Synthetic Data Factory

Automated, Statistically-Driven Synthetic Data Generation

and Validation Platform

Project Overview:

The Synthetic Data Factory (SDF) is an intelligent web application that enables users to generate realistic, privacy-safe, and statistically validated synthetic datasets with ease. It allows developers, data scientists, testers, and researchers to create custom datasets for analytics, testing, and machine learning without relying on sensitive real-world data.

This system bridges the gap between data accessibility and privacy by learning the structure, distribution, and relationships within uploaded datasets or defined schemas, and then reproducing that same statistical behaviour in newly generated synthetic data.

By integrating AI-based data pattern detection, Faker-driven content generation, and statistical validation techniques, SDF ensures every dataset is both realistic and unique, making it ideal for simulation, system testing, research, and training environments.

Scenario 1: Data Science and Machine Learning Development

Data scientists and machine learning engineers often struggle to find large, high-quality datasets that are both diverse and privacy-compliant. The Synthetic Data Factory (SDF) addresses this challenge by generating realistic synthetic data that mirrors the statistical patterns of real-world datasets without exposing sensitive information. For instance, a financial technology startup developing a loan prediction model can use SDF to create synthetic data representing customer profiles, income ranges, credit scores, and loan histories. Instead of relying on confidential client data, they can upload a small sample dataset or define the required schema, and SDF will generate thousands of synthetic records maintaining the same relationships and distributions.

With SDF's built-in visualization and validation tools, teams can ensure that the generated data accurately reflects the original structure. This allows safe, large-scale model training and testing while remaining compliant with data privacy standards. Through its integration of Streamlit, Faker, and statistical validation frameworks, the Synthetic Data Factory not only accelerates data preparation but also empowers organizations to innovate responsibly. Ultimately, SDF transforms data scarcity into data abundance driving progress in AI, analytics, and decision intelligence securely and efficiently.

Scenario 2: Application Development and Database Testing

In software development and testing, teams often require large, diverse, and realistic datasets to validate performance, database operations, and system integration. However, using production data introduces serious privacy and compliance challenges. The Synthetic Data Factory (SDF)

overcomes this by generating synthetic datasets that replicate the structure and statistical behaviour of real data without revealing any sensitive information.

For example, a banking enterprise can use SDF to generate synthetic customer profiles, account details, and transaction histories to test APIs, data pipelines, and application performance under production-like conditions. Through its intelligent combination of Streamlit, Faker, and validation frameworks, SDF enables secure, scalable, and automated data population for development environments.

Its integrated MySQL connectivity module supports direct schema detection and synthetic data insertion, simplifying test data management. By ensuring realistic simulation, privacy compliance, and data consistency, SDF helps teams accelerate release cycles, enhance reliability, and minimize time and cost in test data generation.

Architecture Overview:

The Synthetic Data Factory is a modular, intelligent data generation platform designed to create realistic, privacy-safe, and statistically balanced synthetic datasets for AI, analytics, and application testing. Built using Streamlit as the front-end framework, SDF integrates a seamless, interactive interface with a robust backend powered by Python, Faker, NumPy, Pandas, and PyMySQL for advanced data handling and simulation.

The system follows a multi-layered architecture consisting of three core backend modules: UniversalDataGenerator, DatabaseHandler, and DataValidator, each serving a distinct role. The *UniversalDataGenerator* module generates synthetic datasets either from uploaded CSV files or user-defined column schemas. The *DatabaseHandler* enables direct integration with MySQL databases, allowing users to generate and insert synthetic data directly based on table structures. The *DataValidator* module ensures statistical fidelity by comparing real and synthetic datasets using distribution metrics and visual validations.

The Streamlit-based front-end ties these components together through an intuitive tabbed interface featuring CSV extension, custom data creation, and database connectivity sections. With integrated validation, progress tracking, and responsive visual design, SDF provides users with a complete synthetic data ecosystem enabling them to generate, validate, and visualize data securely and efficiently for diverse enterprise and research applications.

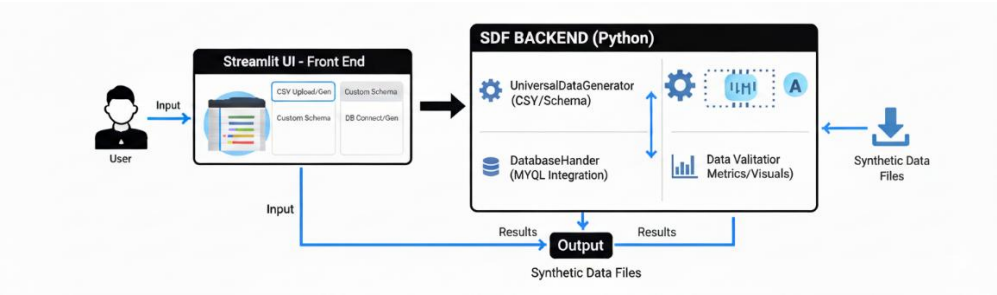


Figure 1: System architecture diagram

Core Technologies:

- **Streamlit (Frontend Framework):** Provides an intuitive and interactive web interface for dataset generation, schema customization, and visualization. It handles real-time user inputs, displays synthetic data previews, and integrates dynamic components like tabs and progress indicators.
- **Faker (Synthetic Data Engine):** Powers the data generation process by creating realistic yet artificial values for various data types such as names, addresses, transactions, and demographic details. It supports both predefined templates and custom schema definitions.
- **NumPy & Pandas (Data Processing Core):** Enable efficient data manipulation, statistical computation, and tabular structure handling. These libraries ensure scalable and optimized performance during large-scale synthetic dataset creation and validation.
- **PyMySQL (Database Integration):** Allows direct connection with MySQL databases for real-time schema extraction and automatic insertion of generated synthetic data, enabling seamless backend testing and database population.
- **Matplotlib & Seaborn (Visualization Framework):** Used to compare and validate real vs. synthetic datasets through visual metrics such as histograms, distributions, and correlation heatmaps, ensuring statistical integrity and data balance.
- **Data Validation & Comparison Module:** Ensures that synthetic datasets maintain consistency, diversity, and similarity to real datasets by applying distribution checks, statistical summaries, and visualization-based validation.
- **Python Backend (Core Logic Layer):** Manages communication between all modules data generation, validation, and database operations that's ensuring smooth execution, modular scalability, and high-performance processing.

Component-Wise Architecture:

Component	Description
User Interface (Streamlit)	Interactive dashboard for CSV upload, schema creation, and database connection with real-time data display.
Universal Data Generator	Creates realistic synthetic datasets using Faker, NumPy, and Pandas based on sample or custom schema.
Database Handler (MySQL)	Connects to MySQL, detects table schemas, and inserts generated data directly for testing and simulation.
Data Validator & Visualizer	Compares real vs. synthetic data through statistical checks and visualizations using Matplotlib and Seaborn.
Custom Schema Builder	Let users define columns, types, and record limits for domain-specific synthetic data generation.

Visualization & Reporting	Generates charts and summaries for quick validation of data quality and distribution patterns.
Backend Logic Controller	Manages coordination between data generation, validation, and visualization modules.
Help & Documentation	Provides usage guidance and workflow support within the Streamlit interface.

Pre-requisites:

1. **Python Environment Setup**: The Synthetic Data Factory (SDF) is developed using Python 3.9+, leveraging its extensive data handling and visualization libraries. Create and activate a dedicated virtual environment (e.g., synthetic_data_factory) to manage dependencies and maintain a clean setup.

   Official Documentation: https://www.python.org/downloads/

2. **Streamlit Installation and Configuration**: Streamlit provides the front-end interface for data generation, visualization, and validation. Install Streamlit to build and run the interactive web dashboard.

   Docs: https://docs.streamlit.io/library/get-started/installation

   Tutorial: https://www.youtube.com/watch?v=JwSS70SZdyM

3. **Library Installation and Core Dependencies**: SDF relies on several Python libraries for data generation, analysis, and visualization. Install all dependencies listed in requirements.txt.

   Key Libraries:

   streamlit – Interactive UI and data display

   faker – Synthetic data generation

   pandas – Data manipulation and structure handling

   numpy – Numerical and statistical computation

   matplotlib & seaborn – Data visualization and validation plots

   pymysql – MySQL database connectivity and data insertion

4. **Database Setup**: For projects requiring live database testing, install and configure MySQL Server. SDF connects via PyMySQL, enabling table schema reading and direct data insertion.

   MySQL Installation: https://dev.mysql.com/downloads/

   PyMySQL Docs: https://pypi.org/project/PyMySQL/

5. **Development Environment**:
   Recommended IDEs for efficient development and debugging

   Visual Studio Code: https://code.visualstudio.com/

   PyCharm (Community Edition): https://www.jetbrains.com/pycharm/download/

6. **Optional Learning Resources**:
   Enhance understanding of the tools and concepts used

   Streamlit Components Gallery: https://streamlit.io/components

   Faker Library Guide: https://faker.readthedocs.io/

Project Flow:

1. Environment Setup and Dependency Configuration

- **Activity 1.1:** Create and activate a virtual environment and install required dependencies.
- **Activity 1.2:** Organize project structure and add UI styling with styles.css.
- **Activity 1.3:** Initialize Streamlit app and validate integration between modules.

2. Core Logic Development (Synthetic Data Generation Engine)

- **Activity 2.1:** Implement UniversalDataGenerator for CSV and schema-based synthetic data.
- **Activity 2.2:** Develop validation module to analyze and compare real vs synthetic data.
- **Activity 2.3:** Integrate MySQL database for schema retrieval and synthetic data insertion.

3. Streamlit UI Implementation and User Interaction

- **Activity 3.1:** Design multi-tab layout with navigation and professional styling.
- **Activity 3.2:** Configure input system for columns, rows, and database details with session state.
- **Activity 3.3:** Display synthetic data and visualizations with computed metrics.

4. Testing, Optimization, and Deployment

- **Activity 4.1:** Test UI components and verify accurate data generation.
- **Activity 4.2:** Validate database integration and large dataset performance.
- **Activity 4.3:** Prepare deployment and perform end-to-end validation for stability and performance.

MILESTONE 1: Environment Setup and Dependency Configuration

This foundational milestone establishes the technical environment required for building and deploying the Synthetic Data Factory (SDF). It ensures that all dependencies, frameworks, and integrations are configured correctly for seamless execution of synthetic data generation, visualization, and validation workflows.

Activity 1.1: Python Environment and Dependency Installation

- Create and activate a dedicated virtual environment for SDF to maintain dependency isolation.

