Results"),
    title="Customer Churn Prediction",
    description="Predict customer churn probability using ML model",
    examples=[
        [45, 24, 75.50, 1810.00],
        [32, 6, 120.00, 720.00],
        [58, 60, 55.25, 3315.00]
    ]
)

if __name__ == "__main__":
    interface.launch(server_name="0.0.0.0", server_port=7860)
```

Databricks Lakehouse

Architecture Overview

The **Databricks Lakehouse** represents a paradigm shift in data architecture, combining the scalability and flexibility of data lakes with the performance and ACID guarantees of data warehouses. It provides a unified platform for all data workloads—from ETL and BI to machine learning and real-time analytics.

Core Principles

1. **Unified Platform**: Single platform for all data workloads (batch, streaming, ML, BI)
2. **Open Standards**: Built on open formats (Delta Lake, Apache Iceberg, Parquet)
3. **ACID Transactions**: Full transactional guarantees for data reliability
4. **Schema Evolution**: Support for schema changes without breaking existing queries
5. **Time Travel**: Query historical versions of data for auditing and recovery
6. **Unified Governance**: Centralized governance through Unity Catalog

Core Components

1. Delta Lake

Delta Lake is the foundational storage layer providing ACID transactions on data lakes:

```python
# Create Delta table
df.write.format("delta").mode("overwrite").save("/mnt/delta/events")

# Read Delta table
df = spark.read.format("delta").load("/mnt/delta/events")

# Time travel
df_yesterday = spark.read.format("delta") \
    .option("versionAsOf", 1) \
    .load("/mnt/delta/events")

# Update data with ACID guarantees
from delta.tables import DeltaTable

deltaTable = DeltaTable.forPath(spark, "/mnt/delta/events")
deltaTable.update(
    condition = "eventType = 'click'",
    set = { "processed": "true" }
)

# Merge (upsert) operation
deltaTable.alias("target").merge(
    source.alias("source"),
    "target.id = source.id"
).whenMatchedUpdate(set = {
    "value": "source.value",
    "updated_at": "current_timestamp()"
}).whenNotMatchedInsert(values = {
    "id": "source.id",
    "value": "source.value",
    "created_at": "current_timestamp()"
}).execute()
```

Key Features: - **ACID Transactions**: Atomicity, Consistency, Isolation, Durability - **Scalable Metadata**: Handles petabyte-scale tables efficiently - **Time Travel**: Access historical data versions - **Schema Enforcement**: Prevents bad data from corrupting tables - **Audit History**: Complete history of all changes

2. Unity Catalog

Unity Catalog provides centralized governance for data and AI assets:

```sql
-- Create catalog
CREATE CATALOG production;

-- Create schema
CREATE SCHEMA production.sales;

-- Create managed table
CREATE TABLE production.sales.transactions (
    transaction_id STRING,
    customer_id STRING,
    amount DECIMAL(10,2),
    transaction_date DATE
) USING DELTA;

-- Grant permissions
GRANT SELECT ON TABLE production.sales.transactions TO `data_analysts`;
GRANT MODIFY ON TABLE production.sales.transactions TO `data_engineers`;

-- Row-level security
CREATE FUNCTION production.sales.filter_region(region STRING)
RETURNS BOOLEAN
RETURN current_user() IN (
    SELECT user_email FROM production.sales.regional_access
    WHERE allowed_region = region
);

ALTER TABLE production.sales.transactions
SET ROW FILTER production.sales.filter_region(region) ON (region);
```

Capabilities: - **Multi-cloud governance**: Works across AWS, Azure, GCP - **Fine-grained access control**: Table, column, and row-level security - **Data lineage**: Track data flow across transformations - **Audit logging**: Complete audit trail of data access - **Centralized metadata**: Single source of truth for all data assets

3. Apache Spark and Photon

Apache Spark provides distributed processing, while **Photon** is Databricks' vectorized query engine:

```python
# Spark DataFrame operations
df = spark.read.table("production.sales.transactions")

# Complex transformations
result = df.filter(col("amount") > 1000) \
    .groupBy("customer_id") \
    .agg(
        sum("amount").alias("total_spent"),
        count("*").alias("transaction_count"),
        avg("amount").alias("avg_transaction")
    ) \
    .orderBy(desc("total_spent"))

# Write results
result.write.mode("overwrite").saveAsTable("production.sales.customer_summary")

# Photon automatically accelerates queries
spark.conf.set("spark.databricks.photon.enabled", "true")
```

Reference Architecture

The Databricks Lakehouse reference architecture consists of seven functional layers:

Layer 1: Source

Data originates from multiple sources:

- **Structured Sources**: Relational databases (PostgreSQL, MySQL, SQL Server, Oracle)
- **Semi-Structured**: JSON, XML, Avro, Parquet files
- **Unstructured**: Logs, images, videos, documents
- **Streaming**: Kafka, Kinesis, Event Hubs, IoT devices
- **SaaS Applications**: Salesforce, Workday, SAP via Lakeflow Connect

Layer 2: Ingest

Multiple ingestion patterns:

Batch Ingestion:

```python
# Auto Loader for incremental file ingestion
df = spark.readStream.format("cloudFiles") \
    .option("cloudFiles.format", "json") \
    .option("cloudFiles.schemaLocation", "/mnt/schema/events") \
    .load("/mnt/landing/events/")

df.writeStream \
    .format("delta") \
    .option("checkpointLocation", "/mnt/checkpoints/events") \
    .table("bronze.events")
```

Streaming Ingestion:

```python
# Kafka streaming
df = spark.readStream \
    .format("kafka") \
    .option("kafka.bootstrap.servers", "kafka:9092") \
    .option("subscribe", "transactions") \
    .load()

# Parse and write to Delta
parsed_df = df.selectExpr("CAST(value AS STRING) as json") \
    .select(from_json("json", schema).alias("data")) \
    .select("data.*")

parsed_df.writeStream \
    .format("delta") \
    .outputMode("append") \
    .option("checkpointLocation", "/mnt/checkpoints/kafka") \
    .table("bronze.transactions")
```

Lakeflow Connect (Built-in Connectors):

```python
# Ingest from Salesforce
import dlt

@dlt.table
def salesforce_accounts():
    return spark.read.format("salesforce") \
        .option("sfObject", "Account") \
        .load()
```

Layer 3: Storage

Data stored in cloud object storage (S3, ADLS, GCS) using open formats:

- **Delta Lake tables**: Primary storage format
- **Apache Iceberg**: Alternative open table format
- **Parquet**: Columnar storage for analytics
- **Unity Catalog Volumes**: For unstructured data (PDFs, images, models)

Layer 4: Transform

Data transformation using multiple engines:

Delta Live Tables (Declarative ETL):

```
import dlt
from pyspark.sql.functions import *

@dlt.table(
    comment="Raw events from source systems",
    table_properties={"quality": "bronze"}
)
def bronze_events():
    return spark.readStream.table("source.events")

