AnonimaData – Scalable Service for Privacy-Preserving Dataset Anonymization

Valentina de Respinis, Danila Meleleo
Università di Bologna
{valentina.derespinis, danila.meleleo}@studio.unibo.it

Abstract
AnonimaData is a scalable web application that enables privacy-preserving anonymization of tabular datasets through a user-friendly interface. It supports multiple anonymization models—such as k-anonymity, l-diversity, t-closeness, and differential privacy—and allows users to upload data, configure algorithm parameters, and generate anonymized outputs that retain analytical utility.

The platform is built with a cloud-native architecture that ensures both scalability and reliability, making it suitable for secure data sharing in privacy-sensitive scenarios.

1 Introduction

In the era of big data, **sharing datasets for research and analysis** has become increasingly common across domains such as healthcare, finance, marketing and public administration. However, this practice presents significant challenges, especially when it involves **sensitive personal information**.

Even when datasets are stripped of direct identifiers such as names or email addresses, individuals can often still be re-identified. These attacks exploit the combination of seemingly innocuous attributes known as quasi-identifiers, such as age, gender and ZIP code[1]; for example, Latanya Sweeney's famous study showed that 87% of the U.S. population could be uniquely identified using just these three attributes[2]. These risks are particularly relevant when data are made publicly available or shared with third parties. Consider, for example, a hospital or financial institution that holds large volumes of sensitive information about patients or clients. If such an organization is asked to share data with external researchers for medical or economic studies, how can it ensure that individuals' identities remain protected? How can it allow data analysis while guaranteeing that people cannot be re-identified, even indirectly?[3]

As data privacy concerns grow, so does the need for stronger privacy-preserving techniques. Traditional methods, such as pseudonymization, are no longer sufficient, leading to the development of privacy models like *k-anonymity*, *l-diversity*, *t-closeness* and *differential privacy*. However, many existing tools lack scalability, adaptability, or cloud integration.

To address these gaps, this project introduces **AnonimaData**: a scalable, cloud-based and user-friendly web application for anonymizing tabular datasets. Through a simple interface, users can securely upload datasets, select anonymization algorithms, configure their parameters and obtain privacy-preserving datasets that retain analytical value.

2 Project Overview: AnonimaData

AnonimaData is a a scalable web interface enabling flexible data anonymization workflows. The system is easily configurable to support different types of anonymization strategies and dataset formats.

The main goal is to build an application that allows users to:

- Upload datasets in **CSV** or **JSON** format
- Select from multiple anonymization algorithms, like *k-anonymity*, *l-diversity*, *t-closeness* and *differential privacy*;
- Dynamically configure algorithm parameters (e.g. the value of k in k-anonymity);
- Preview the anonymized dataset before downloading it;
- Store anonymized datasets both in a cloud database and as downloadable CSV files;
- Manage the entire process through a web-based user interface;
- Access the system via secure user authentication (e.g., Google OAuth 2.0).

To ensure broad applicability, the system is designed to handle *arbitrary tabular datasets*, regardless of their schema, data types or column structure. At the same time, it must guarantee key properties such as *scalability*, *reliability* and *efficient resource management*.

The entire infrastructure is deployed on **Google Cloud Platform (GCP)**, with all resources provisioned and maintained using **Terraform**, ensuring full reproducibility, portability and ease of configuration across environments.

3 Technologies and system architecture

3.1 Technologies used

The system consists of components built on diverse technologies across the backend, frontend, and infrastructure layers. Below is an overview of the main tools and frameworks used.

3.1.1 Backend

- **Python**: Core backend logic, including anonymization algorithms and API endpoints;
- **FastAPI**: A lightweight Python web framework used to expose RESTful API endpoints, such as the */anonymize* route;
- **Firebase Authentication**: Secure user authentication with Google OAuth 2.0;
- **Google Cloud Storage (GCS)**: Used to store the anonymized datasets and generate signed URLs for secure, temporary access;
- **Google Firestore**: Stores metadata related to anonymized files, including user ID, algorithm used, and file reference;
- **Pandas**: Utilized for reading and transforming tabular data before applying anonymization techniques;

- **Docker**: A Dockerfile is provided to containerize the backend service for deployment and local testing;
- **Modular architecture**: Anonymization algorithms are isolated in separate modules to promote scalability and maintainability.

3.1.2 Frontend

- **React.js**: The frontend is developed using the React JavaScript library, structured into reusable components;
- **Vite**: A fast build tool and development server used to bundle and serve the React application;
- **Nginx**: Used as a reverse proxy and static file server in the production environment, with configuration in *nginx.conf*;
- **Docker**: Used to containerize the frontend application for consistent deployment across environments;
- **CSS Modules**: Application-specific styles are included under the styles/directory.