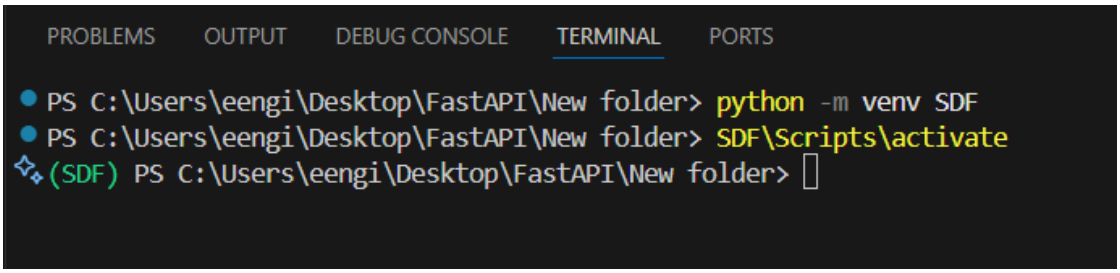


Figure 2: Creating & Activating Environment

- Install all required dependencies listed in requirements.txt using:

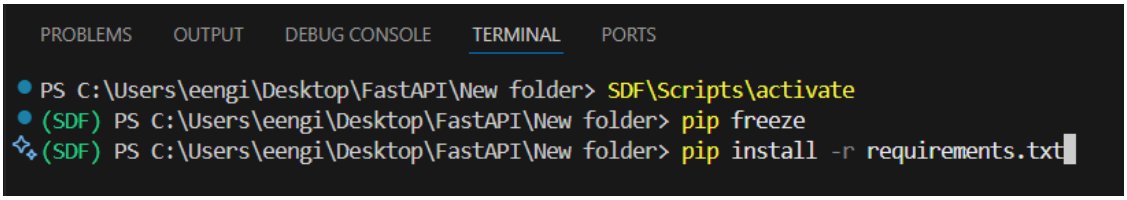


Figure 3: Installing requirements

- Verify compatibility across Python 3.9+ and confirm successful installation of each module.

- Test library imports and ensure data-related packages (NumPy, Pandas) and visualization tools (Matplotlib, Seaborn) work correctly.

Activity 1.2: Project Structure Initialization

- Organize project modules (streamlit_app.py, data_generator.py, database_handler.py, validation.py) into a structured hierarchy for clarity and maintainability.

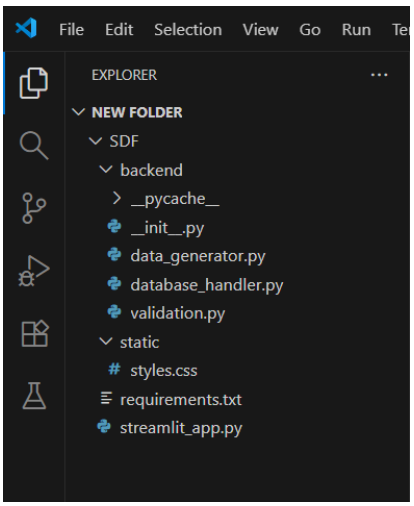


Figure 4: Folder Structure

- Add styles.css to define Streamlit UI design, theme consistency, and responsive layout behaviour.

- Verify file paths and imports to ensure modular independence and scalability.

Activity 1.3: Streamlit Application Initialization

- Set up streamlit_app.py as the central entry point of the system using:

```
st.set_page_config(
    page_title="Synthetic Data Factory",
    layout="wide",
    initial_sidebar_state="expanded"
)

# Load custom CSS
with open(r"C:\Users\eengi\Desktop\SDG\New folder\SDF\static\styles.css") as f:
    st.markdown(f"<style>{f.read()}</style>", unsafe_allow_html=True)

# Initialize session state
if 'generator' not in st.session_state:
    st.session_state.generator = UniversalDataGenerator()
if 'db_handler' not in st.session_state:
    st.session_state.db_handler = DatabaseHandler(st.session_state.generator)
if 'validator' not in st.session_state:
    st.session_state.validator = DataValidator()
if 'synthetic_data' not in st.session_state:
    st.session_state.synthetic_data = None
```

Figure 5: Streamlit Configuration

- Ensure consistent theming, responsive layout, and component rendering across devices.
- Validate integration between all modules, confirming smooth data flow between UI and backend functions.

MILESTONE 2: Core Logic Development (Synthetic Data Generation Engine)

This milestone builds the analytical and generation backbone of SDF. The goal is to automate realistic, statistically valid data generation using the UniversalDataGenerator and connect it with analysis, validation, and storage pipelines.

Activity 2.1: Universal Data Generator Implementation (data_generator.py)

- Develop the UniversalDataGenerator class to handle both CSV-based and custom schema-based synthetic data generation.

```python
class UniversalDataGenerator:
    def __init__(self):
        self.fake = Faker()

    def generate_from_csv(self, csv_file: str, num_rows: int = 200, validate: bool = True):
        progress_bar = st.progress(0)
        status_text = st.empty()

        status_text.text("Uploading CSV file...")

        try:
            original_df = pd.read_csv(csv_file)
            progress_bar.progress(25)

            status_text.text("Analyzing data structure...")

            with st.expander("Original Data Preview", expanded=True):
                st.write(f"**Dataset Info:** {len(original_df)} rows, {len(original_df.columns)} columns")
                st.dataframe(original_df.head(), use_container_width=True)

            synthetic = {}
            total_columns = len(original_df.columns)

            for i, col in enumerate(original_df.columns):
                status_text.text(f"Generating column: {col} ({i+1}/{total_columns})")
                progress_bar.progress(25 + int(25 * i / total_columns))

                if pd.api.types.is_numeric_dtype(original_df[col]):
                    synthetic[col] = self._generate_numeric_data(original_df[col], num_rows)
                elif pd.api.types.is_string_dtype(original_df[col]) or pd.api.types.is_object_dtype(original_df[col]):
                    synthetic[col] = self._generate_text_data(col, original_df[col], num_rows)
                else:
                    synthetic[col] = [f"Data_{i}" for i in range(num_rows)]

            synthetic_df = pd.DataFrame(synthetic)

            status_text.text("Processing relationships between columns...")
            progress_bar.progress(75)

            synthetic_df = self._handle_age_dob_relationship(synthetic_df)

            progress_bar.progress(100)

            status_text.text("Generation complete!")

            with st.expander("Synthetic Data Preview", expanded=True):
                st.dataframe(synthetic_df.head(10), use_container_width=True)

            return synthetic_df, original_df

        except Exception as e:
            st.error(f"Error reading CSV file: {e}")
            return None, None
```

Figure 6: Dataset Generator from CSV

```python
    def generate_from_columns(self, columns: List[str], num_rows: int = 200):
        progress_bar = st.progress(0)
        status_text = st.empty()

        status_text.text(f"Generating data for {len(columns)} columns...")

        synthetic = {}
        total_columns = len(columns)

        for i, col in enumerate(columns):
            status_text.text(f"Generating: {col} ({i+1}/{total_columns})")
            progress_bar.progress(50)
            synthetic[col] = self._generate_from_column_name(col, num_rows)

        synthetic_df = pd.DataFrame(synthetic)

        status_text.text("Processing relationships between columns...")
        progress_bar.progress(75)

        synthetic_df = self._handle_age_dob_relationship(synthetic_df)

        progress_bar.progress(100)

        status_text.text("Generation complete!")

        with st.expander("Generated Data Preview", expanded=True):
            st.dataframe(synthetic_df.head(10), use_container_width=True)

        return synthetic_df
```

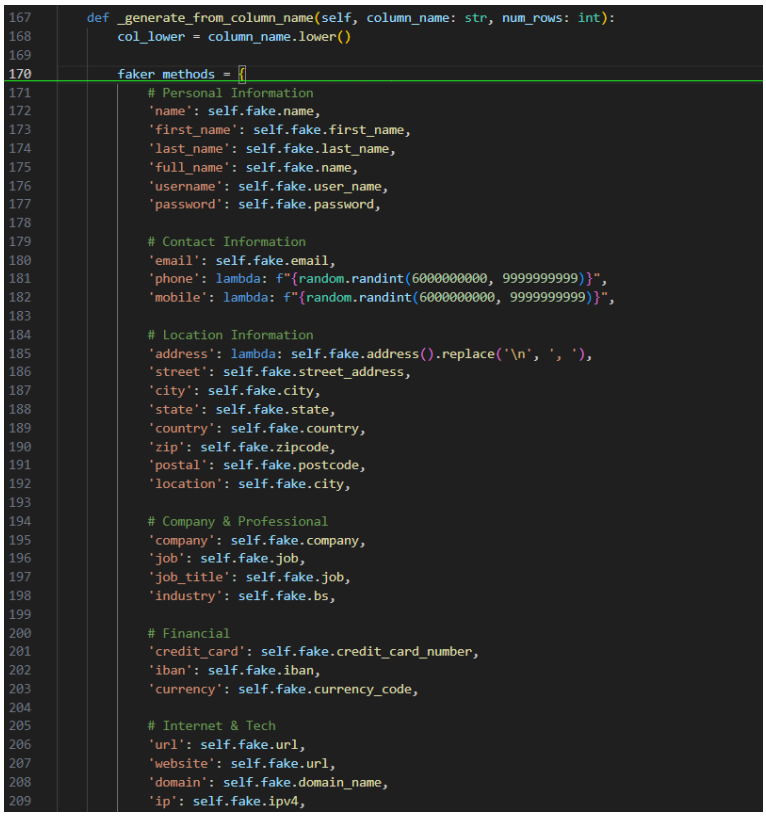
Figure 7: Data Generator from Column Names

- Implement configurable parameters for number of rows, data types, and randomization logic.

```python
    def _generate_numeric_data(self, column_data: pd.Series, num_rows: int):
        clean_data = column_data.dropna()

        if len(clean_data) == 0:
            if pd.api.types.is_integer_dtype(column_data):
                return np.random.randint(0, 100, num_rows)
            else:
                return np.round(np.random.uniform(0, 100, num_rows), 2)

        min_val = clean_data.min()
        max_val = clean_data.max()
        mean_val = clean_data.mean()
        std_val = clean_data.std()

        if std_val > 0:
            generated = np.random.normal(mean_val, std_val, num_rows)
        else:
            generated = np.random.uniform(min_val, max_val, num_rows)

        generated = np.clip(generated, min_val * 0.8, max_val * 1.2)

        if pd.api.types.is_integer_dtype(column_data):
            return generated.astype(int)
        else:
            return np.round(generated, 2)

    def _generate_text_data(self, column_name: str, column_data: pd.Series, num_rows: int):
        clean_data = column_data.dropna()

        if len(clean_data) > 0 and clean_data.nunique() <= 15:
            return np.random.choice(clean_data.unique(), num_rows)

        return self._generate_from_column_name(column_name, num_rows)
```

