```python
"""
Agricultural Sensor Data Pipeline
==================================

A production-grade data pipeline for ingesting, transforming, validating,
and storing agricultural sensor data.

Author: Kapil yadav
"""

import os
import sys
import logging
import json
from datetime import datetime, timedelta
from pathlib import Path
from typing import Dict, List, Optional, Tuple
import pandas as pd
import duckdb
import numpy as np
from dataclasses import dataclass
import pytz


# Configure logging
logging.basicConfig(
    level=logging.INFO,
    format='%(asctime)s - %(name)s - %(levelname)s - %(message)s',
    handlers=[
        logging.FileHandler('pipeline.log'),
        logging.StreamHandler()
    ]
)
logger = logging.getLogger(__name__)


@dataclass
class PipelineConfig:
    """Configuration for the data pipeline."""
    raw_data_path: str = "data/raw"
    processed_data_path: str = "data/processed"
    quality_report_path: str = "data_quality_report.csv"
    timezone: str = "Asia/Kolkata"
    batch_size: int = 10000

    # Calibration parameters (example values)
    calibration_params: Dict[str, Dict[str, float]] = None

    # Expected value ranges for anomaly detection
    value_ranges: Dict[str, Tuple[float, float]] = None

    def __post_init__(self):
        if self.calibration_params is None:
            self.calibration_params = {
                'temperature': {'multiplier': 1.0, 'offset': 0.0},
```

```python
            'humidity': {'multiplier': 1.0, 'offset': 0.0},
            'soil_moisture': {'multiplier': 1.0, 'offset': 0.0},
            'light_intensity': {'multiplier': 1.0, 'offset': 0.0}
        }

    if self.value_ranges is None:
        self.value_ranges = {
            'temperature': (-20.0, 60.0),
            'humidity': (0.0, 100.0),
            'soil_moisture': (0.0, 100.0),
            'light_intensity': (0.0, 100000.0)
        }


class DataIngestionEngine:
    """Handles data ingestion from raw Parquet files."""

    def __init__(self, config: PipelineConfig):
        self.config = config
        self.conn = duckdb.connect(':memory:')
        self.checkpoint_file = "ingestion_checkpoint.json"
        self.stats = {
            'files_read': 0,
            'records_processed': 0,
            'records_skipped': 0,
            'files_failed': 0
        }

    def load_checkpoint(self) -> Dict:
        """Load ingestion checkpoint for incremental loading."""
        if os.path.exists(self.checkpoint_file):
            try:
                with open(self.checkpoint_file, 'r') as f:
                    return json.load(f)
            except Exception as e:
                logger.warning(f"Failed to load checkpoint: {e}")
        return {'last_processed_date': None, 'processed_files': []}

    def save_checkpoint(self, checkpoint: Dict):
        """Save ingestion checkpoint."""
        try:
            with open(self.checkpoint_file, 'w') as f:
                json.dump(checkpoint, f, indent=2)
        except Exception as e:
            logger.error(f"Failed to save checkpoint: {e}")

    def validate_file_schema(self, file_path: str) -> bool:
        """Validate file schema using DuckDB."""
        try:
            # Query to check schema
            schema_query = f"""
            SELECT column_name, data_type
            FROM information_schema.columns
            WHERE table_name = 'temp_table'
            """
```

```python
            # Load file into temporary table
            self.conn.execute(f"CREATE OR REPLACE TABLE temp_table AS SELECT * FROM '{file_path}' LIMIT 1")
            schema_result = self.conn.execute(schema_query).fetchall()

            # Expected columns
            expected_columns = {
                'sensor_id': 'VARCHAR',
                'timestamp': 'VARCHAR',  # Will be converted to datetime
                'reading_type': 'VARCHAR',
                'value': 'DOUBLE',
                'battery_level': 'DOUBLE'
            }

            actual_columns = {row[0]: row[1] for row in schema_result}

            # Check if all expected columns exist
            missing_columns = set(expected_columns.keys()) - set(actual_columns.keys())
            if missing_columns:
                logger.error(f"Missing columns in {file_path}: {missing_columns}")
                return False

            logger.info(f"Schema validation passed for {file_path}")
            return True

        except Exception as e:
            logger.error(f"Schema validation failed for {file_path}: {e}")
            return False

    def validate_data_quality(self, file_path: str) -> Dict:
        """Validate data quality using DuckDB."""
