🎬 Wistia Video Analytics - Complete Azure Implementation Guide

📋 Table of Contents

Phase 1: Azure Resources Setup

Step 1.1: Create Resource Group

1. Navigate to **Azure Portal** (portal.azure.com)
2. Click **Resource Groups** → **+ Create**
3. Enter details:
    - **Subscription**: Your subscription
    - **Resource Group**: `rg-wistia-analytics`
    - **Region**: `East US` (or your preferred region)
4. Click **Review + Create** → **Create**

Step 1.2: Create Storage Account

1. Search for **Storage Accounts** → **+ Create**
2. Configure:
    - **Resource Group**: `rg-wistia-analytics`
    - **Storage Account Name**: `stwistiaanalytics` (must be unique)
    - **Region**: Same as resource group
    - **Performance**: Standard
    - **Redundancy**: LRS (for dev) or GRS (for prod)

3. Click **Review + Create** → **Create**

Step 1.3: Create Key Vault (for secrets)

1. Search for **Key Vaults** → **+ Create**
2. Configure:
    - **Resource Group**: `rg-wistia-analytics`
    - **Key Vault Name**: `kv-wistia-analytics`
    - **Region**: Same as resource group
    - **Pricing Tier**: Standard
3. Click **Review + Create** → **Create**
4. After creation, go to **Secrets** → **+ Generate/Import**:
    - **Name**: `wistia-api-token`
    - **Value**: `" "`
    - Click **Create**

Phase 2: Storage Configuration

Step 2.1: Create Blob Containers

1. Go to your Storage Account → **Containers**
2. Create the following containers:
    - **+ Container** → Name: `bronze-layer` → Private access
    - **+ Container** → Name: `silver-layer` → Private access
    - **+ Container** → Name: `gold-layer` → Private access
    - **+ Container** → Name: `logs` → Private access

Step 2.2: Create Folder Structure

Within each container, create folders using Azure Storage Explorer or by uploading placeholder files:

bronze-layer/

- `raw-media-stats/`
- `raw-visitor-stats/`
- `checkpoints/`

silver-layer/

- `dim-media/`

- `dim-visitor/`
- `fact-engagement/`

gold-layer/

- `reports/`

Phase 3: Azure Databricks Setup

Step 3.1: Create Databricks Workspace

1. Search for **Azure Databricks** → **+ Create**
2. Configure:
    - **Resource Group**: `rg-wistia-analytics`
    - **Workspace Name**: `dbw-wistia-analytics`
    - **Region**: Same as resource group
    - **Pricing Tier**: Premium (required for secrets)
3. Click **Review + Create** → **Create**

Step 3.2: Launch Workspace

1. Go to your Databricks resource → **Launch Workspace**
2. Click **Launch Workspace** button

Step 3.3: Create Databricks Cluster

1. In Databricks UI, click **Compute** (left sidebar)
2. Click **Create Cluster**
3. Configure:
    - **Cluster Name**: `wistia-processing-cluster`
    - **Cluster Mode**: Standard
    - **Databricks Runtime**: 13.3 LTS (includes Apache Spark 3.4.1)
    - **Node Type**: Standard_DS3_v2 (or similar)
    - **Min Workers**: 1
    - **Max Workers**: 2
    - **Auto Termination**: 30 minutes
4. Click **Create Cluster**

Step 3.4: Configure Databricks Secrets

1. In Databricks, click **Settings** (gear icon) → **User Settings**

2. Click **Generate New Token**:
    - **Comment**: `ADF Integration Token`
    - **Lifetime**: 90 days
    - Copy and save the token securely
3. Install Databricks CLI locally or use Azure Cloud Shell:

```
databricks configure --token
# Enter host: https://adb-<workspace-id>.<region>.azuredatabricks.net
# Enter token: <your-token>

# Create secret scope
databricks secrets create-scope --scope wistia-secrets --scope-backend-type
AZURE_KEYVAULT --resource-id
/subscriptions/<sub-id>/resourceGroups/rg-wistia-analytics/providers/Microsoft.KeyVault/vaults/k
v-wistia-analytics --dns-name https://kv-wistia-analytics.vault.azure.net/
```

Step 3.5: Mount Storage to Databricks

Create a new notebook in Databricks and run:

```
# Mount Storage Account
storage_account = "stwistiaanalytics"
container_name = "bronze-layer"
mount_point = "/mnt/bronze"

# Get storage key from Azure Portal
storage_key = "YOUR_STORAGE_KEY"

try:
    dbutils.fs.mount(
        source=f"wasbs://{container_name}@{storage_account}.blob.core.windows.net",
        mount_point=mount_point,
        extra_configs={
            f"fs.azure.account.key.{storage_account}.blob.core.windows.net": storage_key
        }
    )
    print(f"✅ Mounted {container_name}")
except Exception as e:
    print(f"Already mounted or error: {e}")

# Repeat for silver-layer and gold-layer
```

Phase 4: API Ingestion Notebooks

Step 4.1: Create Notebook - wistia-01 (Media Stats Ingestion)

1. In Databricks, click **Workspace** → **Users** → **Your Email**
2. Right-click → **Create** → **Notebook**
3. Name: `wistia-01`, Language: Python
4. Attach to your cluster

Notebook Code:

```
# ============================================
# Notebook: 01_Setup_Storage_Access
# Purpose: Configure storage access for shared clusters
# ============================================

# CONFIGURATION - UPDATE THESE VALUES!
storage_account_name = ""  # ← YOUR STORAGE ACCOUNT NAME
storage_account_key = ""  # ← YOUR STORAGE KEY

# Set Spark configuration for direct access
spark.conf.set(
    f"fs.azure.account.key.{storage_account_name}.blob.core.windows.net",
    storage_account_key
)

print("=" * 70)
print("🔧 STORAGE ACCESS CONFIGURATION")
print("=" * 70)
print(f"✓ Configured access for: {storage_account_name}")

# Storage paths
RAW_PATH = f"wasbs://raw-data@{storage_account_name}.blob.core.windows.net"
PROCESSED_PATH =
f"wasbs://processed-data@{storage_account_name}.blob.core.windows.net"

# Test write access using DataFrame (not RDD)
try:
    from pyspark.sql import Row
```

```
    from datetime import datetime

    # Create test data as DataFrame
    test_data = spark.createDataFrame([
        Row(message=f"Test successful at {datetime.now()}", status="OK")
    ])

    # Write test file
    test_path = f"{RAW_PATH}/test_access.json"
    test_data.write.mode("overwrite").json(test_path)
    print(f"✓ Write test successful")

    # Read it back
    read_test = spark.read.json(test_path)
    print(f"✓ Read test successful: {read_test.first().message}")

    # List containers
    containers =
dbutils.fs.ls(f"wasbs://@{storage_account_name}.blob.core.windows.net/")
    print(f"✓ Found {len(containers)} container(s)")
    for container in containers:
        print(f"   - {container.name}")

    print("\n" + "=" * 70)
    print("✅ STORAGE ACCESS CONFIGURED SUCCESSFULLY!")
    print("=" * 70)

except Exception as e:
    print(f"✗ Error: {str(e)}")
    print("\nPlease verify:")
    print("  1. Storage account name is correct")
    print("  2. Storage account key is correct")
    print("  3. Containers exist in the storage account")
```