3.1.3 Infrastructure & DevOps

- **Terraform**: Infrastructure as Code is managed using Terraform, with modules that automate deployment to Google Cloud Run, Firestore, Cloud Storage, and other GCP resources;
- **Google Cloud Run**: Used to deploy and host both backend and frontend as containerized, stateless services supporting auto-scaling;
- **Firestore**: Used for storing metadata and anonymized dataset information in a NoSQL database;
- **Google Cloud Storage**: Used for storing anonymized datasets as downloadable CSV files;
- **Docker Compose**: For local development, docker-compose.yml sets up backend and frontend services in containers;
- **GitHub**: Used for code collaboration, version control, and CI/CD integration;
- **Firebase Authentication**: Provides secure user authentication and management via OAuth 2.0 integration.
- **Cloud Logging**: Enables centralized logging and monitoring of backend and frontend services for troubleshooting and observability.

3.2 System architecture overview

The system is based on a **client-server architecture**:

- **Frontend**: built with *React* using the *Vite* build tool, containerized with *Docker* and deployed on *Google Cloud Run*.

- **Backend**: developed with *FastAPI* (Python), containerized with *Docker* and deployed on *Google Cloud Run*.

Frontend and backend communicate via RESTful APIs. All services are stateless and containerized, which facilitates scaling and deployment on Google Cloud Platform.

The anonymized datasets are:

- **Stored persistently** in Google Cloud Storage (GCS) as CSV files, organized by folder and uniquely identified using UUIDs.
- **Associated with user metadata** (e.g., user ID, algorithm used, timestamp) and stored in Google Firestore for traceability and retrieval.
- Provided to the user via signed URLs that allow temporary and secure file **downloads** directly from GCS.

This architecture ensures a clear separation of concerns between presentation, processing, and storage, enabling maintainability, scalability, and cloud-native deployment.

3.3 Project structure

```
gruppo-5
├── backend
│   ├── algorithms
│   │   ├── differential_privacy.py
│   │   ├── k_anonymity.py
│   │   ├── l_diversity.py
│   │   └── t_closeness.py
│   ├── auth
│   │   └── firebase_auth.py ..... Handles user authentication via Firebase
│   ├── config
│   │   └── keywords.py ..... Contains reserved keywords for column detection
│   ├── firestore
│   │   └── firestore_client.py ..... Initializes Firestore client for metadata storage
│   ├── routes
│   │   └── anonymize.py ..... API endpoint that triggers anonymization logic
│   ├── storage
│   │   └── storage.py ..... Handles file upload/download with GCS
│   ├── tools
│   │   └── hierarchy_builders.py ..... Builds generalization hierarchies for QIDs
│   ├── Dockerfile
│   └── main.py ..... FastAPI entry point for the backend application
└── frontend
    └── src
        └── components
            ├── AlgorithmSelector.jsx
            ├── FileUpload.jsx
            ├── HomePage.jsx
            ├── LoginPage.jsx ..... Google OAuth 2.0 login page
            └── PrivateRoute.jsx ..... Protects routes from unauthenticated access
```

```
        │   └── WhiteBox.jsx
        ├── styles
        ├── firebase.js ..... Firebase configuration and init for frontend
        └── MainApp.jsx
    ├── Dockerfile
    └── Dockerfile.dev
├── terraform
    ├── backend.tf ..... Defines Cloud Run for backend
    ├── firestore.tf ..... Creates Firestore DB instance
    ├── frontend.tf ..... Defines Cloud Run for frontend
    ├── main.tf ..... Main Terraform entry point with module references
    ├── storage.tf ..... Defines Cloud Storage buckets and permissions
    ├── terraform.tfvars
    ├── variables.tf
    └── vpc.tf ..... Creates custom VPC and subnetworks
├── docker-compose.yml
└── README.md
```

3.4 Backend workflow

The backend is built with FastAPI, ensuring clean API design, asynchronous handling, and automatic documentation. The following describes the end-to-end process, from receiving a request to returning an anonymized dataset:

3.4.1 Server initialization – main.py

The FastAPI app is instantiated and configured with:

- CORS middleware, allowing the frontend to interact with the backend from controlled origins.
- Route registration, importing the anonymize router from `routes/anonymize.py`, which defines all backend endpoints related to file upload, processing, and download.

3.4.2 API endpoint – routes/anonymize.py

The backend exposes multiple endpoints under the `/anonymize` route:

- `GET /anonymize`: A basic health-check endpoint to verify that the service is running.
- `POST /anonymize`: Main endpoint. Receives an uploaded file (CSV or JSON), the selected anonymization algorithm, and its corresponding parameter.
- `GET /download/file_id`: Returns a downloadable CSV file previously anonymized and stored in Google Cloud Storage, identified by a unique file_id.

All endpoints are protected by Firebase token-based authentication using a custom `verify_token` dependency.

3.4.3 File parsing and validation

Once a file is uploaded via the `/anonymize` endpoint, the backend performs the following steps:

1. Validates the MIME type (`text/csv` or `application/json`). If invalid, returns HTTP 400.
2. Reads the file into a `pandas.DataFrame` using in-memory buffers (`BytesIO`).
3. If reading fails due to malformed data or format mismatch, returns HTTP 500 with a descriptive error message.

