SCHOOL OF COMPUTATION, INFORMATION
AND TECHNOLOGY - INFORMATICS

TECHNICAL UNIVERSITY OF MUNICH

Master's Thesis in Informatics

Improving the Functionality and Performance of a Text Privatization Benchmarking Platform

Ahmet Bilal Akın

SCHOOL OF COMPUTATION, INFORMATION
AND TECHNOLOGY - INFORMATICS

TECHNICAL UNIVERSITY OF MUNICH

Master's Thesis in Informatics

Improving the Functionality and Performance of a Text Privatization Benchmarking Platform

Verbesserung der Funktionalität und Leistung einer Benchmarking-Plattform für Textprivatisierung

Author:	Ahmet Bilal Akın
Supervisor:	Prof. Dr. Florian Matthes
Advisor:	Stephen Meisenbacher
Submission Date:	26.11.2025

I confirm that this master's thesis in informatics is my own work and I have documented all sources and material used.

Munich, 26.11.2025 Ahmet Bilal Akın

Location, Submission Date Author

AI Assistant Usage Disclosure

Introduction

Performing work or conducting research at the Chair of Software Engineering for Business Information Systems (sebis) at TUM often entails dynamic and multi-faceted tasks. At sebis, we promote the responsible use of *AI Assistants* in the effective and efficient completion of such work. However, in the spirit of ethical and transparent research, we require all student researchers working with sebis to disclose their usage of such assistants.

For examples of correct and incorrect AI Assistant usage, please refer to the original, unabridged version of this form, located at this link.

Use of *AI Assistants* for Research Purposes

I have used AI Assistant(s) for the purposes of my research as part of this thesis.

Yes ○ No ○

Explanation: I used AI assistants to support my academic writing, specifically to improve grammar and sentence structure. Additionally, I utilized them for code review and debugging.

I confirm in signing below, that I have reported all usage of AI Assistants for my research, and that the report is truthful and complete.

Munich, 26.11.2025	Ahmet Bilal Akın
Location, Date	Author

Acknowledgments

First, I would like to thank my wife, Ecem. Your patience and understanding made this work possible. I am also grateful to my family for their constant support. I would like to express my gratitude to Prof. Dr. Florian Matthes for the opportunity to write this thesis at the sebis chair. Finally, I appreciate the valuable guidance provided by my advisor, Stephen Meisenbacher, throughout this research.

Abstract

The increasing volume of digital text data containing sensitive information has increased the development of automated text privatization methods. However, we know that evaluating these methods is not easy due to the trade-off between privacy preservation and text utility. While the existing *PrivBench* platform offers a modular framework for such evaluation, its initial implementation had limitations such as task execution latency caused by container initialization, and the system also lacked version control.

This thesis presents the design and implementation of an extended PrivBench platform architecture. We address operational and structural problems to improve the system. Regarding performance, a Container Manager service is developed to manage Docker containers, and we are able to shift the execution model from cold to warm starts. For reproducibility, a versioning system is integrated. It is capable of tracking module updates and linking benchmark scores to specific system configurations. Additionally, we improve the user experience via a submission queue mechanism and template-based submission metadata.

Experimental evaluation results across ten repeated runs show improved runtime stability and reduced overhead. Also, the introduction of versioning and structured queuing elevate the platform to a production-ready benchmarking environment. Thanks to these contributions, researchers can conduct standardized and reproducible assessments in our platform.

Overall, this work delivers a more reliable, reproducible, and user-friendly *PrivBench* platform. Furthermore, it establishes a solid foundation for future research in text privacy within Natural Language Processing (NLP).

Kurzfassung

Das zunehmende Volumen digitaler Textdaten, die sensible Informationen enthalten, hat die Entwicklung automatisierter Methoden zur Textprivatisierung verstärkt. Es ist jedoch bekannt, dass die Evaluierung dieser Methoden aufgrund des Zielkonflikts zwischen der Wahrung der Privatsphäre und dem Textnutzen nicht einfach ist. Während die bestehende PrivBench-Plattform ein modulares Framework für solche Evaluierungen bietet, wies ihre ursprüngliche Implementierung Einschränkungen auf, wie etwa Latenzen bei der Task-Ausführung durch Container-Initialisierung; zudem fehlte dem System eine Versionskontrolle.

Diese Arbeit präsentiert den Entwurf und die Implementierung einer erweiterten Architektur der PrivBench-Plattform. Wir adressieren operative und strukturelle Probleme, um das System zu verbessern. Im Hinblick auf die Leistung wird ein Container-Manager-Dienst entwickelt, um Docker-Container zu verwalten, wodurch wir in der Lage sind, das Ausführungsmodell von Kalt- auf Warmstarts umzustellen. Für die Reproduzierbarkeit wird ein Versionierungssystem integriert, das Modulaktualisierungen nachverfolgen und Benchmark-Ergebnisse mit spezifischen Systemkonfigurationen verknüpfen kann. Zusätzlich verbessern wir die Benutzererfahrung durch einen Warteschlangenmechanismus für Einreichungen sowie vorlagenbasierte Metadaten.

Die Ergebnisse der experimentellen Evaluierung über zehn wiederholte Durchläufe zeigen eine verbesserte Laufzeitstabilität und einen reduzierten Overhead. Zudem heben die Einführung der Versionierung und das strukturierte Queuing die Plattform auf das Niveau einer produktionsreifen Benchmarking-Umgebung. Dank dieser Beiträge können Forschende standardisierte und reproduzierbare Bewertungen auf unserer Plattform durchführen.

Insgesamt liefert diese Arbeit eine zuverlässigeren, reproduzierbareren und benutzerfreundlicheren PrivBench-Plattform. Darüber hinaus schafft sie eine solide Grundlage für zukünftige Forschung zur Textprivatsphäre im Bereich Natural Language Processing (NLP).

Contents

1 Introduction

1.1 Motivation and Context

In the digital age, the data produced from various resources, such as social media and online websites, grows remarkably. Although this data is helpful for research, analytics, or technology, it also brings some problems. For instance, digitalization in the healthcare industry generates a large amount of electronic data containing sensitive and confidential information [1]. This type of information can also be found on social media, and addressing this problem with strong methodologies is crucial.

Text privatization is one of the promising and primary solutions to this issue. It is the process of removing, masking, or replacing sensitive information from the text [2]. Since it is one of the well-known solutions, researchers have developed various privatization techniques. We observe various methods, ranging from rule-based applications to neural network-based generative models.

Having several text privatization methods introduces an important challenge. Comparing the original text and the privatized text is not straightforward. The trade-off between privacy and utility is the primary issue in this area [3]. A method with strong privacy properties might render the text useless. On the other hand, if we use a method that preserves the text's utility completely, we might end up with a text that fails to protect the sensitive data. Developers and researchers require a set of tools to evaluate the quality of their models. Therefore, a benchmarking platform that allows them to compare their approaches is beneficial [4].

1.2 Problem Statement

To tackle this problem, the benchmarking application *PrivBench* was implemented. It is a system that provides various text privatization methods, allowing users to evaluate their submissions. This platform consists of different modules, and each of which runs the corresponding text privatization method. For each module, an algorithm and a dataset are required.

The initial implementation is functional and has a good baseline. However, it suffers from certain limitations that create bottlenecks in the application. The core issues are related to performance, usability, and reproducibility. There is room for improvement in the performance of the evaluation pipeline, and it lacks the important versioning component that allows users to compare their results over time. These items prevent the scaling of the application, and to meet the demands of the researchers and developers, they need to be resolved.

1.3 Research Questions

This thesis aims to address the limitations mentioned earlier by answering the following research questions:

- **RQ1:** How can an existing modular evaluation framework for text privatization be extended to enable the standardized, reproducible, and interoperable measurement of privacy, utility, and performance?
  - How can modules be refined to improve overall evaluation?
  - What should each module measure (e.g., privacy, utility)?
  - How to standardize these measurements?
- **RQ2:** How can the platform be designed and engineered to fulfill key software quality attributes such as performance, maintainability, extensibility, and fairness in the benchmarking process?
  - How to enhance performance, maintainability, and extensibility?
  - How to manage versioning for evolving systems?
  - What technical or conceptual needs ensure robustness and fairness?

1.4 Thesis Objectives and Contributions

The main objective of this thesis is to enhance the current platform, to increase its usability, performance, and to make it a reproducible and production-ready system. Although the details of the implementation will be stated in Chapter 4, we briefly describe the contributions of this thesis:

- **Performance and Scalability:** We start with the platform's performance issues. Since they are the most critical points to address, resolving them is the priority. The issues include an asynchronous queue mechanism with relevant information, a container manager to eliminate task startup delays, and optimizing the runner script for different use cases.
- **Reproducibility and Interoperability:** We design and build a versioning system. This new feature enables users to view their benchmark results over time and compare them as needed. This architecture is integrated into the application's admin panel and user interfaces.
- **Extensibility and Operational Readiness:** For the computationally intensive modules, we added device configuration support for Central Processing Units (CPUs)/Graphics Processing Units (GPUs). This enhancement helps the platform to manage multiple

submissions when there is a high load. Furthermore, we adapt the system for production deployment.

- **User Experience Improvements:** We introduce reusable metadata templates to simplify the submission flow and enhance the User Interface (UI) to provide a better user experience.