Step 4.2: Create Notebook - wistia-02 (Visitor Stats Ingestion)

1. Create new notebook: `wistia-02`

Notebook Code:

```
# ============================================
# Notebook: 02_Wistia_API_Ingestion
# Purpose: Fetch Wistia API data - Shared cluster compatible
# ============================================

import requests
import json
from datetime import datetime, timedelta
from pyspark.sql import Row
from pyspark.sql.functions import *
import time

# STORAGE CONFIGURATION - UPDATE THESE!
storage_account_name = ""  # ← YOUR STORAGE ACCOUNT NAME
storage_account_key = ""  # ← YOUR STORAGE KEY

# Set Spark configuration
spark.conf.set(
    f"fs.azure.account.key.{storage_account_name}.blob.core.windows.net",
    storage_account_key
)

# API CONFIGURATION
API_TOKEN = ""
MEDIA_IDS = ["gskhw4w4lm", "v08dlrgr7v"]
BASE_URL = "https://api.wistia.com/v1/stats/medias"

# Storage paths
RAW_PATH = f"wasbs://raw-data@{storage_account_name}.blob.core.windows.net"

# ============================================
# HELPER FUNCTIONS
# ============================================

def fetch_with_retry(url, headers, params=None, max_retries=3):
```

```
    """Fetch data with retry logic"""
    for attempt in range(max_retries):
        try:
            response = requests.get(url, headers=headers, params=params,
timeout=30)

            if response.status_code == 200:
                return response
            elif response.status_code == 429:
                wait_time = (2 ** attempt) * 5
                print(f"  ⚠️  Rate limited. Waiting {wait_time}s...")
                time.sleep(wait_time)
            elif response.status_code == 404:
                print(f"  ⚠️  Resource not found (404)")
                return None
            else:
                print(f" ✗ HTTP {response.status_code}")
                return None

        except Exception as e:
            print(f" ✗ Attempt {attempt + 1}: {str(e)[:100]}")
            if attempt < max_retries - 1:
                time.sleep(5)

    return None

def fetch_media_stats(media_id):
    """Fetch media statistics from Wistia API"""
    url = f"{BASE_URL}/{media_id}.json"
    headers = {"Authorization": f"Bearer {API_TOKEN}"}

    print(f"  → Fetching media stats...")
    response = fetch_with_retry(url, headers)

    if response and response.status_code == 200:
        data = response.json()
        print(f" ✓ Retrieved: {data.get('name', 'Unknown')}")
        return data
```

```
    else:
        print(f" ✗ Failed to fetch media stats")
        return None

def fetch_visitor_data(media_id, since_date=None):
    """Fetch visitor data with pagination"""
    all_visitors = []
    page = 1
    headers = {"Authorization": f"Bearer {API_TOKEN}"}

    print(f"  → Fetching visitor data...")

    while True:
        url = f"{BASE_URL}/{media_id}/visitors.json"
        params = {"page": page, "per_page": 100}

        if since_date:
            params["since"] = since_date.strftime("%Y-%m-%d")

        response = fetch_with_retry(url, headers, params)

        if not response or response.status_code != 200:
            break

        visitors = response.json()

        if not visitors:
            if page == 1:
                print(f"  ⚠️  No visitors found")
            else:
                print(f" ✓ Total visitors: {len(all_visitors)}")
            break

        all_visitors.extend(visitors)
        print(f" ✓ Page {page}: +{len(visitors)} (total: {len(all_visitors)})")
        page += 1
        time.sleep(1)
```

```
        if page > 100:
            print(f"  ⚠️  Pagination limit reached")
            break

    return all_visitors

def save_json_as_dataframe(data, path):
    """Save JSON data as DataFrame (shared cluster compatible)"""
    if not data:
        print(f"  ⚠️  No data to save")
        return False

    try:
        # Convert Python dict/list to DataFrame
        if isinstance(data, list):
            # For list of dicts (like visitors)
            df = spark.createDataFrame([Row(**item) for item in data])
        else:
            # For single dict (like media)
            df = spark.createDataFrame([Row(**data)])

        # Write as JSON
        df.write.mode("overwrite").json(path)

        record_count = df.count()
        print(f" ✓ Saved: {record_count} record(s)")
        return True

    except Exception as e:
        print(f" ✗ Save failed: {str(e)[:200]}")
        return False

def get_last_run_date():
    """Get last successful run date"""
    try:
        metadata_path = f"{RAW_PATH}/metadata/last_run.json"
        df = spark.read.json(metadata_path)
        if df.count() > 0:
```

```
            last_date_str = df.first().timestamp
            last_date = datetime.fromisoformat(last_date_str)
            print(f"ℹ️  Last run: {last_date}")
            return last_date
    except:
        print("ℹ️  No previous run found (using 7-day lookback)")

    return datetime.utcnow() - timedelta(days=7)

def update_last_run_date():
    """Update last run timestamp"""
    try:
        current_time = datetime.utcnow().isoformat()
        df = spark.createDataFrame([Row(timestamp=current_time,
status="success")])
        metadata_path = f"{RAW_PATH}/metadata/last_run.json"
        df.write.mode("overwrite").json(metadata_path)
        print(f"✓ Updated last run timestamp")
    except Exception as e:
        print(f"⚠️  Could not update timestamp: {str(e)[:100]}")

# ============================================
# MAIN EXECUTION
# ============================================

def main():
    print("=" * 70)
    print("🚀 WISTIA API DATA INGESTION (Shared Cluster)")
    print("=" * 70)

    # Verify storage access
    try:
        dbutils.fs.ls(RAW_PATH)
        print("✓ Storage access verified")
    except Exception as e:
        print(f"❌ Storage access failed: {str(e)[:200]}")
        print("   Run notebook 01_Setup_Storage_Access first")
        return
```

```
last_run_date = get_last_run_date()
run_timestamp = datetime.utcnow().strftime('%Y%m%d_%H%M%S')

total_media_success = 0
total_visitors_fetched = 0

for idx, media_id in enumerate(MEDIA_IDS, 1):
    print(f"\n{'=' * 70}")
    print(f"🎥 MEDIA {idx}/{len(MEDIA_IDS)}: {media_id}")
    print(f"{'=' * 70}")

    # Fetch and save media stats
    media_data = fetch_media_stats(media_id)
    if media_data:
        media_path = f"{RAW_PATH}/media/{media_id}_{run_timestamp}"
        if save_json_as_dataframe(media_data, media_path):
            total_media_success += 1

    # Fetch and save visitor data
    visitor_data = fetch_visitor_data(media_id, last_run_date)
    if visitor_data:
        visitor_path = f"{RAW_PATH}/visitors/{media_id}_{run_timestamp}"
        if save_json_as_dataframe(visitor_data, visitor_path):
            total_visitors_fetched += len(visitor_data)

    if idx < len(MEDIA_IDS):
        print(f" ⏸️ Pausing 3 seconds...")
        time.sleep(3)

update_last_run_date()

print(f"\n{'=' * 70}")
print("✅ INGESTION COMPLETE")
print(f"{'=' * 70}")
print(f"  Media processed: {total_media_success}/{len(MEDIA_IDS)}")
print(f"  Total visitors: {total_visitors_fetched}")
```

