Utils:

import logging

import os

from datetime import datetime

# Configure logging

logging.basicConfig(

    level=logging.INFO,

    format='%(asctime)s - %(name)s - %(levelname)s - %(message)s',

    handlers=[

        logging.StreamHandler(),

        logging.FileHandler('comparison\_framework.log')

    ]

)

logger = logging.getLogger(\_\_name\_\_)

def log\_error(error\_message: str) -> None:

    """

    Log error messages to both console and file.

    Args:

        error\_message (str): The error message to log

    """

    logger.error(error\_message)

def check\_file\_size(file) -> bool:

    """

    Check if file size exceeds 3GB threshold.

    Args:

        file: File object or path

    Returns:

        bool: True if file size is within limits, False otherwise

    """

    try:

        if hasattr(file, 'seek'):

            file.seek(0, os.SEEK\_END)

            size = file.tell()

            file.seek(0)  # Reset file pointer

        else:

            size = os.path.getsize(file)

        # 3GB in bytes

        GB\_3 = 3 \* 1024 \* 1024 \* 1024

        return size <= GB\_3

    except Exception as e:

        log\_error(f"Error checking file size: {str(e)}")

        return False

def format\_timestamp() -> str:

    """

    Get formatted timestamp for file naming.

    Returns:

        str: Formatted timestamp string

    """

    return datetime.now().strftime("%Y%m%d\_%H%M%S")

def validate\_connection\_params(params: dict) -> bool:

    """

    Validate database connection parameters.

    Args:

        params (dict): Dictionary containing connection parameters

    Returns:

        bool: True if parameters are valid, False otherwise

    """

    try:

        if not isinstance(params, dict):

            log\_error("Connection parameters must be a dictionary")

            return False

        # Check if using Windows Authentication

        if params.get('use\_windows\_auth', True):

            # For Windows Auth, we need either host or server, and database

            if not params.get('database'):

                log\_error("Database name is required")

                return False

            if not (params.get('host') or params.get('server')):

                log\_error("Server/Host is required")

                return False

            return True

        else:

            # For SQL Auth, we need host/server, database, username, and password

            required\_params = ['database', 'username', 'password']

            if not (params.get('host') or params.get('server')):

                log\_error("Server/Host is required")

                return False

            for param in required\_params:

                if not params.get(param):

                    log\_error(f"{param} is required for SQL Authentication")

                    return False

            return True

    except Exception as e:

        log\_error(f"Error validating connection parameters: {str(e)}")

        return False

def sanitize\_filename(filename: str) -> str:

    """

    Sanitize filename by removing invalid characters.

    Args:

        filename (str): Original filename

    Returns:

        str: Sanitized filename

    """

    # Replace invalid characters with underscore

    invalid\_chars = '<>:"/\\|?\*'

    for char in invalid\_chars:

        filename = filename.replace(char, '\_')

    return filename

def get\_file\_extension(filename: str) -> str:

    """

    Get file extension from filename.

    Args:

        filename (str): Name of the file

    Returns:

        str: File extension without dot

    """

    return os.path.splitext(filename)[1][1:].lower()

def clean\_df\_columns(df) -> 'pd.DataFrame':

    """

    Returns a copy of the DataFrame with whitespace trimmed from all column names.

    Args:

        df (pd.DataFrame): Input DataFrame.

    Returns:

        pd.DataFrame: DataFrame with cleaned column names.

    """

    try:

        df\_copy = df.copy()

        df\_copy.columns = df\_copy.columns.str.strip()

        return df\_copy

    except Exception as e:

        log\_error(f"Error cleaning DataFrame columns: {str(e)}")

        return df  # Return original DataFrame if cleaning fails

db\_connector

import pandas as pd

from sqlalchemy import create\_engine, text

from typing import Dict, Union, Optional, Any

import pyodbc

import teradatasql

from utils import log\_error, validate\_connection\_params, clean\_df\_columns

class DatabaseConnector:

    """Class to handle database connections and data retrieval"""

    @staticmethod

    def get\_sqlserver\_data(connection\_params: Dict[str, str], query: str) -> Optional[pd.DataFrame]:

        """

        Retrieve data from SQL Server using provided connection parameters and query.

        Args:

            connection\_params: Dictionary containing connection parameters

            query: SQL query to execute

        Returns:

            Optional[pd.DataFrame]: DataFrame containing query results, None if error occurs

        """

        if not validate\_connection\_params(connection\_params):

            log\_error("Invalid connection parameters for SQL Server")

            return None

        try:

            # Get server name from either 'server' or 'host' parameter

            server = connection\_params.get('server') or connection\_params.get('host')

            if not server:

                raise ValueError("Server/Host is required")

            database = connection\_params.get('database')

            if not database:

                raise ValueError("Database name is required")

            # Build connection string based on authentication type

            if connection\_params.get('use\_windows\_auth', True):

                conn\_str = (

                    f"Driver={{ODBC Driver 17 for SQL Server}};"

                    f"Server={server};"

                    f"Database={database};"

                    "Trusted\_Connection=yes"

                )

            else:

                username = connection\_params.get('username')

                password = connection\_params.get('password')

                if not username or not password:

                    raise ValueError("Username and password are required for SQL authentication")

                conn\_str = (

                    f"Driver={{ODBC Driver 17 for SQL Server}};"

                    f"Server={server};"

                    f"Database={database};"

                    f"Uid={username};"

                    f"Pwd={password}"

                )

            # Create engine using the pyodbc connection string

            # Format: mssql+pyodbc://username:password@server/database?driver=ODBC+Driver+17+for+SQL+Server

            if connection\_params.get('use\_windows\_auth', True):

                engine = create\_engine(

                    f"mssql+pyodbc://{server}/{database}?driver=ODBC+Driver+17+for+SQL+Server&trusted\_connection=yes",

                    fast\_executemany=True

                )

            else:

                engine = create\_engine(

                    f"mssql+pyodbc://{username}:{password}@{server}/{database}?driver=ODBC+Driver+17+for+SQL+Server",

                    fast\_executemany=True

                )

            # Connect and execute query

            with engine.connect() as connection:

                df = pd.read\_sql(text(query), connection)

                df = clean\_df\_columns(df)  # Clean column names for consistency

                return df

        except Exception as e:

            log\_error(f"Error retrieving data from SQL Server: {str(e)}")

            return None

    @staticmethod

    def get\_teradata\_data(connection\_params: Dict[str, str], query: str) -> Optional[pd.DataFrame]:

        """

        Retrieve data from Teradata using provided connection parameters and query.

        Args:

            connection\_params: Dictionary containing connection parameters

            query: SQL query to execute

        Returns:

            Optional[pd.DataFrame]: DataFrame containing query results, None if error occurs

        """

        try:

            if not validate\_connection\_params(connection\_params):

                log\_error("Invalid connection parameters for Teradata")

                return None

            # Create Teradata connection

            conn = teradatasql.connect(

                host=connection\_params['host'],

                user=connection\_params['username'],

                password=connection\_params['password'],

                database=connection\_params['database']

            )

            # Execute query and return results as DataFrame

            df = pd.read\_sql(query, conn)

            conn.close()

            return df

        except Exception as e:

            log\_error(f"Error retrieving data from Teradata: {str(e)}")

            return None

    @staticmethod

    def get\_data\_from\_stored\_proc(connection\_params: Dict[str, str],

                                 proc\_name: str,

                                 params: Optional[Dict[str, Any]] = None) -> Optional[pd.DataFrame]:

        """

        Execute a stored procedure and retrieve its results.

