HIGH THROUGHPUT PERSONAL INFORMATION CLEANING OF TEXTUAL DOCUMENTS

Ahmet Erdem, Burak Aydın, Abdulkadir Külçe
AI & Data Engineering
Istanbul Technical University
Istanbul, Turkey
{erdemah22,aydinbu22,kulce21}@itu.edu.tr

1 PROBLEM DEFINITION

This section has been kept from the project proposal for readers reference.

This project addresses the critical need for efficient and accurate sanitization of sensitive personal information embedded within large volumes of textual documents. Companies frequently handle vast and rapidly expanding datasets containing unstructured text where Personally Identifiable Information (PII), such as names, addresses, phone numbers, and financial data, is interspersed **?**. The core challenge is two-fold: (1), reliably performing Named Entity Recognition (NER) to precisely locate and categorize these various PII entities across textual documents; (2), executing the cleaning process with high throughput to handle the scale of modern data operations.

The key stakeholders for this project include:

- **Compliance and Legal Teams**: They require documented assurance that privacy regulations (GDPR) are being met.
- **Data Engineers/Scientists**: They need cleaned, compliant data to use for analytics, machine learning, and reporting.
- **Security Officers (CISO/CSO)**: They are responsible for protecting organizational data assets from unauthorized disclosure.
- **End Users/Customers**: Their personal information is directly protected by the system.

The code for the project is made public at https://github.com/ahmeterdem1/YZV448E-term-project

2 INTRODUCTION

The rapid digitization of corporate and educational workflows has resulted in the accumulation of vast repositories of unstructured textual data. While these datasets are invaluable for analytics and the development of large-scale machine learning models, they frequently contain sensitive Personally Identifiable Information (PII), such as names, addresses, and financial identifiers **?**. The exposure of such data not only compromises individual privacy but also poses significant legal and financial risks under stringent regulatory frameworks like the General Data Protection Regulation (GDPR). Consequently, there is an urgent need for automated systems capable of performing high-precision PII cleaning at scale.

The challenge of PII sanitization is twofold. First, the system must achieve high-fidelity Named Entity Recognition (NER) across diverse, unstructured domains—such as educational essays and administrative documents—where entity density and context vary significantly. Second, modern data operations require high-throughput processing capabilities; a solution must be performant enough to handle massive batch volumes without becoming a bottleneck in the data engineering pipeline.

In this work, we present a robust, scalable architecture for high-throughput PII cleaning. Our approach leverages fine-tuned transformer-based models (BERT) optimized via torch.compile and advanced inference engines like vLLM to maximize hardware utilization. To balance the trade-off between individual request latency and system-wide throughput, we implement an asynchronous batch

processing paradigm utilizing FastAPI and Redis. This system employs a dual-condition flushing logic—based on both queue size N and a temporal threshold T—ensuring that the GPU operates at peak efficiency while guaranteeing a maximum waiting time for any single document.

Our contributions include:

- A performance-optimized NER pipeline specifically tuned for complex, unstructured educational data.
- A scalable, Dockerized API architecture designed for horizontal scaling and real-time monitoring.
- An asynchronous queuing strategy that optimizes GPU dynamic batching for compliance-heavy environments.
- We also heavily document our approach, aiming for high reusability and integrability

3 DATA

The foundation of this project's NER training and evaluation will be the PII Detection and Removal from Educational Data dataset, sourced from the recent Kaggle competition. This dataset is particularly valuable because it comprises unstructured, domain-specific text (educational documents, student essays, etc.) that closely mimics the complex, real-world data environments where sensitive PII must be protected.

The dataset consists of thousands of text samples that have been meticulously annotated for various types of Personally Identifiable Information (PII). Key characteristics include:

- **Document Type:** The data is drawn from educational documents, providing a specific context that introduces unique challenges, such as the frequent appearance of student and teacher names, institution names, and specific document identifiers.
- **PII Entities:** The annotations cover a diverse range of PII categories, including but not limited to names, email addresses, phone numbers, addresses, urls and IDs.
- **Annotation Format:** The PII entities are typically provided with their start and end character indices within the raw text, along with their respective entity labels. This format is ideal for training sequence labeling models like BERT, which operate at the token or sub-word level.