        try:
            # Load data for validation
            self.conn.execute(f"CREATE OR REPLACE TABLE validation_table AS SELECT * FROM '{file_path}'")

            # Run validation queries
            validation_results = {}

            # Check for missing values
            missing_query = """
            SELECT
                COUNT(*) as total_rows,
                COUNT(CASE WHEN sensor_id IS NULL THEN 1 END) as missing_sensor_id,
                COUNT(CASE WHEN timestamp IS NULL THEN 1 END) as missing_timestamp,
                COUNT(CASE WHEN reading_type IS NULL THEN 1 END) as missing_reading_type,
                COUNT(CASE WHEN value IS NULL THEN 1 END) as missing_value,
                COUNT(CASE WHEN battery_level IS NULL THEN 1 END) as missing_battery_level
            FROM validation_table
            """

            missing_result = self.conn.execute(missing_query).fetchone()
            validation_results['missing_values'] = {
                'total_rows': missing_result[0],
                'missing_sensor_id': missing_result[1],
                'missing_timestamp': missing_result[2],
```

```
            'missing_reading_type': missing_result[3],
            'missing_value': missing_result[4],
            'missing_battery_level': missing_result[5]
        }

        # Check value ranges
        range_query = """
        SELECT
            reading_type,
            MIN(value) as min_value,
            MAX(value) as max_value,
            COUNT(*) as count
        FROM validation_table
        GROUP BY reading_type
        """

        range_results = self.conn.execute(range_query).fetchall()
        validation_results['value_ranges'] = {
            row[0]: {'min': row[1], 'max': row[2], 'count': row[3]}
            for row in range_results
        }

        return validation_results

    except Exception as e:
        logger.error(f"Data quality validation failed for {file_path}: {e}")
        return {}

def ingest_file(self, file_path: str) -> Optional[pd.DataFrame]:
    """Ingest a single Parquet file."""
    try:
        logger.info(f"Ingesting file: {file_path}")

        # Validate schema
        if not self.validate_file_schema(file_path):
            self.stats['files_failed'] += 1
            return None

        # Validate data quality
        quality_results = self.validate_data_quality(file_path)
        if quality_results:
            logger.info(f"Data quality check completed for {file_path}")
            total_rows = quality_results['missing_values']['total_rows']
            missing_values = sum(quality_results['missing_values'].values()) - total_rows
            logger.info(f"Total rows: {total_rows}, Missing values: {missing_values}")

        # Load data
        df = pd.read_parquet(file_path)

        # Basic data validation
        if df.empty:
            logger.warning(f"Empty file: {file_path}")
            return None

        # Log ingestion summary using DuckDB
```

```python
            self.conn.execute("CREATE OR REPLACE TABLE ingestion_summary AS SELECT * FROM df")

            summary_query = """
            SELECT
                COUNT(*) as total_records,
                COUNT(DISTINCT sensor_id) as unique_sensors,
                COUNT(DISTINCT reading_type) as unique_reading_types,
                MIN(timestamp) as earliest_timestamp,
                MAX(timestamp) as latest_timestamp
            FROM ingestion_summary
            """

            summary = self.conn.execute(summary_query).fetchone()
            logger.info(f"Ingestion summary - Records: {summary[0]}, "
                      f"Sensors: {summary[1]}, Reading types: {summary[2]}")

            self.stats['files_read'] += 1
            self.stats['records_processed'] += len(df)

            return df

        except Exception as e:
            logger.error(f"Failed to ingest {file_path}: {e}")
            self.stats['files_failed'] += 1
            return None

    def ingest_batch(self, start_date: Optional[str] = None) -> List[pd.DataFrame]:
        """Ingest batch of files with incremental loading."""