3.4.4 Column classification

The system **dynamically** classifies the dataset's columns into:

- **Identifiers**: Attributes that uniquely identify an individual on their own (e.g., names, email addresses, social security numbers).
- **Quasi-identifiers (QIs)**: Attributes that do not directly identify an individual but can be used in combination to re-identify them (e.g., age, ZIP code, gender). Quasi-identifiers are a key target of anonymization techniques.
- **Sensitive attributes**: Information that, while not identifying, is private and must be protected (e.g., income, medical conditions, political affiliation).

3.4.5 Algorithm execution

Before running the algorithm, **direct identifiers** are **dropped** from the dataset and the parameter value is parsed and validated (as int or float depending on the algorithm). The supported algorithms are *k-anonymity, l-diversity, t-closeness, differential-privacy.*

Each algorithm is **modularized** and imported from the `algorithms/` folder. Errors during execution (e.g., invalid parameters, edge-case failures) return HTTP 500 responses with debug-friendly logs.

3.4.6 Result preview and serialization

After successful anonymization the result is cleaned for safe JSON serialization (e.g., converting NaNs and floats).

A preview of the first rows is returned to the frontend in JSON format.

3.4.7 Storage and download

The full anonymized dataset is **saved** as a CSV file and made available for **download**.
If Google Cloud Storage (GCS) is enabled:

- The file is uploaded to a **GCS bucket**.
- A signed URL (valid for 15 minutes) is generated to allow secure temporary access.
- Metadata is stored in **Google Firestore**, including user ID, algorithm, parameter, and file path.

If GCS is disabled (for local development):

- The file is saved in a local folder and made downloadable via a local `/download/file_id` route.

3.4.8 Error handling and log system

The backend uses FastAPI's HTTPException to manage errors:

- Unsupported file types → HTTP 400
- Missing or invalid algorithm parameters → HTTP 400
- Missing QIs or sensitive attributes → HTTP 400
- Internal processing errors → HTTP 500

To ensure observability and facilitate post-deployment monitoring, the backend integrates with **Google Cloud Logging**. Logging is implemented using Python's standard logging module and configured through the Uvicorn logger, which seamlessly integrates with Cloud Run's execution environment. As a result, structured log entries are **automatically collected** and made **accessible** via the Google Cloud Console, enabling **traceability** across all stages of the anonymization pipeline.

3.4.9 Summary table

Step	File	Purpose
1	`main.py`	FastAPI server setup, middleware registration (CORS), and route mounting
2	`routes/anonymize.py`	Route definitions for `/anonymize` (GET, POST) and `/download/file_id`
3	`anonymize.py`	File validation and DataFrame creation
4	`anonymize.py` + `keywords.py`	Column classification and fallback mechanisms
5	`anonymize.py` + `algorithms/`	Execution of selected anonymization algorithm
6	`anonymize.py`	Return of JSON-safe data preview
7	`anonymize.py` + `storage.py` + `firestore_client.py`	Cloud storage of anonymized file, metadata tracking, and generation of signed download URL

Table 1: Summary of the backend workflow components

3.5 Frontend workflow

The frontend is developed using **React** and is structured as a modular Single Page Application (SPA) with client-side routing. The application uses **Firebase Authentication** to manage user sessions and ensure that only authenticated users can access the main features.

The main components of the frontend are organized as follows:

- **assets/**: contains static resources, such as images used for the web layout.
- **components/**: includes all the React components responsible for user interaction.
- **styles/**: stores CSS files for customizing the appearance of each page and component.
- `firebase.js`: manages the configuration of Firebase Authentication.

- `MainApp.jsx`: defines the application entry point and routes.

3.5.1 Core components

- **HomePage.jsx**: the protected page that allows users to upload datasets, select anonymization algorithms, start the anonymization process, preview the anonymized data, and download the result.
- **LoginPage.jsx**: the login page where users can authenticate using Google OAuth via Firebase.
- **AlgorithmSelector.jsx**: provides a dropdown menu for selecting the anonymization algorithm and dynamically displays the required parameters based on the user's selection.
- **FileUpload.jsx**: manages file uploads, allowing users to select and load CSV or JSON files. The interface clearly shows the selected file name.
- **PrivateRoute.jsx**: implements route protection by verifying if the user is authenticated before granting access to protected pages. If the user is not authenticated, it automatically redirects to the login page.
- **WhiteBox.jsx**: a reusable UI component that visually separates different sections of the interface, such as file upload, algorithm selection, and data preview.

3.5.2 Routing and navigation

The frontend uses **React Router** to manage navigation between pages:

- `/` → Login page.
- `/home` → Homepage (accessible only if the user is logged in).

Routes are defined in `MainApp.jsx`, where the `PrivateRoute` component ensures that only authenticated users can access the homepage.

3.5.3 Frontend workflow

The flow of the application from the user's perspective is as follows:

1. The user accesses the web application and is immediately redirected to the login page if not authenticated.
2. After successful **login** via Firebase, the user is redirected to the homepage.
3. On the homepage, the user can **upload** a CSV or JSON dataset using the upload button.
4. The user **selects** one of the **anonymization algorithms** and enters the **parameter**.
5. By clicking the "Anonymize" button, the user **sends** the dataset, selected algorithm and parameter to the backend via a POST request, including the Firebase authentication token in the header.
6. The backend processes the dataset and returns a JSON **preview** of the anonymized data, which is displayed in a table in the frontend.