4 SYSTEM OVERVIEW

The PII Cleaner system is a high-performance, asynchronous microservice designed to detect and redact Personally Identifiable Information (PII) from unstructured text. The system employs a hybrid detection strategy combining deep learning (BERT-based NER) with deterministic Regular Expressions (Regex) to ensure high recall and precision. It features a decoupled architecture using FastAPI for ingestion and Redis for task queuing and state management, allowing for scalable, batch-oriented processing.

The architecture follows a producer-consumer pattern to handle high-throughput text processing without blocking the API interface.

4.1 API INTERFACE

- **Framework:** Built on FastAPI (Python 3.11), providing asynchronous endpoints and automatic OpenAPI documentation.
- **Rate Limiting:** Implements a fixed-window rate limiter per IP address using Redis to prevent abuse.
- **Validation:** Enforces max text length of 10000 characters and a max queue size of a thousand items to prevent memory overflows.

4.2 TASK MANAGEMENT

- **Queue Mechanism:** Uses Redis lists to implement a First-In-First-Out (FIFO) task queue
- **State Storage:** Stores task metadata (status, original text, cleaned text, PII stats) in Redis with a configurable Time-To-Live
- **Concurrency Control:** Utilizes atomic locks to ensure only one worker processes a batch at a time, preventing race conditions in distributed environments.

4.3 PII CLEANER MODEL

- A BERT and a Regex engine is combined to achieve high recall.
- The system intelligently merges overlapping spans from both engines, prioritizing high-confidence Regex matches while preserving context-aware BERT detections.
- A dataset partition is automatically generated for training and development for the BERT model.
- The model is optimized via **torch.compile** and vLLM.

4.4 FURTHER DETAILS

To maximize GPU throughput, the system does not process requests immediately. Instead, the system employs a dynamic batching strategy. The tasks are accumulated in the Redis queue. Processing is triggered when either the queue size reaches a predefined maximum size, or the age of the oldest item in the queue exceeds a predefined maximum timeout. When triggered, a batch of documents is pulled, processed in parallel by the model, and results are bulk-updated in Redis. Figure 3 depicts a flowchart that outlines the described backend logic.

5 MLOPS LIFECYCLE

The FastAPI Redis PII Cleaner implements a semi-automated MLOps pipeline designed to maintain high PII detection accuracy (specifically recall) with minimal human intervention. The cycle revolves around continuous performance monitoring and threshold-based retraining. Figure 1 shows a flowchart depicting the full MLOps cycle of the BERT system.

5.1 DATA INGESTION & MANAGEMENT

The system decouples code from data, allowing datasets to be updated dynamically via API without redeploying the container. Datasets are persisted to the local data/ volume, ensuring they survive container restarts. The preprocessing of the data is automatically handled by the training services of the backend.

5.2 MONITORING

A background routine acts as the system's watchdog, ensuring the model does not drift or degrade. It runs periodically alongside the main application. It calculates the F5 Score on the validation split. The F5 metric is chosen over F1 to heavily penalize false negatives (missed PII), prioritizing privacy protection over precision. The current score is published to Redis and visualized on the live dashboard.

5.3 ADAPTIVE RETRAINING

The system features a closed-loop retraining mechanism triggered by performance metrics rather than a fixed schedule. If the evaluated F5 score falls below the a predefined threshold, a training job is automatically initiated. The BERT is finetuned by the training service. Training progress (loss, epoch, steps) is piped to the application's standard logging system.