Figure 8: Numeric data processing

- Integrate Faker to simulate text, numeric, categorical, and temporal data types.



```python
    def _generate_from_column_name(self, column_name: str, num_rows: int):
        col_lower = column_name.lower()

        faker_methods = {
            # Personal Information
            'name': self.fake.name,
            'first_name': self.fake.first_name,
            'last_name': self.fake.last_name,
            'full_name': self.fake.name,
            'username': self.fake.user_name,
            'password': self.fake.password,

            # Contact Information
            'email': self.fake.email,
            'phone': lambda: f"{random.randint(6000000000, 9999999999)}",
            'mobile': lambda: f"{random.randint(6000000000, 9999999999)}",

            # Location Information
            'address': lambda: self.fake.address().replace('\n', ', '),
            'street': self.fake.street_address,
            'city': self.fake.city,
            'state': self.fake.state,
            'country': self.fake.country,
            'zip': self.fake.zipcode,
            'postal': self.fake.postcode,
            'location': self.fake.city,

            # Company & Professional
            'company': self.fake.company,
            'job': self.fake.job,
            'job_title': self.fake.job,
            'industry': self.fake.bs,

            # Financial
            'credit_card': self.fake.credit_card_number,
            'iban': self.fake.iban,
            'currency': self.fake.currency_code,

            # Internet & Tech
            'url': self.fake.url,
            'website': self.fake.url,
            'domain': self.fake.domain_name,
            'ip': self.fake.ipv4,
```

Figure 9: Faker Mapping

```python
            # Dates & Times
            'date': self.fake.date,
            'time': self.fake.time,
            'year': lambda: self.fake.year(),
            'month': lambda: self.fake.month_name(),

            # Date of Birth variations
            'dob': self.fake.date_of_birth,
            'date_of_birth': self.fake.date_of_birth,
            'birth_date': self.fake.date_of_birth,
            'birthdate': self.fake.date_of_birth,
            'birthday': self.fake.date_of_birth,

            # Products & Commerce
            'product': self.fake.word,
            'brand': self.fake.company,
            'color': self.fake.color_name,

            # Education
            'school': self.fake.company,
            'university': self.fake.company,
            'grade': lambda: random.choice(['A', 'B', 'C', 'D', 'F']),

            # Medical
            'hospital': self.fake.company,
            'doctor': self.fake.name,
            'disease': lambda: random.choice(['Flu', 'Cold', 'Headache', 'Fever', 'Allergy']),

            # Vehicles
            'car': lambda: f"{self.fake.company()} {self.fake.word()}",
            'license': self.fake.license_plate,

            # Identification
            'id': lambda: f"ID_{self.fake.random_int(1000, 9999)}",
            'ssn': self.fake.ssn,
            'passport': self.fake.passport_number,

            # Numeric types
            'age': lambda: random.randint(18, 70),
            'salary': lambda: random.randint(30000, 150000),
            'price': lambda: round(random.uniform(10, 1000), 2),
            'quantity': lambda: random.randint(1, 100),
```

Figure 10: Faker Mapping

```
            # Numeric types
            'age': lambda: random.randint(18, 70),
            'salary': lambda: random.randint(30000, 150000),
            'price': lambda: round(random.uniform(10, 1000), 2),
            'quantity': lambda: random.randint(1, 100),
            'score': lambda: random.randint(0, 100),
            'rating': lambda: random.randint(1, 5),
            'serial': lambda: f"SN{self.fake.random_number(digits=8)}",

            # Boolean types
            'is_': lambda: random.choice([True, False]),
            'has_': lambda: random.choice([True, False]),
            'active': lambda: random.choice([True, False]),
            'status': lambda: random.choice(['Active', 'Inactive', 'Pending']),
        }

        for pattern, faker_method in faker_methods.items():
            if pattern in col_lower:
                try:
                    return [faker_method() for _ in range(num_rows)]
                except:
                    continue

        if any(word in col_lower for word in ['first', 'given']):
            return [self.fake.first_name() for _ in range(num_rows)]
        elif any(word in col_lower for word in ['last', 'surname', 'family']):
            return [self.fake.last_name() for _ in range(num_rows)]
        elif any(word in col_lower for word in ['street', 'road', 'avenue']):
            return [self.fake.street_address() for _ in range(num_rows)]
        elif any(word in col_lower for word in ['state', 'province', 'region']):
            return [self.fake.state() for _ in range(num_rows)]
        elif any(word in col_lower for word in ['gender', 'sex']):
            return [random.choice(['Male', 'Female']) for _ in range(num_rows)]

        elif any(word in col_lower for word in ['number', 'count', 'total', 'amount']):
            return [random.randint(1, 1000) for _ in range(num_rows)]
        elif any(word in col_lower for word in ['percent', 'percentage', 'rate']):
            return [round(random.uniform(0, 100), 2) for _ in range(num_rows)]

        else:
            return [self.fake.word() for _ in range(num_rows)]
```

Figure 11: Faker Mapping

Activity 2.2: Data Analysis and Validation Module (validation.py)

- Develop analytical functions to calculate core statistical metrics such as mean, median, mode, variance, and correlation.

```
class DataValidator:
    def __init__(self):
        pass

    def validate_synthetic_data(self, original_df: pd.DataFrame, synthetic_df: pd.DataFrame, alpha: float = 0.05):
        st.subheader("Validation and Quality Metrics")

        validation_results = {
            'columns': [],
            'ks_statistic': [],
            'ks_pvalue': [],
            'ks_significant': [],
            'mean_original': [],
            'mean_synthetic': [],
            'mean_diff': [],
            'mean_diff_percent': []
        }

        common_columns = set(original_df.columns) & set(synthetic_df.columns)

        for col in common_columns:
            if not pd.api.types.is_numeric_dtype(original_df[col]) or not pd.api.types.is_numeric_dtype(synthetic_df[col]):
                continue

            orig_data = original_df[col].dropna()
            synth_data = synthetic_df[col].dropna()

            if len(orig_data) == 0 or len(synth_data) == 0:
                continue

            ks_stat, ks_pvalue = ks_2samp(orig_data, synth_data)

            mean_orig = orig_data.mean()
            mean_synth = synth_data.mean()
            mean_diff = abs(mean_orig - mean_synth)
            mean_diff_percent = (mean_diff / abs(mean_orig)) * 100 if mean_orig != 0 else 0

            validation_results['columns'].append(col)
            validation_results['ks_statistic'].append(ks_stat)
            validation_results['ks_pvalue'].append(ks_pvalue)
            validation_results['ks_significant'].append(ks_pvalue < alpha)
            validation_results['mean_original'].append(mean_orig)
            validation_results['mean_synthetic'].append(mean_synth)
            validation_results['mean_diff'].append(mean_diff)
            validation_results['mean_diff_percent'].append(mean_diff_percent)

        validation_df = pd.DataFrame(validation_results)
```

Figure 12: Validation of dataset

- Implement comparison logic for real vs synthetic data distributions to validate fidelity.

```
        if len(validation_df) > 0:
            col1, col2, col3 = st.columns(3)

            significant_cols = validation_df['ks_significant'].sum()
            total_cols = len(validation_df)
            quality_score = self.calculate_quality_score(validation_df)

            with col1:
                st.metric("Quality Score", f"{quality_score:.1f}/100")

            with col2:
                st.metric("Distribution Match", f"{(total_cols - significant_cols)}/{total_cols}")

            with col3:
                st.metric("Avg Mean Difference", f"{validation_df['mean_diff_percent'].mean():.2f}%")

            if quality_score >= 80:
                st.success("EXCELLENT: Synthetic data closely matches original distribution")
            elif quality_score >= 60:
                st.warning("GOOD: Synthetic data reasonably matches original distribution")
            elif quality_score >= 40:
                st.info("FAIR: Some differences detected in synthetic data")
            else:
                st.error("POOR: Significant differences in synthetic data")

            st.subheader("Detailed Column Analysis")
            display_df = validation_df.copy()
            display_df['ks_significant'] = display_df['ks_significant'].map({True: '❌', False: '✅'})
            display_df = display_df.round(4)
            st.dataframe(display_df, use_container_width=True)

            self._plot_distributions(original_df, synthetic_df, common_columns)

        else:
            st.warning("No numeric columns available for validation.")

        return validation_df
```

Figure 13: Comparing Logic

- Structure results using **Pandas DataFrames** for compatibility with the visualization layer.

```
    def _plot_distributions(self, original_df: pd.DataFrame, synthetic_df: pd.DataFrame, common_columns: set):
        st.subheader("Distribution Comparison")
        for col in common_columns:
            if pd.api.types.is_numeric_dtype(original_df[col]):
                fig = go.Figure()
                fig.add_trace(go.Histogram(
                    x=original_df[col].dropna(),
                    name='Original',
                    opacity=0.7,
                    nbinsx=20
                ))
                fig.add_trace(go.Histogram(
                    x=synthetic_df[col].dropna(),
                    name='Synthetic',
                    opacity=0.7,
                    nbinsx=20
                ))
                fig.update_layout(
                    title=f'Distribution of {col}',
                    xaxis_title=col,
                    yaxis_title='Frequency',
                    barmode='overlay'
                )
                st.plotly_chart(fig, use_container_width=True)
```

Figure 14: Plotting Function

- Integrate automated summary reports highlighting statistical differences and similarities.

```python
    def calculate_quality_score(self, validation_df: pd.DataFrame) -> float:
        if len(validation_df) == 0:
            return 0

        ks_penalty = validation_df['ks_significant'].mean() * 40
        mean_diff_penalty = min(30, validation_df['mean_diff_percent'].mean() / 2)
        base_score = 100
        quality_score = base_score - ks_penalty - mean_diff_penalty

        return max(0, quality_score)
```