        Args:

            connection\_params: Dictionary containing connection parameters

            proc\_name: Name of the stored procedure to execute

            params: Optional dictionary of parameters for the stored procedure

        Returns:

            Optional[pd.DataFrame]: DataFrame containing procedure results, None if error occurs

        """

        if not validate\_connection\_params(connection\_params):

            log\_error("Invalid connection parameters for stored procedure execution")

            return None

        try:

            # Get server name from either 'server' or 'host' parameter

            server = connection\_params.get('server') or connection\_params.get('host')

            if not server:

                raise ValueError("Server/Host is required")

            database = connection\_params.get('database')

            if not database:

                raise ValueError("Database name is required")

            # Build connection string

            if connection\_params.get('use\_windows\_auth', True):

                conn\_str = f"DRIVER={{ODBC Driver 17 for SQL Server}};SERVER={server};DATABASE={database};Trusted\_Connection=yes"

            else:

                username = connection\_params.get('username')

                password = connection\_params.get('password')

                if not username or not password:

                    raise ValueError("Username and password are required for SQL authentication")

                conn\_str = f"DRIVER={{ODBC Driver 17 for SQL Server}};SERVER={server};DATABASE={database};UID={username};PWD={password}"

            # Connect and execute stored procedure using pyodbc

            with pyodbc.connect(conn\_str, timeout=30) as conn:

                cursor = conn.cursor()

                if params:

                    # Build parameter string

                    param\_str = ', '.join([f"@{k}=?" for k in params.keys()])

                    exec\_str = f"EXEC {proc\_name} {param\_str}"

                    cursor.execute(exec\_str, list(params.values()))

                else:

                    cursor.execute(f"EXEC {proc\_name}")

                # Fetch results

                columns = [column[0] for column in cursor.description]

                results = cursor.fetchall()

                # Convert to DataFrame and clean column names

                df = pd.DataFrame.from\_records(results, columns=columns)

                df = clean\_df\_columns(df)  # Clean column names for consistency

                return df

        except Exception as e:

            log\_error(f"Error executing stored procedure: {str(e)}")

            return None

    @staticmethod

    def test\_connection(connection\_params: Dict[str, str], db\_type: str) -> bool:

        """

        Test database connection without executing any queries.

        Args:

            connection\_params: Dictionary containing connection parameters

            db\_type: Type of database ('sqlserver' or 'teradata')

        Returns:

            bool: True if connection successful, False otherwise

        """

        try:

            if not validate\_connection\_params(connection\_params):

                return False

            if db\_type.lower() == 'sqlserver':

                try:

                    # Get server name from either 'server' or 'host' parameter

                    server = connection\_params.get('server') or connection\_params.get('host')

                    if not server:

                        raise ValueError("Server/Host is required")

                    database = connection\_params.get('database')

                    if not database:

                        raise ValueError("Database name is required")

                    # Build connection string

                    if connection\_params.get('use\_windows\_auth', True):

                        conn\_str = f"DRIVER={{ODBC Driver 17 for SQL Server}};SERVER={server};DATABASE={database};Trusted\_Connection=yes"

                    else:

                        username = connection\_params.get('username')

                        password = connection\_params.get('password')

                        if not username or not password:

                            raise ValueError("Username and password are required for SQL authentication")

                        conn\_str = f"DRIVER={{ODBC Driver 17 for SQL Server}};SERVER={server};DATABASE={database};UID={username};PWD={password}"

                    # Test connection with timeout

                    conn = pyodbc.connect(conn\_str, timeout=30)

                    conn.close()

                    return True

                except Exception as e:

                    log\_error(f"SQL Server connection test failed: {str(e)}")

                    return False

            elif db\_type.lower() == 'teradata':

                conn = teradatasql.connect(

                    host=connection\_params['host'],

                    user=connection\_params['username'],

                    password=connection\_params['password'],

                    database=connection\_params['database']

                )

                conn.close()

                return True

            else:

                log\_error(f"Unsupported database type: {db\_type}")

                return False

        except Exception as e:

            log\_error(f"Error testing connection: {str(e)}")

            return False

    @staticmethod

    def get\_table\_schema(connection\_params: Dict[str, str],

                        table\_name: str,

                        db\_type: str) -> Optional[pd.DataFrame]:

        """

        Retrieve schema information for a specified table.

        Args:

            connection\_params: Dictionary containing connection parameters

            table\_name: Name of the table

            db\_type: Type of database ('sqlserver' or 'teradata')

        Returns:

            Optional[pd.DataFrame]: DataFrame containing schema information, None if error occurs

        """

        try:

            if db\_type.lower() == 'sqlserver':

                query = f"""

                SELECT

                    COLUMN\_NAME,

                    DATA\_TYPE,

                    CHARACTER\_MAXIMUM\_LENGTH,

                    IS\_NULLABLE

                FROM INFORMATION\_SCHEMA.COLUMNS

                WHERE TABLE\_NAME = '{table\_name}'

                ORDER BY ORDINAL\_POSITION

                """

                return DatabaseConnector.get\_sqlserver\_data(connection\_params, query)

            elif db\_type.lower() == 'teradata':

                query = f"""

                SELECT

                    ColumnName,

                    ColumnType,

                    CharacterLength,

                    Nullable

                FROM DBC.Columns

                WHERE TableName = '{table\_name}'

                ORDER BY ColumnId

                """

                return DatabaseConnector.get\_teradata\_data(connection\_params, query)

            else:

                log\_error(f"Unsupported database type: {db\_type}")

                return None

        except Exception as e:

            log\_error(f"Error retrieving table schema: {str(e)}")

            return None

mapping\_manager.py

import pandas as pd

from typing import Dict, List, Tuple, Any, Optional

from utils import log\_error

class MappingManager:

    """Class to handle column mappings and data type conversions"""

    # Define the type mapping dictionary as a class variable

    TYPE\_MAPPING = {

        'int': 'int32',

        'int64': 'int64',

        'numeric': 'int64',

        'bigint': 'int64',

        'smallint': 'int64',

        'varchar': 'string',

        'nvarchar': 'string',

        'char': 'string',

        'date': 'datetime64[ns]',

        'datetime': 'datetime64[ns]',

        'decimal': 'float',

        'float': 'float',

        'bit': 'bool',

        'nchar': 'char',

        'boolean': 'bool'

    }

    @staticmethod

    def normalize\_column\_name(col: str) -> str:

        """

        Normalize column name by converting to lowercase, removing spaces and special characters.

        Args:

            col (str): Column name to normalize

        Returns:

            str: Normalized column name

        """

        # Convert to lowercase and strip whitespace

        normalized = col.lower().strip()

        # Remove special characters and spaces, keep only alphanumeric

        normalized = ''.join(e for e in normalized if e.isalnum())

        return normalized

    @classmethod

    def auto\_map\_columns(cls, source\_df: pd.DataFrame, target\_df: pd.DataFrame) -> Dict[str, str]:

        """

        Automatically map columns between source and target DataFrames.

        Args:

            source\_df: Source DataFrame

            target\_df: Target DataFrame

        Returns:

            dict: Mapping of source columns to target columns

        """

        try:

            mapping = {}

            # Get column names

            source\_cols = source\_df.columns

            target\_cols = target\_df.columns

            # Create normalized versions of column names

            normalized\_source = {col: cls.normalize\_column\_name(col) for col in source\_cols}

            normalized\_target = {col: cls.normalize\_column\_name(col) for col in target\_cols}

            # First pass: exact matches based on normalized names

            for source\_col, norm\_s in normalized\_source.items():

                for target\_col, norm\_t in normalized\_target.items():

                    if norm\_s == norm\_t:

                        mapping[source\_col] = target\_col

                        break

            # Second pass: fuzzy matches based on column names and data patterns

            unmapped\_source = [col for col in source\_cols if col not in mapping]

            unmapped\_target = [col for col in target\_cols if col not in mapping.values()]

            for source\_col in unmapped\_source:

                best\_match = None

                best\_score = 0

                # Get sample values and normalized name from source column

                source\_sample = source\_df[source\_col].head(5).astype(str).tolist()

                source\_dtype = str(source\_df[source\_col].dtype)

                norm\_source = normalized\_source[source\_col]

                for target\_col in unmapped\_target:

                    try:

                        # Compare data types

                        target\_dtype = str(target\_df[target\_col].dtype)

                        type\_match = source\_dtype == target\_dtype

                        # Compare normalized column names using fuzzy matching

                        norm\_target = normalized\_target[target\_col]

                        name\_score = cls.\_calculate\_similarity(norm\_source, norm\_target)

                        # Compare data patterns

                        target\_sample = target\_df[target\_col].head(5).astype(str).tolist()

                        data\_score = cls.\_calculate\_data\_similarity(source\_sample, target\_sample)

                        # Calculate combined score with adjusted weights

                        # Increased weight for name similarity and type matching

                        combined\_score = (name\_score \* 0.7) + (data\_score \* 0.3)

                        if type\_match:

                            combined\_score += 0.15  # Increased bonus for type match

                        # Increased threshold for more accurate matching

                        if combined\_score > best\_score and combined\_score > 0.75:  # 75% similarity threshold