        checkpoint = self.load_checkpoint()
        processed_files = set(checkpoint.get('processed_files', []))

        # Get list of files to process
        raw_path = Path(self.config.raw_data_path)
        if not raw_path.exists():
            logger.error(f"Raw data path does not exist: {raw_path}")
            return []

        parquet_files = list(raw_path.glob("*.parquet"))
        parquet_files.sort()

        # Filter files for incremental loading
        files_to_process = []
        for file_path in parquet_files:
            if str(file_path) not in processed_files:
                files_to_process.append(file_path)

        logger.info(f"Found {len(files_to_process)} files to process")

        ingested_data = []
        newly_processed = []

        for file_path in files_to_process:
            df = self.ingest_file(str(file_path))
            if df is not None:
                ingested_data.append(df)
```

```python
            newly_processed.append(str(file_path))

        # Update checkpoint
        if newly_processed:
            checkpoint['processed_files'].extend(newly_processed)
            checkpoint['last_processed_date'] = datetime.now().isoformat()
            self.save_checkpoint(checkpoint)

        logger.info(f"Ingestion completed. Stats: {self.stats}")
        return ingested_data


class DataTransformationEngine:
    """Handles data transformation and enrichment."""

    def __init__(self, config: PipelineConfig):
        self.config = config
        self.conn = duckdb.connect(':memory:')
        self.tz = pytz.timezone(config.timezone)

    def clean_data(self, df: pd.DataFrame) -> pd.DataFrame:
        """Clean data - remove duplicates, handle missing values, detect outliers."""
        logger.info("Starting data cleaning...")

        # Remove duplicates
        initial_count = len(df)
        df = df.drop_duplicates()
        logger.info(f"Removed {initial_count - len(df)} duplicate records")

        # Handle missing values
        # Fill missing battery_level with median
        df['battery_level'] = df['battery_level'].fillna(df['battery_level'].median())

        # Drop rows with missing critical values
        df = df.dropna(subset=['sensor_id', 'timestamp', 'reading_type', 'value'])

        # Detect and handle outliers using z-score
        for reading_type in df['reading_type'].unique():
            mask = df['reading_type'] == reading_type
            values = df.loc[mask, 'value']

            if len(values) > 1:
                z_scores = np.abs((values - values.mean()) / values.std())
                outlier_mask = z_scores > 3

                if outlier_mask.any():
                    logger.info(f"Found {outlier_mask.sum()} outliers for {reading_type}")
                    # Cap outliers at 3 standard deviations
                    mean_val = values.mean()
                    std_val = values.std()

                    df.loc[mask & (z_scores > 3), 'value'] = mean_val + 3 * std_val
                    df.loc[mask & (z_scores < -3), 'value'] = mean_val - 3 * std_val

        logger.info(f"Data cleaning completed. Final record count: {len(df)}")
```

```python
        return df

    def process_timestamps(self, df: pd.DataFrame) -> pd.DataFrame:
        """Process timestamps - convert to ISO format and adjust timezone."""
        logger.info("Processing timestamps...")

        # Convert to datetime
        df['timestamp'] = pd.to_datetime(df['timestamp'])

        # Convert to target timezone
        df['timestamp'] = df['timestamp'].dt.tz_localize('UTC').dt.tz_convert(self.tz)

        # Convert back to ISO format
        df['timestamp'] = df['timestamp'].dt.strftime('%Y-%m-%dT%H:%M:%S%z')

        return df

    def apply_calibration(self, df: pd.DataFrame) -> pd.DataFrame:
        """Apply calibration logic to normalize values."""
        logger.info("Applying calibration...")