1.5 Thesis Outline

The remaining part of this thesis is structured as follows:

- **Chapter 2** provides the background and related work. It describes the fundamentals of text privatization, benchmarks in Natural Language Processing (NLP), and includes a separate section on related research.
- **Chapter 3** presents an analysis of the existing *PrivBench* system. It outlines the architecture, principal components, and modules, while also mentioning the existing challenges and limitations within the system.
- **Chapter 4** is the main chapter of the thesis. It details the design and implementation of the architectural enhancements, which include the asynchronous queue, container manager, and versioning system.
- **Chapter 5** describes the evaluation of the extended platform. It provides quantitative results from performance benchmarks and usability validations.
- **Chapter 6** discusses the implications of the results, addresses the research questions, and outlines future work and limitations.
- **Chapter 7** concludes the thesis by summarizing the key contributions and providing final remarks on the work.

2 Background and Related Work

2.1 Fundamentals of Text Privatization

Texts often include different forms of personal and sensitive data. In some cases, this data can become publicly accessible, pushing people to consider privacy concerns and potential harms. Sensitive information can include personal identifiers, such as a name or email address, or it can be something that indirectly reveals a person's identity, such as job details or age. To prevent issues that can harm people or specific groups, sensitive parts of the text must be handled properly before the data becomes public or is handed over to researchers for analysis [5].

Text privatization mainly refers to changing the content of the text so that individuals cannot be recognized from the privatized information. The application of this concept can be found in various areas, such as medical records, social media, and other text-based content that may contain sensitive information. Regulations such as General Data Protection Regulation (GDPR) in Europe [6] and Health Insurance Portability and Accountability Act (HIPAA) in the United States [7] have specific requirements about data protection and how it should be handled. This pushes researchers to incorporate privacy methods more into their work.

There are different types of sensitive information. The obvious ones, such as personal name and address, are called identifiers. If some information becomes identifying when combined with other knowledge, it is called a quasi-identifier. A job, age, or zip code can be good examples. Furthermore, information such as political opinions and diagnoses about someone's health requires extra caution and can be categorized as highly sensitive [8].

Various approaches protect sensitive information. Anonymization attempts to hide the original identity from the text, so that no one can determine its actual content. The main idea of de-identification is to remove or hide identifiable values. Moreover, pseudonymization aims to ensure that the text structure remains understandable after replacing the identifiers with specific placeholders [9]. Although these methods are pretty helpful, anonymizing an entire text is still challenging. This is because the language contains many parameters and details, such as context and implication. Therefore, using such information, it may be possible to identify the individuals [10].

This issue gives rise to the well-known trade-off in text privatization: privacy versus utility. When we add more privacy to the text, we risk losing useful information. If it is hidden too much, we may not be able to use it in tasks such as machine learning and information retrieval.

Conversely, if we decide to keep more information, the identification of individuals can still be easily possible. Thus, balancing privacy and utility is one of the primary challenges of text privatization researchers [11].

Finally, since there are different ways to test privacy methods, each researcher might have a system with different criteria or a set of tools. Therefore, comparing different approaches is hard, especially in a reproducible way. A standardized evaluation workflow helps researchers understand how their methods protect privacy while maintaining relevant information at the same time. The benchmark this thesis improves also shares the same motivation behind it.

2.2 Text Privatization Approaches

We have seen various methods developed to protect sensitive data in text over time. Naturally, these approaches may have certain differences in preserving the level of sensitive data or in how they handle large amounts of text. We can generally divide the existing methods into three categories: rule-based techniques, machine learning–based approaches, and neural or generative privacy models.

2.2.1 Rule-based Approaches

Rule-based methods typically follow a pattern to find sensitive data. These applications are usually straightforward and work better in well-defined use cases such as hospital records [5]. For example, phone numbers can be replaced with a regular expression, and this can work properly. However, when it comes to more complex and ambiguous texts, this approach may not work as intended.

2.2.2 Machine Learning–based Approaches

Unlike rule-based methods, these models learn from the statistical patterns that exist in the data. In this way, they have the flexibility of learning from the examples. Famous ones include Support Vector Machines (SVMs) and Conditional Random Fields (CRFs), which have been used in Named Entity Recognition (NER) related problems. They handle the unstructured data better, but without feature engineering and labeled data, they may also perform poorly. One further issue is that if there is not enough data present, they may struggle with generalization across different fields [12].

2.2.3 Neural and Generative Privacy Models

Recent improvements in NLP have facilitated the implementation of new and powerful Deep Learning (DL) models that better understand the context. Particularly, transformer-based language models, such as Bidirectional Encoder Representations from Transformers (BERT) and Generative Pre-trained Transformer (GPT) variations, can maintain the grammar quality of the output and are good at identifying sensitive information in the text [11]. In certain cases, they generate modified outputs without losing the meaning, i.e., they do not remove information but replace it with something meaningful [13]. Reaching a better privacy-utility trade-off is comparatively easier in these types of models, but they are costly because of their computationally intensive nature. Also, they may leak private data if not implemented carefully.

2.2.4 Comparison Overview

The table below summarizes the main characteristics of each approach:

Table 2.1: Comparison Overview of Text Privatization Approaches

Method Type	**Strengths**	**Weaknesses**
Rule-based	Easy to implement; performs well on structured data (e.g., medical records).	Fails on ambiguous or informal data; poor generalization across domains.
Machine Learning–based	Learns from annotated data; adaptable to new text types	Requires large labeled datasets; feature-engineering effort; may not generalize well to unseen domains.
Neural / Generative	Understand context and semantics; produces coherent output; strong privacy–utility balance.	Computationally expensive; possible privacy leakage; lower interpretability.

2.2.5 Summary

To summarize, we see a shift from rule-based methods to more advanced neural techniques. Each approach has certain advantages and disadvantages, but their evaluation still depends on the trade-off between privacy and utility. Therefore, benchmark platforms that measure these properties in a standardized way are relevant [10]. In the next section, we will look at the evaluation of privacy and utility.

2.3 Measuring Privacy and Utility in Text Privatization

Evaluation of text privatization methods is possible using quantifiable metrics that explain how much valuable content remains in the text and how effectively the model hides sensitive information. This section discusses how privacy, utility, and performance are measured in the current research, considering their interaction with benchmarking.

2.3.1 Privacy Metrics

Privacy refers to the degree to which identifiable or sensitive information has been hidden in the transformed text. Traditional evaluation approaches are usually based on NER systems, and they look for whether the personal information, such as name and address, is removed or not. Rule-based methods also verify if the recall of sensitive data is still available in the text.

Advanced techniques aim to find out re-identification risk, which refers to the likelihood that an individual can be inferred from the privatized text. This approach has been discussed in de-identification frameworks and synthetic data generation studies [14].

Additionally, Differential Privacy (DP) provides a formal mathematical definition of privacy. It ensures that the probability of identifying an individual does not change radically, regardless of whether that person's data is included in the dataset [15]. Although originally developed for structured data, recent research work [11] has shown its adaptation for various natural language tasks.

2.3.2 Utility Metrics

The utility explains the remaining useful information and readability after privatization. The most straightforward way to measure it is by making a comparison between the original text and its privatized version using similarity metrics such as Bilingual Evaluation Understudy (BLEU) and Recall-Oriented Understudy for Gisting Evaluation (ROUGE) [16] [17]. Although these metrics are initially implemented to evaluate summarization or machine translation, they are also commonly used in text privacy.

There are more advanced metrics such as BERTScore [18] developed recently with the recent progress in the NLP field. BERTScore uses contextual embeddings to understand how semantically similar two sentences are. It utilizes transformer-based models like BERT for the embeddings. These types of metrics can be particularly useful when evaluating a paraphrased or rearranged text.

Researchers show that utility can also be measured indirectly in a downstream task. In the PrivacyGLUE benchmark [10], the model's performance is demonstrated on various NLP tasks when trained on original versus privatized datasets. If there is a slight decrease, it is a better utility.

2.3.3 The Privacy–Utility Trade-off

Balancing privacy and utility is a well-known challenge in this field. When there is stronger privacy, we have the cost of lower utility. This concept is named the privacy-utility trade-off.

This trade-off has been defined formally in the DP literature [19]. As can be seen in Figure

2.1, increasing privacy usually results in a decrease in utility. The aim is to find a balanced region where both privacy and utility are at reasonable numbers. The figure is a conceptual illustration adapted from the work of Dwork and Roth. We should also note that researchers state that optimality in this curve depends on the context; i.e., we might need higher privacy in one area, such as healthcare, whereas in text summarization, better utility is more crucial.

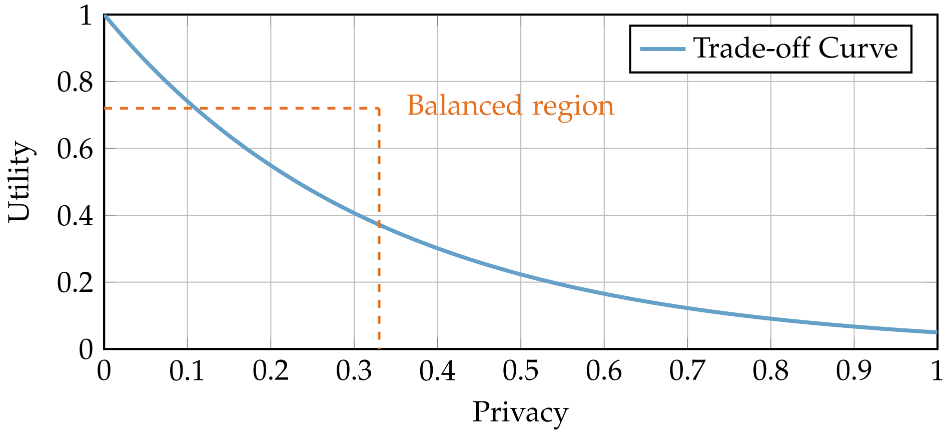


Figure 2.1: Illustration of the privacy–utility trade-off, adapted from [19]. Increasing privacy (rightward) typically reduces text utility (downward). The balanced region indicates optimal trade-offs.