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

# Execute
main()
```

Phase 5: Data Processing & Transformation

Step 5.1: Create Notebook - wistia-03 (Transform to Star Schema)

Notebook Code:

```
# ============================================
# Notebook: 03_Wistia_Data_Processing (UC-safe)
# Purpose: Transform raw data - Shared cluster compatible
# ============================================

from pyspark.sql import functions as F
from pyspark.sql.window import Window
from pyspark.sql.types import *
from datetime import datetime

# STORAGE CONFIGURATION - UPDATE THESE!
storage_account_name = ""  # ← YOUR STORAGE ACCOUNT NAME
storage_account_key = ""  # ← YOUR STORAGE KEY

# Set Spark configuration
spark.conf.set(
    f"fs.azure.account.key.{storage_account_name}.blob.core.windows.net",
    storage_account_key
)

# Storage paths
RAW_PATH = f"wasbs://raw-data@{storage_account_name}.blob.core.windows.net"
PROCESSED_PATH =
f"wasbs://processed-data@{storage_account_name}.blob.core.windows.net"
```

```
print("=" * 70)
print("🔄 WISTIA DATA PROCESSING (Shared Cluster, Unity Catalog-safe)")
print("=" * 70)

# ============================================
# READ RAW DATA
# ============================================

print("\n📂 Reading raw data...")

try:
    # List available files (best-effort)
    try:
        media_folders = dbutils.fs.ls(f"{RAW_PATH}/media/")
        visitor_folders = dbutils.fs.ls(f"{RAW_PATH}/visitors/")
        print(f" ✓ Found {len(media_folders)} media folder(s)")
        print(f" ✓ Found {len(visitor_folders)} visitor folder(s)")
    except:
        print("  ⚠️  Could not list folders (they may not exist yet)")

    # Read media data - wildcard handles nested structure
    media_df = (
        spark.read
        .option("multiline", "true")
        .json(f"{RAW_PATH}/media/*/*.json")
    )
    media_count = media_df.count()
    print(f" ✓ Loaded {media_count} media record(s)")

    # Read visitor data
    visitor_df = (
        spark.read
        .option("multiline", "true")
        .json(f"{RAW_PATH}/visitors/*/*.json")
    )
    visitor_count = visitor_df.count()
    print(f" ✓ Loaded {visitor_count} visitor record(s)")
```

```
except Exception as e:
    print(f" ❌ Error reading data: {str(e)[:300]}")
    print("\n  Troubleshooting:")
    print("  1. Run 02_Wistia_API_Ingestion or 00_Generate_Dummy_Data first")
    print("  2. Verify data exists in storage account")
    raise

# Display sample data
print("\n📊 Sample Media Data:")
if media_count > 0:
    display(media_df.limit(3))
else:
    print("  ⚠️  No media data found")

print("\n📊 Sample Visitor Data:")
if visitor_count > 0:
    display(visitor_df.limit(3))
else:
    print("  ⚠️  No visitor data found")

# ============================================
# TRANSFORM: dim_media
# ============================================

print("\n🔧 Processing dim_media...")

if media_count > 0:
    dim_media = media_df.select(
        F.col("hashed_id").alias("media_id"),  # generator uses 'hashed_id'
        F.coalesce(F.col("name"), F.lit("Unknown")).alias("title"),
        F.concat(F.lit("https://wistia.com/series/health/videos/"),
F.col("hashed_id")).alias("url"),
        F.when(F.lower(F.coalesce(F.col("name"),
F.lit(""))).contains("facebook"), "Facebook")
         .when(F.lower(F.coalesce(F.col("name"),
F.lit(""))).contains("youtube"), "YouTube")
```

```
          .when(F.lower(F.coalesce(F.col("name"),
F.lit(""))).contains("instagram"), "Instagram")
          .otherwise("Wistia").alias("channel"),
        F.when(F.col("created").cast("bigint").isNotNull(),
               F.from_unixtime(F.col("created")).cast("timestamp"))
          .otherwise(F.current_timestamp()).alias("created_at"),
        F.current_timestamp().alias("processed_at")
    ).distinct()

    # ENHANCEMENT: Ensure media_id is never NULL
    print("  🔒 Ensuring media_id integrity...")
    dim_media = dim_media.withColumn(
        "media_id",
        F.when(F.col("media_id").isNull() | (F.col("media_id") == ""),
               F.concat(F.lit("media_"), F.monotonically_increasing_id()))
        .otherwise(F.col("media_id"))
    )

    dim_media_count = dim_media.count()
    print(f" ✓ Processed {dim_media_count} media record(s)")
else:
    print("  ⚠️  Skipping - no media data")
    dim_media = None
    dim_media_count = 0