        df['raw_value'] = df['value'].copy()

        for reading_type, params in self.config.calibration_params.items():
            mask = df['reading_type'] == reading_type
            if mask.any():
                df.loc[mask, 'value'] = (
                    df.loc[mask, 'raw_value'] * params['multiplier'] + params['offset']
                )

        return df

    def detect_anomalies(self, df: pd.DataFrame) -> pd.DataFrame:
        """Detect anomalous readings based on expected ranges."""
        logger.info("Detecting anomalies...")

        df['anomalous_reading'] = False

        for reading_type, (min_val, max_val) in self.config.value_ranges.items():
            mask = df['reading_type'] == reading_type
            if mask.any():
                anomaly_mask = (df['value'] < min_val) | (df['value'] > max_val)
                df.loc[mask & anomaly_mask, 'anomalous_reading'] = True

        anomaly_count = df['anomalous_reading'].sum()
        logger.info(f"Detected {anomaly_count} anomalous readings")

        return df

    def compute_aggregations(self, df: pd.DataFrame) -> pd.DataFrame:
        """Compute daily averages and rolling averages."""
        logger.info("Computing aggregations...")

        # Convert timestamp back to datetime for aggregation
        df['timestamp_dt'] = pd.to_datetime(df['timestamp'])
```

```
df['date'] = df['timestamp_dt'].dt.date

# Load data into DuckDB for efficient aggregation
self.conn.execute("CREATE OR REPLACE TABLE sensor_data AS SELECT * FROM df")

# Daily averages
daily_avg_query = """
SELECT
    sensor_id,
    reading_type,
    date,
    AVG(value) as daily_avg_value,
    COUNT(*) as daily_record_count
FROM sensor_data
GROUP BY sensor_id, reading_type, date
ORDER BY sensor_id, reading_type, date
"""

daily_avg_df = self.conn.execute(daily_avg_query).df()

# Join back to original data
df = df.merge(
    daily_avg_df,
    on=['sensor_id', 'reading_type', 'date'],
    how='left'
)

# 7-day rolling average (simplified - using daily averages)
rolling_avg_query = """
SELECT
    sensor_id,
    reading_type,
    date,
    AVG(daily_avg_value) OVER (
        PARTITION BY sensor_id, reading_type
        ORDER BY date
        ROWS BETWEEN 6 PRECEDING AND CURRENT ROW
    ) as rolling_7day_avg
FROM (
    SELECT DISTINCT sensor_id, reading_type, date, daily_avg_value
    FROM sensor_data
)
ORDER BY sensor_id, reading_type, date
"""

rolling_avg_df = self.conn.execute(rolling_avg_query).df()

# Join rolling averages
df = df.merge(
    rolling_avg_df,
    on=['sensor_id', 'reading_type', 'date'],
    how='left'
)

logger.info("Aggregations completed")
```

```python
        return df

    def transform_data(self, df: pd.DataFrame) -> pd.DataFrame:
        """Apply complete transformation pipeline."""
        logger.info("Starting data transformation...")

        # Clean data
        df = self.clean_data(df)

        # Process timestamps
        df = self.process_timestamps(df)

        # Apply calibration
        df = self.apply_calibration(df)

        # Detect anomalies
        df = self.detect_anomalies(df)

        # Compute aggregations
        df = self.compute_aggregations(df)

        logger.info("Data transformation completed")
        return df


class DataQualityValidator:
    """Validates data quality and generates reports."""

    def __init__(self, config: PipelineConfig):
        self.config = config
        self.conn = duckdb.connect(':memory:')

    def validate_data_types(self, df: pd.DataFrame) -> Dict:
        """Validate data types."""
        logger.info("Validating data types...")

        type_issues = {}

        # Check timestamp format
        try:
            pd.to_datetime(df['timestamp'])
        except Exception:
            type_issues['timestamp'] = "Invalid timestamp format"

        # Check numeric types
        if not pd.api.types.is_numeric_dtype(df['value']):
            type_issues['value'] = "Value column is not numeric"

        if not pd.api.types.is_numeric_dtype(df['battery_level']):
            type_issues['battery_level'] = "Battery level column is not numeric"

        return type_issues

    def check_value_ranges(self, df: pd.DataFrame) -> Dict:
        """Check value ranges per reading type."""