2.3.4 Performance as a Third Dimension

Performance is an important aspect of evaluation, together with privacy and utility. It can be considered as a third evaluation dimension, since runtime efficiency, scalability, or reproducibility are vital metrics that indicate the success of the model. For example, suppose a method with good privacy and utility fails to become consistent or is not fast enough. In that case, it might be impractical to use it in a research or production environment [20]. One recent benchmark framework Holistic Evaluation of Language Models (HELM) [21] emphasizes this important issue, mentioning that performance should be considered alongside accuracy.

Likewise, it is essential to assess how quickly the text privatization benchmark processes submissions or handles large datasets. That said, considering performance, privacy, and utility together gives us a more comprehensive and realistic overview.

2.3.5 Summary

To have a thorough text privatization evaluation, we should look at aspects of privacy, utility, and performance together, as they cover different parts of the evaluation's quality. Modern benchmarks such as PrivacyGLUE and HELM show that combining these aspects yields a more realistic scenario when comparing different models. In the next section, we review the existing benchmark systems that integrate these concepts.

2.4 Benchmarks and Evaluation Platforms

Benchmarking plays an important role in comparing advanced NLP systems. Standardized workflows and datasets for evaluating different models are the core properties of the benchmarks. It is especially important for text privatization systems, because researchers can measure how well their models perform, and do so systematically or periodically.

2.4.1 Evolution of Benchmarking in NLP

General NLP evaluation systems such as General Language Understanding Evaluation (GLUE) [22] and its successor SuperGLUE [23] introduced the concept of scaled, standardized benchmarking. These frameworks have various tasks, such as question answering and sentiment analysis, with a unified scoring mechanism. Additionally, leaderboards, where researchers can compare their models against standardized data and metrics, become more common.

Following the introduction of these, we have begun to see domain-specific benchmarks in various areas. For example, Biomedical Language Understanding Evaluation (BLUE) [24], which has a focus on biomedical text, conducts work that combines technical language with privacy-concerned datasets. Furthermore, SQuAD [25] is a notable example of a question-answering benchmark, which is an important task in text anonymization systems.

The primary focus of these benchmarks is general language understanding or measuring accuracy. In other words, they are not specifically implemented for text privacy. In any case, their modular benchmarking approaches affected the privacy research.

2.4.2 Benchmarks for Privacy and De-identification

The benchmarks with a focus on privacy emerged later. Comparing anonymization and de-identification systems triggers this. One early work was the Informatics for Integrating Biology and the Bedside (i2b2) de-identification challenge [26]. It focuses on removing protected health information from medical records and provides annotated datasets and standardized metrics, such as precision and recall. One of the important aspects of this research is that it shows the feasibility of evaluating text anonymization models in a controlled environment.

A more recent work, PrivacyGLUE [10], integrates the GLUE's multi-task benchmarking idea to privacy tasks. It measures both utility and privacy. Another important study is Text Anonymization Benchmark (TAB) [27], which creates a specific corpus and evaluation framework for text anonymization. TAB is one of the most extensive resources for the assessment of anonymized data. It provides original and anonymized text as a pair, and has also standardized metrics for privacy measurement.

Other than privacy-oriented platforms, more general evaluation systems, such as HELM [21] and Beyond the Imitation Game Benchmark (BIG-Bench) [28], are also relevant to text privacy benchmarking, as they share common principles, including reproducibility and transparency.

2.4.3 Challenges in Benchmarking Text Privatization

Although there has been a significant progress, we see various challenges in benchmarking text privatization [29].

- Data is limited because real sensitive text cannot be shared easily. That is why most current and recent datasets use synthetic or anonymized data, which may not accurately capture real-world diversity.
- We still have metric inconsistency. Studies often use incompatible measures of privacy and utility, which is undesirable if we want to compare results across different systems.
- Versioning and reproducibility are not appropriately addressed. Updates to datasets or models make the older benchmark results obsolete and incomparable with the newer ones.

Performance aspects are also not frequently reported, although these criteria do not directly measure privacy properties, but rather determine if a system can be realistically deployed.

2.4.4 Toward Integrated Benchmarking Frameworks

The benchmarks in NLP and privacy have a trend from accuracy-specific comparisons to comprehensive, modular evaluation systems [30]. We identify three important characteristics of the future benchmarks:

- **Standardization:** Consistent tasks and metrics across privacy, utility, and performance.
- **Reproducibility:** Version-controlled datasets and models together with transparent evaluation environments.
- **Extensibility:** The ability to integrate new modules or privacy techniques without invalidating previous results.

By combining these principles, benchmarking platforms provide a more insightful and scalable evaluation of text privacy.

2.5 Summary and Research Gap

Text privatization has evolved from simpler rule-based systems to neural approaches with more data-driven techniques. Initial methods had patterns that mostly worked for structured data but were not successful in terms of generalization and flexibility. Machine Learning

(ML) techniques such as CRFs and SVMs helped identify statistical patterns and learn from the annotated data. Moreover, modern transformer-based models such as BERT and GPT brought a better contextual understanding and semantic ability. Despite these improvements, we still have the privacy-utility trade-off, i.e., achieving a balance between the two remains a bottleneck.

Researchers use different metrics, such as precision and recall, or similarity measures, such as BLEU, ROUGE, and BERTScore, to evaluate utility. These are often used inconsistently, and this creates a comparability problem. Additionally, some systems prioritize accuracy over other factors, such as reproducibility and efficiency, for real-world deployment.

Frameworks such as GLUE, SuperGLUE, and domain-specific benchmarks like BLUE, i2b2, and the TAB showed the importance of shared datasets and standardized evaluation in NLP. PrivacyGLUE and HELM, improve this idea with fairness, reproducibility, and modular design. However, it is not easy to design a benchmark that has full integration of privacy, utility, and performance metrics.

Current approaches offer valuable insights, but there is still room for improvement. We lack a benchmarking platform that evaluates text privatization systems in a comprehensive and performance-aware way [31]. Therefore, this thesis aims to fill this gap by enhancing *PrivBench* into a benchmarking system that measures privacy and utility under standardized conditions in a reproducible manner. The following chapter analyzes the existing system and describes its key limitations.

3 Analysis of the Existing PrivBench System

3.1 Introduction and Context

PrivBench platform was developed as a standardized environment for evaluating text privatization methods. Its main goal is to support researchers and developers while they assess their privacy modelling approaches. It also integrates different evaluation modules and provides a system that users can use to measure their model's performance on a shared dataset.

We first describe the current system in detail before explaining the new enhancements. Understanding the initial design and workflows is important to catch the strengths and weaknesses, so that we can achieve a conclusive result. This analysis is important and explains the technical architecture that serves as guidance for further improvements.

This chapter begins with an overview of the system and its main actors, followed by a discussion of the overall architecture, workflow, and data model. The last sections identify the platform's current limitations and challenges. These findings help us to build the motivation for the enhancements that we explain in the subsequent implementation chapter.

3.2 System Overview

In the *PrivBench* platform, users can evaluate text privatization models through a defined benchmarking workflow. We have two actor groups: users (researchers or developers) and admins.

Users can submit their privatized text or model outputs, which are then evaluated by the benchmark's automated modules in the background. In the flow, they need to download the dataset for each module and upload their privatized versions. During the evaluation phase, they can see the submission's progress and view the aggregated evaluation result on a web-based UI. Thanks to this flow and interaction, we have consistency across different submissions, and the evaluation process is more straightforward.

Administrators maintain the overall module infrastructure, and they manage datasets and modules. In other words, they can update the evaluation pipeline by adding or deleting new modules or datasets. Use case diagram of the platform can be found in Figure 3.1.

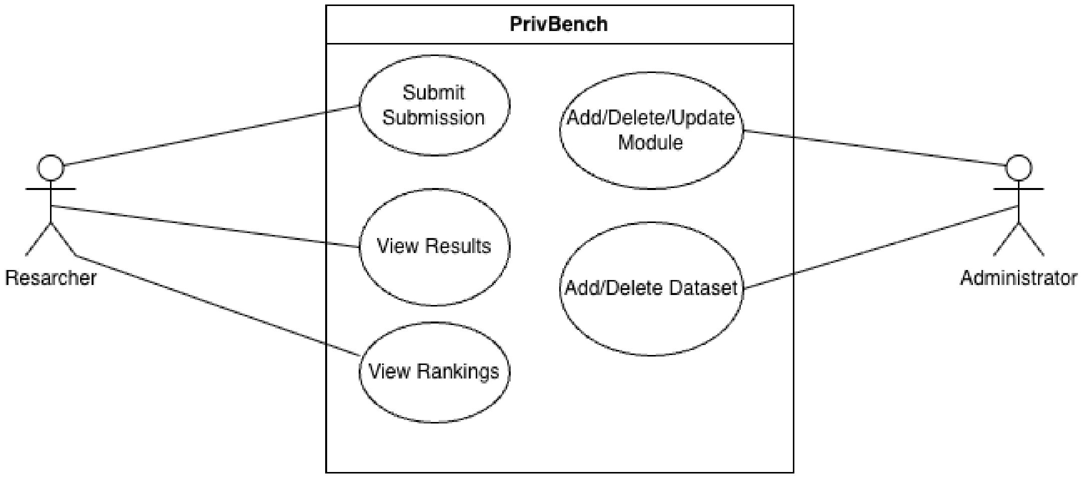


Figure 3.1: Use Case Diagram of the *PrivBench* platform showing user and administrator interactions with core functionalities.