7. The user can **download** the complete anonymized dataset via a dedicated download button.
8. Finally, the user can **log out** using the logout button, which clears the session and redirects to the login page.

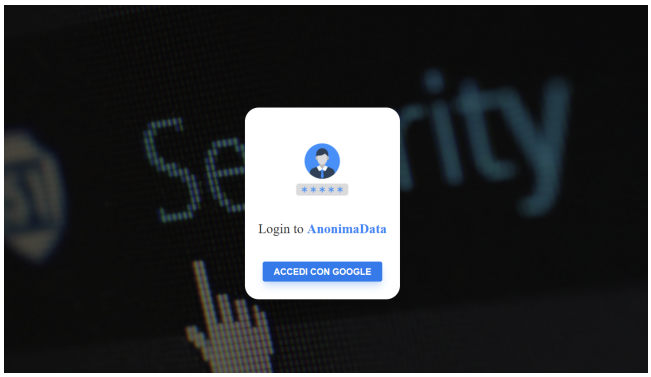

Figure 1: User authentication via Google OAuth (Firebase Authentication)

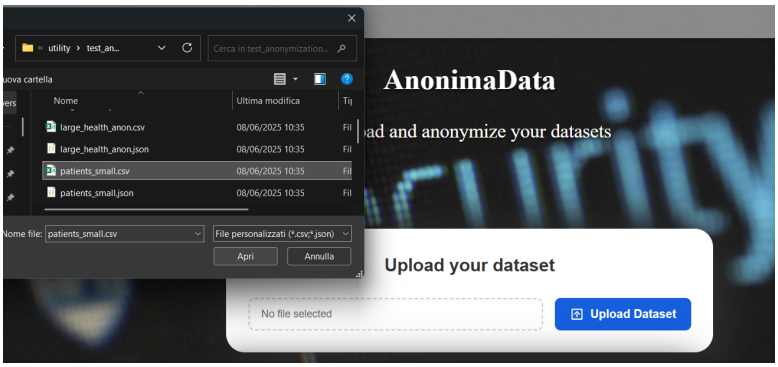

Figure 2: Dataset upload interface

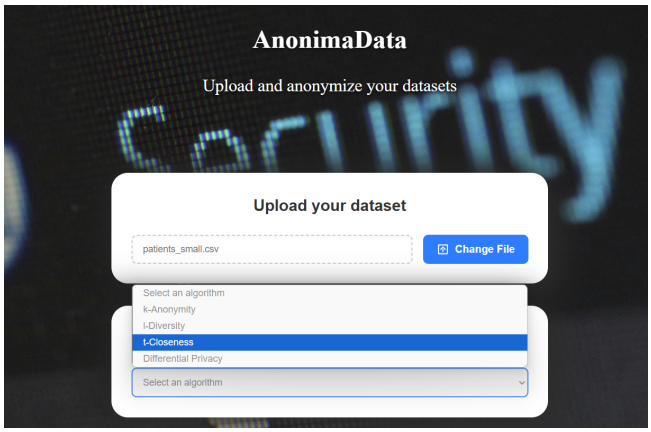

Figure 3: Anonymization algorithm selection and configuration

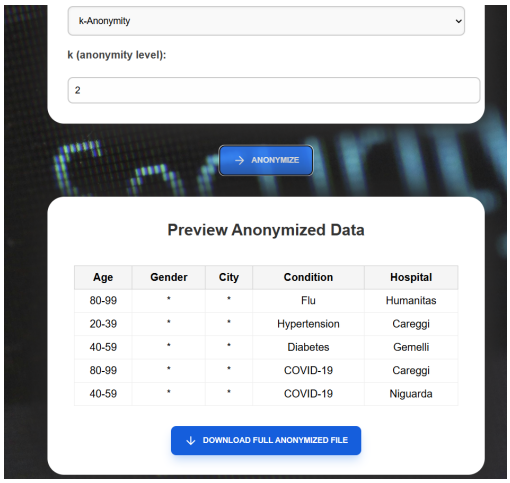

Figure 4: Preview of the anonymized dataset

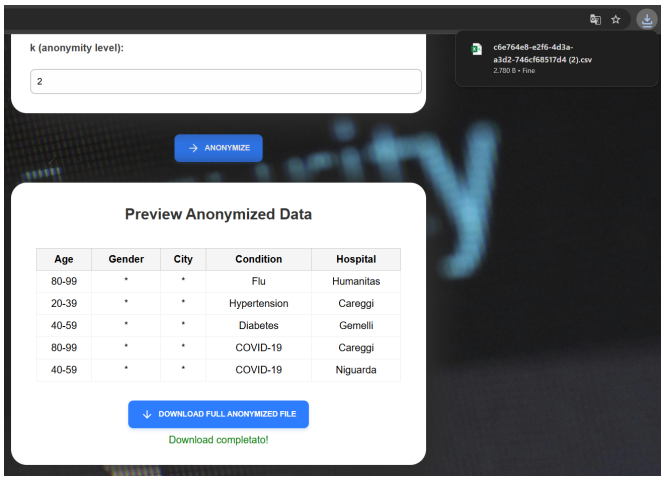

Figure 5: Download of the anonymized dataset

3.5.4 Summary table

Component	File	Purpose
App Entry Point	`MainApp.jsx`	Configures routing and initializes Firebase
Login Page	`LoginPage.jsx`	Manages Google OAuth login via Firebase
Protected Route	`PrivateRoute.jsx`	Protects routes and verifies authentication status
Home Page	`HomePage.jsx`	Main page for uploading files, selecting algorithms, previewing results, and downloading files
File Upload	`FileUpload.jsx`	Handles file selection and input validation
Algorithm Selection	`AlgorithmSelector.jsx`	Allows algorithm selection and parameter configuration
Reusable UI Box	`WhiteBox.jsx`	Provides a clean layout container for UI sections

Table 2: Summary of the frontend components

The frontend is fully integrated with the backend via authenticated REST API calls and provides a responsive, modular, and secure interface that guides the user throughout the anonymization process.

4 Implementation details

4.1 Generalization Hierarchies

A core component of our anonymization pipeline is the dynamic generation of attribute-specific **generalization hierarchies**. These hierarchies are used to transform quasi-identifiers into more abstract representations during privacy-preserving transformations such as k-anonymity,

ℓ-diversity, and t-closeness. This process is known as **generalization** — one of the classical techniques in data anonymization. By replacing specific attribute values (e.g., "34", "10001") with broader categories (e.g., "30–39", "100**"), generalization reduces the risk of re-identification while retaining the overall structure and analytical value of the data.

4.1.1 Purpose and Role of Hierarchies

Generalization hierarchies define how detailed attribute values are transformed into broader categories. For example, a birthdate can be generalized stepwise into month, year, decade, and century. This structure allows anonymization algorithms to **choose the minimal generalization level needed** to satisfy privacy constraints—maximizing data utility when possible.

This approach allows our system to be both **scalable**, by automatically adjusting to the structure and diversity of the input data, and **reliable**, by ensuring a clear, deterministic transformation path for each quasi-identifier.

4.1.2 Implementation and Strategy

Each hierarchy is constructed by a *dedicated function* selected through a *mapping system*. **Predefined builders** exist for common quasi-identifiers such as age, ZIP code, salary, dates, categorical labels, and email domains. If no exact match is found, the system applies heuristics based on data type and column name.