Figure 15: Score Calculation

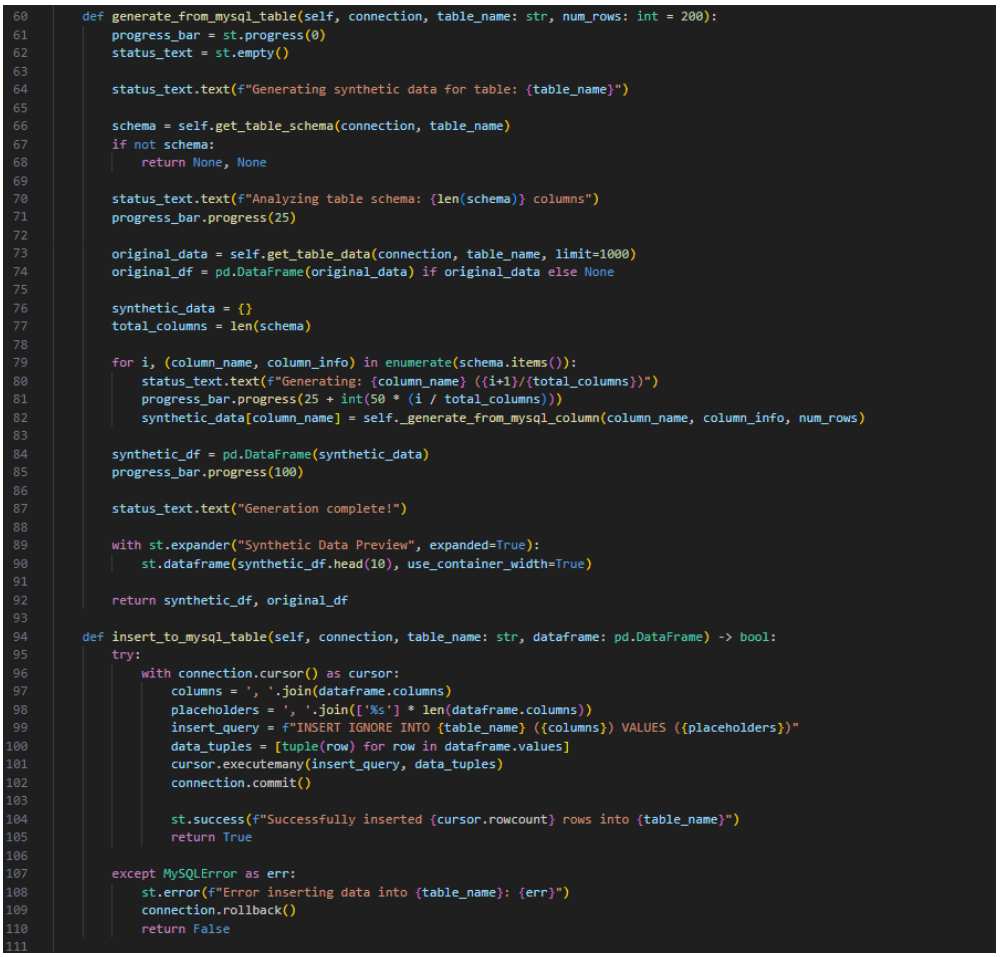
Activity 2.3: Database Integration (database_handler.py)

- Implement secure and optimized **MySQL** database integration using **PyMySQL**.
- Allow users to:
  - Retrieve existing table schemas automatically.
  - Generate data conforming to those schemas.

```python
class DatabaseHandler:
    def __init__(self, data_generator):
        self.generator = data_generator

    def connect_to_mysql(self, host: str, user: str, password: str, database: str):
        try:
            connection = pymysql.connect(
                host=host,
                user=user,
                password=password,
                database=database,
                charset='utf8mb4',
                cursorclass=pymysql.cursors.DictCursor
            )
            st.success("Successfully connected to MySQL database!")
            return connection
        except MySQLError as err:
            st.error(f"Failed to connect to database: {err}")
            return None

    def get_mysql_tables(self, connection) -> List[str]:
        try:
            with connection.cursor() as cursor:
                cursor.execute("SHOW TABLES")
                tables = [list(table.values())[0] for table in cursor.fetchall()]
                return tables
        except MySQLError as err:
            st.error(f"Error fetching tables: {err}")
            return []

    def get_table_schema(self, connection, table_name: str) -> Dict[str, Any]:
        try:
            with connection.cursor() as cursor:
                cursor.execute(f"DESCRIBE {table_name}")
                schema = cursor.fetchall()

                schema_info = {}
                for column in schema:
                    schema_info[column['Field']] = {
                        'field': column['Field'],
                        'type': column['Type'],
                        'null': column['Null'],
                        'key': column['Key'],
                        'default': column['Default'],
                        'extra': column['Extra']
                    }
                return schema_info
        except MySQLError as err:
            st.error(f"Error fetching schema for {table_name}: {err}")
            return {}
```

Figure 16: SQL operation function

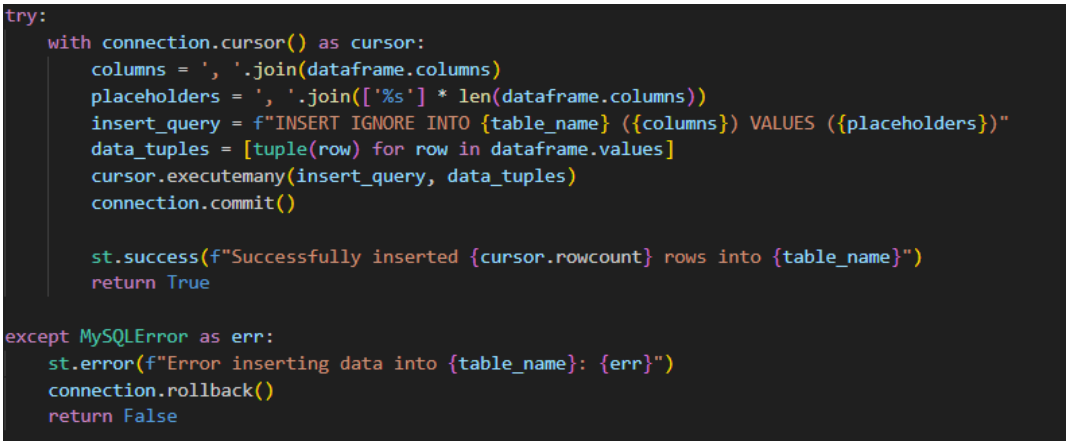
- Insert synthetic records directly into connected databases.



```python
    def generate_from_mysql_table(self, connection, table_name: str, num_rows: int = 200):
        progress_bar = st.progress(0)
        status_text = st.empty()

        status_text.text(f"Generating synthetic data for table: {table_name}")

        schema = self.get_table_schema(connection, table_name)
        if not schema:
            return None, None

        status_text.text(f"Analyzing table schema: {len(schema)} columns")
        progress_bar.progress(25)

        original_data = self.get_table_data(connection, table_name, limit=1000)
        original_df = pd.DataFrame(original_data) if original_data else None

        synthetic_data = {}
        total_columns = len(schema)

        for i, (column_name, column_info) in enumerate(schema.items()):
            status_text.text(f"Generating: {column_name} ({i+1}/{total_columns})")
            progress_bar.progress(25 + int(50 * (i / total_columns)))
            synthetic_data[column_name] = self._generate_from_mysql_column(column_name, column_info, num_rows)

        synthetic_df = pd.DataFrame(synthetic_data)
        progress_bar.progress(100)

        status_text.text("Generation complete!")

        with st.expander("Synthetic Data Preview", expanded=True):
            st.dataframe(synthetic_df.head(10), use_container_width=True)

        return synthetic_df, original_df

    def insert_to_mysql_table(self, connection, table_name: str, dataframe: pd.DataFrame) -> bool:
        try:
            with connection.cursor() as cursor:
                columns = ', '.join(dataframe.columns)
                placeholders = ', '.join(['%s'] * len(dataframe.columns))
                insert_query = f"INSERT IGNORE INTO {table_name} ({columns}) VALUES ({placeholders})"
                data_tuples = [tuple(row) for row in dataframe.values]
                cursor.executemany(insert_query, data_tuples)
                connection.commit()

                st.success(f"Successfully inserted {cursor.rowcount} rows into {table_name}")
                return True

        except MySQLError as err:
            st.error(f"Error inserting data into {table_name}: {err}")
            connection.rollback()
            return False

```

Figure 17: Creating and Inserting the dummy data to database

- Include robust error handling for connection failures, authentication issues, and SQL exceptions.



```python
try:
    with connection.cursor() as cursor:
        columns = ', '.join(dataframe.columns)
        placeholders = ', '.join(['%s'] * len(dataframe.columns))
        insert_query = f"INSERT IGNORE INTO {table_name} ({columns}) VALUES ({placeholders})"
        data_tuples = [tuple(row) for row in dataframe.values]
        cursor.executemany(insert_query, data_tuples)
        connection.commit()

        st.success(f"Successfully inserted {cursor.rowcount} rows into {table_name}")
        return True

except MySQLError as err:
    st.error(f"Error inserting data into {table_name}: {err}")
    connection.rollback()
    return False
```

Figure 18: Exception Handling

MILESTONE 3: Streamlit UI Implementation and User Interaction

This milestone focuses on building an **interactive, intuitive user interface** that connects backend intelligence with real-time visualization. The UI allows users to configure schemas, generate data, validate results, and interact with synthetic datasets dynamically.

Activity 3.1: Layout Design and Navigation

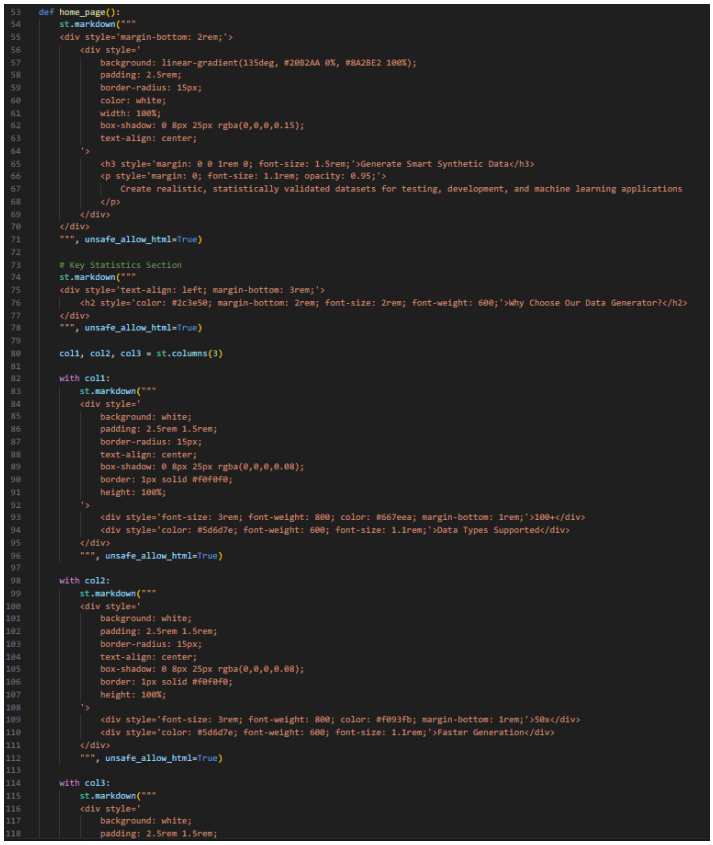
- Develop a multi-tabbed interface for:

  CSV Extension
  Dummy Data Creation
  SQL Operations

```
# Header
st.markdown('<h1 class="main-header">Synthetic Data Factory</h1>', unsafe_allow_html=True)
st.markdown('<h6 class="main-header">Generate intelligent, realistic, and statistically balanced synthetic data — with precision and style.</h1>', unsafe_allow_html=True
st.markdown(" ")

# Main Tabs Navigation
tab1, tab2, tab3, tab4, tab5 = st.tabs([
    "Home",
    "CSV Extension",
    "Dummy Data Creation",
    "MySQL Operations",
    "About"
])

with tab1:
    home_page()
with tab2:
    csv_extension_page()
with tab3:
    dummy_data_creation_page()
with tab4:
    mysql_operations_page()
with tab5:
    about_page()
```