3.3 Architectural Design

The *PrivBench* platform has a modular and service-oriented architecture that provides flexibility and scalability for text privatization methods. The UI, evaluation logic, and execution environments are separated as layers. Thanks to that, different components can extend on their own. Additionally, we need to maintain our system to ensure it remains consistent and controlled.

From a high-level perspective, we have three layers in our platform: the frontend, the backend, and the evaluation environment. These layers are supported by a shared database for persistent data management. Figure 3.2 shows the overall high-level architecture and the interactions of components.

3.3.1 Frontend (User Interface)

We have a frontend that provides a web UI where users can upload submissions, see their results, or view the leaderboard. Moreover, the communication with the backend is handled by Representational State Transfer (REST) Application Programming Interface (API) endpoints. The frontend's focus is on creating an application that users can easily understand. In this way, they do not need to worry about the complex benchmarking process that is handled in the backend. For instance, they can submit their submission and compare their results to other users directly from the ranking dashboard.

3.3.2 Backend (Core Services and APIs)

Our backend is the core component of the application. It handles the incoming requests from the frontend and manages the evaluation task pipeline. Each submission has a unique identifier that links the user input. This makes the submissions traceable and visible. One of the most important responsibilities of the backend is delegating time-consuming evaluation jobs to an asynchronous task queue. In this way, it does not execute them directly, and blocking requests are avoided.

3.3.3 Task Queue (Asynchronous Orchestration)

We distribute the evaluation tasks to different workers through a message-based queue system. Each worker completes one module evaluation and returns the result to the backend with the respective score. We are also able to see the status of the tasks to check their progress. Thanks to this architecture, we have a better throughput.

3.3.4 Evaluation Modules (Containerized Components)

Each evaluation module runs independently and is responsible for a specific analysis task. For example, one module may concentrate on privacy using a NER model, while another computes semantic similarity to assess utility. We use Docker to containerize the modules, and this brings consistency and isolated dependencies. For each module, we have distinct dependencies, since each has a different functionality.

The advantage of the modular approach is that it is easily extensible. We can add new modules with their configurations without changing the features in the platform. This design decision is compatible with modern benchmarking practices.

3.3.5 Database

All persistent information, such as user profiles, submissions, module metadata, and benchmark results, is stored in a relational database. It helps to ensure data integrity and traceability.

Overall, the PrivBench architecture is designed for modularity and transparency. The separation between components allows the system to manage complex evaluations. As we discuss in the later sections, there are also certain technical limitations identified. They are mostly related to performance, versioning, and container management.

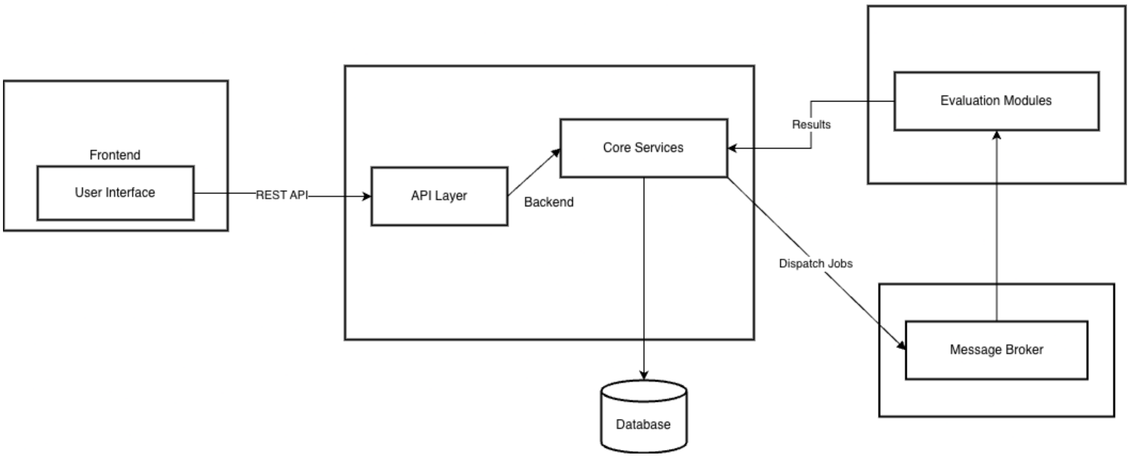


Figure 3.2: Component-level architecture of the *PrivBench* system illustrating the interactions between the frontend, backend, task queue, evaluation module, and database.

3.4 Workflow Analysis

We explain the execution of the workflow in the benchmark in this section. We describe how data and control flow when a user makes a submission. It starts with when a user submits their privatized text and ends with the generation of the aggregated scores, which summarize the overall performance. To identify the bottlenecks in the application, it is important to have a clear understanding of the submission workflow. Also, activity diagram of the platform can be found in Figure 3.3.

At a high level, the workflow can be divided into five sequential stages:

- Dataset Preparation
- Submission Upload
- Task Scheduling and Execution
- Evaluation and Aggregation
- Result

3.4.1 Dataset Preparation

Before any evaluation, the user downloads a predefined benchmark dataset from the platform. Each dataset has a reference text and an expected output format. Therefore, the user uploads the privatized version of this dataset, and these datasets are then used as an input for the evaluation modules.

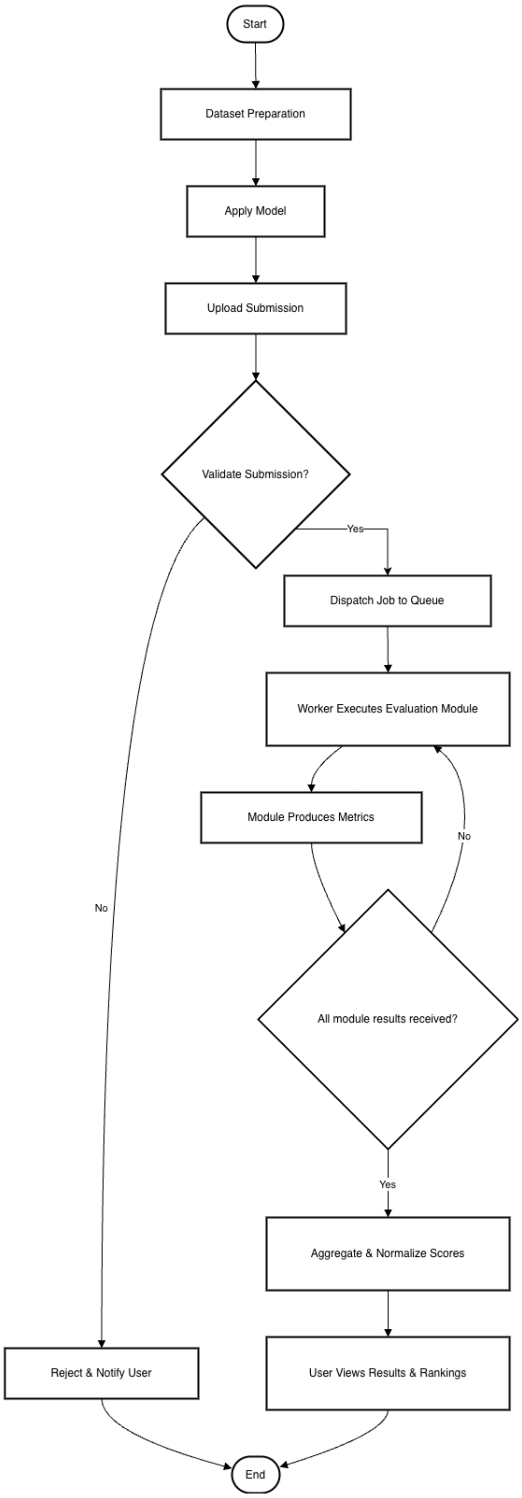


Figure 3.3: Activity Diagram of the *PrivBench* workflow from dataset preparation to result visualization.

3.4.2 Submission Upload

Once the user applies their model and generates privatized text, the output can be uploaded via the web interface. The frontend sends the submission to the backend using REST API endpoints. When backend services get the request, they validate the input (submission) and store the input data and the metadata in the database.

3.4.3 Task Scheduling and Execution

After the submission is uploaded, the backend sends it to the message queue as an asynchronous task. The queue has the ability to manage multiple tasks at the same time, so this allows us to evaluate multiple modules in parallel. Worker processes consume tasks from the queue, then execute the modules in isolated Docker containers. Each evaluation module focuses on a specific aspect, such as privacy protection or text utility, and computes a score based on a metric. When the results are ready, they are sent back to the backend.

3.4.4 Evaluation and Aggregation

When all modules finish processing, the benchmark service in the backend computes an overall score. This result represents the trade-off between privacy and utility. Users can also see the scores of each respective module and understand how their model balances these objectives.

3.4.5 Result

After aggregation, the results are stored in the database and sent to the frontend dashboard for visualization. Users can see their submissions and compare their results to other participants on the ranking page.