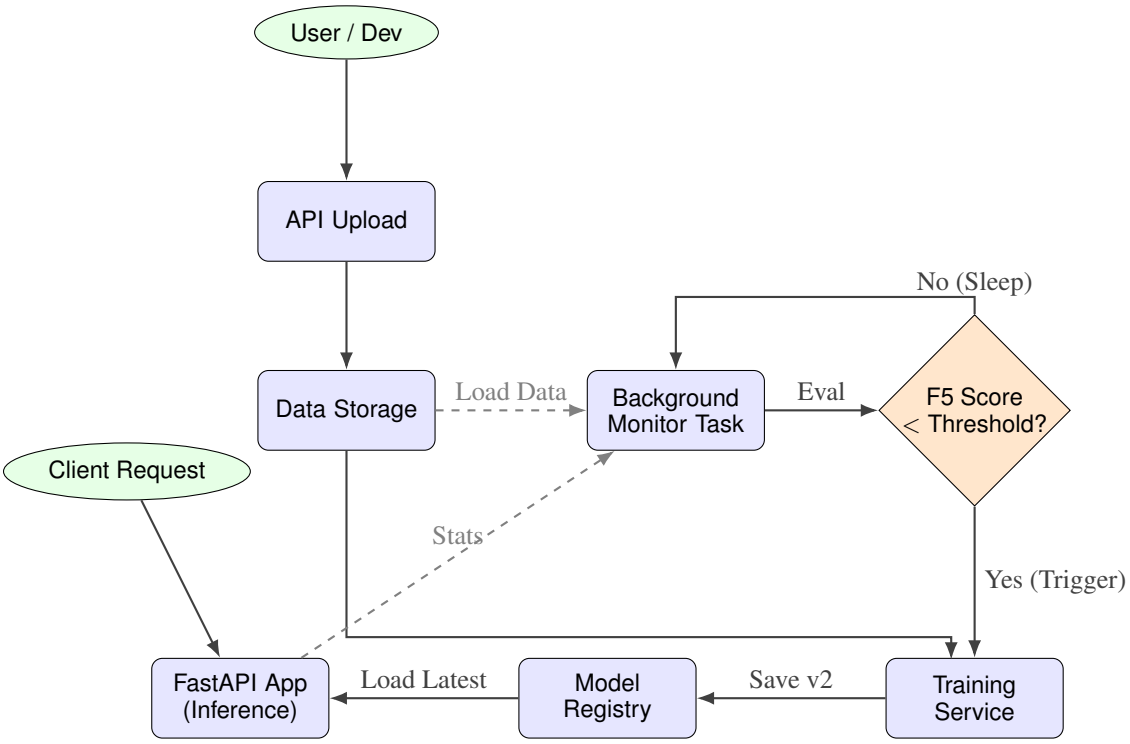

Figure 1: The MLOps lifecycle of our project.

5.4 Model Registry & Dynamic Deployment

The project implements a file-system-based model registry to manage versioning. Upon successful training, the new model artifacts (weights, config, tokenizer) are saved to a timestamped directory. The application includes logic to identify the most recently created model folder. When the application restarts (or when a reload is triggered), it automatically loads the latest version from the registry, ensuring the production environment always runs the best available model.

6 Evaluation Results

Our evaluation results are 3 fold. 2 of the evaluation tests are shared on the Github page of the project. The first such evaluation is a general evaluation of the API. All endpoints are tested thoroughly through a series of test cases. The second such evaluation is a load test. This test send heavy requests to the API, both countwise and sizewise. Here we share the expected outputs of tests and recommend the reader to perform the test on their own too, to validate our claims. We however claim that all validation and load tests are passing on the criteria that:

- All validation test must run successfully. (`test_api.py` on Github)
- Certain load tests run successfully, all load tests must fail immediately after running them again (`stress_test_api.py`)

Load tests fail when run again, because limitations are applied on the backend. An IP address is limited to 20 requests per minute at most. There are also text length limits configurable in the backend.

We also report our dashboard as the final evaluation in Figure 2. We however, strongly encourage the reader to run the app locally and view the dashboard themselves.