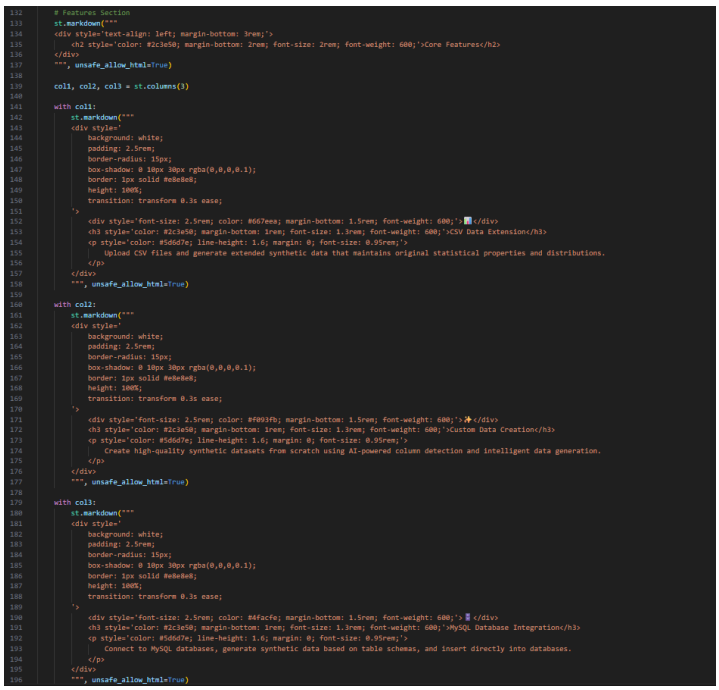
Figure 19: Navigation buttons

- Used columns, custom Markdown, and HTML formatting for adding custom style and structure.



```
def home_page():
    st.markdown("""
    <div style='margin-bottom: 2rem;'>
        <div style='
            background: linear-gradient(135deg, #2082AA 0%, #8A2BE2 100%);
            padding: 2.5rem;
            border-radius: 15px;
            color: white;
            width: 100%;
            box-shadow: 0 8px 25px rgba(0,0,0,0.15);
            text-align: center;
        '>
            <h3 style='margin: 0 0 1rem 0; font-size: 1.5rem;'>Generate Smart Synthetic Data</h3>
            <p style='margin: 0; font-size: 1.1rem; opacity: 0.95;'>
                Create realistic, statistically validated datasets for testing, development, and machine learning applications
            </p>
        </div>
    </div>
    """, unsafe_allow_html=True)

    # Key Statistics Section
    st.markdown("""
    <div style='text-align: left; margin-bottom: 3rem;'>
        <h2 style='color: #2c3e50; margin-bottom: 2rem; font-size: 2rem; font-weight: 600;'>Why Choose Our Data Generator?</h2>
    </div>
    """, unsafe_allow_html=True)

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

    with col1:
        st.markdown("""
        <div style='
            background: white;
            padding: 2.5rem 1.5rem;
            border-radius: 15px;
            text-align: center;
            box-shadow: 0 8px 25px rgba(0,0,0,0.08);
            border: 1px solid #f0f0f0;
            height: 100%;
        '>
            <div style='font-size: 3rem; font-weight: 800; color: #667eea; margin-bottom: 1rem;'>100+</div>
            <div style='color: #5d6d7e; font-weight: 600; font-size: 1.1rem;'>Data Types Supported</div>
        </div>
        """, unsafe_allow_html=True)

    with col2:
        st.markdown("""
        <div style='
            background: white;
            padding: 2.5rem 1.5rem;
            border-radius: 15px;
            text-align: center;
            box-shadow: 0 8px 25px rgba(0,0,0,0.08);
            border: 1px solid #f0f0f0;
            height: 100%;
        '>
            <div style='font-size: 3rem; font-weight: 800; color: #f093fb; margin-bottom: 1rem;'>50x</div>
            <div style='color: #5d6d7e; font-weight: 600; font-size: 1.1rem;'>Faster Generation</div>
        </div>
        """, unsafe_allow_html=True)

    with col3:
        st.markdown("""
        <div style='
            background: white;
            padding: 2.5rem 1.5rem;
```

Figure 20: Column separation output



```
    # Features Section
    st.markdown("""
    <div style='text-align: left; margin-bottom: 3rem;'>
        <h2 style='color: #2c3e50; margin-bottom: 2rem; font-size: 2rem; font-weight: 600;'>Core Features</h2>
    </div>
    """, unsafe_allow_html=True)

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

    with col1:
        st.markdown("""
        <div style='
            background: white;
            padding: 2.5rem;
            border-radius: 15px;
            box-shadow: 0 10px 30px rgba(0,0,0,0.1);
            border: 1px solid #e8e8e8;
            height: 100%;
            transition: transform 0.3s ease;
        '>
            <div style='font-size: 2.5rem; color: #667eea; margin-bottom: 1.5rem; font-weight: 600;'>📊</div>
            <h3 style='color: #2c3e50; margin-bottom: 1rem; font-size: 1.3rem; font-weight: 600;'>CSV Data Extension</h3>
            <p style='color: #5d6d7e; line-height: 1.6; margin: 0; font-size: 0.95rem;'>
                Upload CSV files and generate extended synthetic data that maintains original statistical properties and distributions.
            </p>
        </div>
        """, unsafe_allow_html=True)

    with col2:
        st.markdown("""
        <div style='
            background: white;
            padding: 2.5rem;
            border-radius: 15px;
            box-shadow: 0 10px 30px rgba(0,0,0,0.1);
            border: 1px solid #e8e8e8;
            height: 100%;
            transition: transform 0.3s ease;
        '>
            <div style='font-size: 2.5rem; color: #f093fb; margin-bottom: 1.5rem; font-weight: 600;'>✨</div>
            <h3 style='color: #2c3e50; margin-bottom: 1rem; font-size: 1.3rem; font-weight: 600;'>Custom Data Creation</h3>
            <p style='color: #5d6d7e; line-height: 1.6; margin: 0; font-size: 0.95rem;'>
                Create high-quality synthetic datasets from scratch using AI-powered column detection and intelligent data generation.
            </p>
        </div>
        """, unsafe_allow_html=True)

    with col3:
        st.markdown("""
        <div style='
            background: white;
            padding: 2.5rem;
            border-radius: 15px;
            box-shadow: 0 10px 30px rgba(0,0,0,0.1);
            border: 1px solid #e8e8e8;
            height: 100%;
            transition: transform 0.3s ease;
        '>
            <div style='font-size: 2.5rem; color: #4facfe; margin-bottom: 1.5rem; font-weight: 600;'>🗄️</div>
            <h3 style='color: #2c3e50; margin-bottom: 1rem; font-size: 1.3rem; font-weight: 600;'>MySQL Database Integration</h3>
            <p style='color: #5d6d7e; line-height: 1.6; margin: 0; font-size: 0.95rem;'>
                Connect to MySQL databases, generate synthetic data based on table schemas, and insert directly into databases.
            </p>
        </div>
        """, unsafe_allow_html=True)
```

Figure 21: Markdowns

- Integrate visual separators and typography for a professional, data-centric dashboard experience.

Activity 3.2: Input System and Schema Configuration

Added text area for user input including:

- Column names and types format for CSV extension page

```python
def csv_extension_page():
    st.markdown("""...

    uploaded_file = st.file_uploader("Upload CSV File", type=['csv'])
    st.markdown(" ")

    if uploaded_file is not None:
        with open("temp_upload.csv", "wb") as f:
            f.write(uploaded_file.getbuffer())

        original_df = pd.read_csv("temp_upload.csv")

        col1, col2 = st.columns(2)

        with col1:
            st.metric("Original Rows", len(original_df))
        with col2:
            st.metric("Columns", len(original_df.columns))

        with st.expander("Original Data Preview", expanded=True):
            st.dataframe(original_df.head(), use_container_width=True)

        st.markdown("### Generation Settings")
        col1, col2 = st.columns(2)

        with col1:
            num_rows = st.number_input("Number of Synthetic Rows", min_value=1, value=200, step=100)
        with col2:
            output_file = st.text_input("Output Filename", value="synthetic_data.csv")

        validate = st.checkbox("Run Validation", value=True)

        if st.button("Generate Synthetic Data", type="primary", use_container_width=True):
            with st.spinner("Generating synthetic data..."):
                synthetic_df, original_df_full = st.session_state.generator.generate_from_csv(
                    "temp_upload.csv",
                    num_rows,
                    validate=False
                )

                if synthetic_df is not None:
                    st.session_state.synthetic_data = synthetic_df

                    if validate and original_df_full is not None:
                        st.session_state.validator.validate_synthetic_data(original_df_full, synthetic_df)

        if st.session_state.synthetic_data is not None:
            st.markdown("### Download Generated Data")
            csv = st.session_state.synthetic_data.to_csv(index=False)
            st.download_button(...
```

Figure 22: Input to CSV extension

- Row column name for generation for Dummy data creation

```python
def dummy_data_creation_page():
    st.markdown("""...

    columns_input = st.text_area(
        "Enter Column Names (one per line or comma-separated)",
        height=80,
        placeholder="name, email, age, salary, city, phone_number"
    )

    col1, col2 = st.columns(2)

    with col1:
        num_rows = st.number_input("Number of Rows", min_value=1, value=200, step=100)

    with col2:
        output_file = st.text_input("Output Filename", value="dummy_data.csv")

    if st.button("Generate dummy Data", type="primary", use_container_width=True):
        if columns_input:
            columns = []
            if ',' in columns_input:
                columns = [col.strip() for col in columns_input.split(',') if col.strip()]
            else:
                columns = [col.strip() for col in columns_input.split('\n') if col.strip()]

            if columns:
                with st.spinner(f"Generating {num_rows} rows with {len(columns)} columns..."):
                    synthetic_df = st.session_state.generator.generate_from_columns(columns, num_rows)
                    st.session_state.synthetic_data = synthetic_df

                if st.session_state.synthetic_data is not None:
                    st.markdown("### Download Generated Data")
                    csv = st.session_state.synthetic_data.to_csv(index=False)
                    st.download_button(
                        label="Download CSV",
                        data=csv,
                        file_name=output_file,
                        mime="text/csv",
                        use_container_width=True
                    )
            else:
                st.error("Please enter at least one column name.")
        else:
            st.error("Please enter column names.")
```