3.5 Data Model and Module Design

The *PrivBench* platform follows a relational data model that includes users, submissions, datasets, modules, and results. This schema is followed across all benchmark executions. Figure 3.4 provides a conceptual overview of the main entities and their interactions.

3.5.1 User Entity

Each user in the system has their own profile with personal information and authentication credentials. Users can submit multiple benchmark results over time and can see their submission history with respective performance scores.

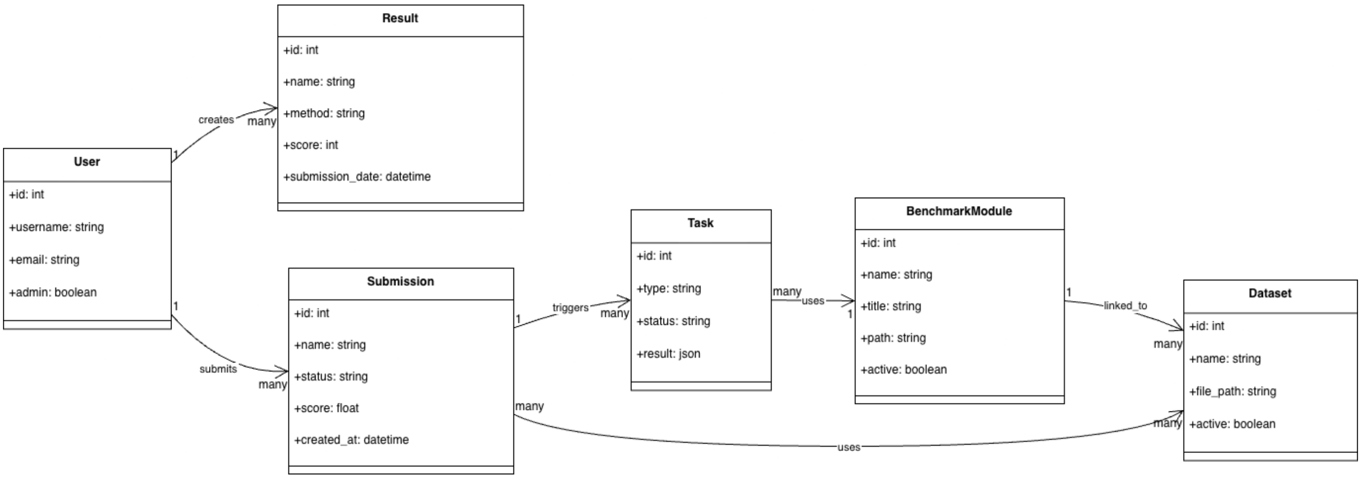


Figure 3.4: Class diagram of the initial *PrivBench* data model.

3.5.2 Dataset Entity

Datasets include the text data and metadata used in the evaluation. Each dataset has information like the dataset name, text, and specific attributes depending on the module. Additionally, datasets are linked to submissions, allowing users to view which dataset is used for each submission.

3.5.3 Submission Entity

Submission is one of the main entities of our data model. It represents outputs provided by the user, and the output is a privatized text. Each submission is tied to the datasets of each module, and the user makes the submission. It also stores certain metadata such as timestamp, score, and status. We can say that submission is the central element that links users and datasets.

3.5.4 Task Entity

For each submission, the backend creates an evaluation job, which is a task entity. A task shows the evaluation module, its execution status, and timestamps regarding the completion time of the task. Tasks are the critical structures that handle the asynchronous evaluation running in the background.

3.5.5 Module Entity

Modules contain metadata such as title and description, defined when the module is first created. Moreover, they implement the model algorithm for the specific evaluation metric, such as utility and privacy. Modules are containerized to ensure isolation during execution, and this simplifies the task-module interaction. However, in the initial system, module versions are not tracked, and this makes it difficult to reproduce results over time.

3.5.6 Result Entity

The result entity stores the outputs, such as aggregated benchmark scores of each module for a given submission. Results are used to calculate rankings, so that users can view feedback about their submissions. Since the initial implementation does not have the version information, we are unable to reproduce historical results based on the changes to the component. This limitation is addressed in improvements.

The current data model supports basic reproducibility through relationships between entities, but lacks formal versioning. This limitation restricts comparisons and traceability. This issue motivates a major extension, which is discussed in the next chapter.

3.6 Identified Limitations and Challenges

While the initial implementation of the *PrivBench* platform successfully shows the feasibility of benchmarking text privatization methods, there are certain limitations related to architecture, and usability. These issues create problems in the system's performance, maintainability, and reproducibility. Especially in the long-term usefulness, resolving these challenges is highly important. We discuss the key issues identified in the upcoming subsections.

3.6.1 Performance and Scalability

One of the most critical issues in the current system is related to performance and scalability. The backend uses an asynchronous task queue to manage evaluation jobs. However, optimization in the scheduling and resource mechanisms is required.

Tasks are executed in containerized environments, and each new evaluation causes a new container startup. In a high workload, these startup delays may lead to longer queues and lower throughput. Furthermore, there is no task distribution or queue mechanism that can relieve the computational resources.

These inefficiencies limit the system's ability to handle concurrent submissions, which is an important aspect in a benchmarking platform.

3.6.2 Reproducibility and Data Traceability

The lack of formal versioning mechanisms brings a reproducibility problem. In other words, we do not have a comparison mechanism for previous benchmark results in the existing system. When a module is updated or added, it is not possible to see the former results.

Since *PrivBench* does not maintain module version histories, it is not possible to see the specific configuration that produced the given output. Reproducibility is an important feature of a benchmark, therefore, introducing a systematic version tracking increases the credibility of the benchmark. This limitation is one of the primary motivations behind the enhancements proposed in the next chapter.

3.6.3 Usability and User Interaction

Although the frontend provides a functional interface for submitting and viewing results, its usability is limited in some pages. For instance, there is no option to cancel queued submissions. Also, the current leaderboard view lacks filtering and advanced filtering options. This makes it difficult for users to check their results in detail. The admin view has room for improvement for some UI enhancements, and there also need some advancements in module configuration logic.

3.6.4 Operational and Maintenance Complexity

Another challenge is the platform's deployment and operational management. Evaluation modules are containerized individually. Each time a submission is sent to the backend, the container starts running and stops when the evaluation is complete. This brings a container startup delay, as it always starts from scratch when a submission is made. Therefore, it introduces an extra overhead.

Additionally, the current infrastructure does not have GPU support. Some models can be computationally extensive, and using only a CPU limits the execution of such evaluations. Thus, to adapt the system for larger workloads, we need to optimize runtime performance and resource management.

3.6.5 Summary of Limitations

In summary, *PrivBench* establishes a strong baseline for modular evaluation, but we need enhancements for a production-ready benchmarking platform. The absence of versioning and limited performance optimization are the main challenges. These form the foundation for the improvements that we explain in the next chapter.

Table 3.1 summarizes the main limitations and their impact on system functionality.

Table 3.1: Summary of Key Limitations in the Initial *PrivBench* System.

Performance	Slow container startup	Low throughput under heavy workloads
Reproducibility	No version tracking for datasets or modules	Inability to reproduce or compare historical results
Usability	Limited feedback and no task control features for users	Poor user experience
Maintenance	Improvable container management and admin panel	Administrative overhead and configuration inconsistencies
Scalability	Lack of task queueing mechanism	Limited parallelization and inefficient resource utilization

3.7 Summary

This chapter presents a detailed analysis of the initial *PrivBench* system. We explain the architecture, workflow, data model, and operational challenges. The platform has a good infrastructure for benchmarking text privatization thanks to its modular design, asynchronous task execution, and automated evaluation workflow. Also, separating the components as frontend, backend, task queue, and evaluation modules allows us to extend and maintain the platform.

However, we find certain limitations that restrict the system's performance and usability. The current implementation of container management and task queueing creates a bottleneck in the performance of evaluations. Also, the versioning mechanism is missing, and this causes problems with comparison and reproducibility. User interaction also needs enhancements for a better user experience.

In the next chapter, we propose improvements based on the findings in this analysis chapter. Chapter 4 focuses on addressing the identified challenges by enhancing the current system's architecture and redesigning some of the components. We also integrate version control, improve the user experience, and have a production deployment.

4 System Design and Implementation

4.1 Introduction and Objectives

The analysis of the initial *PrivBench* platform in Chapter 3 shows us that there are certain limitations related to performance, reproducibility, and usability. The system provides a modular architecture for benchmarking text privatization; however, we have identified issues in the design that constrain scalability and reproducibility. To support multiple users and have a consistent result comparison, we need to overcome these deficiencies and develop a more capable benchmarking platform.

This chapter describes the redesign and implementation of an extended version of *PrivBench*. In the new architecture, we introduce operational, functional, and structural enhancements in the platform. The aim of these ideas is to have a more scalable and reproducible benchmarking environment. We want to explain the critical design objectives in the following subsections.

4.1.1 Scalability and Performance Optimization

The first objective is to improve the system's performance when there is a higher workload. Introducing a submission queue mechanism, reducing container startup latency, and improving the UI performance are the main changes that contribute to the platform's efficiency.

4.1.2 Reproducibility through Versioning

The second objective is to implement a versioning mechanism that tracks changes to modules and submission results. We integrate different version entities into our data model so that the benchmark results can become comparable over time. That is, when a module is added, modified, or removed, the administrator creates a new version, and we record the history of the module updates. This new version is created using the `AppVersion` model, and the records of the specific module changes are stored in `ModuleUpdate` entity.