@dlt.table(
    comment="Cleaned and validated events",
    table_properties={"quality": "silver"}
)
@dlt.expect_or_drop("valid_timestamp", "timestamp IS NOT NULL")
@dlt.expect_or_drop("valid_user", "user_id IS NOT NULL")
def silver_events():
    return dlt.read_stream("bronze_events") \
        .withColumn("processed_at", current_timestamp()) \
        .dropDuplicates(["event_id"])

@dlt.table(
    comment="Aggregated user metrics",
    table_properties={"quality": "gold"}
)
def gold_user_metrics():
    return dlt.read("silver_events") \
        .groupBy("user_id", window("timestamp", "1 day")) \
        .agg(
            count("*").alias("event_count"),
            countDistinct("session_id").alias("session_count")
        )
```

Spark SQL:

```sql
-- Complex analytical query
WITH customer_segments AS (
    SELECT
        customer_id,
        SUM(amount) as total_spent,
        COUNT(*) as transaction_count,
        CASE
            WHEN SUM(amount) > 10000 THEN 'VIP'
            WHEN SUM(amount) > 5000 THEN 'Premium'
            ELSE 'Standard'
        END as segment
    FROM production.sales.transactions
    WHERE transaction_date >= '2024-01-01'
    GROUP BY customer_id
)
SELECT
    segment,
    COUNT(*) as customer_count,
    AVG(total_spent) as avg_spent,
    SUM(total_spent) as segment_revenue
FROM customer_segments
GROUP BY segment
ORDER BY segment_revenue DESC;
```

Layer 5: Query/Process

Multiple compute options for different workloads:

Compute Type	Best For	Characteristics
SQL Warehouses	BI, analytics, ad-hoc queries	Serverless, auto-scaling, optimized for SQL
All-Purpose Clusters	Interactive development, notebooks	Persistent, customizable, multi-language
Job Clusters	Scheduled ETL, batch processing	Ephemeral, cost-effective, isolated
Serverless Compute	On-demand workloads	Instant startup, pay-per-use

Layer 6: Serve

Data serving for different consumption patterns:

Data Warehousing:

```sql
-- Create materialized view for BI
CREATE MATERIALIZED VIEW production.analytics.sales_summary AS
SELECT
    DATE_TRUNC('month', transaction_date) as month,
    product_category,
    region,
    SUM(amount) as total_revenue,
    COUNT(DISTINCT customer_id) as unique_customers
FROM production.sales.transactions
GROUP BY month, product_category, region;

-- Optimize for query performance
OPTIMIZE production.analytics.sales_summary
ZORDER BY (month, product_category);
```

Model Serving:

```python
# Deploy ML model for real-time serving
from databricks.sdk import WorkspaceClient
from databricks.sdk.service.serving import ServedEntityInput,
EndpointCoreConfigInput

w = WorkspaceClient()

w.serving_endpoints.create(
    name="customer-churn-model",
    config=EndpointCoreConfigInput(
        served_entities=[
            ServedEntityInput(
                entity_name="main.ml_models.customer_churn",
                entity_version="3",
                workload_size="Small",
                scale_to_zero_enabled=True
            )
        ]
    )
)
```

Lakebase (OLTP):

```sql
-- Create OLTP database instance
CREATE DATABASE INSTANCE my_oltp_db
WITH (
    instance_type = 'db.t3.medium',
    storage_size = 100
);

-- Create transactional table
CREATE TABLE my_oltp_db.orders (
    order_id SERIAL PRIMARY KEY,
    customer_id INTEGER NOT NULL,
    order_date TIMESTAMP DEFAULT CURRENT_TIMESTAMP,
    status VARCHAR(20),
    total_amount DECIMAL(10,2)
);

-- Sync to Delta Lake for analytics
CREATE SYNCED TABLE production.sales.orders_sync
AS SELECT * FROM my_oltp_db.orders;
```

Layer 7: Analysis

Final consumption layer:

- **BI Tools**: Tableau, Power BI, Looker connected via JDBC/ODBC
- **Databricks SQL Editor**: Native SQL interface with dashboards
- **Notebooks**: Interactive analysis with Python, R, Scala, SQL
- **Applications**: Databricks Apps for custom interfaces
- **APIs**: REST APIs for programmatic access

Medallion Architecture

The **Medallion Architecture** is a data design pattern that organizes data into three layers—Bronze, Silver, and Gold—representing progressive levels of data quality and refinement.

Bronze Layer (Raw Data)

Purpose: Ingest and preserve raw data in its original form

Characteristics: - Append-only, immutable data - Minimal transformation - Preserves data lineage - Enables reprocessing - Schema-on-read approach

Implementation:

```
# Bronze layer: Raw ingestion
from pyspark.sql.functions import current_timestamp, input_file_name

bronze_df = spark.readStream \
    .format("cloudFiles") \
    .option("cloudFiles.format", "json") \
    .option("cloudFiles.schemaLocation", "/mnt/schemas/events") \
    .load("/mnt/landing/events/") \
    .withColumn("ingestion_timestamp", current_timestamp()) \
    .withColumn("source_file", input_file_name())

bronze_df.writeStream \
    .format("delta") \
    .option("checkpointLocation", "/mnt/checkpoints/bronze_events") \
    .option("mergeSchema", "true") \
    .table("bronze.events")
```

Best Practices: - Store all fields as STRING or VARIANT to handle schema changes - Add metadata columns (ingestion_timestamp, source_file, source_system) - Use Auto Loader for incremental ingestion - Enable schema evolution - Retain all historical data

Silver Layer (Validated Data)

Purpose: Clean, validate, and enrich data for reliable consumption

Characteristics: - Schema enforcement - Data quality checks - Deduplication - Standardization - Type casting - Enrichment with reference data

Implementation:

```python
from delta.tables import DeltaTable
from pyspark.sql.functions import *

# Silver layer: Validation and cleaning
def process_to_silver():
    # Read from bronze
    bronze_df = spark.readStream.table("bronze.events")

    # Data quality transformations
    silver_df = bronze_df \
        .filter(col("event_id").isNotNull()) \
        .filter(col("timestamp").isNotNull()) \
        .filter(col("timestamp") >= "2020-01-01") \
        .withColumn("event_date", to_date("timestamp")) \
        .withColumn("event_hour", hour("timestamp")) \
        .withColumn("user_id", col("user_id").cast("long")) \
        .withColumn("amount", col("amount").cast("decimal(10,2)")) \
        .dropDuplicates(["event_id"]) \
        .withColumn("processed_timestamp", current_timestamp())

    # Write to silver with data quality expectations
    silver_df.writeStream \
        .format("delta") \
        .outputMode("append") \
        .option("checkpointLocation", "/mnt/checkpoints/silver_events") \
        .foreachBatch(lambda batch_df, batch_id:
            write_with_quality_checks(batch_df, "silver.events")
        ) \
        .start()

def write_with_quality_checks(df, table_name):
    # Quality metrics
    total_records = df.count()
    null_user_ids = df.filter(col("user_id").isNull()).count()
    invalid_amounts = df.filter(col("amount") < 0).count()