# ============================================
# TRANSFORM: dim_visitor
# ============================================

print("\n🔧 Processing dim_visitor...")

if visitor_count > 0:
    dim_visitor = visitor_df.select(
        F.col("visitor_key").alias("visitor_id"),
        F.coalesce(F.col("ip_address"), F.lit("Unknown")).alias("ip_address"),
        F.coalesce(F.col("country"), F.lit("Unknown")).alias("country"),
        F.current_timestamp().alias("processed_at")
    ).distinct()
```

```
    # ENHANCEMENT: Ensure visitor_id is never NULL
    print("  🔒 Ensuring visitor_id integrity...")
    dim_visitor = dim_visitor.withColumn(
        "visitor_id",
        F.when(F.col("visitor_id").isNull() | (F.col("visitor_id") == ""),
               F.concat(F.lit("visitor_"), F.monotonically_increasing_id()))
        .otherwise(F.col("visitor_id"))
    )

    dim_visitor_count = dim_visitor.count()
    print(f" ✓ Processed {dim_visitor_count} visitor(s)")
else:
    print("  ⚠️  Skipping - no visitor data")
    dim_visitor = None
    dim_visitor_count = 0

# ============================================
# TRANSFORM: fact_media_engagement (UC-safe path extraction)
# ============================================

print("\n🔧 Processing fact_media_engagement...")

if visitor_count > 0:
    # Unity Catalog: use _metadata.file_path instead of input_file_name()
    # Example file_path:
    #
wasbs://raw-data@<acct>.blob.core.windows.net/visitors/gskhw4w4lm_20250101_120000/part-0000-....
    visitor_with_path = visitor_df.withColumn("file_path",
F.col("_metadata.file_path"))

    # Extract media_id safely; try multiple patterns and coalesce
    # Pattern 1: /visitors/<media_id>_YYYYMMDD_HHMMSS/
    p1 = F.regexp_extract(F.col("file_path"),
r"/visitors/([a-z0-9]+)_\d{8}_\d{6}/", 1)
    # Pattern 2 (fallback): /visitors/<media_id>_ (stop at first underscore)
    p2 = F.regexp_extract(F.col("file_path"), r"/visitors/([a-z0-9]+)_", 1)
```

```
    visitor_with_media = visitor_with_path.withColumn("media_id",
F.coalesce(p1, p2))

    # ENHANCEMENT: Ensure media_id is never NULL in fact table
    visitor_with_media = visitor_with_media.withColumn(
        "media_id",
        F.when(F.col("media_id").isNull() | (F.col("media_id") == ""),
               F.concat(F.lit("media_"), F.monotonically_increasing_id()))
        .otherwise(F.col("media_id"))
    )

    if "events" in visitor_df.columns:
        fact_engagement = (
            visitor_with_media
            .filter(F.col("events").isNotNull() & (F.size(F.col("events")) >
0))
            .select(
                F.col("media_id"),
                F.col("visitor_key").alias("visitor_id"),
                F.explode(F.col("events")).alias("event")
            )
            .filter(F.col("event.type") == "play")
            .withColumn("event_date",
F.to_date(F.from_unixtime(F.col("event.time"))))
            .groupBy(
                "media_id",
                "visitor_id",
                F.col("event_date").alias("date")
            )
            .agg(
                F.count("*").alias("play_count"),
                F.round(F.count("*") / F.lit(10.0), 2).alias("play_rate"),
                # Sum duration watched (seconds). Ensure cast to double.
                F.round(

F.sum(F.coalesce(F.col("event.duration_watched").cast("double"), F.lit(0.0))),
                    2
                ).alias("total_watch_time_seconds"),
```

```
                F.round(
F.avg(F.coalesce(F.col("event.percent_watched").cast("double"), F.lit(0.0))),
                    2
                ).alias("avg_percent_watched")
            )
            .withColumn("loaded_at", F.current_timestamp())
        )

        # ENHANCEMENT: Ensure no NULL keys in fact table
        fact_engagement = fact_engagement.filter(
            F.col("media_id").isNotNull() &
            F.col("visitor_id").isNotNull() &
            F.col("date").isNotNull()
        )

        fact_engagement_count = fact_engagement.count()
        print(f" ✓ Processed {fact_engagement_count} engagement fact(s)")
    else:
        print("  ⚠️  No events column found in visitor data")
        fact_engagement = None
        fact_engagement_count = 0
else:
    print("  ⚠️  Skipping - no visitor data")
    fact_engagement = None
    fact_engagement_count = 0

# ============================================
# DATA VALIDATION & CLEANING
# ============================================

print("\n🔍 Performing data validation...")

def validate_and_log(df, df_name, key_columns):
    """Validate DataFrame for NULL keys and log issues"""
    if df is None:
        print(f"\n  📊 Validating {df_name}: NO DATA")
        return df
```

```
    total_count = df.count()
    print(f"\n  📊 Validating {df_name}:")
    print(f"     - Total records: {total_count}")

    for col_name in key_columns:
        null_count = df.filter(F.col(col_name).isNull()).count()
        empty_count = df.filter((F.col(col_name) == "") |
(F.trim(F.col(col_name)) == "")).count()

        if null_count > 0:
            print(f"     ⚠️  WARNING: {null_count} records with NULL
{col_name}")
        if empty_count > 0:
            print(f"     ⚠️  WARNING: {empty_count} records with empty
{col_name}")

        # Show sample of problematic records
        if null_count > 0:
            print(f"     📝 Sample records with NULL {col_name}:")
            problematic = df.filter(F.col(col_name).isNull()).limit(3)
            display(problematic)

    return df

# Validate each DataFrame
dim_media = validate_and_log(dim_media, "dim_media", ["media_id"])
dim_visitor = validate_and_log(dim_visitor, "dim_visitor", ["visitor_id"])
fact_engagement = validate_and_log(fact_engagement, "fact_engagement",
["media_id", "visitor_id"])

# Remove records with NULL keys (final safety check)
print("\n🧹 Final data cleaning...")

if dim_media is not None:
    initial_media_count = dim_media.count()
    dim_media = dim_media.filter(F.col("media_id").isNotNull())
    cleaned_media_count = dim_media.count()
```