Results are linked to their corresponding versions using the `SubmissionVersionScore` model. Thanks to this approach, we are able to maintain reproducibility even when the benchmark has certain updates over time. It is possible since each submission is associated with the set of modules and the version information.

We do not currently have dataset versioning support in the existing design, but a potential extension could be tracking dataset revisions with links to module versions. This can extend the reproducibility capabilities of the application even further.

4.1.3 Extensibility and Operational Readiness

The third objective is about simplifying deployment and modular extensibility. In the updated architecture, we allow the admin to choose a device (CPU/GPU) for each module. Also, we introduce a container management layer and make enhancements in the integration of new and updated evaluation modules. These features make the application more maintainable and adaptable for potential use cases.

4.1.4 Enhanced User Experience and Administrative Control

The fourth objective is to improve platform usability for both administrators and researchers. Frontend components that are implemented and improved contributed to the usability of the application. For example, viewing historical versions of their results, reusing metadata templates, and being able to cancel a submission are critical enhancements that improve user experience. Furthermore, we also improve the administrative control by expanding the module management panel and introducing versioning features.

The following sections describe the design and implementation of these improvements in detail. Section 4.2 discusses the updated architecture, and mentions the newly added backend components and how they interact. Section 4.3 explains the evaluation workflow with redesigned and extended services. In the later sections, we focus on versioning implementation, queue management, and container optimization. We also mention data model extensions, UI improvements, and deployment configuration.

4.2 Improved System Architecture

The new version of *PrivBench* introduces an updated architecture that improves scalability, reproducibility, and maintainability. We keep the core structure of the application and add the new components and services on top of it. In this way, we try to solve the issues mentioned in the previous system. Therefore, the improved architecture integrates these additions without removing the existing ones.

Figure 4.1 shows the updated architecture with new models and services integrated into the platform. The Version Controller and Container Manager extend the backend layer, whereas the Benchmark Queue is the component between the backend and the evaluation environment. Also, the database layer now has entities that store module updates and application versions.

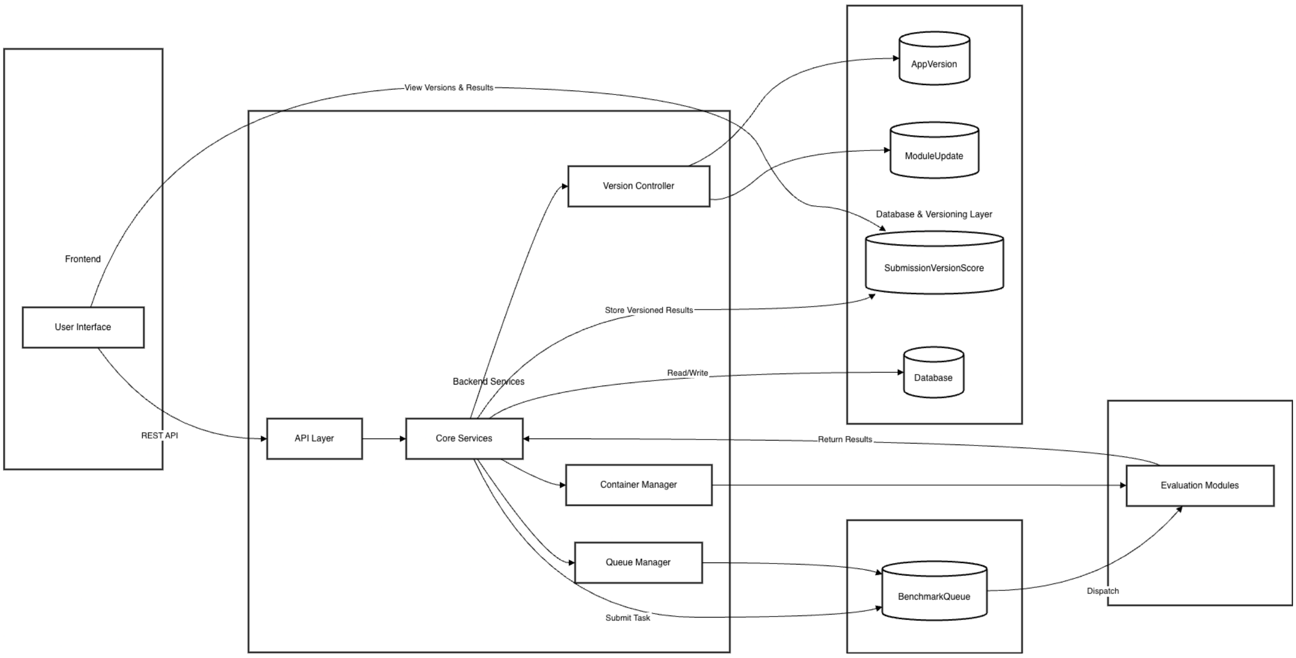


Figure 4.1: Updated component-level architecture of the *PrivBench* system showing new backend services for version control, container management, and queue orchestration.

4.2.1 Overview

As discussed, the new system enhances the previous architecture by integrating additional layers and services. The main architectural components are:

1. **Frontend (User Interface)**
   Users make their submissions and can monitor the progress via the provided UI. Also, they can see their previous benchmark results. We extend the frontend component to have:
   - Access to historical benchmark versions and comparison views.
   - Template-based submission through the new `TemplateMetadata` model. It is possible to create templates for the submission metadata.
   - A ranking page enhanced by basic and advanced filtering.
   - Better progress monitoring with an asynchronous queue mechanism.
2. **Backend (Core Services and APIs)**
   The backend is still the primary component of the application that controls most of the workflows. It now includes new services to support reproducibility and performance:
   - **Version Controller:** Maintains consistency between system versions and handles the version changes when a change happens in the benchmarking modules.

- **Container Manager:** It controls the lifecycle of the evaluation containers. To minimize startup latency, it reuses containers and loads them during the application startup.
- **Queue Manager:** Handles asynchronous submission scheduling using the`BenchmarkQueue` model. Moreover, it provides job tracking with information such as status and position.

These components make sure that the backend can process multiple submissions concurrently.

3. **Submission Queue (Asynchronous Execution Layer)**
   The asynchronous queue is integrated with the backend via the `BenchmarkQueue` entity. This new approach allows us to monitor position tracking in the queue. Each queue entry has links to the submission, user, and benchmark module.

4. **Evaluation Environment (Containerized Modules)**
   Evaluation modules are still in isolated containers. However, the improved architecture introduces a container manager that handles the container-related operations of the modules. In this way, we reduce the redundant and repetitive container startup and setup operations. Furthermore, the container manager communicates with the corresponding backend service to have the required information related to modules.

5. **Database and Version Layer**
   The database schema has been extended to support versioning.

   - The `AppVersion` model stores global benchmark versions.
   - The `ModuleUpdate` model tracks changes to evaluation modules.
   - The `SubmissionVersionScore` model records benchmark results per version.

   Thanks to these additions, the system is able to maintain complete traceability of different versions.

4.2.2 Architectural Interactions

The redesigned architecture involves more asynchronous tasks that interact with components. Also, it handles the versioning while evaluating the submissions. For instance, after the backend validates the submission, it enqueues the task until it is ready to be evaluated. The queue then sends this task to the respective module, where the evaluation is executed.

Once results are ready, they are stored in `SubmissionVersionScore`. This model links the submission to the benchmark version and the corresponding modules. Having this structure allows users to review their evaluations later on and check which set of modules produced each result.

4.2.3 Architectural Improvements Summary

The main architectural enhancements introduced in the redesigned *PrivBench* can be summarized as follows:

Table 4.1: Summary of Architectural Improvements in the Extended *PrivBench* System.

Improvement Area	Description	Related Component(s)
Version Control	Tracks application and result versions to ensure reproducibility and historical traceability.	`AppVersion`, `ModuleUpdate`, `SubmissionVersionScore`
Queue Management	Implements submission scheduling through a persistent job queue.	`BenchmarkQueue`
Container Optimization	Introduces container management and reuse to minimize startup delays and improve concurrency.	Container Manager
User Transparency	Enhances visibility for users by displaying version details, module histories, and real-time task progress.	Frontend Interface

4.3 Workflow Redesign

The new workflow changes the evaluation process because of the architectural changes we introduced, such as version tracking and an asynchronous queue mechanism. The goal is to minimize evaluation delays and have an intuitive workflow that users can easily follow. Figure 4.2 illustrates the updated workflow.