    # Log quality metrics
    quality_metrics = spark.createDataFrame([{
        "table": table_name,
        "timestamp": datetime.now(),
        "total_records": total_records,
        "null_user_ids": null_user_ids,
        "invalid_amounts": invalid_amounts,
        "quality_score": 1 - ((null_user_ids + invalid_amounts) /
total_records)
    }])

    quality_metrics.write.mode("append").saveAsTable("monitoring.data_quality")

    # Write validated data
    df.write.format("delta").mode("append").saveAsTable(table_name)
```

Data Quality Checks:

```sql
-- Add constraints to silver tables
ALTER TABLE silver.events ADD CONSTRAINT valid_amount CHECK (amount >= 0);
ALTER TABLE silver.events ADD CONSTRAINT valid_timestamp CHECK (timestamp IS
NOT NULL);

-- Create expectations with Delta Live Tables
@dlt.expect_or_drop("valid_user_id", "user_id IS NOT NULL")
@dlt.expect_or_drop("valid_event_type", "event_type IN ('click', 'view',
'purchase')")
@dlt.expect_or_fail("critical_data", "amount IS NOT NULL AND amount > 0")
```

Gold Layer (Business-Level Aggregates)

Purpose: Provide curated, business-ready datasets optimized for analytics

Characteristics: - Business logic applied - Aggregated metrics - Dimensional modeling - Optimized for query performance - Aligned with business requirements

Implementation:

```python
# Gold layer: Business aggregates
from pyspark.sql.window import Window

# Customer 360 view
customer_360 = spark.sql("""
    SELECT
        c.customer_id,
        c.customer_name,
        c.customer_segment,
        c.registration_date,

        -- Transaction metrics
        COUNT(DISTINCT t.transaction_id) as total_transactions,
        SUM(t.amount) as lifetime_value,
        AVG(t.amount) as avg_transaction_value,
        MAX(t.transaction_date) as last_transaction_date,
        DATEDIFF(CURRENT_DATE(), MAX(t.transaction_date)) as
days_since_last_purchase,

        -- Product preferences
        COLLECT_LIST(DISTINCT t.product_category) as purchased_categories,

        -- Engagement metrics
        COUNT(DISTINCT e.session_id) as total_sessions,
        SUM(CASE WHEN e.event_type = 'view' THEN 1 ELSE 0 END) as total_views,
        SUM(CASE WHEN e.event_type = 'click' THEN 1 ELSE 0 END) as
total_clicks,

        -- Risk indicators
        CASE
            WHEN DATEDIFF(CURRENT_DATE(), MAX(t.transaction_date)) > 90 THEN
'High'
            WHEN DATEDIFF(CURRENT_DATE(), MAX(t.transaction_date)) > 30 THEN
'Medium'
            ELSE 'Low'
        END as churn_risk

    FROM silver.customers c
    LEFT JOIN silver.transactions t ON c.customer_id = t.customer_id
    LEFT JOIN silver.events e ON c.customer_id = e.user_id
    GROUP BY c.customer_id, c.customer_name, c.customer_segment,
c.registration_date
""")

customer_360.write.mode("overwrite").saveAsTable("gold.customer_360")

# Optimize for analytics
spark.sql("OPTIMIZE gold.customer_360 ZORDER BY (customer_id,
customer_segment)")
```

Dimensional Modeling:

```sql
-- Fact table: Sales transactions
CREATE TABLE gold.fact_sales (
    transaction_key BIGINT,
    date_key INT,
    customer_key INT,
    product_key INT,
    store_key INT,
    quantity INT,
    unit_price DECIMAL(10,2),
    discount_amount DECIMAL(10,2),
    tax_amount DECIMAL(10,2),
    total_amount DECIMAL(10,2)
) USING DELTA
PARTITIONED BY (date_key);

-- Dimension table: Customers
CREATE TABLE gold.dim_customer (
    customer_key INT,
    customer_id STRING,
    customer_name STRING,
    customer_segment STRING,
    customer_tier STRING,
    registration_date DATE,
    effective_date DATE,
    end_date DATE,
    is_current BOOLEAN
) USING DELTA;

-- Dimension table: Products
CREATE TABLE gold.dim_product (
    product_key INT,
    product_id STRING,
    product_name STRING,
    product_category STRING,
    product_subcategory STRING,
    brand STRING,
    supplier STRING
) USING DELTA;

-- Dimension table: Date
CREATE TABLE gold.dim_date (
    date_key INT,
    date DATE,
    day_of_week STRING,
    day_of_month INT,
    day_of_year INT,
    week_of_year INT,
    month INT,
    month_name STRING,
    quarter INT,
    year INT,
    is_weekend BOOLEAN,
    is_holiday BOOLEAN
) USING DELTA;
```

Performance Optimization:

```sql
-- Partition large tables
ALTER TABLE gold.fact_sales ADD PARTITION (date_key=20250101);

-- Z-Order for multi-dimensional clustering
OPTIMIZE gold.fact_sales ZORDER BY (customer_key, product_key);

-- Create materialized views for common queries
CREATE MATERIALIZED VIEW gold.monthly_sales_summary AS
SELECT
    d.year,
    d.month,
    d.month_name,
    p.product_category,
    c.customer_segment,
    SUM(f.total_amount) as total_revenue,
    COUNT(DISTINCT f.transaction_key) as transaction_count,
    COUNT(DISTINCT f.customer_key) as unique_customers,
    AVG(f.total_amount) as avg_transaction_value
FROM gold.fact_sales f
JOIN gold.dim_date d ON f.date_key = d.date_key
JOIN gold.dim_product p ON f.product_key = p.product_key
JOIN gold.dim_customer c ON f.customer_key = c.customer_key
GROUP BY d.year, d.month, d.month_name, p.product_category, c.customer_segment;
```

Databricks Lakebase (OLTP)

Lakebase is a fully managed PostgreSQL-based OLTP database engine integrated into the Databricks platform, enabling transactional workloads alongside analytical processing.

Architecture

Lakebase provides: - **PostgreSQL Compatibility**: Standard PostgreSQL wire protocol and SQL dialect - **Decoupled Storage**: Compute and storage separation for scalability - **Unity Catalog Integration**: Governed access to OLTP data - **Sync Tables**: Automatic synchronization to Delta Lake for analytics - **High Availability**: Built-in replication and failover

Use Cases

1. **Feature Store**: Low-latency feature serving for ML models
2. **Application State**: Store application state for Databricks Apps
3. **Real-time Data Serving**: Serve insights from gold tables to applications
4. **Operational Workflows**: Manage workflow state and orchestration metadata

Implementation Example

```sql
-- Create Lakebase instance
CREATE DATABASE INSTANCE ecommerce_oltp
WITH (
    instance_type = 'db.r5.xlarge',
    storage_size = 500,
    backup_retention_days = 7
);