```
    print(f" ✓ dim_media: {initial_media_count} → {cleaned_media_count} (removed
{initial_media_count - cleaned_media_count})")

if dim_visitor is not None:
    initial_visitor_count = dim_visitor.count()
    dim_visitor = dim_visitor.filter(F.col("visitor_id").isNotNull())
    cleaned_visitor_count = dim_visitor.count()
    print(f" ✓ dim_visitor: {initial_visitor_count} → {cleaned_visitor_count}
(removed {initial_visitor_count - cleaned_visitor_count})")

if fact_engagement is not None:
    initial_fact_count = fact_engagement.count()
    fact_engagement = fact_engagement.filter(
        F.col("media_id").isNotNull() &
        F.col("visitor_id").isNotNull()
    )
    cleaned_fact_count = fact_engagement.count()
    print(f" ✓ fact_engagement: {initial_fact_count} → {cleaned_fact_count}
(removed {initial_fact_count - cleaned_fact_count})")

# ============================================
# DEDUPLICATION - CRITICAL FIX FOR PRIMARY KEY VIOLATIONS
# ============================================

print("\n🔍 Removing duplicates to prevent primary key violations...")

def remove_duplicates(df, df_name, key_columns):
    """Remove duplicate records based on key columns"""
    if df is None:
        return df

    initial_count = df.count()

    # Show duplicates before removal
    duplicate_count = df.groupBy(key_columns).count().filter(F.col("count") >
1).count()
    if duplicate_count > 0:
        print(f"  ⚠️  Found {duplicate_count} duplicate groups in {df_name}")
```

```
        # Show sample duplicates
        duplicates = df.groupBy(key_columns).count().filter(F.col("count") >
1).limit(3)
        print(f"  📝 Sample duplicate keys:")
        display(duplicates)

    # Remove duplicates - keep first occurrence
    window_spec = Window.partitionBy(key_columns).orderBy(F.lit(1))
    df_deduped = df.withColumn("row_num", F.row_number().over(window_spec))
    df_deduped = df_deduped.filter(F.col("row_num") == 1).drop("row_num")

    final_count = df_deduped.count()
    removed_count = initial_count - final_count

    if removed_count > 0:
        print(f" ✅ Removed {removed_count} duplicate records from {df_name}")

    return df_deduped

# Apply deduplication
dim_media = remove_duplicates(dim_media, "dim_media", ["media_id"])
dim_visitor = remove_duplicates(dim_visitor, "dim_visitor", ["visitor_id"])
if fact_engagement is not None:
    fact_engagement = remove_duplicates(fact_engagement, "fact_engagement",
["media_id", "visitor_id", "date"])

# Additional: Check for duplicate keys after cleaning
print(f"\n🔎 Final duplicate check...")

if dim_media is not None:
    media_duplicates =
dim_media.groupBy("media_id").count().filter(F.col("count") > 1)
    if media_duplicates.count() > 0:
        print(f" ❌ CRITICAL: Still found duplicate media_id after cleaning!")
        display(media_duplicates)
    else:
        print(f" ✅ No duplicate media_id found")
```

```
if dim_visitor is not None:
    visitor_duplicates =
dim_visitor.groupBy("visitor_id").count().filter(F.col("count") > 1)
    if visitor_duplicates.count() > 0:
        print(f" ❌ CRITICAL: Still found duplicate visitor_id after cleaning!")
        display(visitor_duplicates)
    else:
        print(f" ✅ No duplicate visitor_id found")

# ============================================
# FINAL DATA QUALITY CHECK
# ============================================

print("\n📊 Final Data Quality Report:")

if dim_media is not None:
    print(f"  📹 dim_media: {dim_media.count():,} records")
    print(f"     - Unique media_id:
{dim_media.select('media_id').distinct().count()}")

if dim_visitor is not None:
    print(f"  👥 dim_visitor: {dim_visitor.count():,} records")
    print(f"     - Unique visitor_id:
{dim_visitor.select('visitor_id').distinct().count()}")

if fact_engagement is not None:
    print(f"  📈 fact_engagement: {fact_engagement.count():,} records")
    print(f"     - Unique combinations: {fact_engagement.select('media_id',
'visitor_id', 'date').distinct().count()}")

# ============================================
# WRITE PROCESSED DATA
# ============================================

print("\n💾 Writing processed data...")

try:
```

```
    if dim_media is not None and dim_media.count() > 0:

dim_media.write.mode("overwrite").parquet(f"{PROCESSED_PATH}/dim-media/")
        print(f" ✅ Saved dim_media: {dim_media.count()} records")

        # Display final dim_media sample
        print(f"\n📊 Final dim_media sample:")
        display(dim_media.limit(3))
    else:
        print(f"  ⚠️  Skipped dim_media (no data)")

    if dim_visitor is not None and dim_visitor.count() > 0:

dim_visitor.write.mode("overwrite").parquet(f"{PROCESSED_PATH}/dim-visitor/")
        print(f" ✅ Saved dim_visitor: {dim_visitor.count()} records")
    else:
        print(f"  ⚠️  Skipped dim_visitor (no data)")

    if fact_engagement is not None and fact_engagement.count() > 0:

fact_engagement.write.mode("overwrite").parquet(f"{PROCESSED_PATH}/fact-engagem
ent/")
        print(f" ✅ Saved fact_engagement: {fact_engagement.count()} records")
    else:
        print(f"  ⚠️  Skipped fact_engagement (no data)")

except Exception as e:
    print(f" ❌ Error writing: {str(e)[:300]}")
    raise

# ============================================
# SUMMARY
# ============================================

print("\n" + "=" * 70)
print("✅ DATA PROCESSING COMPLETE")
print("=" * 70)
```

```
print(f"  dim_media:            {dim_media.count() if dim_media else 0:,} records")
print(f"  dim_visitor:          {dim_visitor.count() if dim_visitor else 0:,} records")
print(f"  fact_engagement:      {fact_engagement.count() if fact_engagement else 0:,} records")
print(f"  Processed at:         {datetime.now().strftime('%Y-%m-%d %H:%M:%S')}")
print("=" * 70)

print(f"\n🎯 Next Steps:")
print(f"  1. Verify no duplicate keys in the final data")
print(f"  2. Run ADF pipeline to load to SQL Database")
print(f"  3. Check pipeline_execution_log for any issues")
print("=" * 70 + "\n")
```

Phase 6: Azure SQL Database Setup

Step 6.1: Create Azure SQL Database

1. Search for **SQL databases** → **+ Create**
2. Configure:
    - **Resource Group**: `rg-wistia-analytics`
    - **Database Name**: `sql-wistia-analytics`
    - **Server**: Create new
        - **Server Name**: `sql-wistia-server`
        - **Admin Login**: `sqladmin`
        - **Password**: (create strong password)
        - **Location**: Same as resource group
    - **Compute + Storage**: Basic (5 DTUs) for dev
3. Click **Review + Create** → **Create**

Step 6.2: Configure Firewall

1. Go to SQL Server → **Networking**
2. Add your client IP address
3. Enable **Allow Azure services and resources to access this server**
4. Click **Save**

Step 6.3: Create Tables

1. Go to **Query Editor** in Azure Portal
2. Login with SQL authentication
3. Run the following SQL:

```
-- Create staging tables
CREATE TABLE stg_dim_media (
    media_id NVARCHAR(50) PRIMARY KEY,
    title NVARCHAR(500),
    url NVARCHAR(1000),
    channel NVARCHAR(50),
    created_at DATETIME2,
    processed_at DATETIME2
);

CREATE TABLE stg_dim_visitor (
    visitor_id NVARCHAR(100) PRIMARY KEY,
    ip_address NVARCHAR(50),
    country NVARCHAR(100),
    processed_at DATETIME2
);

CREATE TABLE stg_fact_engagement (
    media_id NVARCHAR(50),
    visitor_id NVARCHAR(100),
    date DATE,
    play_count INT,
    play_rate DECIMAL(5,2),
    total_watch_time INT,
    watched_percent DECIMAL(5,2),
    loaded_at DATETIME2,
    PRIMARY KEY (media_id, visitor_id, date)
);

-- Create production tables (optional)
CREATE TABLE dim_media AS SELECT * FROM stg_dim_media WHERE 1=0;
CREATE TABLE dim_visitor AS SELECT * FROM stg_dim_visitor WHERE 1=0;
CREATE TABLE fact_engagement AS SELECT * FROM stg_fact_engagement WHERE 1=0;
```

Phase 7: Azure Data Factory Pipeline

Step 7.1: Create Data Factory

1. Search for **Data factories** → **+ Create**
2. Configure:
    - **Resource Group**: `rg-wistia-analytics`
    - **Name**: `adf-wistia-analytics`
    - **Region**: Same as resource group
    - **Version**: V2
3. Click **Review + Create** → **Create**
4. Click **Launch Studio**

Step 7.2: Create Linked Services

A. Azure Databricks Linked Service

1. Click **Manage** (toolbox icon) → **Linked Services** → **+ New**
2. Select **Azure Databricks** → **Continue**
3. Configure:
    - **Name**: `AzureDatabricks_ls`
    - **Databricks Workspace**: Select your workspace
    - **Cluster**: Use **New Job Cluster** or **Existing Interactive Cluster**
    - **Access Token**: Use your Databricks token from Key Vault
4. Test connection → **Create**

B. Azure Blob Storage Linked Service

1. **+ New** → **Azure Blob Storage** → **Continue**
2. Configure:
    - **Name**: `AzureBlobStorage_ls`
    - **Storage Account**: `stwistiaanalytics`
    - **Authentication**: Account Key or Managed Identity
3. Test connection → **Create**

C. Azure SQL Database Linked Service

1. **+ New** → **Azure SQL Database** → **Continue**
2. Configure:
    - **Name**: `AzureSqlDatabase_ls`
    - **Server**: `sql-wistia-server`
    - **Database**: `sql-wistia-analytics`
    - **Authentication**: SQL authentication
    - **Username**: `sqladmin`
    - **Password**: (from Key Vault or enter directly)
3. Test connection → **Create**

Step 7.3: Create Datasets

A. Parquet Datasets (Silver Layer)

1. Click **Author** (pencil icon) → **+** → **Dataset**
2. Create three datasets:

ds_Parquet_dim_media:

- **Linked Service**: `AzureBlobStorage_ls`
- **File Path**: Container: `silver-layer`, Directory: `dim-media`
- **Format**: Parquet

ds_parquet_dim_visitor:

- **Linked Service**: `AzureBlobStorage_ls`
- **File Path**: Container: `silver-layer`, Directory: `dim-visitor`
- **Format**: Parquet

ds_parquet_fact_engagement:

- **Linked Service**: `AzureBlobStorage_ls`
- **File Path**: Container: `silver-layer`, Directory: `fact-engagement`
- **Format**: Parquet

B. SQL Datasets (Staging Tables) Create three SQL datasets:

ds_sql_stg_dim_media:

- **Linked Service**: `AzureSqlDatabase_ls`
- **Table**: `dbo.stg_dim_media`

ds_sql_stg_dim_visitor:

- **Linked Service**: `AzureSqlDatabase_ls`
- **Table**: `dbo.stg_dim_visitor`

ds_sql_stg_fact_engagement:

- **Linked Service**: `AzureSqlDatabase_ls`
- **Table**: `dbo.stg_fact_engagement`

Step 7.4: Create Pipeline

1. Click **Author** → **+** → **Pipeline**

2. Name: `pl_wistia_main_pipeline`

Add Activities in Order:

Activity 1: Run API Ingestion 00

- Type: Databricks Notebook
- **Notebook Path**: `/Users/your-email/wistia-01`
- **Linked Service**: `AzureDatabricks_ls`

Activity 2: Run API Ingestion 01

- Type: Databricks Notebook
- **Notebook Path**: `/Users/your-email/wistia-02`
- **Depends On**: Run_API_Ingestion_00 (Success)

Activity 3: Run Data Processing

- Type: Databricks Notebook
- **Notebook Path**: `/Users/your-email/wistia-03`
- **Depends On**: Run_API_Ingestion_01 (Success)

Activity 4-6: Copy Activities (Parallel) All three depend on Run_Data_Processing (Success):

Copy_dim_media_to_SQL:

- **Source**: `ds_Parquet_dim_media`
- **Sink**: `ds_sql_stg_dim_media`
- **Pre-copy Script**: `TRUNCATE TABLE stg_dim_media`

Copy_dim_visitor_to_SQL:

- **Source**: `ds_parquet_dim_visitor`
- **Sink**: `ds_sql_stg_dim_visitor`
- **Pre-copy Script**: `TRUNCATE TABLE stg_dim_visitor`

Copy_fact_engagement_to_SQL:

- **Source**: `ds_parquet_fact_engagement`
- **Sink**: `ds_sql_stg_fact_engagement`
- **Pre-copy Script**: `TRUNCATE TABLE stg_fact_engagement`

3. Click **Validate All**
4. Click **Publish All**

Step 7.5: Create Pipeline Trigger

1. In your pipeline, click **Add Trigger** → **New/Edit**
2. Click **+ New**
3. Configure:
    - **Name**: `daily_trigger`
    - **Type**: Schedule
    - **Start Date**: Today
    - **Recurrence**: Every 1 Day at 2:00 AM
    - **End**: After 7 days
4. Click **OK** → **Publish**

Phase 8: CI/CD with GitHub Actions

Step 8.1: Create GitHub Repository

1. Go to GitHub.com → Create new repository
2. Name: `wistia-video-analytics`
3. Initialize with README
4. Clone locally

Step 8.2: Export ADF Resources

1. In ADF Studio, click **Manage** → **Git Configuration**
2. Connect to your GitHub repository
3. Set **Collaboration Branch**: `main`
4. Set **Publish Branch**: `adf_publish`
5. Click **Apply**

All your pipelines, datasets, and linked services will be exported as JSON files to the repo.

Step 8.3: Create GitHub Secrets

1. Go to your repo → **Settings** → **Secrets and variables** → **Actions**
2. Add these secrets:
    - `AZURE_CREDENTIALS` (Service Principal JSON)
    - `DATABRICKS_TOKEN`
    - `SQL_PASSWORD`

Step 8.4: Create GitHub Actions Workflow

Create `.github/workflows/deploy-adf.yml`:

```
name: Deploy ADF Pipeline

on:
  push:
    branches: [ main ]
  workflow_dispatch:

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

      - name: Azure Login
        uses: azure/login@v1
        with:
          creds: ${{ secrets.AZURE_CREDENTIALS }}

      - name: Validate ADF Resources
        run: |
          echo "Validating Data Factory resources..."
          # Add validation scripts here

      - name: Deploy to ADF
        run: |
          az datafactory pipeline create \
            --resource-group rg-wistia-analytics \
            --factory-name adf-wistia-analytics \
            --name pl_wistia_main_pipeline \
            --pipeline @pipeline/pl_wistia_main_pipeline.json
```

Phase 9: Production Execution

Step 9.1: Manual Pipeline Test

1. Go to ADF Studio → **Author**
2. Open `pl_wistia_main_pipeline`
3. Click **Debug** to test
4. Monitor execution in **Monitor** tab

Step 9.2: Enable Scheduled Trigger

1. Go to **Manage** → **Triggers**
2. Find `daily_trigger`
3. Click **Activate**

Step 9.3: Daily Monitoring Checklist

For each of the 7 days:

- [ ] Check pipeline run status in ADF Monitor
- [ ] Verify new data in Bronze layer
- [ ] Verify transformed data in Silver layer
- [ ] Verify SQL tables have new records
- [ ] Check for any failures or warnings
- [ ] Document daily data volumes

Phase 10: Monitoring & Validation

Step 10.1: Create Monitoring Dashboard

1. In Azure Portal, create **Dashboard**
2. Add tiles:
    - ADF pipeline runs (success/failure)
    - Databricks job status
    - SQL database DTU usage
    - Storage account capacity

📈 Phase 11: Create Power BI Dashboard

Step 11.1: Connect Power BI to Azure SQL

1. Open **Power BI Desktop**
2. Click **Get Data** → **Azure** → **Azure SQL Database**
3. Enter server details:
    - **Server**: `sql-wistia-server.database.windows.net`
    - **Database**: `sql-wistia-analytics`

4. Authentication: **Database** (use sqladmin credentials)
5. Select tables:
    - `stg_dim_media`
    - `stg_dim_visitor`
    - `stg_fact_engagement`
6. Click **Load**

Step 11.2: Create Data Model Relationships

1. Go to **Model** view
2. Create relationships:
    - `fact_engagement.media_id` → `dim_media.media_id`
    - `fact_engagement.visitor_id` → `dim_visitor.visitor_id`

Step 11.3: Create DAX Measures

```
// Total Plays
Total Plays = SUM(fact_engagement[play_count])

// Average Completion Rate
Avg Completion = AVERAGE(fact_engagement[watched_percent])

// Total Watch Hours
Total Watch Hours = SUM(fact_engagement[total_watch_time]) / 3600

// Engagement Rate
Engagement Rate =
    DIVIDE(
        SUM(fact_engagement[play_count]),
        DISTINCTCOUNT(fact_engagement[visitor_id])
    )