The new workflow consists of five main stages:

- Submission Initialization
- Version Resolution
- Task Scheduling and Execution
- Result Aggregation and Version Linking
- Visualization and Feedback

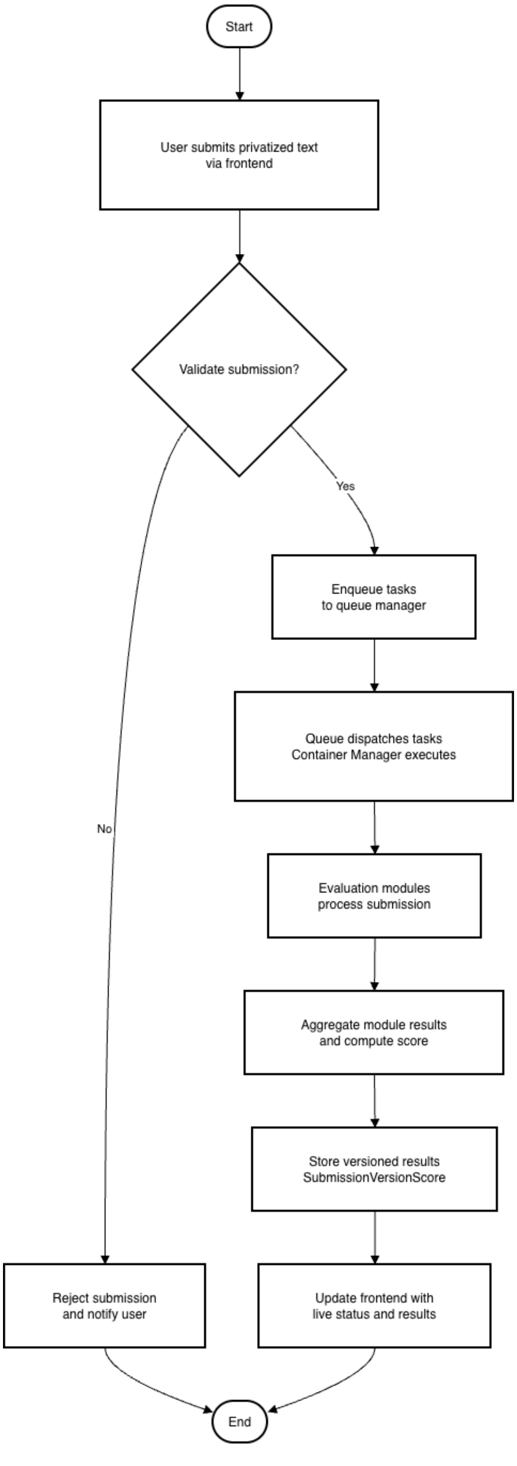

Start
User submits privatized text via frontend
Validate submission?
Yes
No
Enqueue tasks to queue manager
Queue dispatches tasks Container Manager executes
Evaluation modules process submission
Aggregate module results and compute score
Store versioned results SubmissionVersionScore
Reject submission and notify user
Update frontend with live status and results
End

Figure 4.2: Activity diagram of the *PrivBench* workflow showing the asynchronous task execution and version-linked result storage.

4.3.1 Submission Initialization

The evaluation process starts with the user's submission of the privatized dataset through the UI. The backend validates the submission to ensure its format is as expected and the uploaded dataset has certain properties. Users may also optionally reuse metadata templates that can be defined during the submission process. It is a valuable feature for repeated submissions.

After the validation, the backend records the submission in the database and creates the queue entry for benchmarking.

4.3.2 Version Resolution

Before execution, the system determines the current benchmark version by querying the `AppVersion` table. If we have a module update since the last benchmark execution, a new version is automatically created. Thus, the submission to be executed uses the latest version, as it has already been updated. This makes sure that the end result is linked to the respective modules and the latest version.

4.3.3 Task Scheduling and Execution

After version resolution, the submission enters the asynchronous queue managed by the backend's `Queue Manager`. Task positions and current states of the submissions are decided via the information in `BenchmarkQueue`.

The queue is responsible for sending the tasks to the evaluation environment, i.e., containers. Container Manager provides a setup in which tasks can be executed efficiently. Furthermore, each module has its own environment in the container with its logic file and requirements, and processes the submission independently.

4.3.4 Result Aggregation and Version Linking

When the evaluation modules are finished with the submission, the backend calculates an average score based on each metric derived from the modules. It then stores them as a new entry in `SubmissionVersionScore`. Thus, each record links the submission, benchmark version, and the score.

Having such a versioning structure brings advantages, as if modules are updated later, previous results remain accessible and comparable. Thanks to this, users can review their past submissions in different benchmark versions. It also contributes to reproducibility.

4.3.5 Visualization and Feedback

The final step presents the results to the user. The frontend displays:

- Overall benchmark scores per version.

- Module-specific results.
- The benchmark version used for the evaluation.
- Real-time progress updates during execution.

Users can transparently see their submission's progress in the UI, which increases the application's usability and user-friendliness.

The redesigned workflow improves both efficiency and reproducibility. Integrating version tracking and an asynchronous task queue brings application persistent results and version histories.

4.4 Versioning System and Reproducibility

Reproducibility is an important property of a benchmarking platform. Since the evaluation logic of the modules might change over time, having a versioned application brings a significant advantage. In the initial version, evaluation modules were not tracked, and results were not linked to versions. Therefore, we could not understand the difference in the produced results, as there is no historical tracking or a change list.

The extended platform introduces a versioning system centered around the versioning of modules and submission results. This improvement allows users to understand exactly which module configuration produced that score. This feature also gives users the opportunity to have valid comparisons across different versions.

4.4.1 Module Versioning

Each evaluation module is a component with specific configurations and logic, and they work independently. However, each of them computes one or more metrics such as privacy accuracy or utility preservation. As we mentioned earlier, as these modules may evolve due to bug fixes or performance improvements, tracking their changes gives us an overview, and it is important for benchmarking.

The platform introduces module versioning through two new entities:

- `ModuleUpdate`**:** Stores module changes, such as additions, modifications, or deletions. Each entry specifies:
    - the module affected,
    - the type of update (e.g., “new module”, “modified”, “deleted”),
    - the level of the change (major or minor),
    - a description, and
    - a timestamp.

- `AppVersion`**:** Represents a global version of the benchmark. When we have a module update, the administrator publishes the new version in the platform. Therefore, every module change is linked to a unique version. The `AppVersion` table stores the list of benchmark versions.

These two entities allow the system to have a complete history of modules.

4.4.2 Versioned Submission Results

Additionally, the platform stores versioned evaluation results. We use `SubmissionVersionScore` model to achive that. It connects each submission to:

- the submission identifier,
- the benchmark version used during evaluation,
- the calculated score,
- the timestamp of evaluation,
- and the responsible modules.

If a module is updated, users can compare how their results changed thanks to this model. The advantage of this approach is that, unlike the previous system, results are not overwritten when modules change. Each new evaluation generates a new `SubmissionVersionScore` entry. Thus, we are able to see the entire history of the submissions with their version information.

4.4.3 Versioning Workflow

The versioning process is integrated into the benchmarking workflow as follows:

1. **Module changes are applied.**
   An administrator modifies a module or adds a new one. The change is recorded in `ModuleUpdate`. When the administrator publishes the change, we have a new entry in `AppVersion`.
2. **A submission is evaluated.**
   When a user submits privatized text, we determine the latest benchmark version by querying the `AppVersion`.
3. **Tasks are executed under a specific version.**
   All evaluations executed for the submission have the same current version.
4. **Results are stored with their versions.** After evaluation, a `SubmissionVersionScore` entry is created and it has:
   - the submission ID,
   - the benchmark version,

- the evaluation score, and
- the modules involved.

Hence, this process makes sure that users can compare results across versions, module changes are visible, and we have consistency. We can see the sequence diagram explaining the versioning workflow in Figure 4.3.

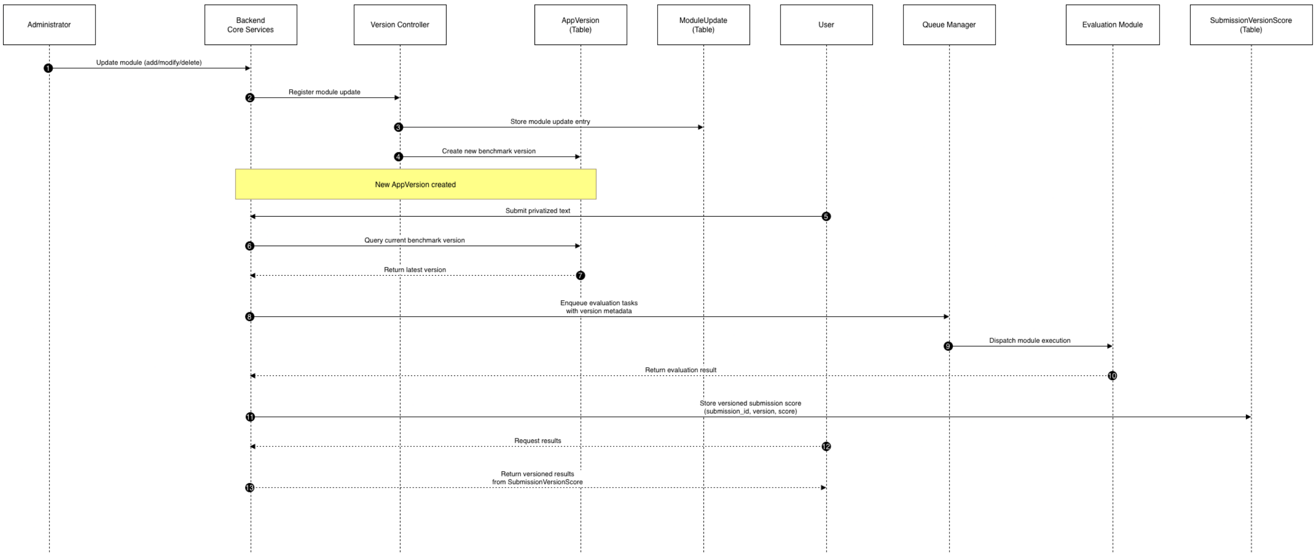

Administrator
Backend Core Services
Version Controller
AppVersion (Table)
ModuleUpdate (Table)
User
Queue Manager
Evaluation Module
SubmissionVersionScore (Table)
1
Update module (add/modify/delete)
2
Register module update
3
Store module update entry
4
Create new benchmark version
New AppVersion created
5
Submit privatized text
6
Query current benchmark version
7
Return latest version
8
Enqueue evaluation tasks with version metadata
9
Dispatch module execution
10
Return evaluation result
11
Store versioned submission score (submission_id, version, score)
12
Request results
13
Return versioned results from SubmissionVersionScore

Figure 4.3: Sequence diagram illustrating the versioning workflow in the improved *PrivBench* system.