                            best\_score = combined\_score

                            best\_match = target\_col

                    except Exception as e:

                        log\_error(f"Error comparing columns {source\_col} and {target\_col}: {str(e)}")

                        continue

                if best\_match:

                    mapping[source\_col] = best\_match

                    unmapped\_target.remove(best\_match)

            # Log mapping results

            if not mapping:

                log\_error("No column mappings found between source and target DataFrames")

            else:

                print(f"Successfully mapped {len(mapping)} columns")

            return mapping

        except Exception as e:

            log\_error(f"Error in auto\_map\_columns: {str(e)}")

            return {}

    @staticmethod

    def \_calculate\_similarity(str1: str, str2: str) -> float:

        """Calculate similarity between two strings using Levenshtein distance."""

        if not str1 or not str2:

            return 0

        # Simple implementation of Levenshtein distance

        if len(str1) < len(str2):

            str1, str2 = str2, str1

        if not str2:

            return 0

        previous\_row = range(len(str2) + 1)

        for i, c1 in enumerate(str1):

            current\_row = [i + 1]

            for j, c2 in enumerate(str2):

                insertions = previous\_row[j + 1] + 1

                deletions = current\_row[j] + 1

                substitutions = previous\_row[j] + (c1 != c2)

                current\_row.append(min(insertions, deletions, substitutions))

            previous\_row = current\_row

        # Convert distance to similarity score

        max\_len = max(len(str1), len(str2))

        similarity = 1 - (previous\_row[-1] / max\_len)

        return similarity

    @staticmethod

    def \_calculate\_data\_similarity(source\_sample: List[str], target\_sample: List[str]) -> float:

        """Calculate similarity between two lists of data samples."""

        if not source\_sample or not target\_sample:

            return 0

        # Convert all values to lowercase strings for comparison

        source\_values = [str(val).lower() for val in source\_sample]

        target\_values = [str(val).lower() for val in target\_sample]

        # Calculate pattern similarity

        total\_score = 0

        comparisons = 0

        # Compare data patterns

        for s\_val in source\_values:

            best\_match = 0

            for t\_val in target\_values:

                # Check for exact matches

                if s\_val == t\_val:

                    best\_match = 1

                    break

                # Check for numeric patterns

                s\_numeric = ''.join(c for c in s\_val if c.isdigit())

                t\_numeric = ''.join(c for c in t\_val if c.isdigit())

                if s\_numeric and t\_numeric:

                    if s\_numeric == t\_numeric:

                        best\_match = max(best\_match, 0.9)

                        continue

                # Check for string patterns

                s\_alpha = ''.join(c for c in s\_val if c.isalpha())

                t\_alpha = ''.join(c for c in t\_val if c.isalpha())

                if s\_alpha and t\_alpha:

                    alpha\_sim = MappingManager.\_calculate\_similarity(s\_alpha, t\_alpha)

                    best\_match = max(best\_match, alpha\_sim \* 0.8)

            total\_score += best\_match

            comparisons += 1

        return total\_score / comparisons if comparisons > 0 else 0

    @classmethod

    def generate\_data\_type\_mapping(cls, df: pd.DataFrame) -> Dict[str, str]:

        """

        Generate data type mapping for DataFrame columns.

        Args:

            df: Input DataFrame

        Returns:

            dict: Mapping of column names to suggested data types

        """

        type\_mapping = {}

        for column in df.columns:

            current\_type = str(df[column].dtype)

            # Map pandas/numpy types to our type system

            if 'int' in current\_type:

                type\_mapping[column] = 'int64'

            elif 'float' in current\_type:

                type\_mapping[column] = 'float'

            elif 'datetime' in current\_type:

                type\_mapping[column] = 'datetime64[ns]'

            elif 'bool' in current\_type:

                type\_mapping[column] = 'bool'

            else:

                type\_mapping[column] = 'string'

        return type\_mapping

    @classmethod

    def apply\_mapping(cls, df: pd.DataFrame, type\_mapping: Dict[str, str]) -> pd.DataFrame:

        """

        Apply data type mapping to DataFrame.

        Args:

            df: Input DataFrame

            type\_mapping: Dictionary mapping column names to desired data types

        Returns:

            pd.DataFrame: DataFrame with applied type conversions

        """

        try:

            df\_copy = df.copy()

            for column, dtype in type\_mapping.items():

                if column in df\_copy.columns:

                    try:

                        if dtype == 'datetime64[ns]':

                            df\_copy[column] = pd.to\_datetime(df\_copy[column], errors='coerce')

                        else:

                            df\_copy[column] = df\_copy[column].astype(dtype)

                    except Exception as e:

                        log\_error(f"Error converting column {column} to type {dtype}: {str(e)}")

            return df\_copy

        except Exception as e:

            log\_error(f"Error applying type mapping: {str(e)}")

            return df

    @staticmethod

    def validate\_mapping(source\_df: pd.DataFrame, target\_df: pd.DataFrame,

                        column\_mapping: Dict[str, str]) -> Tuple[bool, List[str]]:

        """

        Validate column mapping between source and target DataFrames.

        Args:

            source\_df: Source DataFrame

            target\_df: Target DataFrame

            column\_mapping: Dictionary mapping source columns to target columns

        Returns:

            tuple: (is\_valid, list of validation messages)

        """

        messages = []

        is\_valid = True

        # Check if all mapped columns exist

        for source\_col, target\_col in column\_mapping.items():

            if source\_col not in source\_df.columns:

                messages.append(f"Source column '{source\_col}' not found")

                is\_valid = False

            if target\_col not in target\_df.columns:

                messages.append(f"Target column '{target\_col}' not found")

                is\_valid = False

        # Check for duplicate mappings

        if len(set(column\_mapping.values())) != len(column\_mapping.values()):

            messages.append("Duplicate target columns in mapping")

            is\_valid = False

        return is\_valid, messages

    @staticmethod

    def get\_unmapped\_columns(source\_df: pd.DataFrame, target\_df: pd.DataFrame,

                            column\_mapping: Dict[str, str]) -> Tuple[List[str], List[str]]:

        """

        Get lists of unmapped columns from both source and target DataFrames.

        Args:

            source\_df: Source DataFrame

            target\_df: Target DataFrame

            column\_mapping: Dictionary mapping source columns to target columns

        Returns:

            tuple: (unmapped source columns, unmapped target columns)

        """

        unmapped\_source = [col for col in source\_df.columns if col not in column\_mapping]

        unmapped\_target = [col for col in target\_df.columns if col not in column\_mapping.values()]

        return unmapped\_source, unmapped\_target

app.py

import streamlit as st

import pandas as pd

from typing import Dict, List, Tuple, Optional, Any

import os

from datetime import datetime

import datacompy

from ydata\_profiling import ProfileReport

import streamlit.components.v1 as components

# Import our modules

from data\_reader import DataReader

from mapping\_manager import MappingManager

from report\_generator import ReportGenerator

from db\_connector import DatabaseConnector

from api\_fetcher import APIFetcher

from utils import log\_error, format\_timestamp

# Set page config

st.set\_page\_config(

    page\_title="Data Comparison Framework",

    page\_icon="📊",

    layout="wide"

)

# Define source types

SOURCE\_TYPES = [

    "CSV File",

    "DAT File",

    "SQL Server",

    "Stored Procedure",

    "Teradata",

    "API",

    "Parquet File",

    "Zipped Flat Files"

]

def main():

    """Main application function"""

    # Add modern header with enhanced styling

    st.markdown(

        """

        <div style='text-align: center; background-color: #f8f9fa; padding: 2rem;

             border-radius: 10px; margin-bottom: 2rem; border: 1px solid #dee2e6;'>

            <h1 style='color: #1f77b4; margin-bottom: 0.5rem; font-size: 2.5em;'>

                Data Comparison Framework

            </h1>

            <p style='color: #6c757d; font-size: 1.1em; margin: 1rem 0;'>

                Compare data between SQL Server databases and feed files with advanced mapping capabilities

            </p>

            <div style='display: flex; justify-content: center; gap: 1rem; margin-top: 1rem;'>

                <div style='background-color: #e8f4f8; padding: 0.5rem 1rem; border-radius: 5px;'>

                    <span style='color: #1f77b4;'>✨ Automatic Column Mapping</span>

                </div>

                <div style='background-color: #e8f4f8; padding: 0.5rem 1rem; border-radius: 5px;'>