Figure 23: Input to the Dummy data creation

- Database details (host, user, password, table)

```python
def mysql_operations_page():

    with col1:
        host = st.text_input("Host", value="localhost")
        password = st.text_input("Password", type="password")
    with col2:
        user = st.text_input("Username", value="root")
        database = st.text_input("Database", value="test")

    if st.button("Connect to Database", use_container_width=True):
        with st.spinner("Connecting to database..."):
            connection = st.session_state.db_handler.connect_to_mysql(host, user, password, database)
            if connection:
                st.session_state.db_connection = connection
                st.session_state.db_tables = st.session_state.db_handler.get_mysql_tables(connection)

    if 'db_connection' in st.session_state and st.session_state.db_connection:
        st.success("Connected to database!")

        if st.session_state.db_tables:
            st.subheader("Available Tables")
            selected_table = st.selectbox("Select Table", st.session_state.db_tables)

            if selected_table:
                # Show table schema
                schema = st.session_state.db_handler.get_table_schema(st.session_state.db_connection, selected_table)

                st.subheader("Table Schema")
                schema_df = pd.DataFrame.from_dict(schema, orient='index')
                st.dataframe(schema_df, use_container_width=True)

                num_rows = st.number_input("Number of Rows", min_value=1, value=100, key="mysql_rows")
                validate = st.checkbox("Run Validation", value=True, key="mysql_validate")
                insert_db = st.checkbox("Insert into Database", value=False)

                if st.button("Generate from Table Schema", type="primary", use_container_width=True):
                    with st.spinner(f"Generating data for table '{selected_table}'..."):
                        synthetic_df, original_df = st.session_state.db_handler.generate_from_mysql_table(
                            st.session_state.db_connection,
                            selected_table,
                            num_rows
                        )
                        st.session_state.synthetic_data = synthetic_df

                        if synthetic_df is not None:
                            if validate and original_df is not None and len(original_df) > 0:
                                st.info(f"Validating against {len(original_df)} original rows...")
                                st.session_state.validator.validate_synthetic_data(original_df, synthetic_df)

                            if insert_db:
                                success = st.session_state.db_handler.insert_to_mysql_table(
                                    st.session_state.db_connection,
                                    selected_table,
                                    synthetic_df
                                )
```

Figure 24: Database detail input section

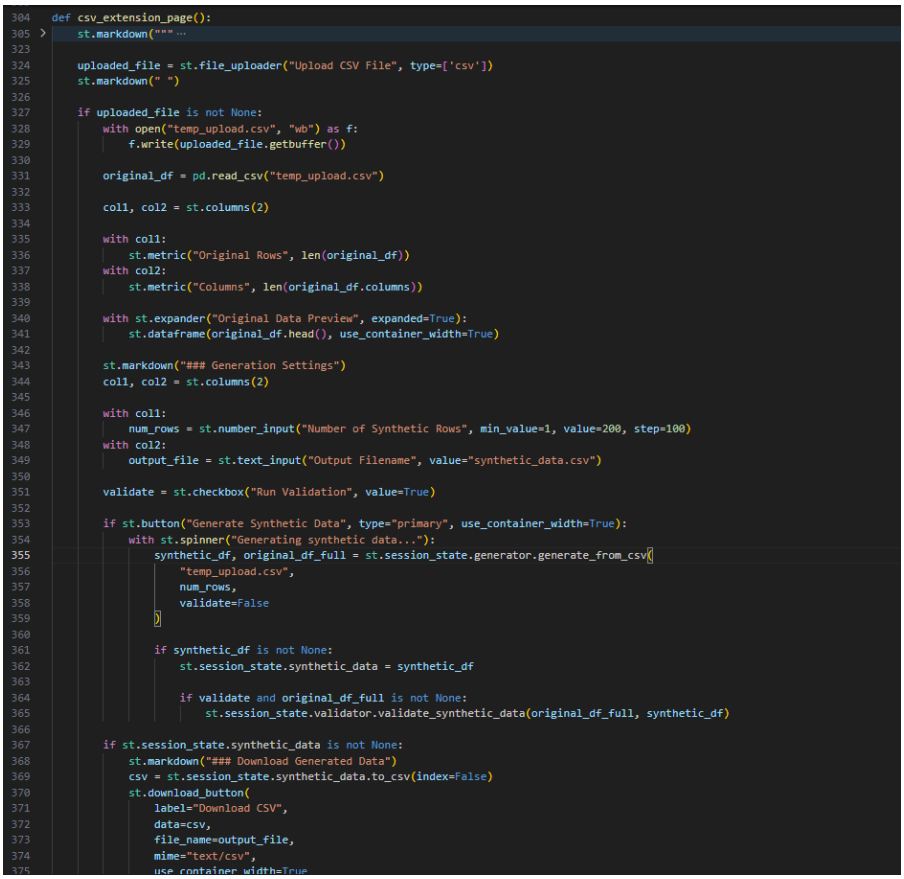
- Integrate progress indicators and success messages using Streamlit's feedback components (st.spinner, st.success, etc.).
- Persist inputs via session state for consistent user experience.

```python
# Initialize session state
if 'generator' not in st.session_state:
    st.session_state.generator = UniversalDataGenerator()
if 'db_handler' not in st.session_state:
    st.session_state.db_handler = DatabaseHandler(st.session_state.generator)
if 'validator' not in st.session_state:
    st.session_state.validator = DataValidator()
if 'synthetic_data' not in st.session_state:
    st.session_state.synthetic_data = None
```

Figure 25: Session State

Activity 3.3: Data Visualization and Result Display

- Display generated synthetic data dynamically using **Pandas DataFrames** in Streamlit tables.



```python
def csv_extension_page():
    st.markdown(""" ···

    uploaded_file = st.file_uploader("Upload CSV File", type=['csv'])
    st.markdown(" ")

    if uploaded_file is not None:
        with open("temp_upload.csv", "wb") as f:
            f.write(uploaded_file.getbuffer())

        original_df = pd.read_csv("temp_upload.csv")

        col1, col2 = st.columns(2)

        with col1:
            st.metric("Original Rows", len(original_df))
        with col2:
            st.metric("Columns", len(original_df.columns))

        with st.expander("Original Data Preview", expanded=True):
            st.dataframe(original_df.head(), use_container_width=True)

        st.markdown("### Generation Settings")
        col1, col2 = st.columns(2)

        with col1:
            num_rows = st.number_input("Number of Synthetic Rows", min_value=1, value=200, step=100)
        with col2:
            output_file = st.text_input("Output Filename", value="synthetic_data.csv")

        validate = st.checkbox("Run Validation", value=True)

        if st.button("Generate Synthetic Data", type="primary", use_container_width=True):
            with st.spinner("Generating synthetic data..."):
                synthetic_df, original_df_full = st.session_state.generator.generate_from_csv(
                    "temp_upload.csv",
                    num_rows,
                    validate=False
                )

                if synthetic_df is not None:
                    st.session_state.synthetic_data = synthetic_df

                    if validate and original_df_full is not None:
                        st.session_state.validator.validate_synthetic_data(original_df_full, synthetic_df)

        if st.session_state.synthetic_data is not None:
            st.markdown("### Download Generated Data")
            csv = st.session_state.synthetic_data.to_csv(index=False)
            st.download_button(
                label="Download CSV",
                data=csv,
                file_name=output_file,
                mime="text/csv",
                use_container_width=True
```

Figure 26: Streamlit Output section for CSV extension

- Integrate Matplotlib and Plotly for generating statistical charts histograms with correlation of the inputted dataset.

```python
    def _plot_distributions(self, original_df: pd.DataFrame, synthetic_df: pd.DataFrame, common_columns: set):
        st.subheader("Distribution Comparison")
        for col in common_columns:
            if pd.api.types.is_numeric_dtype(original_df[col]):
                fig = go.Figure()
                fig.add_trace(go.Histogram(
                    x=original_df[col].dropna(),
                    name='Original',
                    opacity=0.7,
                    nbinsx=20
                ))
                fig.add_trace(go.Histogram(
                    x=synthetic_df[col].dropna(),
                    name='Synthetic',
                    opacity=0.7,
                    nbinsx=20
                ))
                fig.update_layout(
                    title=f'Distribution of {col}',
                    xaxis_title=col,
                    yaxis_title='Frequency',
                    barmode='overlay'
                )
                st.plotly_chart(fig, use_container_width=True)
```

Figure 27: Plotting the hist graph

- Show computed metrics like mean, standard deviation, and column count in summary cards.

```
    if uploaded_file is not None:
        with open("temp_upload.csv", "wb") as f:
            f.write(uploaded_file.getbuffer())

        original_df = pd.read_csv("temp_upload.csv")

        col1, col2 = st.columns(2)

        with col1:
            st.metric("Original Rows", len(original_df))
        with col2:
            st.metric("Columns", len(original_df.columns))

        with st.expander("Original Data Preview", expanded=True):
            st.dataframe(original_df.head(), use_container_width=True)

        st.markdown("### Generation Settings")
        col1, col2 = st.columns(2)

        with col1:
            num_rows = st.number_input("Number of Synthetic Rows", min_value=1, value=200, step=100)
        with col2:
            output_file = st.text_input("Output Filename", value="synthetic_data.csv")

        validate = st.checkbox("Run Validation", value=True)

        if st.button("Generate Synthetic Data", type="primary", use_container_width=True):
            with st.spinner("Generating synthetic data..."):
                synthetic_df, original_df_full = st.session_state.generator.generate_from_csv(
                    "temp_upload.csv",
                    num_rows,
                    validate=False
                )

                if synthetic_df is not None:
                    st.session_state.synthetic_data = synthetic_df

                    if validate and original_df_full is not None:
                        st.session_state.validator.validate_synthetic_data(original_df_full, synthetic_df)
```

Figure 28: Evaluation Matrix

MILESTONE 4: Testing, Optimization, and Final Deployment

This final milestone ensures reliability, speed, and stability of the Synthetic Data Factory through rigorous front-end testing, optimization, and deployment preparation.

Activity 4.1: Functional Testing

- Test all UI components including buttons, input forms, chart rendering, and data preview tables.