Supported hierarchies include:

- **Age and numeric values**: Values are grouped into progressively larger buckets (e.g., 5-year, 10-year, 20-year ranges).
- **Salaries and incomes**: Rounded and grouped into labeled income bands (e.g., "30k–39k").
- **ZIP/postal codes**: Generalized by truncating digits (e.g., "12345" $\rightarrow$ "1234*" $\rightarrow$ "123**" $\rightarrow$ "*").
- **Dates**: Converted into month, year, decade, and century representations.
- **Categorical values**: Mapped directly to a wildcard "*" level, or optionally grouped (e.g., email domains).
- **Email domains**: Normalized and generalized to hide organization-specific structures.

Each builder is designed to be **stateless** and **idempotent**, allowing reuse across large datasets and making the system robust in **concurrent** or **parallel** processing environments.

4.1.3 Design for Scalability

To support datasets with unknown or heterogeneous structures, the system **avoids hardcoding** rules. Instead, it relies on:

- **Automatic** type-based inspection,
- An **extensible** builder registry,
- Step-wise generalization that adapts to the required privacy level.

This makes the framework **adaptable to different contexts** with minimal configuration.

4.1.4 Design for Reliability

The hierarchy design guarantee consistent, predictable transformations through:

- Standardized **value cleaning**,
- **Fallback** rules for unparseable inputs,
- Unified wildcard use (e.g., *),
- Data-type aware logic (e.g., for dates and numbers).

Generalization **avoids excessive abstraction** when unnecessary—for example, stopping ZIP code generalization at "123**" to preserve granularity elsewhere. This reflects a **balanced trade-off between privacy and utility**.

4.1.5 Future Improvements

While the current hierarchy system is flexible and functional, several improvements could further enhance performance and adaptability:

- Adaptive bucketing based on data distribution (e.g., quantile-based instead of fixed-step generalization),
- Ontology-based hierarchies for categorical fields (e.g., job titles, medical conditions),
- User-defined hierarchies via JSON or YAML configuration for custom domains,
- Hierarchy visualization tools to assist with debugging and validation.

This modular and extensible hierarchy system forms the foundation for scalable generalization in our anonymization framework. It supports the reliable transformation of diverse datasets while keeping the implementation maintainable and extensible for future enhancements.

4.2 Column Classification and API Integration

To support flexible anonymization across varied datasets, the system includes:

Column Classification: The system classifies columns into identifiers, quasi-identifiers, and sensitive attributes using a **combination** of **keyword matching** (configured in `config/keywords.py`), **data type inference**, and **statistical heuristics** such as value range analysis and entropy.