```

```python
        logger.info("Checking value ranges...")

        range_issues = {}

        for reading_type in df['reading_type'].unique():
            if reading_type in self.config.value_ranges:
                min_expected, max_expected = self.config.value_ranges[reading_type]
                type_data = df[df['reading_type'] == reading_type]

                min_actual = type_data['value'].min()
                max_actual = type_data['value'].max()

                issues = []
                if min_actual < min_expected:
                    issues.append(f"Min value {min_actual} below expected {min_expected}")
                if max_actual > max_expected:
                    issues.append(f"Max value {max_actual} above expected {max_expected}")

                if issues:
                    range_issues[reading_type] = issues

        return range_issues

    def detect_time_gaps(self, df: pd.DataFrame) -> Dict:
        """Detect gaps in hourly data using DuckDB."""
        logger.info("Detecting time gaps...")

        # Load data into DuckDB
        self.conn.execute("CREATE OR REPLACE TABLE gap_analysis AS SELECT * FROM df")

        # Generate expected hourly timestamps and find gaps
        gap_query = """
        WITH hourly_expected AS (
            SELECT
                sensor_id,
                reading_type,
                generate_series(
                    date_trunc('hour', MIN(timestamp_dt)),
                    date_trunc('hour', MAX(timestamp_dt)),
                    INTERVAL '1 hour'
                ) as expected_hour
            FROM gap_analysis
            GROUP BY sensor_id, reading_type
        ),
        actual_hours AS (
            SELECT
                sensor_id,
                reading_type,
                date_trunc('hour', timestamp_dt) as actual_hour,
                COUNT(*) as record_count
            FROM gap_analysis
            GROUP BY sensor_id, reading_type, date_trunc('hour', timestamp_dt)
        )
        SELECT
            e.sensor_id,
```

```
        e.reading_type,
        COUNT(*) as expected_hours,
        COUNT(a.actual_hour) as actual_hours,
        COUNT(*) - COUNT(a.actual_hour) as missing_hours
    FROM hourly_expected e
    LEFT JOIN actual_hours a ON e.sensor_id = a.sensor_id
        AND e.reading_type = a.reading_type
        AND e.expected_hour = a.actual_hour
    GROUP BY e.sensor_id, e.reading_type
    HAVING COUNT(*) - COUNT(a.actual_hour) > 0
    ORDER BY missing_hours DESC
    """

    gap_results = self.conn.execute(gap_query).fetchall()

    gaps = {}
    for row in gap_results:
        sensor_id, reading_type, expected, actual, missing = row
        gaps[f"{sensor_id}_{reading_type}"] = {
            'expected_hours': expected,
            'actual_hours': actual,
            'missing_hours': missing
        }

    return gaps

def profile_data(self, df: pd.DataFrame) -> Dict:
    """Profile data quality metrics."""
    logger.info("Profiling data quality...")

    profile = {}

    # Missing values percentage
    total_rows = len(df)
    missing_percentages = {}

    for reading_type in df['reading_type'].unique():
        type_data = df[df['reading_type'] == reading_type]
        missing_count = type_data['value'].isna().sum()
        missing_percentages[reading_type] = (missing_count / len(type_data)) * 100

    profile['missing_value_percentages'] = missing_percentages

    # Anomalous readings percentage
    anomaly_percentages = {}
    for reading_type in df['reading_type'].unique():
        type_data = df[df['reading_type'] == reading_type]
        anomaly_count = type_data['anomalous_reading'].sum()
        anomaly_percentages[reading_type] = (anomaly_count / len(type_data)) * 100

    profile['anomaly_percentages'] = anomaly_percentages

    # Time coverage per sensor
    coverage = {}
    for sensor_id in df['sensor_id'].unique():
```

```python
        sensor_data = df[df['sensor_id'] == sensor_id]
        min_time = sensor_data['timestamp_dt'].min()
        max_time = sensor_data['timestamp_dt'].max()
        coverage[sensor_id] = {
            'start_time': min_time.isoformat(),
            'end_time': max_time.isoformat(),
            'duration_hours': (max_time - min_time).total_seconds() / 3600
        }

    profile['time_coverage'] = coverage

    return profile

def generate_quality_report(self, df: pd.DataFrame) -> Dict:
    """Generate comprehensive data quality report."""