-- Create transactional tables
CREATE TABLE ecommerce_oltp.orders (
    order_id SERIAL PRIMARY KEY,
    customer_id INTEGER NOT NULL,
    order_date TIMESTAMP DEFAULT CURRENT_TIMESTAMP,
    status VARCHAR(20) NOT NULL,
    total_amount DECIMAL(10,2) NOT NULL,
    CONSTRAINT valid_status CHECK (status IN ('pending', 'confirmed',
'shipped', 'delivered', 'cancelled'))
);

CREATE TABLE ecommerce_oltp.order_items (
    item_id SERIAL PRIMARY KEY,
    order_id INTEGER REFERENCES orders(order_id),
    product_id INTEGER NOT NULL,
    quantity INTEGER NOT NULL,
    unit_price DECIMAL(10,2) NOT NULL,
    CONSTRAINT positive_quantity CHECK (quantity > 0)
);

CREATE INDEX idx_orders_customer ON ecommerce_oltp.orders(customer_id);
CREATE INDEX idx_orders_date ON ecommerce_oltp.orders(order_date);

-- Sync to Delta Lake for analytics
CREATE SYNCED TABLE production.sales.orders_analytics
AS SELECT
    order_id,
    customer_id,
    order_date,
    status,
    total_amount
FROM ecommerce_oltp.orders;

-- Synced table automatically updates as OLTP data changes
```

Integration with Databricks Apps

```
 # Flask app using Lakebase
from flask import Flask, request, jsonify
import psycopg2
import os

app = Flask(__name__)

# Connect to Lakebase
def get_db_connection():
    return psycopg2.connect(
        host=os.getenv("LAKEBASE_HOST"),
        database="ecommerce_oltp",
        user=os.getenv("LAKEBASE_USER"),
        password=os.getenv("LAKEBASE_PASSWORD")
    )

@app.route('/orders', methods=['POST'])
def create_order():
    data = request.get_json()
    conn = get_db_connection()
    cursor = conn.cursor()

    try:
        # Insert order (ACID transaction)
        cursor.execute("""
            INSERT INTO orders (customer_id, status, total_amount)
            VALUES (%s, %s, %s)
            RETURNING order_id
        """, (data['customer_id'], 'pending', data['total_amount']))

        order_id = cursor.fetchone()[0]

        # Insert order items
        for item in data['items']:
            cursor.execute("""
                INSERT INTO order_items (order_id, product_id, quantity,
unit_price)
                VALUES (%s, %s, %s, %s)
            """, (order_id, item['product_id'], item['quantity'],
item['unit_price']))

        conn.commit()
        return jsonify({"order_id": order_id, "status": "created"}), 201

    except Exception as e:
        conn.rollback()
        return jsonify({"error": str(e)}), 500
    finally:
        cursor.close()
        conn.close()

@app.route('/orders/<int:order_id>', methods=['GET'])
def get_order(order_id):
    conn = get_db_connection()
    cursor = conn.cursor()

    cursor.execute("""
        SELECT o.order_id, o.customer_id, o.order_date, o.status,
o.total_amount,
               json_agg(json_build_object(
                   'product_id', oi.product_id,
                   'quantity', oi.quantity,
                   'unit_price', oi.unit_price
```

```
                )) as items
        FROM orders o
        LEFT JOIN order_items oi ON o.order_id = oi.order_id
        WHERE o.order_id = %s
        GROUP BY o.order_id
    """, (order_id,))

    result = cursor.fetchone()
    cursor.close()
    conn.close()

    if result:
        return jsonify({
            "order_id": result[0],
            "customer_id": result[1],
            "order_date": result[2].isoformat(),
            "status": result[3],
            "total_amount": float(result[4]),
            "items": result[5]
        }), 200
    else:
        return jsonify({"error": "Order not found"}), 404
```

Implementation Guide

Step 1: Set Up Unity Catalog

```
 -- Create catalog hierarchy
CREATE CATALOG IF NOT EXISTS production;
CREATE SCHEMA IF NOT EXISTS production.bronze;
CREATE SCHEMA IF NOT EXISTS production.silver;
CREATE SCHEMA IF NOT EXISTS production.gold;

-- Set up permissions
GRANT USE CATALOG ON CATALOG production TO `data_team`;
GRANT CREATE SCHEMA ON CATALOG production TO `data_engineers`;
GRANT SELECT ON SCHEMA production.gold TO `data_analysts`;
```

Step 2: Implement Bronze Layer

```
# Auto Loader for continuous ingestion
from pyspark.sql.functions import *

# Configure Auto Loader
bronze_stream = spark.readStream \
    .format("cloudFiles") \
    .option("cloudFiles.format", "json") \
    .option("cloudFiles.schemaLocation", "/mnt/schemas/events") \
    .option("cloudFiles.inferColumnTypes", "true") \
    .option("cloudFiles.schemaEvolutionMode", "addNewColumns") \
    .load("s3://my-bucket/landing/events/")

# Add metadata
bronze_enriched = bronze_stream \
    .withColumn("_ingestion_timestamp", current_timestamp()) \
    .withColumn("_source_file", input_file_name()) \
    .withColumn("_bronze_date", current_date())

# Write to bronze
bronze_enriched.writeStream \
    .format("delta") \
    .option("checkpointLocation", "/mnt/checkpoints/bronze_events") \
    .option("mergeSchema", "true") \
    .partitionBy("_bronze_date") \
    .table("production.bronze.events")
```

Step 3: Implement Silver Layer with DLT

```
import dlt
from pyspark.sql.functions import *

@dlt.table(
    name="silver_events",
    comment="Validated and cleaned events",
    table_properties={"quality": "silver", "pipelines.autoOptimize.zOrderCols":
"user_id,event_date"}
)
@dlt.expect_or_drop("valid_event_id", "event_id IS NOT NULL")
@dlt.expect_or_drop("valid_timestamp", "event_timestamp IS NOT NULL AND
event_timestamp >= '2020-01-01'")
@dlt.expect_or_drop("valid_user", "user_id IS NOT NULL AND user_id > 0")
@dlt.expect("valid_amount", "amount >= 0")
def create_silver_events():
    return (
        dlt.read_stream("production.bronze.events")
        .select(
            col("event_id"),
            col("user_id").cast("long"),
            to_timestamp("event_timestamp").alias("event_timestamp"),
            to_date("event_timestamp").alias("event_date"),
            col("event_type"),
            col("amount").cast("decimal(10,2)"),
            col("_ingestion_timestamp")
        )
        .dropDuplicates(["event_id"])
        .withColumn("_silver_processed_at", current_timestamp())
    )
```

Step 4: Implement Gold Layer