// Videos by Channel
Videos by Channel = COUNTROWS(dim_media)
```

Step 11.4: Create Visualizations

Page 1: Executive Summary

- Card: Total Plays
- Card: Total Watch Hours
- Card: Average Completion Rate
- Card: Unique Visitors

- Line Chart: Daily Plays Trend
- Donut Chart: Plays by Channel (Facebook vs YouTube)

Page 2: Video Performance

- Table: Media Title, Total Plays, Avg Completion, Watch Time
- Bar Chart: Top 10 Videos by Plays
- Scatter Plot: Play Count vs Completion Rate

Page 3: Visitor Analytics

- Map: Visitors by Country
- Table: Top Countries by Engagement
- Line Chart: New vs Returning Visitors

Step 11.5: Publish to Power BI Service

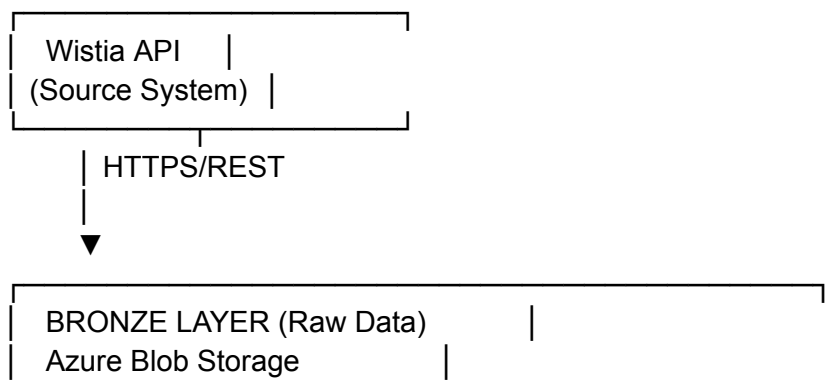
1. Click **Publish** in Power BI Desktop
2. Select workspace
3. Configure **Scheduled Refresh**:
    - Frequency: Daily at 6:00 AM
    - Gateway: Configure if needed

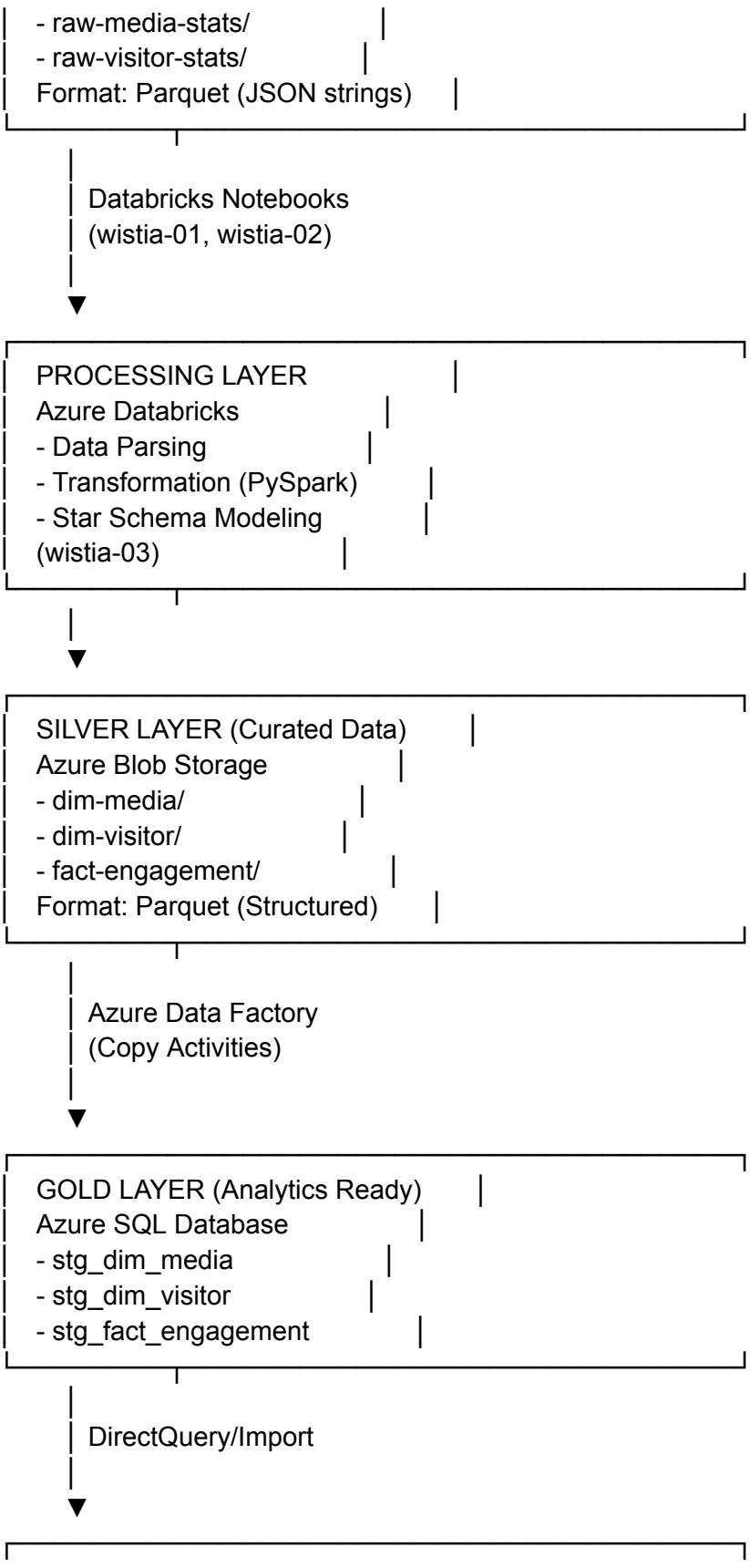
Create Architecture Diagram

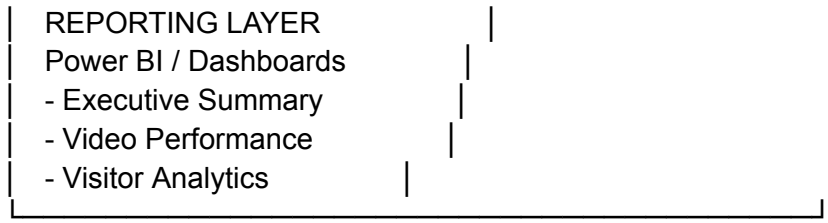
System Architecture

Data Flow Diagram

\`\`\`




- raw-media-stats/
- raw-visitor-stats/
Format: Parquet (JSON strings)
Databricks Notebooks
(wistia-01, wistia-02)
PROCESSING LAYER
Azure Databricks
- Data Parsing
- Transformation (PySpark)
- Star Schema Modeling
(wistia-03)
SILVER LAYER (Curated Data)
Azure Blob Storage
- dim-media/
- dim-visitor/
- fact-engagement/
Format: Parquet (Structured)
Azure Data Factory
(Copy Activities)
GOLD LAYER (Analytics Ready)
Azure SQL Database
- stg_dim_media
- stg_dim_visitor
- stg_fact_engagement
DirectQuery/Import



Technology Stack

Layer	Technology	Purpose
Source	Wistia API	Video analytics data
Ingestion	Azure Databricks	API calls, data extraction
Storage	Azure Blob Storage	Data lake (Bronze/Silver)
Processing	PySpark	Data transformation
Orchestration	Azure Data Factory	Workflow automation
Database	Azure SQL Database	Structured storage (Gold)
Reporting	Power BI	Visualization & analytics
DevOps	GitHub Actions	CI/CD automation
Secrets	Azure Key Vault	Credential management

Daily Operations

Morning Checklist (After 6 AM)
1. Check ADF pipeline run status
2. Verify record counts in SQL database
3. Review data quality report
4. Check Power BI dashboard refresh