    logger.info("Generating data quality report...")

    report = {
        'timestamp': datetime.now().isoformat(),
        'total_records': len(df),
        'unique_sensors': df['sensor_id'].nunique(),
        'reading_types': df['reading_type'].unique().tolist(),
        'data_type_issues': self.validate_data_types(df),
        'value_range_issues': self.check_value_ranges(df),
        'time_gaps': self.detect_time_gaps(df),
        'data_profile': self.profile_data(df)
    }

    # Save report as CSV
    report_df = pd.DataFrame([{
        'metric': 'total_records',
        'value': report['total_records']
    }, {
        'metric': 'unique_sensors',
        'value': report['unique_sensors']
    }])

    # Add detailed metrics
    for reading_type, pct in report['data_profile']['missing_value_percentages'].items():
        report_df = pd.concat([report_df, pd.DataFrame([{
            'metric': f'missing_values_pct_{reading_type}',
            'value': pct
        }])], ignore_index=True)

    for reading_type, pct in report['data_profile']['anomaly_percentages'].items():
        report_df = pd.concat([report_df, pd.DataFrame([{
            'metric': f'anomaly_pct_{reading_type}',
            'value': pct
        }])], ignore_index=True)

    report_df.to_csv(self.config.quality_report_path, index=False)
    logger.info(f"Quality report saved to {self.config.quality_report_path}")

    return report
```

```python
class DataStorageEngine:
    """Handles data loading and storage optimization."""

    def __init__(self, config: PipelineConfig):
        self.config = config
        self.processed_path = Path(config.processed_data_path)
        self.processed_path.mkdir(parents=True, exist_ok=True)

    def optimize_for_analytics(self, df: pd.DataFrame) -> pd.DataFrame:
        """Optimize data for analytical queries."""
        logger.info("Optimizing data for analytics...")

        # Add partition columns
        df['timestamp_dt'] = pd.to_datetime(df['timestamp'])
        df['date'] = df['timestamp_dt'].dt.date
        df['year'] = df['timestamp_dt'].dt.year
        df['month'] = df['timestamp_dt'].dt.month
        df['day'] = df['timestamp_dt'].dt.day

        # Sort by timestamp for better compression
        df = df.sort_values(['timestamp_dt', 'sensor_id'])

        return df

    def save_partitioned_data(self, df: pd.DataFrame):
        """Save data in partitioned Parquet format."""
        logger.info("Saving partitioned data...")

        # Group by date for partitioning
        for date, group in df.groupby('date'):
            # Create date partition directory
            date_str = date.strftime('%Y-%m-%d')
            date_dir = self.processed_path / f"date={date_str}"
            date_dir.mkdir(exist_ok=True)

            # Further partition by sensor_id for large datasets
            for sensor_id, sensor_group in group.groupby('sensor_id'):
                filename = f"sensor_{sensor_id}.parquet"
                filepath = date_dir / filename

                # Save with compression
                sensor_group.to_parquet(
                    filepath,
                    engine='pyarrow',
                    compression='snappy',
                    index=False
                )

        logger.info(f"Data saved to {self.processed_path}")

    def create_summary_tables(self, df: pd.DataFrame):
        """Create summary tables for quick analytics."""
        logger.info("Creating summary tables...")