```
 @dlt.table(
    name="gold_daily_user_metrics",
    comment="Daily aggregated user metrics for analytics",
    table_properties={"quality": "gold"}
)
def create_gold_daily_metrics():
    return (
        dlt.read("silver_events")
        .groupBy("user_id", "event_date")
        .agg(
            count("*").alias("total_events"),
            countDistinct("event_id").alias("unique_events"),
            sum(when(col("event_type") == "purchase",
col("amount")).otherwise(0)).alias("total_revenue"),
            count(when(col("event_type") == "purchase",
1)).alias("purchase_count"),
            count(when(col("event_type") == "view", 1)).alias("view_count"),
            count(when(col("event_type") == "click", 1)).alias("click_count")
        )
        .withColumn("conversion_rate",
            col("purchase_count") / (col("view_count") + col("click_count")))
    )
```

Step 5: Optimize and Monitor

```
 -- Optimize tables regularly
OPTIMIZE production.silver.events ZORDER BY (user_id, event_date);
OPTIMIZE production.gold.daily_user_metrics ZORDER BY (event_date, user_id);

-- Vacuum old files (retain 7 days)
VACUUM production.silver.events RETAIN 168 HOURS;

-- Monitor data quality
SELECT
    table_name,
    COUNT(*) as row_count,
    COUNT(DISTINCT user_id) as unique_users,
    MIN(event_date) as earliest_date,
    MAX(event_date) as latest_date
FROM production.silver.events
GROUP BY table_name;

-- Set up table monitoring
CREATE OR REPLACE TABLE production.monitoring.table_metrics AS
SELECT
    current_timestamp() as check_timestamp,
    'production.silver.events' as table_name,
    COUNT(*) as row_count,
    COUNT(DISTINCT user_id) as unique_users,
    SUM(CASE WHEN amount IS NULL THEN 1 ELSE 0 END) as null_amounts
FROM production.silver.events;
```

Databricks Asset Bundles

Overview and Concepts

Databricks Asset Bundles (DAB) provide an infrastructure-as-code approach to managing Databricks resources. Bundles enable developers to define, version, validate, and deploy Databricks workflows, apps, pipelines, and other resources programmatically.

Key Benefits

1. **Infrastructure as Code**: Define all Databricks resources in YAML or Python
2. **Version Control**: Track changes in Git alongside application code
3. **CI/CD Integration**: Automate testing and deployment pipelines
4. **Environment Management**: Separate dev, staging, and production configurations
5. **Reproducibility**: Ensure consistent deployments across environments
6. **Collaboration**: Enable team-based development with code reviews

Core Concepts

- **Bundle**: Collection of Databricks resources and their configurations
- **Resources**: Jobs, pipelines, apps, models, dashboards, etc.
- **Targets**: Environment-specific configurations (dev, staging, prod)
- **Variables**: Parameterized values for flexibility
- **Deployment Modes**: Controls resource naming and permissions

Bundle Structure

A typical bundle project structure:

```
my_project/
├── databricks.yml           # Main bundle configuration
├── resources/               # Resource definitions
│   ├── jobs/
│   │   ├── etl_job.yml
│   │   └── ml_training_job.yml
│   ├── pipelines/
│   │   └── data_pipeline.yml
│   └── apps/
│       └── analytics_app.yml
├── src/                     # Source code
│   ├── notebooks/
│   │   ├── bronze_ingestion.py
│   │   ├── silver_transformation.py
│   │   └── gold_aggregation.py
│   └── python/
│       ├── __init__.py
│       └── utils.py
├── tests/                   # Unit tests
│   └── test_transformations.py
├── fixtures/                # Test data
│   └── sample_data.json
└── README.md
```

Configuration Reference

databricks.yml Structure

```yaml
# Bundle definition
bundle:
  name: my_data_platform

  # Git integration
  git:
    origin_url: https://github.com/myorg/my-data-platform
    branch: main

# Variables for parameterization
variables:
  catalog_name:
    description: "Unity Catalog name"
    default: "development"

  warehouse_id:
    description: "SQL Warehouse ID"
    default: "abc123def456"

  notification_email:
    description: "Email for job notifications"

# Include additional configuration files
include:
  - resources/**/*.yml

# Workspace configuration
workspace:
  host: https://company.databricks.com
  root_path:
/Workspace/Users/$`{workspace.current_user.userName}/.bundle/`${bundle.name}/${bu

# Define resources
resources:
  jobs:
    etl_pipeline:
      name: "[${bundle.target}] ETL Pipeline"
      tasks:
        - task_key: bronze_ingestion
          notebook_task:
            notebook_path: ../src/notebooks/bronze_ingestion.py
            source: WORKSPACE
          new_cluster:
            spark_version: "13.3.x-scala2.12"
            node_type_id: "i3.xlarge"
            num_workers: 2

        - task_key: silver_transformation
          depends_on:
            - task_key: bronze_ingestion
          notebook_task:
            notebook_path: ../src/notebooks/silver_transformation.py
          new_cluster:
            spark_version: "13.3.x-scala2.12"
            node_type_id: "i3.xlarge"
            num_workers: 4

      schedule:
        quartz_cron_expression: "0 0 2 * * ?"
```

```yaml
        timezone_id: "America/Los_Angeles"

      email_notifications:
        on_failure:
          - ${var.notification_email}

  pipelines:
    dlt_pipeline:
      name: "[${bundle.target}] DLT Pipeline"
      catalog: ${var.catalog_name}
      target: ${bundle.target}_schema
      libraries:
        - notebook:
            path: ../src/notebooks/dlt_definitions.py
      configuration:
        warehouse_id: ${var.warehouse_id}
      continuous: false

  apps:
    analytics_dashboard:
      name: "[${bundle.target}] Analytics Dashboard"
      description: "Real-time analytics dashboard"
      resources:
        - name: warehouse
          sql_warehouse:
            id: ${var.warehouse_id}

# Deployment targets
targets:
  dev:
    mode: development
    default: true
    workspace:
      host: https://dev.databricks.com
    variables:
      catalog_name: "dev"
      notification_email: "dev-team@company.com"

  staging:
    mode: development
    workspace:
      host: https://staging.databricks.com
    variables:
      catalog_name: "staging"
      notification_email: "qa-team@company.com"

  prod:
    mode: production
    workspace:
      host: https://prod.databricks.com
    run_as:
      service_principal_name: "prod-service-principal"
    variables:
      catalog_name: "production"
      notification_email: "data-ops@company.com"

    # Production-specific overrides
    resources:
      jobs:
        etl_pipeline:
          schedule:
            quartz_cron_expression: "0 0 1 * * ?"  # Run at 1 AM in prod
          tasks:
            - task_key: bronze_ingestion
              new_cluster:
                num_workers: 8  # More workers in prod
```

```yaml
        - task_key: silver_transformation
          new_cluster:
            num_workers: 16
```

Development Lifecycle

1. Initialize Bundle

```bash
# Initialize new bundle from template
databricks bundle init

# Choose template
# - default-python: Python-based workflows
# - default-sql: SQL-based workflows
# - dbt-sql: dbt integration
# - mlops-stacks: ML workflows