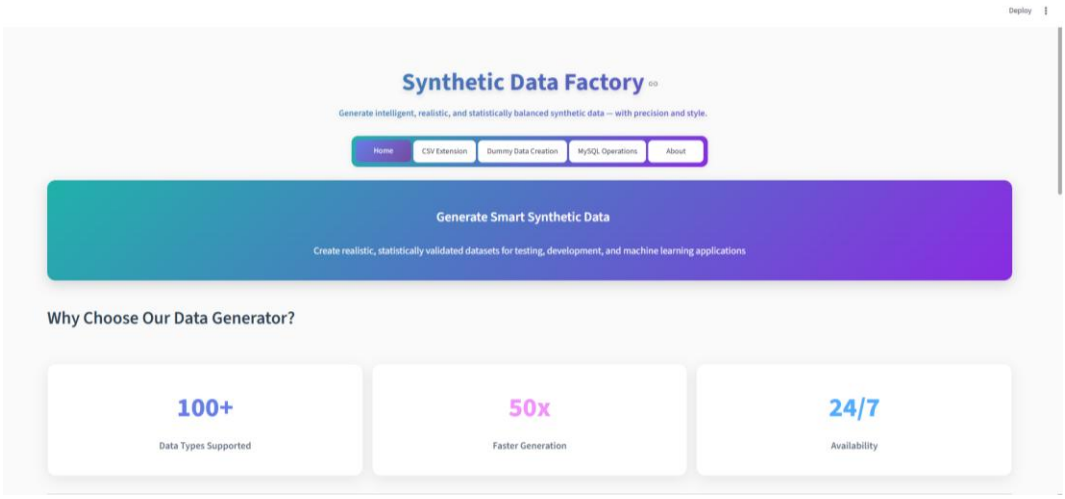


Figure 29: Home Page

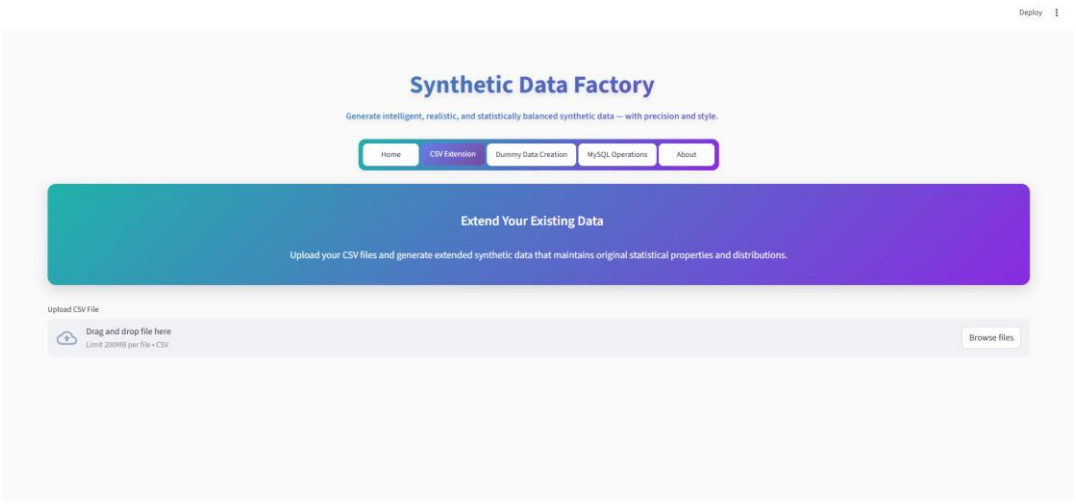


Figure 30: CSV Extention Page

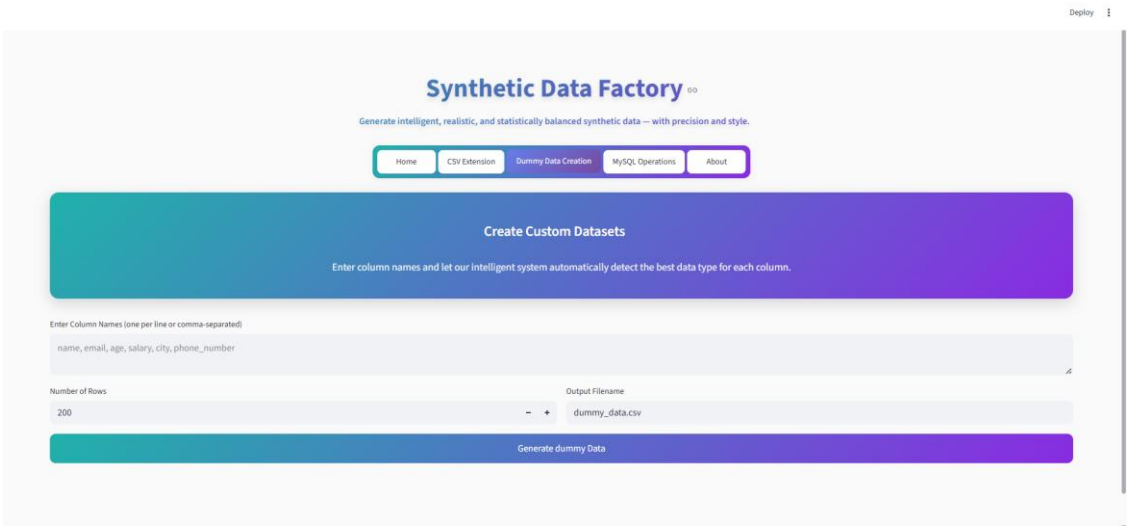


Figure 31: Dummy Data Creation Page

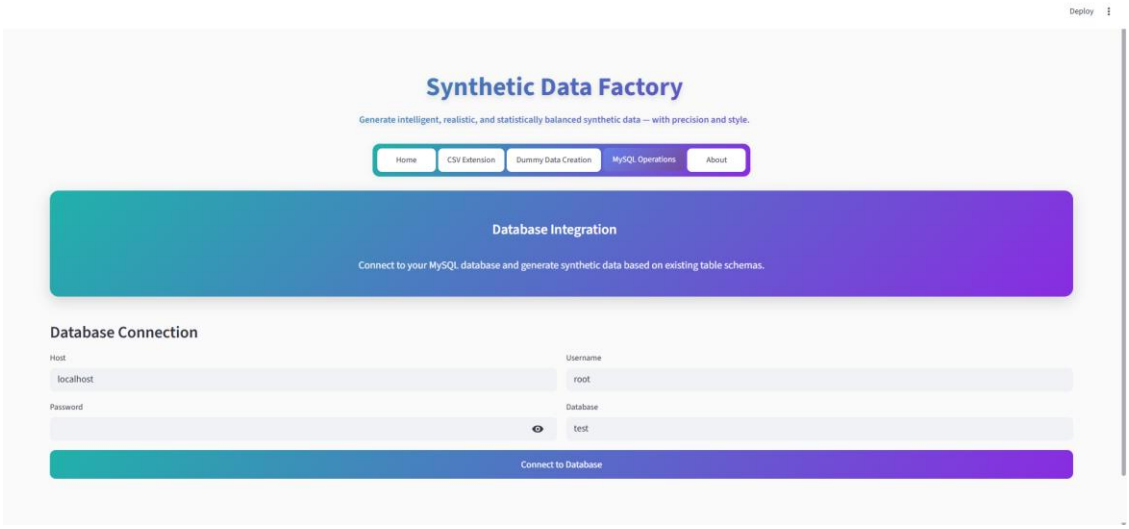


Figure 32: SQL Operation Page

- Ensure data generation is accurate and consistent for both CSV and database modes.

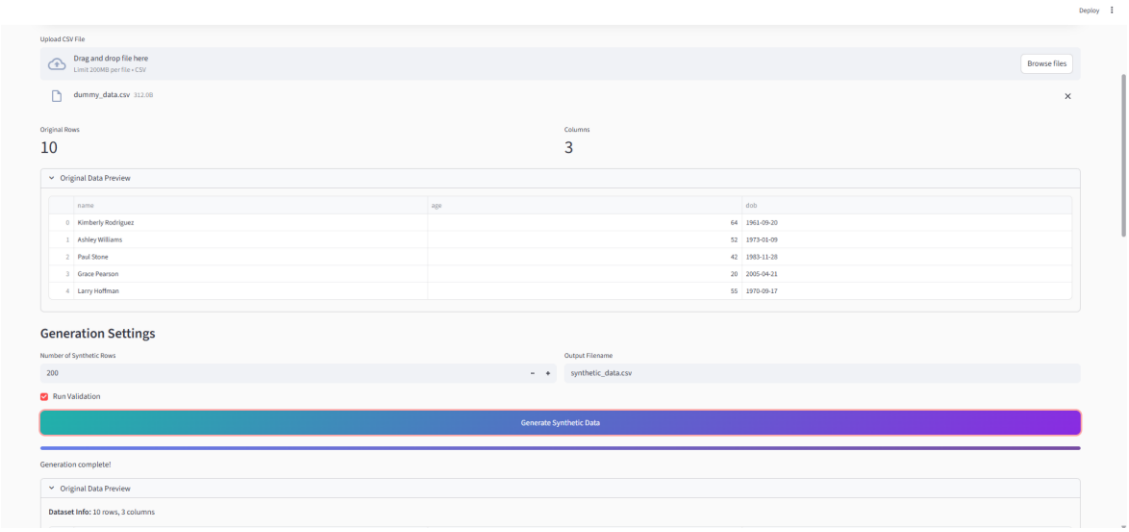


Figure 33: Data Generation from CSV Extension

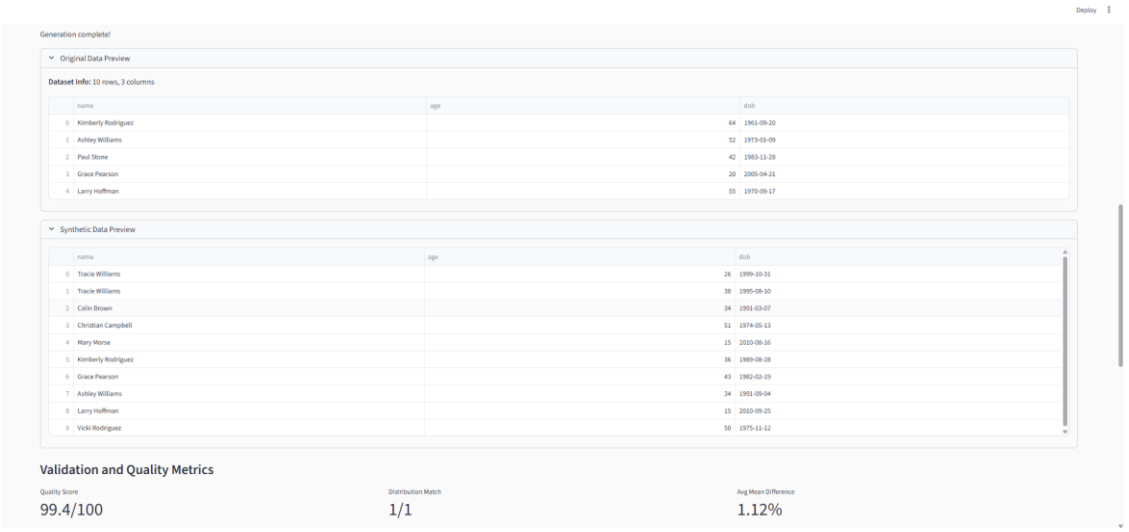


Figure 34: Real VS Synthetic Data

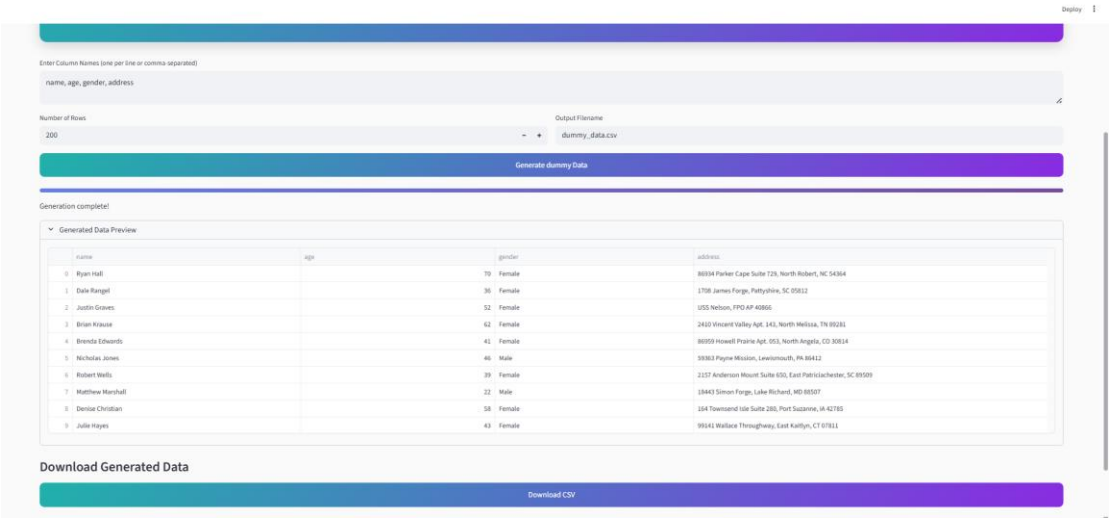


Figure 35: Data Creation from Dummy Data creation Model

- Validate interactive chart responsiveness across different browsers and resolutions.