If no quasi-identifiers are detected, the anonymization process is halted with an error. If a required sensitive attribute is missing—such as in algorithms like *l-diversity* or *t-closeness*—a **fallback** mechanism automatically selects the quasi-identifier with the highest statistical variability (e.g., entropy), assuming it to be the most informative. This fallback logic ensures that classification remains **robust**, even in datasets with ambiguous or unconventional schemas.

API Integration: Exposed via POST `/anonymize` (FastAPI)

- Supports real-time and batch mode
- Stateless and horizontally scalable
- Streaming file input for large datasets
- Clear error handling with fallback paths

4.3 *k*-Anonymity

Theoretical Overview k-Anonymity[4] is one of the foundational models for data anonymization. Its main objective is to prevent re-identification of individuals by ensuring that **each record** in the dataset is **indistinguishable from at least** $k-1$ **other records** with **respect** to a set of **quasi-identifiers.**

Implementation Strategy The system enforces k-anonymity through a *progressive generalization* approach:

1. **Hierarchy Assignment:** Each quasi-identifier is paired with a dedicated generalization hierarchy, constructed by a builder function dynamically selected at runtime.
2. **Grouping and Validation:** At each generalization level, the dataset is grouped by the generalized quasi-identifiers. The system checks whether each group (equivalence class) contains at least k records.
3. **Iteration:** If any group fails the check, the generalization level is increased and the process is repeated until the condition is met or a maximum level is reached.

Column roles (identifier, quasi-identifier, sensitive attribute), as previously described, are inferred automatically using keyword-driven configuration and entropy-based fallbacks. This enables portability across diverse datasets without requiring manual annotations.

Highlights

- Modular architecture based on reusable generalization hierarchies.
- Easily extensible to new domains through the addition of new builder functions.
- Integrated into a REST API (`/anonymize`), supporting both batch and real-time workflows.
- Robust fallback mechanisms ensure operation even in poorly labeled datasets.

4.4 ℓ-Diversity

Theoretical Overview While k-anonymity effectively hides individual identities, it is insufficient in cases where all records in a group share the same sensitive attribute value. In such scenarios, **knowing that an individual belongs to a group can still reveal confidential information**—a situation known as a **homogeneity attack**.

To mitigate this risk, ℓ-diversity[5] extends the k-anonymity model by requiring that **each group** (equivalence class) **contain at least** ℓ **distinct and sufficiently diverse sensitive values**. This increases the uncertainty an attacker faces when trying to infer private information about individuals within a group.

Implementation Strategy The ℓ-diversity algorithm builds directly upon the generalization framework used for k-anonymity, utilizing the same modular hierarchy system and builder infrastructure. The generalization process proceeds uniformly across all quasi-identifiers.

The algorithm performs the following steps:

1. **Initial Generalization:** The dataset is generalized to a certain level using predefined hierarchies.
2. **Validation:** For each group, the number of distinct sensitive values is counted and compared against the ℓ threshold.
3. **Incremental Adjustment:** If any group fails the diversity check, the generalization level is increased and validation is repeated.

Currently, **generalization** is **applied globally and uniformly** across the entire dataset. While this simplifies the implementation, it may lead to **over-generalization** and unnecessary **loss of information**.

Highlights

- Shares all logic and infrastructure with k-anonymity, allowing **code reuse** and **architectural consistency**.
- Designed for **extensibility**; improvements such as Mondrian multidimensional partitioning or the Incognito algorithm are being considered to achieve better data utility.
- Exposed through the same API interface as k-anonymity, ensuring interoperability within privacy-preserving workflows.

4.5 *t*-Closeness

Theoretical Overview t-Closeness[6] addresses the limitations of both k-anonymity and ℓ-diversity by focusing not only on the presence of diversity, but on its **distribution**. The model ensures that **the distribution of sensitive values within any group is statistically similar to the distribution across the entire dataset**.

Formally, a dataset satisfies t-closeness if the distance between the distribution of a sensitive attribute in any equivalence class and the global distribution does not exceed a *threshold* t. This minimizes the potential information gain for an attacker.

Implementation Strategy The algorithm enforces t-closeness through the following iterative process:

1. Partition the dataset into equivalence classes based on generalized quasi-identifiers.
2. Compute the global distribution of the sensitive attribute.
3. For each group:
   - Compute the local distribution of the sensitive attribute.
   - Calculate the L_1 (Total Variation) distance from the global distribution.
   - If the distance exceeds the threshold t, the group is considered non-compliant.
4. Increase the generalization level and repeat until all groups satisfy t-closeness or a maximum level is reached.

Design Decisions

- Sensitive values are **never altered**; only quasi-identifiers are generalized.
- The L_1 **distance metric** was chosen over Earth Mover's Distance for simplicity and better applicability to categorical sensitive attributes.
- Generalization hierarchies are **reused** and expanded dynamically, as in previous models.

Limitations

- Currently optimized for **categorical sensitive attributes**; numerical attributes are not yet supported.
- May result in **high information loss** in skewed datasets requiring deep generalization.

4.6 Differential Privacy

Theoretical Overview Differential Privacy (DP)[7] provides a formal mathematical guarantee of privacy. By injecting calibrated noise into data, DP ensures that the presence or absence of any single individual has a limited impact on the outcome, quantified by the privacy parameter ε.