# Or initialize from custom template
databricks bundle init --template-dir /path/to/template
```

2. Develop Locally

```bash
# Validate bundle configuration
databricks bundle validate

# Preview deployment changes
databricks bundle deploy --dry-run

# Deploy to development
databricks bundle deploy --target dev
```

3. Test and Iterate

```bash
# Run specific job
databricks bundle run etl_pipeline --target dev

# Run with parameters
databricks bundle run etl_pipeline \
  --target dev \
  --params '{"start_date": "2025-01-01", "end_date": "2025-01-31"}'

# View job status
databricks bundle run etl_pipeline --target dev --wait
```

4. Validate and Deploy

```
# Run validation
databricks bundle validate --target prod

# Deploy to production
databricks bundle deploy --target prod

# Verify deployment
databricks bundle summary --target prod
```

5. Clean Up

```
# Destroy resources (use with caution!)
databricks bundle destroy --target dev

# Confirm destruction
# Type 'yes' when prompted
```

CI/CD Integration

GitHub Actions Workflow

```yaml
# .github/workflows/databricks-deploy.yml
name: Databricks Bundle Deployment

on:
  push:
    branches: [main, develop]
  pull_request:
    branches: [main]

env:
  DATABRICKS_HOST: ${{ secrets.DATABRICKS_HOST }}
  DATABRICKS_TOKEN: ${{ secrets.DATABRICKS_TOKEN }}

jobs:
  validate:
    runs-on: ubuntu-latest
    steps:
      - uses: actions/checkout@v3

      - name: Install Databricks CLI
        run: |
          curl -fsSL https://raw.githubusercontent.com/databricks/setup-
cli/main/install.sh | sh

      - name: Validate Bundle
        run: databricks bundle validate

  test:
    runs-on: ubuntu-latest
    needs: validate
    steps:
      - uses: actions/checkout@v3

      - name: Set up Python
        uses: actions/setup-python@v4
        with:
          python-version: '3.10'

      - name: Install dependencies
        run: |
          pip install -r requirements.txt
          pip install pytest

      - name: Run tests
        run: pytest tests/

  deploy-dev:
    runs-on: ubuntu-latest
    needs: [validate, test]
    if: github.ref == 'refs/heads/develop'
    steps:
      - uses: actions/checkout@v3

      - name: Install Databricks CLI
        run: |
          curl -fsSL https://raw.githubusercontent.com/databricks/setup-
cli/main/install.sh | sh

      - name: Deploy to Dev
```

```yaml
        run: |
          databricks bundle deploy --target dev
          databricks bundle run etl_pipeline --target dev

  deploy-prod:
    runs-on: ubuntu-latest
    needs: [validate, test]
    if: github.ref == 'refs/heads/main'
    environment: production
    steps:
      - uses: actions/checkout@v3

      - name: Install Databricks CLI
        run: |
          curl -fsSL https://raw.githubusercontent.com/databricks/setup-
cli/main/install.sh | sh

      - name: Deploy to Production
        run: databricks bundle deploy --target prod
```

Azure DevOps Pipeline

```yaml
# azure-pipelines.yml
trigger:
  branches:
    include:
      - main
      - develop

pool:
  vmImage: 'ubuntu-latest'

variables:
  - group: databricks-credentials

stages:
  - stage: Validate
    jobs:
      - job: ValidateBundle
        steps:
          - task: UsePythonVersion@0
            inputs:
              versionSpec: '3.10'

          - script: |
              curl -fsSL https://raw.githubusercontent.com/databricks/setup-
cli/main/install.sh | sh
              databricks bundle validate
            displayName: 'Validate Databricks Bundle'
            env:
              DATABRICKS_HOST: $(DATABRICKS_HOST)
              DATABRICKS_TOKEN: $(DATABRICKS_TOKEN)

  - stage: Test
    dependsOn: Validate
    jobs:
      - job: RunTests
        steps:
          - task: UsePythonVersion@0
            inputs:
              versionSpec: '3.10'

          - script: |
              pip install -r requirements.txt
              pip install pytest
              pytest tests/
            displayName: 'Run Unit Tests'

  - stage: DeployDev
    dependsOn: Test
    condition: eq(variables['Build.SourceBranch'], 'refs/heads/develop')
    jobs:
      - deployment: DeployToDev
        environment: 'development'
        strategy:
          runOnce:
            deploy:
              steps:
                - checkout: self
                - script: |
                    curl -fsSL
https://raw.githubusercontent.com/databricks/setup-cli/main/install.sh | sh
                    databricks bundle deploy --target dev
                  displayName: 'Deploy to Development'
                  env:
```

```
                        DATABRICKS_HOST: $(DATABRICKS_DEV_HOST)
                        DATABRICKS_TOKEN: $(DATABRICKS_DEV_TOKEN)

  - stage: DeployProd
    dependsOn: Test
    condition: eq(variables['Build.SourceBranch'], 'refs/heads/main')
    jobs:
      - deployment: DeployToProduction
        environment: 'production'
        strategy:
          runOnce:
            deploy:
              steps:
                - checkout: self
                - script: |
                    curl -fsSL
https://raw.githubusercontent.com/databricks/setup-cli/main/install.sh | sh
                    databricks bundle deploy --target prod
                  displayName: 'Deploy to Production'
                  env:
                    DATABRICKS_HOST: $(DATABRICKS_PROD_HOST)
                    DATABRICKS_TOKEN: $(DATABRICKS_PROD_TOKEN)
```

Deployment Modes

Databricks Asset Bundles support three deployment modes:

1. Development Mode

```
targets:
  dev:
    mode: development
```

Characteristics: - Resources prefixed with `[dev username]` - Job schedules paused by default - Permissions: Only creator can access - Use case: Individual developer environments

2. Production Mode

```
targets:
  prod:
    mode: production
    run_as:
      service_principal_name: "prod-sp"
```

Characteristics: - No resource name prefixes - Job schedules active - Runs as service principal - Immutable deployments (prevents accidental changes) - Use case: Production workloads

3. Snapshot Mode (Default)

```yaml
targets:
  staging:
    mode: snapshot
```

Characteristics: - Resources prefixed with target name - Job schedules active - Runs as deploying user - Use case: Staging/QA environments

Best Practices

1. Project Organization

```
my_project/
├── databricks.yml              # Main config
├── resources/
│   ├── common.yml              # Shared configurations
│   ├── jobs/
│   │   ├── ingestion.yml
│   │   └── transformation.yml
│   └── pipelines/
│       └── dlt_pipeline.yml
├── src/
│   ├── common/                 # Shared utilities
│   ├── ingestion/              # Ingestion logic
│   └── transformation/         # Transformation logic
├── tests/
│   ├── unit/
│   └── integration/
└── docs/
    └── architecture.md
```

2. Use Variables for Flexibility

```yaml
variables:
  environment:
    description: "Deployment environment"

  cluster_config:
    description: "Cluster configuration"
    default:
      spark_version: "13.3.x-scala2.12"
      node_type_id: "i3.xlarge"

resources:
  jobs:
    my_job:
      name: "[${var.environment}] My Job"
      new_cluster: ${var.cluster_config}
```

3. Separate Concerns