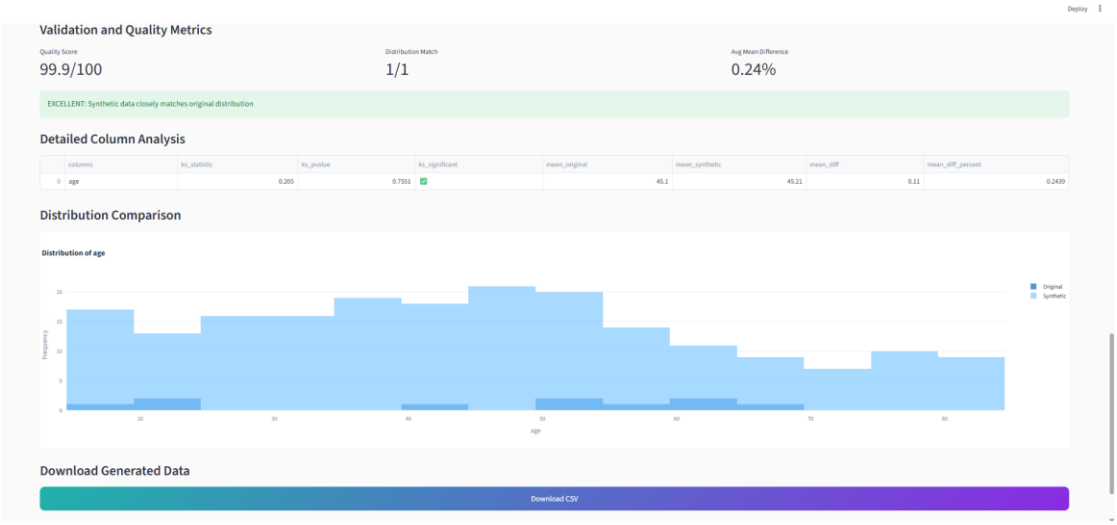


Figure 36: Chart Visualization

Activity 4.2: Integration and Performance Validation

- Test MySQL connection handling and data insertion with large datasets.

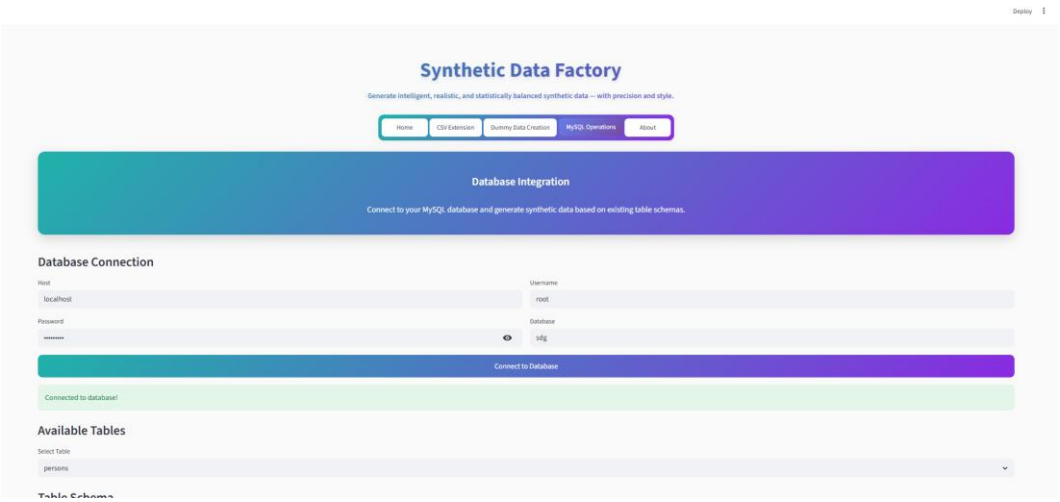


Figure 37: Connection to the Database

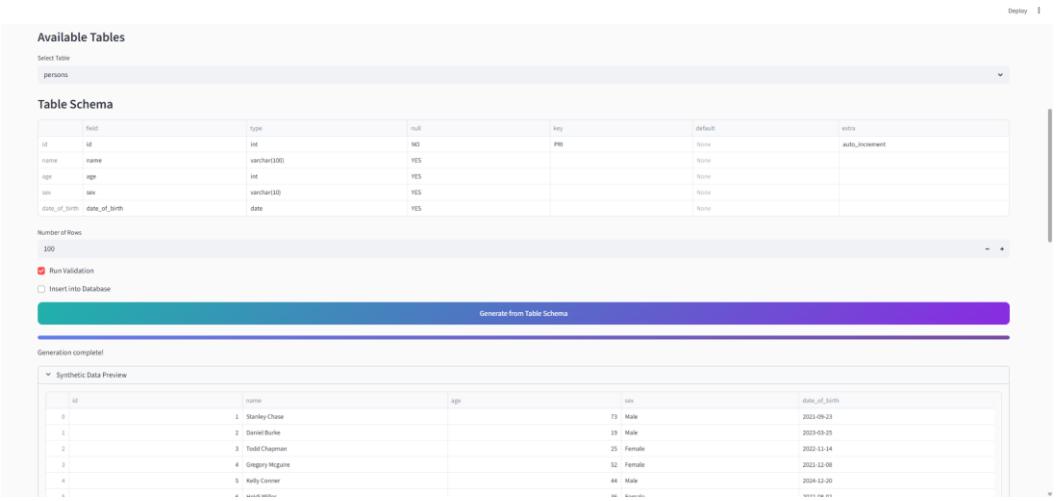


Figure 38: Data Generation from SQL Model

- Configuring the system generation of 10K+ rows efficiently in the Database (MYSQL).

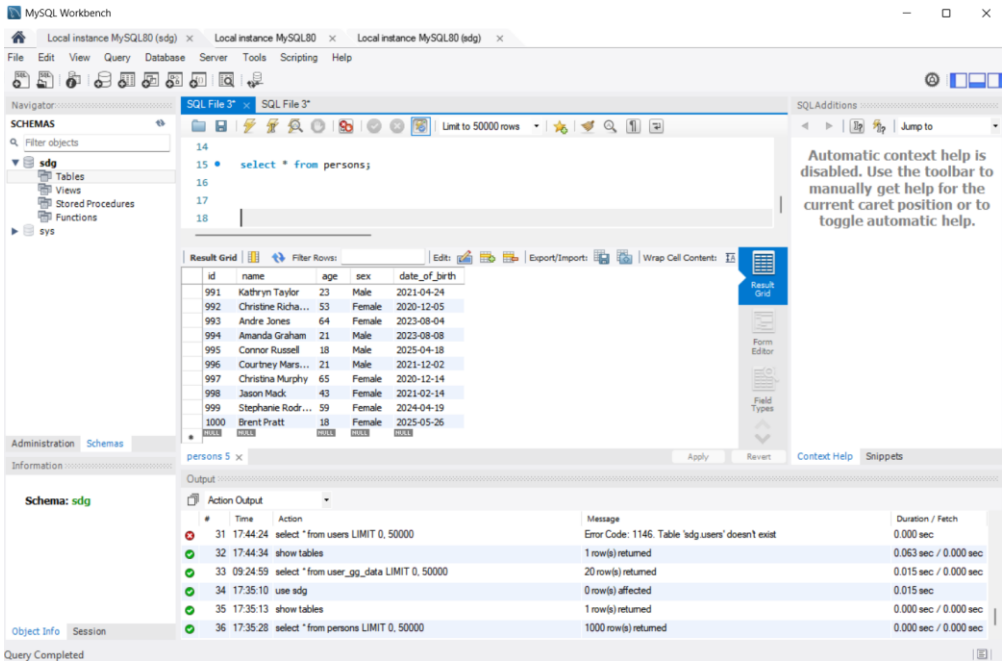


Figure 39: SQL data inserting verification

- Validate that Streamlit's refresh and rerun mechanisms preserve data state correctly with session states.

Activity 4.3: Deployment Preparation and Final Validation

- The project was prepared for deployment with all dependencies documented in requirements.txt and environment setup instructions completed.
- The application was successfully deployed on Streamlit Cloud (or a local server) with proper configuration to ensure scalability and stability.
- Comprehensive end-to-end validation was performed across all modules, confirming UI consistency, functionality, and seamless integration.
- The deployed environment was verified to maintain data integrity, high performance, and responsiveness under real-world usage conditions.

Conclusion

The Synthetic Data Factory (SDF) represents a significant advancement in intelligent data generation, transforming how organizations create, validate, and deploy synthetic datasets for testing, development, and analytics. By integrating AI-powered data generation techniques with robust statistical validation and database compatibility, the system ensures that every dataset is realistic, consistent, and tailored to the user's specific requirements. Whether it's extending existing CSV files, creating custom datasets from scratch, or populating MySQL databases, SDF empowers users to produce high-quality synthetic data efficiently and accurately.

Built on a Streamlit interface, the application delivers a seamless and interactive user experience. Users can intuitively navigate between multiple data generation modes, configure datasets, monitor validation metrics, and download or directly insert synthetic data into databases. The combination of Python's data science libraries, intelligent column detection, and database integration makes SDF both technically sophisticated and practically applicable for real-world workflows.

Looking ahead, Synthetic Data Factory has strong potential for further expansion. Features such as automated pattern detection, advanced anomaly generation, integration with additional database systems, and enhanced analytics dashboards could transform SDF into a comprehensive data simulation platform. Ultimately, this project demonstrates how AI-driven synthetic data generation can bridge gaps in data availability, accelerate development cycles, and empower organizations to make data-driven decisions with confidence and precision.