This implementation uses an *input perturbation* approach, where a differentially private version of the dataset is generated a priori—without the need to track future queries.

Implementation Strategy Unlike generalization-based methods, this model uses input perturbation to generate a privatized version of the dataset before any query is made. The implementation is designed to be fully automated and data-adaptive:

- **Numerical Columns:** Laplace noise is added. Values are clipped using configurable bounds to avoid unrealistic outputs (e.g., negative ages).
- **Age:** Identified automatically via keyword matching. A specific normalization and perturbation pipeline is applied, limiting noise to ± 5 years and enforcing a valid range between 18 and 90.
- **ZIP Codes:** Recognized by name and generalized by truncation to the first three digits. Noise is not applied to preserve interpretability.
- **Categorical Columns:** Processed via randomized response. Each value is kept or changed based on a probability derived from ε and the number of categories.
- **Outlier Control:** Sensitivity is estimated using the range between the 1st and 99th percentiles, minimizing the impact of outliers.
- **Privacy Budget Allocation:** 90% of the total budget is allocated to numerical columns, and 10% to categorical ones.

Privacy Budget Allocation The total privacy budget is distributed asymmetrically:

- 90% is allocated to numerical attributes (where Laplace noise requires tighter control).

- 10% is assigned to categorical columns (where randomized response is effective with less noise).

Strengths

- Fully automated pipeline that adapts to heterogeneous datasets.
- Realistic outputs: ages, ZIPs, and salaries remain semantically coherent.
- Robustagainst skewed distributions and outliers.
- Type-aware processing improves both privacy protection and utility retention.

Limitations

- Attribute-level independence: noise is applied independently, without preserving inter-attribute correlations.
- Utility may degrade significantly for high-dimensional datasets or small ε.

5 Production Deployment on Google Cloud Platform

5.1 Objectives and Architecture

The system has been designed to be cloud-native, scalable, and modular, with deployment entirely managed on Google Cloud Platform (GCP). The primary goal of the production environment is to enable end-users to upload tabular datasets (CSV or JSON), configure anonymization algorithms dynamically, and obtain anonymized outputs in both database and downloadable CSV formats.

The architecture follows a service-oriented design and separates concerns across loosely coupled components, in order to ensure maintainability, reliability, and extensibility.

5.2 GCP Infrastructure Components

The system is composed of the following major GCP-managed services:

1. **Frontend**: A modern single-page application built with React and deployed on Google Cloud Run.
2. **Backend API**: A containerized REST API built with FastAPI (Python), deployed on Cloud Run with autoscaling and HTTPS termination managed by Cloud Load Balancing.
3. **Relational Database**: Firestore is used to store metadata and anonymized datasets.
4. **Object Storage**: Google Cloud Storage buckets store uploaded and anonymized CSV files, with access managed through signed URLs.
5. **Authentication**: Firebase Authentication provides OAuth 2.0 login, enabling secure access via Google accounts.
6. **Logging and Monitoring**: Cloud Logging and Cloud Monitoring are enabled for both frontend and backend services, supporting performance diagnostics and alerting.

7. **Infrastructure as Code**: All resources are provisioned and managed using Terraform, ensuring reproducibility and version control of infrastructure configurations.

5.3 Scalability and Reliability Considerations

The system adopts several best practices to achieve horizontal scalability and high availability:

- **Stateless Services**: Both frontend and backend are containerized and stateless, making them compatible with Cloud Run's autoscaling features.
- **Compute-Storage Separation**: The anonymization logic runs in isolated compute containers, while all persistent data is stored in Google Cloud Store, following the separation-of-concerns principle.
- **Autoscaling and Load Balancing**: Cloud Run services scale automatically based on request load and are protected via GCP's global HTTP(S) Load Balancer.
- **Configurable Privacy Parameters**: All algorithmic parameters (e.g., k, ℓ, t, ε) are configurable via the frontend and passed to the backend dynamically, supporting flexible workloads.
- **Terraform-Based Infrastructure**: The entire environment can be deployed from scratch using Terraform, promoting infrastructure reproducibility and enabling infrastructure-as-code workflows.

5.4 Challenges and Solutions: CORS and Service Access Restrictions

During the production deployment, a significant issue arose related to Cross-Origin Resource Sharing (CORS) preflight requests. The frontend and backend services, both deployed on Cloud Run, were unable to communicate properly due to access restrictions.

Specifically, both services were configured as private, requiring authentication tokens for access. Besides the Firebase token used for user authentication, Cloud Run also enforced Google identity tokens to verify that the calling user was authorized. This double authentication caused conflicts, blocking CORS preflight requests and preventing interaction between the frontend and backend.

The team undertook an extensive troubleshooting process over approximately three weeks, involving:

- Detailed log analysis using Cloud Logging to trace the failure points.
- Multiple curl-based tests to simulate requests and debug token-related errors.
- Attempts to maintain service privacy while enabling frontend-backend communication by experimenting with:
  - API Gateway configurations, including different HTTP methods and endpoint settings.
  - Virtual Private Cloud (VPC) configurations.
  - Serving the frontend through Cloud Storage combined with a Content Delivery Net-

work (CDN) to simplify domain and access management.