                    <span style='color: #1f77b4;'>🔍 Smart Join Detection</span>

                </div>

                <div style='background-color: #e8f4f8; padding: 0.5rem 1rem; border-radius: 5px;'>

                    <span style='color: #1f77b4;'>📊 Detailed Reports</span>

                </div>

            </div>

        </div>

        """,

        unsafe\_allow\_html=True

    )

    # Add error handling for the entire app

    try:

        # Initialize session state variables

        if 'source\_df' not in st.session\_state:

            st.session\_state.source\_df = None

        if 'target\_df' not in st.session\_state:

            st.session\_state.target\_df = None

        if 'column\_mapping' not in st.session\_state:

            st.session\_state.column\_mapping = {}

        if 'excluded\_columns' not in st.session\_state:

            st.session\_state.excluded\_columns = []

        if 'report\_paths' not in st.session\_state:

            st.session\_state.report\_paths = {}

        # Create two columns for source and target selection with enhanced styling

        st.markdown(

            """

            <div style='display: flex; gap: 2rem; margin: 1rem 0;'>

                <div style='flex: 1; background-color: #f8f9fa; padding: 1.5rem; border-radius: 10px; border: 1px solid #dee2e6;'>

                    <h3 style='margin: 0 0 1rem 0; color: #1f77b4;'>Source Configuration</h3>

                    <p style='color: #6c757d; margin-bottom: 1rem;'>Select and configure your source data</p>

                </div>

                <div style='flex: 1; background-color: #f8f9fa; padding: 1.5rem; border-radius: 10px; border: 1px solid #dee2e6;'>

                    <h3 style='margin: 0 0 1rem 0; color: #1f77b4;'>Target Configuration</h3>

                    <p style='color: #6c757d; margin-bottom: 1rem;'>Select and configure your target data</p>

                </div>

            </div>

            """,

            unsafe\_allow\_html=True

        )

        col1, col2 = st.columns(2)

        with col1:

            source\_type = st.selectbox(

                "Select Source Type",

                SOURCE\_TYPES,

                key="source\_type",

                help="Choose the type of source data you want to compare"

            )

            source\_data = handle\_data\_source(source\_type, "source")

            if source\_data is not None:

                st.success(f"✅ Source data loaded successfully: {len(source\_data)} rows, {len(source\_data.columns)} columns")

        with col2:

            target\_type = st.selectbox(

                "Select Target Type",

                SOURCE\_TYPES,

                key="target\_type",

                help="Choose the type of target data you want to compare"

            )

            target\_data = handle\_data\_source(target\_type, "target")

            if target\_data is not None:

                st.success(f"✅ Target data loaded successfully: {len(target\_data)} rows, {len(target\_data.columns)} columns")

    except Exception as e:

        st.error(f"❌ An error occurred: {str(e)}")

        log\_error(f"Application error: {str(e)}")

        return

    # If both source and target data are loaded

    if source\_data is not None and target\_data is not None:

        try:

            st.session\_state.source\_df = source\_data

            st.session\_state.target\_df = target\_data

            # Force re-mapping if source columns have changed

            current\_source\_cols = list(source\_data.columns)

            if (not st.session\_state.get('column\_mapping') or

                st.session\_state.get('last\_source\_columns') != current\_source\_cols):

                st.session\_state.column\_mapping = MappingManager.auto\_map\_columns(source\_data, target\_data)

                st.session\_state.last\_source\_columns = current\_source\_cols

                if st.session\_state.column\_mapping:

                    st.success(f"✅ Successfully mapped {len(st.session\_state.column\_mapping)} columns automatically!")

                else:

                    st.warning("⚠️ No automatic column mappings found. Please map columns manually below.")

        # Show column mapping interface

        st.subheader("Column Mapping")

        show\_column\_mapping\_interface(source\_data, target\_data)

        # Add join key selection with enhanced styling

        st.markdown(

            """

            <div style='background-color: #f8f9fa; padding: 1.5rem; border-radius: 10px; margin: 2rem 0; border: 1px solid #dee2e6;'>

                <h3 style='margin: 0; color: #1f77b4;'>Join Key Selection</h3>

                <p style='margin: 0.5rem 0 0 0; color: #6c757d;'>

                    Select columns to use as join keys for comparing records between source and target data.

                </p>

            </div>

            """,

            unsafe\_allow\_html=True

        )

        # Get mapped columns (where source and target are mapped)

        mapped\_columns = {source\_col: target\_col

                        for source\_col, target\_col in st.session\_state.column\_mapping.items()

                        if source\_col in source\_data.columns and target\_col in target\_data.columns}

        if not mapped\_columns:

            st.warning("⚠️ No mapped columns available for join keys. Please map at least one column first.")

            join\_keys = None

        else:

            # Create columns for join key selection and preview

            col1, col2 = st.columns([1, 2])

            with col1:

                # Select join keys from mapped columns with enhanced styling

                st.markdown("##### Select Join Key Columns")

                selected\_keys = st.multiselect(

                    "Choose columns that uniquely identify records",

                    options=list(mapped\_columns.keys()),

                    help="These columns will be used to match records between source and target data"

                )

                if selected\_keys:

                    join\_keys = [(source\_col, mapped\_columns[source\_col]) for source\_col in selected\_keys]

                    st.markdown(

                        """

                        <div style='background-color: #e8f4f8; padding: 0.75rem; border-radius: 5px; margin-top: 1rem;'>

                            <p style='margin: 0; color: #1f77b4;'>Selected Join Keys:</p>

                        </div>

                        """,

                        unsafe\_allow\_html=True

                    )

                    for source\_col, target\_col in join\_keys:

                        st.markdown(f"- {source\_col} → {target\_col}")

                else:

                    st.info("ℹ️ No join keys selected. Comparison will be done row by row.")

                    join\_keys = None

            with col2:

                # Show preview of mapped columns with samples

                st.markdown("##### Join Keys Preview")

                mapped\_preview = []

                for source\_col, target\_col in mapped\_columns.items():

                    source\_sample = source\_data[source\_col].head(3).tolist()

                    target\_sample = target\_data[target\_col].head(3).tolist()

                    mapped\_preview.append({

                        'Source Column': source\_col,

                        'Target Column': target\_col,

                        'Source Sample': str(source\_sample),

                        'Target Sample': str(target\_sample)

                    })

                if mapped\_preview:

                    st.dataframe(

                        pd.DataFrame(mapped\_preview),

                        use\_container\_width=True,

                        height=200

                    )

        # Store join keys in session state

        st.session\_state.join\_keys = join\_keys

        # Compare button section with enhanced styling

        st.markdown(

            """

            <div style='background-color: #f8f9fa; padding: 1.5rem; border-radius: 10px; margin: 2rem 0;

                 border: 1px solid #dee2e6; text-align: center;'>

                <h3 style='margin: 0 0 1rem 0; color: #1f77b4;'>Generate Comparison Reports</h3>

            </div>

            """,

            unsafe\_allow\_html=True

        )

        # Center the compare button

        col1, col2, col3 = st.columns([2, 1, 2])

        with col2:

            if st.button("🔄 Compare Data", type="primary", use\_container\_width=True):

                with st.spinner("Generating comparison reports..."):

                    perform\_comparison()

def handle\_data\_source(source\_type: str, prefix: str) -> Optional[pd.DataFrame]:

    """Handle different types of data sources"""

    try:

        if source\_type in ["CSV File", "DAT File", "Parquet File", "Zipped Flat Files"]:

            return handle\_file\_upload(source\_type, prefix)

        elif source\_type in ["SQL Server", "Teradata", "Stored Procedure"]:

            return handle\_database\_connection(source\_type, prefix)

        elif source\_type == "API":

            return handle\_api\_connection(prefix)

    except Exception as e:

        log\_error(f"Error handling {source\_type}: {str(e)}")

        st.error(f"Error processing {source\_type}: {str(e)}")

    return None

def handle\_file\_upload(file\_type: str, prefix: str) -> Optional[pd.DataFrame]:

    """Handle file upload for different file types with enhanced UI and error handling"""

    # Add file upload UI with modern styling

    st.markdown(

        f"""

        <div style='background-color: #f8f9fa; padding: 1.5rem; border-radius: 10px; margin-bottom: 1rem; border: 1px solid #dee2e6;'>

            <h4 style='margin: 0; color: #1f77b4;'>{file\_type} Upload</h4>

            <p style='margin: 0.5rem 0 0 0; color: #6c757d;'>

                Upload your {file\_type.lower()} and configure import settings below.