```yaml
# resources/jobs/etl.yml
resources:
  jobs:
    etl_job:
      name: "ETL Job"
      tasks: !include tasks/etl_tasks.yml

# resources/tasks/etl_tasks.yml
- task_key: ingest
  notebook_task:
    notebook_path: ../src/ingest.py
- task_key: transform
  depends_on:
    - task_key: ingest
  notebook_task:
    notebook_path: ../src/transform.py
```

4. Version Control Best Practices

- Store bundles in Git
- Use feature branches for development
- Require code reviews for production changes
- Tag releases for production deployments
- Use `.gitignore` for generated files

```
# .gitignore
.databricks/
__pycache__/
*.pyc
.pytest_cache/
.venv/
```

5. Testing Strategy

```python
# tests/test_transformations.py
import pytest
from pyspark.sql import SparkSession
from src.transformations import clean_data

@pytest.fixture
def spark():
    return SparkSession.builder.master("local[1]").getOrCreate()

def test_clean_data(spark):
    # Arrange
    input_data = [
        (1, "John", None),
        (2, "Jane", "invalid"),
        (3, "Bob", "valid")
    ]
    df = spark.createDataFrame(input_data, ["id", "name", "status"])

    # Act
    result = clean_data(df)

    # Assert
    assert result.count() == 2  # Invalid records dropped
    assert result.filter("status = 'valid'").count() == 1
```

Code Examples

Example 1: Complete ETL Bundle

databricks.yml:

```yaml
bundle:
  name: customer_analytics_etl

variables:
  catalog:
    default: "development"
  schema:
    default: "customer_data"

include:
  - resources/*.yml

targets:
  dev:
    mode: development
    default: true
    variables:
      catalog: "dev"

  prod:
    mode: production
    run_as:
      service_principal_name: "etl-service-principal"
    variables:
      catalog: "prod"
```

resources/jobs.yml:

```yaml
resources:
  jobs:
    customer_etl:
      name: "[${bundle.target}] Customer Analytics ETL"

      job_clusters:
        - job_cluster_key: "etl_cluster"
          new_cluster:
            spark_version: "13.3.x-scala2.12"
            node_type_id: "i3.xlarge"
            num_workers: 4
            spark_conf:
              "spark.databricks.delta.optimizeWrite.enabled": "true"
              "spark.databricks.delta.autoCompact.enabled": "true"

      tasks:
        - task_key: "ingest_raw_data"
          job_cluster_key: "etl_cluster"
          notebook_task:
            notebook_path: "../src/notebooks/01_ingest_raw.py"
            base_parameters:
              catalog: "${var.catalog}"
              schema: "${var.schema}"

        - task_key: "transform_to_silver"
          depends_on:
            - task_key: "ingest_raw_data"
          job_cluster_key: "etl_cluster"
          notebook_task:
            notebook_path: "../src/notebooks/02_transform_silver.py"
            base_parameters:
              catalog: "${var.catalog}"
              schema: "${var.schema}"

        - task_key: "aggregate_to_gold"
          depends_on:
            - task_key: "transform_to_silver"
          job_cluster_key: "etl_cluster"
          notebook_task:
            notebook_path: "../src/notebooks/03_aggregate_gold.py"
            base_parameters:
              catalog: "${var.catalog}"
              schema: "${var.schema}"

        - task_key: "data_quality_checks"
          depends_on:
            - task_key: "aggregate_to_gold"
          job_cluster_key: "etl_cluster"
          python_wheel_task:
            package_name: "data_quality"
            entry_point: "run_checks"
            parameters:
              - "--catalog=${var.catalog}"
              - "--schema=${var.schema}"
          libraries:
            - whl: "../dist/data_quality-0.1.0-py3-none-any.whl"

      schedule:
        quartz_cron_expression: "0 0 2 * * ?"
        timezone_id: "UTC"
        pause_status: "UNPAUSED"

      email_notifications:
        on_failure:
          - "data-eng@company.com"
```

```yaml
        on_success:
          - "data-ops@company.com"

      max_concurrent_runs: 1
      timeout_seconds: 7200
```

Example 2: ML Training Pipeline Bundle

```yaml
bundle:
  name: ml_training_pipeline

variables:
  model_name:
    default: "customer_churn_model"
  experiment_path:
    default: "/Shared/ml_experiments/customer_churn"

resources:
  jobs:
    train_model:
      name: "[$`{bundle.target}] ML Training - `${var.model_name}"

      tasks:
        - task_key: "prepare_features"
          new_cluster:
            spark_version: "13.3.x-cpu-ml-scala2.12"
            node_type_id: "i3.xlarge"
            num_workers: 2
          notebook_task:
            notebook_path: "../src/ml/01_feature_engineering.py"
            base_parameters:
              model_name: "${var.model_name}"

        - task_key: "train_model"
          depends_on:
            - task_key: "prepare_features"
          new_cluster:
            spark_version: "13.3.x-cpu-ml-scala2.12"
            node_type_id: "i3.2xlarge"
            num_workers: 4
          notebook_task:
            notebook_path: "../src/ml/02_train_model.py"
            base_parameters:
              model_name: "${var.model_name}"
              experiment_path: "${var.experiment_path}"
          libraries:
            - pypi:
                package: "scikit-learn==1.3.0"
            - pypi:
                package: "xgboost==2.0.0"

        - task_key: "evaluate_model"
          depends_on:
            - task_key: "train_model"
          new_cluster:
            spark_version: "13.3.x-cpu-ml-scala2.12"
            node_type_id: "i3.xlarge"
            num_workers: 2
          notebook_task:
            notebook_path: "../src/ml/03_evaluate_model.py"
            base_parameters:
              model_name: "${var.model_name}"
              experiment_path: "${var.experiment_path}"

        - task_key: "register_model"
          depends_on:
            - task_key: "evaluate_model"
          new_cluster:
            spark_version: "13.3.x-cpu-ml-scala2.12"
            node_type_id: "i3.xlarge"
            num_workers: 1
```

```yaml
          notebook_task:
            notebook_path: "../src/ml/04_register_model.py"
            base_parameters:
              model_name: "${var.model_name}"
              experiment_path: "${var.experiment_path}"
              registry_stage: "${bundle.target == 'prod' ? 'Production' :
'Staging'}"

  experiments:
    churn_experiment:
      name: "${var.experiment_path}"
      description: "Customer churn prediction experiments"

  models:
    churn_model:
      name: "${var.model_name}"
      description: "Customer churn prediction model"

targets:
  dev:
    mode: development
    variables:
      model_name: "dev_customer_churn"
      experiment_path:
"/Users/${workspace.current_user.userName}/experiments/churn"

  prod:
    mode: production
    run_as:
      service_principal_name: "ml-service-principal"
    variables:
      model_name: "customer_churn_model"
      experiment_path: "/Shared/ml_experiments/customer_churn"
```

Example 3: Multi-App Bundle