```

```python
        # Daily summary
        daily_summary = df.groupby(['date', 'sensor_id', 'reading_type']).agg({
            'value': ['mean', 'min', 'max', 'count'],
            'anomalous_reading': 'sum',
            'battery_level': 'mean'
        }).reset_index()

        daily_summary.columns = [
            'date', 'sensor_id', 'reading_type',
            'avg_value', 'min_value', 'max_value', 'record_count',
            'anomaly_count', 'avg_battery_level'
        ]

        daily_summary.to_parquet(
            self.processed_path / "daily_summary.parquet",
            engine='pyarrow',
            compression='snappy',
            index=False
        )

        # Sensor summary
        sensor_summary = df.groupby(['sensor_id', 'reading_type']).agg({
            'value': ['mean', 'std', 'min', 'max'],
            'anomalous_reading': 'sum',
            'timestamp_dt': ['min', 'max']
        }).reset_index()

        sensor_summary.columns = [
            'sensor_id', 'reading_type',
            'avg_value', 'std_value', 'min_value', 'max_value',
            'total_anomalies', 'first_reading', 'last_reading'
        ]

        sensor_summary.to_parquet(
            self.processed_path / "sensor_summary.parquet",
            engine='pyarrow',
            compression='snappy',
            index=False
        )

        logger.info("Summary tables created")


class AgricultureDataPipeline:
    """Main pipeline orchestrator."""

    def __init__(self, config: PipelineConfig):
        self.config = config
        self.ingestion_engine = DataIngestionEngine(config)
        self.transformation_engine = DataTransformationEngine(config)
        self.quality_validator = DataQualityValidator(config)
        self.storage_engine = DataStorageEngine(config)

    def run_pipeline(self, incremental: bool = True):
        """Run the complete data pipeline."""
```

```
        logger.info("Starting Agriculture Data Pipeline...")

        try:
            # Step 1: Data Ingestion
            logger.info("Step 1: Data Ingestion")
            raw_dataframes = self.ingestion_engine.ingest_batch()

            if not raw_dataframes:
                logger.warning("No data to process")
                return

            # Combine all dataframes
            combined_df = pd.concat(raw_dataframes, ignore_index=True)
            logger.info(f"Combined dataset size: {len(combined_df)} records")

            # Step 2: Data Transformation
            logger.info("Step 2: Data Transformation")
            transformed_df = self.transformation_engine.transform_data(combined_df)

            # Step 3: Data Quality Validation
            logger.info("Step 3: Data Quality Validation")
            quality_report = self.quality_validator.generate_quality_report(transformed_df)

            # Step 4: Data Storage
            logger.info("Step 4: Data Storage")
            optimized_df = self.storage_engine.optimize_for_analytics(transformed_df)
            self.storage_engine.save_partitioned_data(optimized_df)
            self.storage_engine.create_summary_tables(optimized_df)

            # Log final statistics
            logger.info("Pipeline completed successfully!")
            logger.info(f"Processed {len(optimized_df)} records")
            logger.info(f"Ingestion stats: {self.ingestion_engine.stats}")
            logger.info(f"Quality report saved to: {self.config.quality_report_path}")
            logger.info(f"Processed data saved to: {self.config.processed_data_path}")

            return {
                'status': 'success',
                'records_processed': len(optimized_df),
                'ingestion_stats': self.ingestion_engine.stats,
                'quality_report': quality_report
            }

        except Exception as e:
            logger.error(f"Pipeline failed: {e}")
            raise


def main():
    """Main entry point for the pipeline."""
    # Create configuration
    config = PipelineConfig()

    # Create required directories
    Path(config.raw_data_path).mkdir(parents=True, exist_ok=True)
```

```python
    Path(config.processed_data_path).mkdir(parents=True, exist_ok=True)

    # Initialize and run pipeline
    pipeline = AgricultureDataPipeline(config)

    try:
        result = pipeline.run_pipeline(incremental=True)
        print(f"Pipeline completed: {result}")
    except Exception as e:
        print(f"Pipeline failed: {e}")
        sys.exit(1)


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