            </p>

        </div>

        """,

        unsafe\_allow\_html=True

    )

    # File upload section

    uploaded\_file = st.file\_uploader(

        f"Choose {file\_type}",

        key=f"{prefix}\_file",

        help=f"Select a {file\_type.lower()} to upload"

    )

    if uploaded\_file:

        try:

            # Show file info

            file\_details = {

                "Filename": uploaded\_file.name,

                "File size": f"{uploaded\_file.size / 1024:.2f} KB",

                "File type": uploaded\_file.type

            }

            st.markdown("##### File Details")

            for key, value in file\_details.items():

                st.text(f"{key}: {value}")

            # Configuration options based on file type

            with st.expander("Import Configuration", expanded=True):

                if file\_type in ["CSV File", "DAT File"]:

                    col1, col2 = st.columns(2)

                    with col1:

                        delimiter = st.text\_input(

                            "Delimiter",

                            value=',' if file\_type == "CSV File" else '|',

                            key=f"{prefix}\_delimiter",

                            help="Enter the character used to separate columns"

                        )

                    with col2:

                        encoding = st.selectbox(

                            "File Encoding",

                            ["utf-8", "latin1", "ascii"],

                            key=f"{prefix}\_encoding",

                            help="Select the file encoding"

                        )

            # Load the file with progress indicator

            with st.spinner(f"Loading {file\_type}..."):

                if file\_type == "CSV File":

                    df = DataReader.load\_csv(uploaded\_file, delimiter=delimiter)

                elif file\_type == "DAT File":

                    df = DataReader.load\_dat(uploaded\_file, delimiter=delimiter)

                elif file\_type == "Parquet File":

                    df = DataReader.load\_parquet(uploaded\_file)

                elif file\_type == "Zipped Flat Files":

                    df = DataReader.load\_zipped\_flat\_files(uploaded\_file, separator=delimiter)

                if df is not None:

                    st.success(f"✅ {file\_type} loaded successfully!")

                    st.info(f"Retrieved {len(df)} rows and {len(df.columns)} columns")

                    # Show data preview in an expander

                    with st.expander("Data Preview"):

                        st.dataframe(

                            df.head(5),

                            use\_container\_width=True,

                            height=200

                        )

                    return df

                else:

                    st.error(f"❌ Failed to load {file\_type}. Please check the file format and try again.")

        except Exception as e:

            st.error(f"❌ Error reading {file\_type}: {str(e)}")

            log\_error(f"File upload error ({file\_type}): {str(e)}")

            # Show detailed error message in an expander

            with st.expander("Error Details"):

                st.code(str(e))

    return None

def handle\_database\_connection(db\_type: str, prefix: str) -> Optional[pd.DataFrame]:

    """Handle database connections with enhanced UI and error handling"""

    # Add database connection UI with modern styling

    st.markdown(

        f"""

        <div style='background-color: #f8f9fa; padding: 1.5rem; border-radius: 10px; margin-bottom: 1rem; border: 1px solid #dee2e6;'>

            <h4 style='margin: 0; color: #1f77b4;'>{db\_type} Connection</h4>

            <p style='margin: 0.5rem 0 0 0; color: #6c757d;'>

                Enter your {db\_type} connection details below. Column names will be automatically standardized for better mapping.

            </p>

        </div>

        """,

        unsafe\_allow\_html=True

    )

    # Create tabs for connection and query

    conn\_tab, query\_tab = st.tabs(["Connection Details", "Query Configuration"])

    with conn\_tab:

        # Connection details with improved layout

        col1, col2 = st.columns(2)

        with col1:

            host = st.text\_input(

                "Server/Host",

                key=f"{prefix}\_host",

                help="Enter the server name or IP address"

            )

        with col2:

            database = st.text\_input(

                "Database",

                key=f"{prefix}\_database",

                help="Enter the database name"

            )

        # Authentication section

        st.markdown("##### Authentication")

        if db\_type in ["SQL Server", "Stored Procedure"]:

            use\_windows\_auth = st.checkbox(

                "Use Windows Authentication",

                value=True,

                key=f"{prefix}\_use\_windows\_auth",

                help="Check to use Windows Authentication, uncheck for SQL Authentication"

            )

            if not use\_windows\_auth:

                col1, col2 = st.columns(2)

                with col1:

                    username = st.text\_input(

                        "Username",

                        key=f"{prefix}\_username",

                        help="Enter your SQL Server username"

                    )

                with col2:

                    password = st.text\_input(

                        "Password",

                        type="password",

                        key=f"{prefix}\_password",

                        help="Enter your SQL Server password"

                    )

        else:

            use\_windows\_auth = False

            col1, col2 = st.columns(2)

            with col1:

                username = st.text\_input(

                    "Username",

                    key=f"{prefix}\_username"

                )

            with col2:

                password = st.text\_input(

                    "Password",

                    type="password",

                    key=f"{prefix}\_password"

                )

    with query\_tab:

        if db\_type == "Stored Procedure":

            proc\_name = st.text\_input(

                "Stored Procedure Name",

                key=f"{prefix}\_proc",

                help="Enter the name of the stored procedure"

            )

            params = st.text\_area(

                "Parameters (JSON format)",

                key=f"{prefix}\_params",

                help="Enter parameters as JSON, e.g., {\"param1\": \"value1\"}"

            )

        else:

            query = st.text\_area(

                "SQL Query",

                key=f"{prefix}\_query",

                height=150,

                help="Enter your SQL query here. The query will be executed to fetch the data."

            )

    # Connection button with loading state

    if st.button("🔌 Connect to Database", key=f"{prefix}\_connect", use\_container\_width=True):

        with st.spinner(f"Connecting to {db\_type}..."):

            try:

                # Build connection parameters

                conn\_params = {

                    'host': host,

                    'database': database,

                    'use\_windows\_auth': use\_windows\_auth if db\_type in ["SQL Server", "Stored Procedure"] else False

                }

                # Add username/password only if not using Windows Auth

                if not (db\_type in ["SQL Server", "Stored Procedure"] and use\_windows\_auth):

                    if not username or not password:

                        st.error("❌ Username and password are required for SQL Authentication")

                        return None

                    conn\_params.update({

                        'username': username,

                        'password': password

                    })

                # Validate connection parameters

                if not host or not database:

                    st.error("❌ Server/Host and Database are required")

                    return None

                # Execute query based on database type

                if db\_type == "SQL Server":

                    if not query:

                        st.error("❌ SQL Query is required")

                        return None

                    df = DatabaseConnector.get\_sqlserver\_data(conn\_params, query)

                    if df is not None:

                        st.success("✅ SQL Server connection successful!")

                        st.info(f"Retrieved {len(df)} rows and {len(df.columns)} columns")

                    return df

                elif db\_type == "Teradata":

                    if not query:

                        st.error("❌ SQL Query is required")

                        return None

                    return DatabaseConnector.get\_teradata\_data(conn\_params, query)

                elif db\_type == "Stored Procedure":

                    if not proc\_name:

                        st.error("❌ Stored Procedure name is required")

                        return None

                    df = DatabaseConnector.get\_data\_from\_stored\_proc(

                        conn\_params, proc\_name, eval(params) if params else None)

                    if df is not None:

                        st.success("✅ Stored Procedure executed successfully!")

                        st.info(f"Retrieved {len(df)} rows and {len(df.columns)} columns")

                    return df

            except Exception as e:

                st.error(f"❌ Database connection error: {str(e)}")

                log\_error(f"Database connection error ({db\_type}): {str(e)}")

    return None

def handle\_api\_connection(prefix: str) -> Optional[pd.DataFrame]:

    """Handle API connections with enhanced UI and error handling"""

    # Add API connection UI with modern styling

    st.markdown(

        """

        <div style='background-color: #f8f9fa; padding: 1.5rem; border-radius: 10px; margin-bottom: 1rem; border: 1px solid #dee2e6;'>

            <h4 style='margin: 0; color: #1f77b4;'>API Connection</h4>

            <p style='margin: 0.5rem 0 0 0; color: #6c757d;'>

                Configure your API connection details and parameters below.