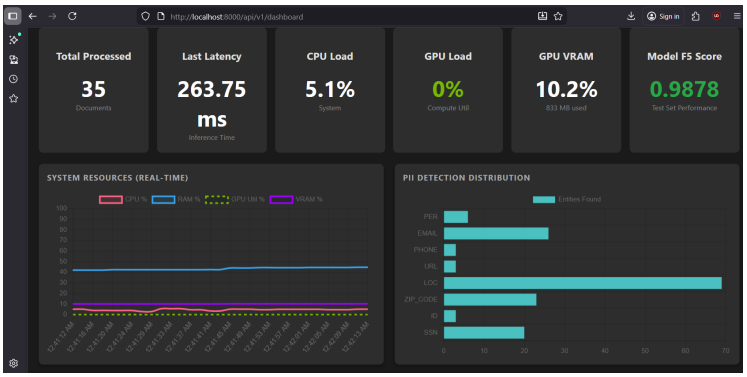

Figure 2: The live dashboard from our API (with GPU)

7 Deployment and Monitoring

The FastAPI Redis PII Cleaner utilizes a container-first deployment strategy designed for portability across development and production environments. Observability is built directly into the application layer, removing the need for external agents for basic monitoring.

The application is fully dockerized and adapted based on the existence of one or multiple GPUs. A base docker configuration sets up the Redis service and FastAPI parameters. And if applicable, another docker configuration sets up the GPU and NVIDIA related configurations. The application attempts to load NVIDIA libraries too, but degrades gracefully to CPU-only monitoring and inference if GPU hardware is unavailable.

To provide monitoring and observability tools, background task polls system resources every 10 seconds. CPU and RAM usage, total documents processed, frequencies of PII objects masked, the last measured latency and if applicable, volatile GPU usage and VRAM usage are tracked by this system. These metrics are shown in a dashboard. And parallel to that dashboard, a drag-and-drop processing page is also provided.

The application also utilizes heavy logging. There exists a separate logging store defined in the configurations of the application, where all logs are saved to for future retrieval. API endpoints, backend services, training and evaluation loops all heavily log their processing stages to facilitate backtracking events and possible bugs.

8 Ethics and Privacy Notes

The FastAPI Redis PII Cleaner is designed with ”Privacy by Design” principles, ensuring that sensitive data is processed securely, minimized, and retained only for as long as necessary. However, as an ML-based probabilistic system, it entails specific ethical considerations regarding reliability and bias.

The system is architected primarily for batch processing with a strict Time-To-Live (TTL) policy. The application logging configuration implements strict retention policies. Error logs are retained for 7 days, while general application/debug logs are retained for only 3 days.

On the side of model limitations, there is a non-zero risk that the model may fail to identify non-standard PII. The model may also incorrectly flag common nouns as entities. The BERT model used may also exhibit demographic biases.

On the implementation of the API, to prevent Denial of Service (DoS) attacks and resource exhaustion, the API enforces a rate limit of 20 requests per minute per IP address. This protects the GPU resources and ensures fair usage. The application runs inside a Docker container, providing a layer of isolation from the host system.

Finally, we reiterate that our system is compliant with the core enforcements of GDPR, minimizing data retention mainly through Redis TTL limits and log retention limits. The dashboard also pro-

vides real-time visibility into what the system is doing (throughput, latency, entity detection rates), allowing operators to audit system behavior continuously.

9 Contribution Breakdown

The API code is built incrementally. All students, at some point have contributed to the code of the project. When a new idea was accepted by the project team, a team member implemented it and updated the project accordingly.

The documentation part of the project, notably reports and the slides have also been shared accordingly. Below is an incomplete and approximate (due to the sheer number of commits and changes to the project structure) list showing who did what:

- **Ahmet Erdem**: API logic backend, Model Training API, Final Report
- **Burak Aydın**: Model API, Dockerization, Slides, Progress Reports
- **Abdülkadir Külçe**: Dashboard and UI, Validation and Load tests, Evaluations

It should be still noted that each listed task is not exclusive to a single teammate, and the listing and matching are approximate.