```yaml
bundle:
  name: analytics_platform

resources:
  apps:
    sales_dashboard:
      name: "[${bundle.target}] Sales Dashboard"
      description: "Real-time sales analytics"
      resources:
        - name: sales_warehouse
          sql_warehouse:
            id: "${var.warehouse_id}"

    customer_insights:
      name: "[${bundle.target}] Customer Insights"
      description: "Customer behavior analytics"
      resources:
        - name: analytics_warehouse
          sql_warehouse:
            id: "${var.warehouse_id}"

    ml_predictions:
      name: "[${bundle.target}] ML Predictions"
      description: "Real-time ML predictions interface"
      resources:
        - name: model_endpoint
          model_serving_endpoint:
            name: "customer-churn-endpoint"

  jobs:
    refresh_dashboards:
      name: "[${bundle.target}] Refresh Dashboard Data"
      tasks:
        - task_key: "refresh_sales"
          sql_task:
            warehouse_id: "${var.warehouse_id}"
            query:
              query_id: "${var.sales_query_id}"

        - task_key: "refresh_customers"
          sql_task:
            warehouse_id: "${var.warehouse_id}"
            query:
              query_id: "${var.customer_query_id}"

      schedule:
        quartz_cron_expression: "0 */15 * * * ?"  # Every 15 minutes
        timezone_id: "UTC"

variables:
  warehouse_id:
    description: "SQL Warehouse ID for apps"
  sales_query_id:
    description: "Sales refresh query ID"
  customer_query_id:
    description: "Customer refresh query ID"

targets:
  dev:
    mode: development
    variables:
      warehouse_id: "dev_warehouse_123"
      sales_query_id: "dev_sales_query"
```

```yaml
      customer_query_id: "dev_customer_query"

  prod:
    mode: production
    run_as:
      service_principal_name: "apps-service-principal"
    variables:
      warehouse_id: "prod_warehouse_456"
      sales_query_id: "prod_sales_query"
      customer_query_id: "prod_customer_query"
```

Integration Patterns

Pattern 1: Apps + Lakehouse

Databricks Apps can leverage the full power of the Lakehouse architecture:

```python
# Streamlit app accessing Lakehouse data
import streamlit as st
from databricks import sql
import os

# Connect to SQL Warehouse
connection = sql.connect(
    server_hostname=os.getenv("DATABRICKS_HOST"),
    http_path=f"/sql/1.0/warehouses/{os.getenv('WAREHOUSE_ID')}"
)

# Query gold layer
@st.cache_data(ttl=600)
def load_customer_360(customer_id):
    cursor = connection.cursor()
    cursor.execute(f"""
        SELECT *
        FROM production.gold.customer_360
        WHERE customer_id = '{customer_id}'
    """)
    return cursor.fetchall_arrow().to_pandas()

# UI
customer_id = st.text_input("Customer ID")
if customer_id:
    df = load_customer_360(customer_id)
    st.dataframe(df)
```

Pattern 2: Bundles + Apps

Manage Databricks Apps as code using Asset Bundles:

```yaml
# databricks.yml
resources:
  apps:
    analytics_app:
      name: "[${bundle.target}] Analytics App"
      description: "Customer analytics dashboard"
      resources:
        - name: warehouse
          sql_warehouse:
            id: "${var.warehouse_id}"
        - name: lakebase
          database_instance:
            name: "analytics_db"

# Deploy app with bundle
# databricks bundle deploy --target prod
```

Pattern 3: Bundles + Lakehouse

Orchestrate complete Lakehouse workflows with bundles:

```yaml
resources:
  pipelines:
    medallion_pipeline:
      name: "Medallion Architecture Pipeline"
      catalog: "${var.catalog}"
      target: "${bundle.target}_schema"
      libraries:
        - notebook:
            path: "../src/bronze_layer.py"
        - notebook:
            path: "../src/silver_layer.py"
        - notebook:
            path: "../src/gold_layer.py"

      configuration:
        bronze_path: "/mnt/bronze"
        silver_path: "/mnt/silver"
        gold_path: "/mnt/gold"
```

Pattern 4: End-to-End Platform

Complete integration of all three technologies:

```yaml
bundle:
  name: customer_intelligence_platform

resources:
  # Data ingestion and transformation
  pipelines:
    customer_data_pipeline:
      name: "Customer Data Pipeline"
      catalog: "production"
      target: "customer_data"
      libraries:
        - notebook:
            path: "../src/pipelines/bronze_customers.py"
        - notebook:
            path: "../src/pipelines/silver_customers.py"
        - notebook:
            path: "../src/pipelines/gold_customer_360.py"

  # ML training
  jobs:
    churn_model_training:
      name: "Churn Model Training"
      tasks:
        - task_key: "train"
          notebook_task:
            notebook_path: "../src/ml/train_churn_model.py"

  # Model serving
  model_serving_endpoints:
    churn_prediction:
      name: "churn-prediction-endpoint"
      config:
        served_entities:
          - entity_name: "production.ml_models.customer_churn"
            entity_version: "1"
            workload_size: "Small"

  # OLTP database for app state
  database_instances:
    app_state_db:
      name: "customer_app_state"
      instance_type: "db.t3.medium"

  # Customer-facing application
  apps:
    customer_portal:
      name: "Customer Intelligence Portal"
      description: "360-degree customer view with churn predictions"
      resources:
        - name: warehouse
          sql_warehouse:
            id: "${var.warehouse_id}"
        - name: model
          model_serving_endpoint:
            name: "churn-prediction-endpoint"
        - name: database
          database_instance:
            name: "customer_app_state"
```

References

Official Databricks Documentation

1. **Databricks Apps**
2. Databricks Apps Overview
3. Get Started with Databricks Apps
4. Develop Databricks Apps
5. Configure App Runtime
6. **Databricks Lakehouse**
7. Lakehouse Architecture
8. Lakehouse Reference Architecture
9. Medallion Architecture
10. What is Lakebase?
11. **Databricks Asset Bundles**
12. What are Databricks Asset Bundles?
13. Bundle Configuration
14. Develop Bundles
15. Bundle Tutorials
16. CI/CD Best Practices

GitHub Repositories

1. **Databricks Apps Examples**
2. databricks-solutions/databricks-apps-examples
3. **Databricks Asset Bundle Examples**
4. databricks/bundle-examples

Community Resources

1. **Medium Articles**
2. Building a Databricks App Project: Architecture, Concepts, and Implementation Guide
3. CI/CD Strategies For Databricks Asset Bundles
4. Understanding Databricks Lakehouse Reference Architectures
5. **Databricks Community**
6. Exploring Code With Databricks Apps

Additional Resources

- Databricks Glossary: Medallion Architecture
- Databricks Product Page: Databricks Apps
- Databricks Blog: Announcing General Availability of Databricks Asset Bundles

---

Document Version: 1.0
Last Updated: October 20, 2025
Compiled By: Comprehensive Research Analysis

This document represents a complete synthesis of official Databricks documentation, community resources, and open-source examples to provide actionable guidance for implementing Databricks Apps, Lakehouse architecture, and Asset Bundles in production environments.