            </p>

        </div>

        """,

        unsafe\_allow\_html=True

    )

    # Create tabs for basic and advanced configuration

    basic\_tab, advanced\_tab = st.tabs(["Basic Configuration", "Advanced Settings"])

    with basic\_tab:

        # URL and Method

        col1, col2 = st.columns([3, 1])

        with col1:

            api\_url = st.text\_input(

                "API URL",

                key=f"{prefix}\_api\_url",

                help="Enter the complete API endpoint URL",

                placeholder="https://api.example.com/data"

            )

        with col2:

            method = st.selectbox(

                "Method",

                ["GET", "POST"],

                key=f"{prefix}\_method",

                help="Select the HTTP method to use"

            )

    with advanced\_tab:

        # Headers

        st.markdown("##### Headers")

        headers = st.text\_area(

            "Headers (JSON format)",

            key=f"{prefix}\_headers",

            help="Enter headers as JSON object, e.g., {\"Authorization\": \"Bearer token\"}",

            height=100,

            placeholder="""

{

    "Authorization": "Bearer your\_token",

    "Content-Type": "application/json"

}"""

        )

        # Parameters

        st.markdown("##### Parameters")

        params = st.text\_area(

            "Parameters (JSON format)",

            key=f"{prefix}\_params",

            help="Enter query parameters or POST body as JSON object",

            height=100,

            placeholder="""

{

    "limit": 1000,

    "offset": 0

}"""

        )

    # Connection button with loading state

    if st.button("🔌 Connect to API", key=f"{prefix}\_connect", use\_container\_width=True):

        with st.spinner("Connecting to API..."):

            try:

                # Validate inputs

                if not api\_url:

                    st.error("❌ API URL is required")

                    return None

                # Parse JSON inputs

                try:

                    headers\_dict = eval(headers) if headers else None

                    params\_dict = eval(params) if params else None

                except Exception as e:

                    st.error(f"❌ Invalid JSON format: {str(e)}")

                    return None

                # Fetch data

                df = APIFetcher.fetch\_api\_data(

                    api\_url=api\_url,

                    method=method,

                    headers=headers\_dict,

                    params=params\_dict

                )

                if df is not None:

                    st.success("✅ API connection successful!")

                    st.info(f"Retrieved {len(df)} rows and {len(df.columns)} columns")

                    # Show data preview in an expander

                    with st.expander("Data Preview"):

                        st.dataframe(

                            df.head(5),

                            use\_container\_width=True,

                            height=200

                        )

                    return df

                else:

                    st.error("❌ No data received from API")

            except Exception as e:

                st.error(f"❌ API connection error: {str(e)}")

                log\_error(f"API connection error: {str(e)}")

                # Show detailed error message in an expander

                with st.expander("Error Details"):

                    st.code(str(e))

    return None

def show\_column\_mapping\_interface(source\_df: pd.DataFrame, target\_df: pd.DataFrame):

    """Show interface for column mapping"""

    st.markdown(

        """

        <div style='background-color: #f8f9fa; padding: 1.5rem; border-radius: 10px; margin-bottom: 1rem; border: 1px solid #dee2e6;'>

            <h3 style='margin: 0; color: #1f77b4;'>Column Mapping Configuration</h3>

            <p style='margin: 0.5rem 0 0 0; color: #6c757d;'>

                Map columns between source and target data sources. Use auto-mapping or manually adjust the mappings below.

            </p>

        </div>

        """,

        unsafe\_allow\_html=True

    )

    # Create two columns for the auto-map button and mapping status

    col1, col2 = st.columns([1, 2])

    with col1:

        # Add auto-map button with enhanced styling

        if st.button("🔄 Auto-Map Columns", key="auto\_map\_btn", use\_container\_width=True):

            with st.spinner("Mapping columns..."):

                st.session\_state.column\_mapping = MappingManager.auto\_map\_columns(source\_df, target\_df)

                if st.session\_state.column\_mapping:

                    st.success(f"✅ Successfully mapped {len(st.session\_state.column\_mapping)} columns!")

                else:

                    st.warning("⚠️ No automatic matches found. Please map columns manually.")

    with col2:

        # Show current mapping status with enhanced styling

        if st.session\_state.column\_mapping:

            total\_cols = len(source\_df.columns)

            mapped\_cols = len(st.session\_state.column\_mapping)

            mapping\_percentage = (mapped\_cols / total\_cols) \* 100

            st.markdown(

                f"""

                <div style='background-color: #e8f4f8; padding: 0.75rem; border-radius: 5px; text-align: center;'>

                    <span style='font-size: 1.1em; color: #1f77b4;'>

                        {mapped\_cols} of {total\_cols} columns mapped ({mapping\_percentage:.1f}%)

                    </span>

                </div>

                """,

                unsafe\_allow\_html=True

            )

    # Show mapping table with enhanced styling

    st.markdown(

        """

        <div style='margin-top: 1rem;'>

            <h4 style='color: #666;'>Current Column Mappings</h4>

        </div>

        """,

        unsafe\_allow\_html=True

    )

    # Create a DataFrame to display the mapping

    mapping\_data = []

    for source\_col in source\_df.columns:

        source\_sample = str(source\_df[source\_col].head(2).tolist())

        target\_col = st.session\_state.column\_mapping.get(source\_col, '')

        target\_sample = str(target\_df[target\_col].head(2).tolist()) if target\_col else ''

        mapping\_data.append({

            'Source Column': source\_col,

            'Source Sample': source\_sample,

            'Target Column': target\_col,

            'Target Sample': target\_sample,

            'Status': '✅' if target\_col else '❌'

        })

    mapping\_df = pd.DataFrame(mapping\_data)

    # Display the mapping table with custom styling

    st.dataframe(

        mapping\_df,

        use\_container\_width=True,

        height=400,

        column\_config={

            "Status": st.column\_config.Column(

                "Mapping Status",

                help="✅: Mapped | ❌: Unmapped",

                width="small"

            )

        }

    )

    # Create three columns for the mapping interface

    col1, col2, col3 = st.columns([2, 2, 1])

    with col1:

        st.markdown("\*\*Source Column\*\*")

    with col2:

        st.markdown("\*\*Target Column\*\*")

    with col3:

        st.markdown("\*\*Exclude\*\*")

    # Show mapping interface for each source column

    for source\_col in source\_df.columns:

        col1, col2, col3 = st.columns([2, 2, 1])

        with col1:

            st.text(source\_col)

            st.caption(f"Sample: {str(source\_df[source\_col].head(2).tolist())}")

        with col2:

            # Dropdown for target column selection

            current\_mapping = st.session\_state.column\_mapping.get(source\_col, '')

            target\_options = [''] + list(target\_df.columns)

            target\_index = target\_options.index(current\_mapping) if current\_mapping in target\_options else 0

            target\_col = st.selectbox(

                f"Map to",

                options=target\_options,

                index=target\_index,

                key=f"mapping\_{source\_col}"

            )

            if target\_col:

                if target\_col != current\_mapping:

                    st.session\_state.column\_mapping[source\_col] = target\_col

                    # Show sample of selected target column

                    st.caption(f"Sample: {str(target\_df[target\_col].head(2).tolist())}")

            elif source\_col in st.session\_state.column\_mapping:

                del st.session\_state.column\_mapping[source\_col]

        with col3:

            # Checkbox for excluding column from comparison

            excluded = st.checkbox(

                "Exclude",

                key=f"exclude\_{source\_col}",

                value=source\_col in st.session\_state.excluded\_columns

            )

            if excluded and source\_col not in st.session\_state.excluded\_columns:

                st.session\_state.excluded\_columns.append(source\_col)

            elif not excluded and source\_col in st.session\_state.excluded\_columns:

                st.session\_state.excluded\_columns.remove(source\_col)

def perform\_comparison():

    """Perform the comparison and generate reports"""

    try:

        st.markdown("### Comparison Results")

        # Create reports directory if it doesn't exist

        os.makedirs("reports", exist\_ok=True)

        timestamp = format\_timestamp()

        # Generate reports

        join\_keys = st.session\_state.get('join\_keys', None)

        with st.spinner("Generating comparison reports..."):

            try:

                from ydata\_profiling import ProfileReport

                import streamlit.components.v1 as components

                # Generate Y-Data HTML profiling reports

                st.write("Generating Y-Data Profiling Reports...")