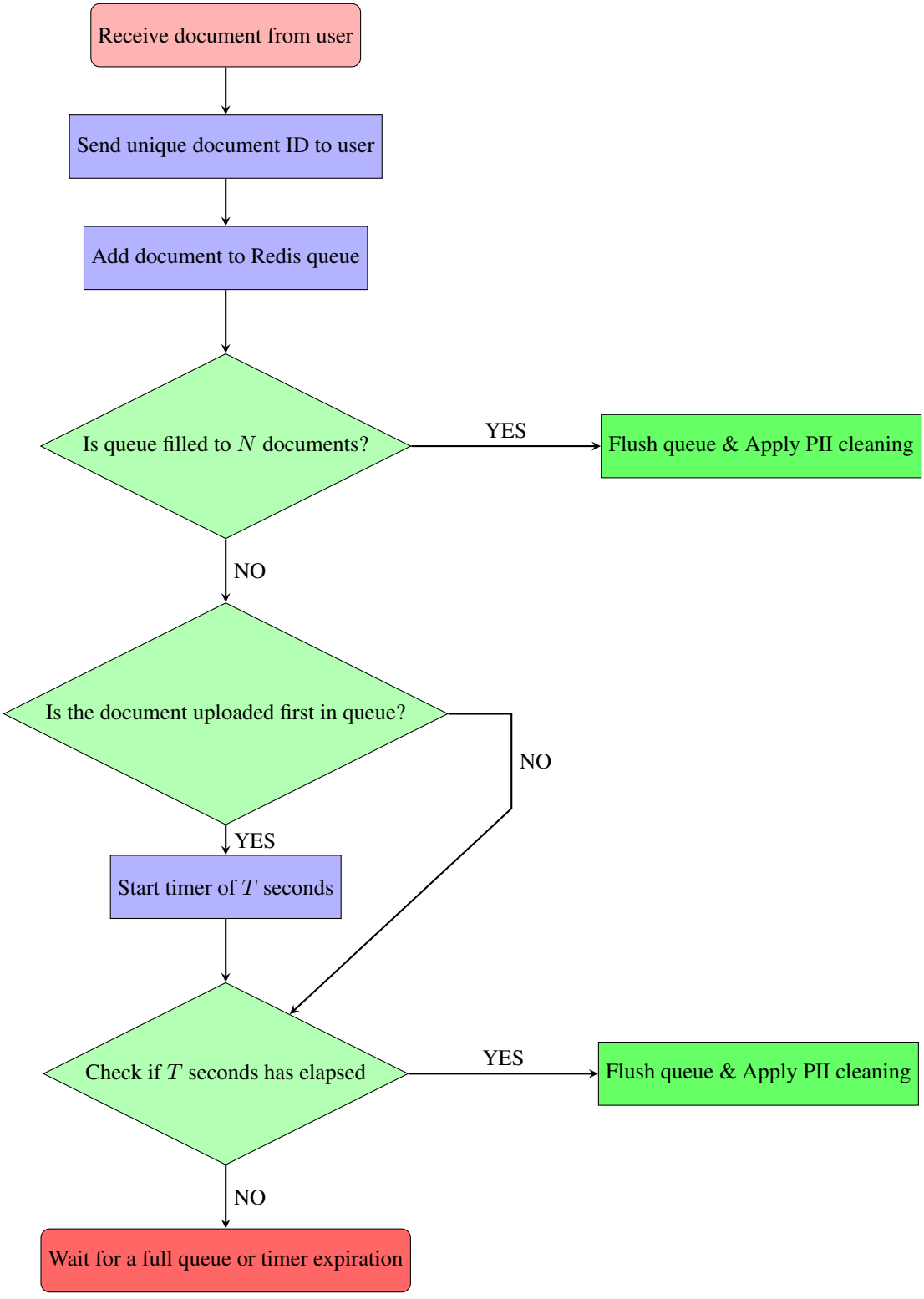

Figure 3: Logic flow of the FastAPI/Redis backend upon a user POST request.