  - Use of the gcloud proxy to forward requests securely.

- Careful review and configuration of IAM policies, with dedicated service accounts assigned to each resource.

Despite these extensive efforts, none of the approaches resolved the CORS issue without compromising service accessibility. The main obstacle was the organizational policy restricting the use of public access ('allUsers') for Cloud Run services, preventing a straightforward public exposure.

Ultimately, the only viable solution was to disable IAM enforcement entirely on both frontend and backend Cloud Run services, effectively making them public. Importantly, exposing the frontend and backend to the internet does not reduce security in this case, since access to the services remains strictly limited to authenticated users through Firebase Authentication. This approach aligns with the best practices recommended by Google Cloud Platform for managing access to Cloud Run services in scenarios requiring cross-origin communication.

This resolution required careful coordination and repeated testing but ensured a functional and responsive production deployment. The experience highlighted the complexities involved in balancing security, service privacy, and cross-origin communication in GCP environments, as well as the importance of persistent troubleshooting and exploration of multiple strategies when facing cloud infrastructure challenges.

5.5 Known Limitations and Trade-offs

While the system fulfills the majority of the production objectives, certain advanced features have been intentionally deferred:

- **No Pub/Sub Integration**: Asynchronous anonymization jobs via Pub/Sub queues were considered but not implemented due to time constraints. Currently, anonymization is processed synchronously during each request. While this is sufficient for small to medium datasets, it may introduce latency for larger payloads.
- **No Distributed Workload Execution**: The system does not yet partition anonymization jobs across multiple workers or instances. All computation is performed within a single container per request. A distributed model (e.g., Pub/Sub pipelines) is a potential future enhancement.
- **No AB Load Testing**: Formal load testing (e.g., via Apache Bench or Locust) has not yet been conducted, although the architecture is expected to scale to modest usage scenarios.

Despite these limitations, the current deployment is fully functional, robust under typical loads, and architecturally designed to support future scale-out strategies without major redesigns.

6 Conclusions

AnonimaData provides a robust and scalable solution for privacy-preserving dataset anonymization, combining a user-friendly interface with powerful, cloud-native technologies. The modular architecture ensures maintainability, while the use of standardized anonymization techniques

like *k-anonymity, ℓ-diversity, t-closeness,* and *differential privacy* allows for flexible deployment across various data sensitivity scenarios.

The system demonstrates strong integration with the Google Cloud ecosystem and achieves a balance between usability, data utility, and privacy. Through the use of modern DevOps tools such as Terraform and Docker, AnonimaData attains reproducibility and simplifies deployment across production and development environments.

Future developments may include support for custom user-defined hierarchies, asynchronous job queues (e.g., via Pub/Sub) and improved visual feedback on anonymization impact. Incorporating formal testing strategies and distributed anonymization pipelines would further strengthen its reliability and scalability.

References

[1] Pierangela Samarati. Protecting respondents' identities in microdata release. *IEEE Transactions on Knowledge and Data Engineering*, 13(6):1010–1027, 2001. doi: 10.1109/69.971193.

[2] LATANYA SWEENEY. k-anonymity: A model for protecting privacy. *International Journal of Uncertainty, Fuzziness and Knowledge-Based Systems*, 10(05):557–570, 2002. doi: 10.1142/S0218488502001648. URL https://doi.org/10.1142/S0218488502001648.

[3] Arvind Narayanan and Vitaly Shmatikov. Robust de-anonymization of large sparse datasets. In *2008 IEEE Symposium on Security and Privacy (sp 2008)*, pages 111–125, 2008. doi: 10.1109/SP.2008.33.

[4] Pierangela Samarati and Latanya Sweeney. Protecting privacy when disclosing information: k-anonymity and its enforcement through generalization and suppression. *IEEE Transactions on Knowledge and Data Engineering*, 10(5):557–570, 1998. doi: 10.1109/69.704210. URL https://doi.org/10.1109/69.704210.

[5] A. Machanavajjhala, J. Gehrke, D. Kifer, and M. Venkitasubramaniam. L-diversity: privacy beyond k-anonymity. In *22nd International Conference on Data Engineering (ICDE'06)*, pages 24–24, 2006. doi: 10.1109/ICDE.2006.1.

[6] Ninghui Li, Tiancheng Li, and Suresh Venkatasubramanian. t-closeness: Privacy beyond k-anonymity and l-diversity. *Proceedings of the 23rd International Conference on Data Engineering*, pages 106–115, 2007. doi: 10.1109/ICDE.2007.367853. URL https://doi.org/10.1109/ICDE.2007.367853.

[7] Cynthia Dwork, Frank McSherry, Kobbi Nissim, and Adam Smith. Calibrating noise to sensitivity in private data analysis. In *Proceedings of the 3rd Theory of Cryptography Conference*, pages 265–284, 2006. doi: 10.1007/11787006_1. URL https://doi.org/10.1007/11787006_1.