                # Source profiling

                source\_profile = ProfileReport(

                    st.session\_state.source\_df,

                    title="Source Data Profile Report",

                    minimal=True

                )

                source\_html = f"reports/SourceProfile\_{timestamp}.html"

                source\_profile.to\_file(source\_html)

                # Target profiling

                target\_profile = ProfileReport(

                    st.session\_state.target\_df,

                    title="Target Data Profile Report",

                    minimal=True

                )

                target\_html = f"reports/TargetProfile\_{timestamp}.html"

                target\_profile.to\_file(target\_html)

                # Generate other reports

                st.write("Generating Comparison Reports...")

                # Generate difference report

                diff\_df, has\_differences = ReportGenerator.generate\_diff\_report(

                    st.session\_state.source\_df,

                    st.session\_state.target\_df,

                    st.session\_state.column\_mapping,

                    st.session\_state.excluded\_columns,

                    join\_keys=join\_keys

                )

                # Generate profiling report

                profile\_df = ReportGenerator.generate\_profiling\_report(

                    st.session\_state.source\_df,

                    st.session\_state.target\_df,

                    st.session\_state.column\_mapping,

                    join\_keys=join\_keys

                )

                # Generate regression report (independent of join keys)

                regression\_path = f"reports/RegressionReport\_{timestamp}.xlsx"

                ReportGenerator.generate\_regression\_report(

                    st.session\_state.source\_df,

                    st.session\_state.target\_df,

                    st.session\_state.column\_mapping,

                    regression\_path

                )

                # Generate side-by-side report

                side\_by\_side\_path = f"reports/DifferenceReport\_{timestamp}.xlsx"

                side\_by\_side\_df, \_ = ReportGenerator.generate\_side\_by\_side\_report(

                    st.session\_state.source\_df,

                    st.session\_state.target\_df,

                    st.session\_state.column\_mapping,

                    side\_by\_side\_path,

                    join\_keys=join\_keys

                )

                # Display Y-Data Profiling Reports in expandable sections

                st.subheader("Y-Data Profiling Reports")

                # Generate comparison profile

                comparison\_profile = ProfileReport(

                    pd.concat([

                        st.session\_state.source\_df.add\_prefix('Source\_'),

                        st.session\_state.target\_df.add\_prefix('Target\_')

                    ], axis=1),

                    title="Source vs Target Comparison Profile",

                    minimal=True

                )

                comparison\_html = f"reports/ComparisonProfile\_{timestamp}.html"

                comparison\_profile.to\_file(comparison\_html)

                # Generate DataCompy comparison report

                try:

                    # Prepare join keys for DataCompy

                    join\_columns = []

                    source\_df = st.session\_state.source\_df.copy()

                    target\_df = st.session\_state.target\_df.copy()

                    if join\_keys:

                        join\_columns = [key[0] for key in join\_keys]  # Use source column names

                    else:

                        # If no join keys specified, use index

                        source\_df['\_index'] = source\_df.index

                        target\_df['\_index'] = target\_df.index

                        join\_columns = ['\_index']

                    # Create DataCompy comparison

                    comparison = datacompy.Compare(

                        source\_df,

                        target\_df,

                        join\_columns=join\_columns,

                        df1\_name='Source',

                        df2\_name='Target'

                    )

                    # Get column statistics safely

                    column\_stats\_html = '<p>No column statistics available.</p>'

                    try:

                        if (hasattr(comparison, 'column\_stats') and

                            isinstance(comparison.column\_stats, pd.DataFrame) and

                            not comparison.column\_stats.empty):

                            column\_stats\_html = comparison.column\_stats.to\_html()

                    except Exception as e:

                        st.warning(f"Could not generate column statistics: {str(e)}")

                    # Generate HTML report

                    datacompy\_html = f"reports/DataCompyReport\_{timestamp}.html"

                    with open(datacompy\_html, 'w') as f:

                        f.write(f"""

                        <html>

                        <head>

                            <title>DataCompy Comparison Report</title>

                            <style>

                                body {{ font-family: Arial, sans-serif; margin: 20px; }}

                                .report {{ max-width: 1200px; margin: 0 auto; }}

                                .section {{ margin: 20px 0; padding: 20px; border: 1px solid #ddd; border-radius: 5px; }}

                                .match {{ color: green; }}

                                .mismatch {{ color: red; }}

                                table {{ border-collapse: collapse; width: 100%; }}

                                th, td {{ border: 1px solid #ddd; padding: 8px; text-align: left; }}

                                th {{ background-color: #f5f5f5; }}

                            </style>

                        </head>

                        <body>

                            <div class="report">

                                <h1>DataCompy Comparison Report</h1>

                                <div class="section">

                                    <h2>Summary</h2>

                                    <pre>{comparison.report()}</pre>

                                </div>

                                <div class="section">

                                    <h2>Detailed Statistics</h2>

                                    <h3>Matches</h3>

                                    <div class="match">

                                        <p>Number of rows match: {comparison.count\_matching\_rows()}</p>

                                        <p>Number of columns match: {len(set(source\_df.columns).intersection(target\_df.columns))}</p>

                                    </div>

                                    <h3>Mismatches</h3>

                                    <div class="mismatch">

                                        <p>Rows only in Source: {len(comparison.df1\_unq\_rows)}</p>

                                        <p>Rows only in Target: {len(comparison.df2\_unq\_rows)}</p>

                                        <p>Source-only columns: {len(set(source\_df.columns) - set(target\_df.columns))}</p>

                                        <p>Target-only columns: {len(set(target\_df.columns) - set(source\_df.columns))}</p>

                                    </div>

                                </div>

                                <div class="section">

                                    <h2>Column Statistics</h2>

                                    {column\_stats\_html}

                                </div>

                            </div>

                        </body>

                        </html>

                        """)

                except Exception as e:

                    st.error(f"Error generating DataCompy report: {str(e)}")

                    datacompy\_html = None

                # Store DataCompy report path

                st.session\_state.report\_paths['datacompy\_html'] = datacompy\_html

                # Store report paths in session state to prevent page reset

                if 'report\_paths' not in st.session\_state:

                    st.session\_state.report\_paths = {}

                st.session\_state.report\_paths.update({

                    'source\_html': source\_html,

                    'target\_html': target\_html,

                    'comparison\_html': comparison\_html

                })

                tab1, tab2, tab3, tab4 = st.tabs(["Source Profile", "Target Profile", "Comparison Profile", "DataCompy Report"])

                with tab1:

                    with open(source\_html, 'r', encoding='utf-8') as f:

                        components.html(f.read(), height=600, scrolling=True)

                    # Download button for source profile

                    with open(source\_html, 'rb') as f:

                        st.download\_button(

                            "Download Source Profile Report",

                            f,

                            file\_name=f"SourceProfile\_{timestamp}.html",

                            mime="text/html",

                            key="download\_source"

                        )

                with tab2:

                    with open(target\_html, 'r', encoding='utf-8') as f:

                        components.html(f.read(), height=600, scrolling=True)

                    # Download button for target profile

                    with open(target\_html, 'rb') as f:

                        st.download\_button(

                            "Download Target Profile Report",

                            f,

                            file\_name=f"TargetProfile\_{timestamp}.html",

                            mime="text/html",

                            key="download\_target"

                        )

                with tab3:

                    with open(comparison\_html, 'r', encoding='utf-8') as f:

                        components.html(f.read(), height=600, scrolling=True)

                    # Download button for comparison profile

                    with open(comparison\_html, 'rb') as f:

                        st.download\_button(

                            "Download Comparison Profile Report",

                            f,

                            file\_name=f"ComparisonProfile\_{timestamp}.html",

                            mime="text/html",

                            key="download\_comparison"

                        )

                with tab4:

                    if datacompy\_html and os.path.exists(datacompy\_html):

                        with open(datacompy\_html, 'r', encoding='utf-8') as f:

                            components.html(f.read(), height=600, scrolling=True)

                        col1, col2 = st.columns([3, 1])

                        with col1:

                            # Download button for DataCompy report

                            with open(datacompy\_html, 'rb') as f:

                                st.download\_button(

                                    "📥 Download DataCompy Report",

                                    f,

                                    file\_name=f"DataCompyReport\_{timestamp}.html",

                                    mime="text/html",

                                    key="download\_datacompy",

                                    use\_container\_width=True

                                )

                        # Show additional DataCompy insights

                        with st.expander("📊 DataCompy Insights", expanded=False):

                            try:

                                st.write("### Match Statistics")

                                col1, col2, col3 = st.columns(3)

                                matching\_rows = comparison.count\_matching\_rows()

                                source\_only\_rows = len(comparison.df1\_unq\_rows)

                                target\_only\_rows = len(comparison.df2\_unq\_rows)

                                with col1:

                                    st.metric(

                                        "Matching Rows",

                                        matching\_rows,

                                        delta=f"{matching\_rows / len(source\_df) \* 100:.1f}%"

                                    )

                                with col2:

                                    st.metric(

                                        "Source Only Rows",

                                        source\_only\_rows,

                                        delta=f"{source\_only\_rows / len(source\_df) \* 100:.1f}%"

                                    )

                                with col3:

                                    st.metric(

                                        "Target Only Rows",

                                        target\_only\_rows,

                                        delta=f"{target\_only\_rows / len(target\_df) \* 100:.1f}%"

                                    )

                                st.write("### Column Analysis")

                                # Get column sets

                                source\_cols = set(source\_df.columns)

                                target\_cols = set(target\_df.columns)

                                common\_cols = source\_cols.intersection(target\_cols)

                                source\_only = source\_cols - target\_cols

                                target\_only = target\_cols - source\_cols

                                # Display column sets

                                col1, col2 = st.columns(2)

                                with col1:

                                    st.write("Source-only Columns:")

                                    if source\_only:

                                        for col in sorted(source\_only):

                                            st.info(f"- {col}")

                                    else:

                                        st.success("No source-only columns")

                                with col2:

                                    st.write("Target-only Columns:")

                                    if target\_only:

                                        for col in sorted(target\_only):

                                            st.info(f"- {col}")

                                    else:

                                        st.success("No target-only columns")

                                st.write("### Common Column Analysis")

                                if common\_cols:

                                    for col in sorted(common\_cols):

                                        # Calculate match rate for common columns

                                        source\_values = set(source\_df[col].dropna())

                                        target\_values = set(target\_df[col].dropna())

                                        common\_values = source\_values.intersection(target\_values)

                                        all\_values = source\_values.union(target\_values)

                                        match\_rate = len(common\_values) / len(all\_values) if all\_values else 1.0

                                        # Display progress bar with details

                                        st.progress(

                                            match\_rate,

                                            text=f"{col}: {match\_rate\*100:.1f}% unique values match"

                                        )

                                        with st.expander(f"Details for {col}", expanded=False):

                                            st.write(f"- Unique values in Source: {len(source\_values)}")

                                            st.write(f"- Unique values in Target: {len(target\_values)}")

                                            st.write(f"- Common unique values: {len(common\_values)}")

                                            if len(source\_values - target\_values) > 0:

                                                st.write("- Values only in Source:", list(source\_values - target\_values)[:5])

                                            if len(target\_values - source\_values) > 0:

                                                st.write("- Values only in Target:", list(target\_values - source\_values)[:5])

                                else:

                                    st.warning("No common columns found between source and target datasets")

                            except Exception as e:

                                st.error(f"Error displaying insights: {str(e)}")

                    else:

                        st.error("❌ DataCompy report generation failed. Please check the data and try again.")

                        with col2:

                            st.write("Target-only Columns:")

                            if target\_only:

                                for col in sorted(target\_only):

                                    st.info(f"- {col}")

                            else:

                                st.success("No target-only columns")

                        st.write("### Common Column Analysis")

                        if common\_cols:

                            for col in sorted(common\_cols):

                                # Calculate match rate for common columns

                                source\_values = set(st.session\_state.source\_df[col].dropna())

                                target\_values = set(st.session\_state.target\_df[col].dropna())

                                common\_values = source\_values.intersection(target\_values)

                                all\_values = source\_values.union(target\_values)

                                match\_rate = len(common\_values) / len(all\_values) if all\_values else 1.0

                                # Display progress bar with details

                                st.progress(

                                    match\_rate,

                                    text=f"{col}: {match\_rate\*100:.1f}% unique values match"

                                )

                                with st.expander(f"Details for {col}", expanded=False):

                                    st.write(f"- Unique values in Source: {len(source\_values)}")

                                    st.write(f"- Unique values in Target: {len(target\_values)}")

                                    st.write(f"- Common unique values: {len(common\_values)}")

                                    if len(source\_values - target\_values) > 0:

                                        st.write("- Values only in Source:", list(source\_values - target\_values)[:5])

                                    if len(target\_values - source\_values) > 0:

                                        st.write("- Values only in Target:", list(target\_values - source\_values)[:5])

                        else:

                            st.warning("No common columns found between source and target datasets")

            except ImportError:

                st.warning("Y-Data Profiling package not installed. Please install ydata-profiling package.")

                # Continue with other reports...

        # Store report paths in session state

        st.session\_state.report\_paths.update({

            'side\_by\_side\_path': side\_by\_side\_path,

            'profiling\_path': f"reports/ProfilingReport\_{timestamp}.xlsx",

            'regression\_path': regression\_path

        })

        # Create zip file containing all reports

        st.divider()

        zip\_path = f"reports/FinalComparison\_{timestamp}.zip"

        try:

            import zipfile

            with zipfile.ZipFile(zip\_path, 'w') as zipf:

                # Add all reports to zip

                for report\_name, report\_path in st.session\_state.report\_paths.items():

                    if report\_path and os.path.exists(report\_path):

                        zipf.write(report\_path, os.path.basename(report\_path))

            # Add download button for zip file

            st.markdown("### 📦 Download Complete Comparison Package")

            st.caption("Contains all reports: Regression, Side by Side, DataCompy, and Profile Reports")

            with open(zip\_path, 'rb') as f:

                st.download\_button(

                    "📥 Download All Reports (ZIP)",

                    f,

                    file\_name=f"FinalComparison\_{timestamp}.zip",

                    mime="application/zip",

                    key="download\_all",

                    use\_container\_width=True

                )

        except Exception as e:

            st.error(f"Error creating zip file: {str(e)}")

        st.divider()

        # Display results and download buttons

        col1, col2 = st.columns(2)

        with col1:

            st.markdown("#### Data Comparison Results")

            st.dataframe(diff\_df)

            if os.path.exists(side\_by\_side\_path):

                with open(side\_by\_side\_path, 'rb') as f:

                    st.download\_button(

                        "Download Difference Report",

                        f,

                        file\_name=f"DifferenceReport\_{timestamp}.xlsx",

                        mime="application/vnd.openxmlformats-officedocument.spreadsheetml.sheet",

                        key="download\_difference"

                    )

        with col2:

            st.markdown("#### Y-Data Profiling Results")

            if not profile\_df.empty:

                # Format the dataframe for better display

                st.write("Column-wise Statistical Comparison:")

                # Create tabs for different views

                metrics\_tab, details\_tab = st.tabs(["Main Metrics", "Detailed Statistics"])

                with metrics\_tab:

                    main\_metrics = profile\_df[['Column', 'Source\_Count', 'Target\_Count', 'Match\_Percentage']]

                    st.dataframe(main\_metrics, use\_container\_width=True)

                with details\_tab:

                    st.dataframe(profile\_df, use\_container\_width=True)

                # Save profiling report

                profiling\_path = st.session\_state.report\_paths['profiling\_path']

                profile\_df.to\_excel(profiling\_path, index=False)

                # Download buttons with unique keys

                col3, col4 = st.columns(2)

                with col3:

                    with open(profiling\_path, 'rb') as f:

                        st.download\_button(

                            "Download Profiling Report",

                            f,

                            file\_name=f"ProfilingReport\_{timestamp}.xlsx",

                            mime="application/vnd.openxmlformats-officedocument.spreadsheetml.sheet",

                            key="download\_profiling"

                        )

                with col4:

                    if os.path.exists(regression\_path):

                        with open(regression\_path, 'rb') as f:

                            st.download\_button(

                                "Download Regression Report",

                                f,

                                file\_name=f"RegressionReport\_{timestamp}.xlsx",

                                mime="application/vnd.openxmlformats-officedocument.spreadsheetml.sheet",

                                key="download\_regression"

                            )

            else:

                st.error("Unable to generate profiling report. Please check the data and try again.")

    except Exception as e:

        log\_error(f"Error performing comparison: {str(e)}")

        st.error(f"Error performing comparison: {str(e)}")

if \_\_name\_\_ == "\_\_main\_\_":

    main()