4.4.4 Future Extensions

We currently cover module configurations and results in the existing versioning system. It can be extended to include dataset versioning and reproducible container images with specific dependencies. These additions would further enhance the capabilities of versioning and reproducibility, particularly for models that are difficult to reproduce or complex.

4.5 Performance Optimization and Container Management

Executing the evaluation tasks efficiently is highly important for benchmarking platforms. If we want to support multiple users or have concurrent submissions, performance is even more crucial. In the initial version, we had a Celery-based task execution that triggered a new container instance for each evaluation module. Although it was functional, we had performance bottlenecks such as high startup latency of containers and a lack of a mechanism to prioritize workload (queue system).

The new and improved system introduces architectural and operational improvements to resolve these limitations. We propose a queue management system, reusable evaluation containers, and improved status tracking. These optimizations reduce runtime overhead and provide users with better feedback in the UI.

4.5.1 Queue Management

A significant limitation of the original system was the lack of submission tracking. There was no queue implementation, and users were not aware of the other users' tasks. Therefore, it was difficult to understand the progress and performance issues.

The improved system introduces a queue model that is supported by a database model, and it provides:

- Submission ordering through a numerical position.
- State transitions via `QueueStatus` values (waiting, processing, completed, failed, cancelled).
- Timing metadata (created_at, started_at, completed_at).
- Traceability through links to users, submissions, and modules.

We now have a dedicated Queue Manager that coordinates the submission scheduling together with Celery's internal task queue. As a result, we have a more predictable and maintainable evaluation pipeline.

4.5.2 Container Reuse with the Container Manager

Container startup time was one of the expensive operations in the initial system. As each evaluation module ran in a new container, it was leading to performance overhead. It could be especially problematic when we have multiple submissions using the same module or multiple evaluations are triggered within a short time.

To address this, we introduce the Container Manager, which is a backend component that handles container orchestration. Its primary responsibilities are:

- Reusing container instances instead of restarting them.
- Restarting containers only when a module update occurs.
- Allocating resources based on device specification (CPU or GPU).

This reduces startup latency and avoids unnecessary container launches. Also, thanks to these enhancements, the execution pipeline becomes more responsive.

4.5.3 Improved Asynchronous Execution Flow

The combination of the updated queue model and container manager produces a more efficient asynchronous pipeline:

1. Backend validates submission and creates queue entries for each module.
2. Queue Manager dispatches jobs in a controlled order.
3. Container Manager assigns an available container.
4. Module executes the evaluation.
5. Timing metadata is recorded. It can be utilized in the analysis of throughput and bottlenecks.
6. Results are computed and stored as versioned scores.

Compared to the original implementation, we now have a system that increases parallel processing capability and reduces container overhead.

4.5.4 Summary of Performance Gains

Table 4.2: Summary of Performance Improvements in the Extended *PrivBench* System.

Improvement	**Effect**
Container reuse	Reduces container cold-start delays by reusing module containers rather than launching new instances for every task.
Structured queue management	Introduces task ordering and job diagnostics through the `BenchmarkQueue` model.
GPU-aware execution	Optimizes container allocation based on device specification (CPU/GPU), reducing runtime for computationally intensive NLP models.
Transparent status reporting	Improves user experience through real-time progress updates and clear visibility into task states.

4.6 Enhanced Data Model and Module Design

To support new features such as versioning, improved asynchronous execution, and user workflow enhancements, we also have to extend the data model we have. The initial system already had a relational structure that connected different entities, but it was missing mechanisms to trace the latest updates in the system.

We address these issues in the enhanced data model by introducing new entities and updating the relationships between them. Figure 4.4 illustrates the updated class diagram, and shows the new components that handle versioning and queue management.

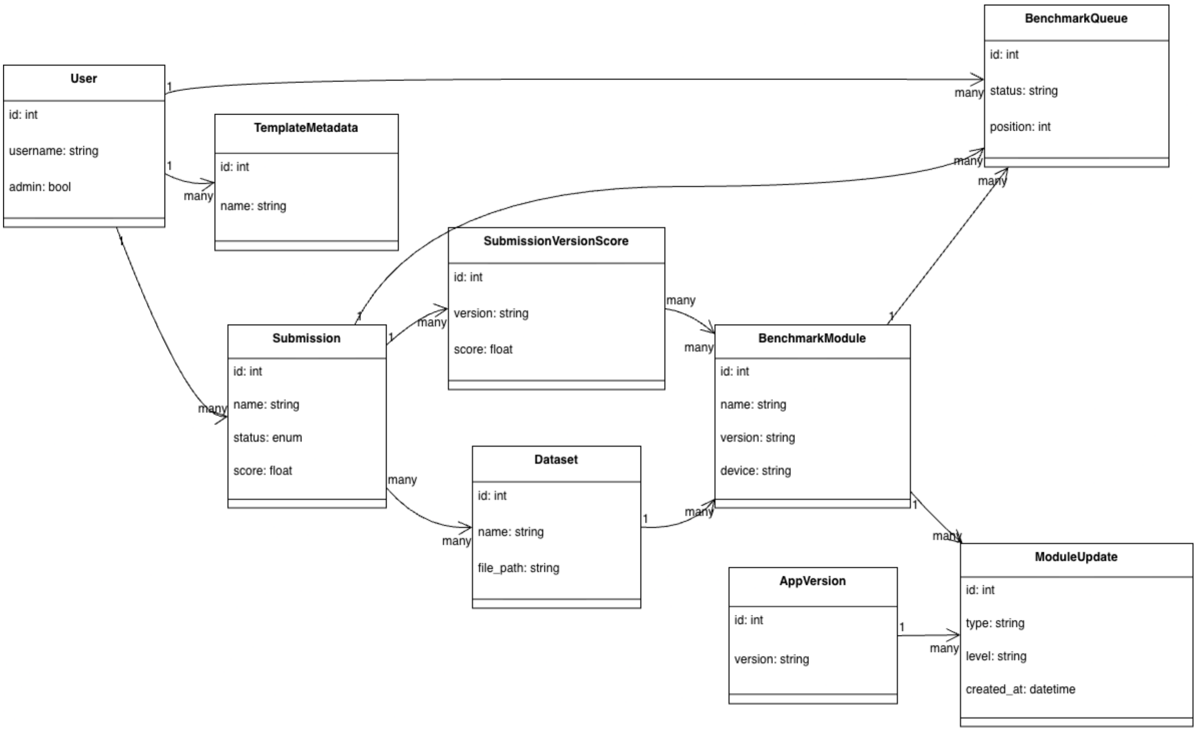


Figure 4.4: Class diagram of the improved *PrivBench* data model, including entities for versioning, queue management, and template metadata.

4.6.1 Overview of New Entities

The updated data model introduces five new entities:

- `AppVersion`, `ModuleUpdate`, `SubmissionVersionScore`, `BenchmarkQueue`, and `TemplateMetadata`.

These models extend the current relational structure to support the improved workflow.

AppVersion

The responsibility of the `AppVersion` model is to maintain the global version for the benchmark. Whenever the administrator adds a new module or updates one, we create a new version. We can think of the version as a snapshot of the platform's module configuration at a specific point.

Thanks to this approach, submissions evaluated under specific configurations are distinct. Additionally, we offer a historical comparison view by linking the results to the benchmark modules and versions.

ModuleUpdate

The `ModuleUpdate` entity records changes applied to evaluation modules. Each entry has the following attributes:

- the module affected,
- the type of change (new module, modified, deleted),
- the importance level (major or minor),
- a description,
- the timestamp,
- and the corresponding `AppVersion`.

This model allows us to see the module evolution in more detail due to its properties.

SubmissionVersionScore

In the initial system, even though we had a module change, the results were overwritten. This was making historical comparison impossible. In the `SubmissionVersionScore` model, we address this issue by introducing a new mechanism with entries that have:

- a submission, a benchmark version, the computed score, the included modules

This entity enables users to compare how the same submission performs under different versions.

BenchmarkQueue

The `BenchmarkQueue` entity implements an asynchronous submission queue by using the relations with other models. Each queue entry stores:

- the user that triggered the task,
- the module to evaluate,
- the submission being processed,
- the Celery task identifier,
- the task position,
- the queue status (waiting, processing, completed, failed, cancelled),
- timestamps for creation, start, and completion.

Users were able to track their submission status in the initial system. We now introduce an observable and traceable queue where everyone can see their status with enhanced information.

TemplateMetadata

The platform now also supports reusable metadata templates. The main idea is to simplify the repetitive submission process that requires filling out metadata.
The `TemplateMetadata` entity stores:

- the template name, the associated user, and inherited metadata fields.

This feature helps users maintain their configurations across submissions.

4.6.2 Updated Relationships Between Entities

The introduction of the new models require additional relationships:

- `AppVersion` → `ModuleUpdate`. Each version records module updates for the version increment.
- `Submission` → `SubmissionVersionScore`. Submissions may now have multiple versioned scores.
- `SubmissionVersionScore` → `BenchmarkModule`. The many-to-many relationship links each version score to the modules involved.
- `Submission` / `User` → `BenchmarkQueue`. Queue entries are tied to users and submissions.
- `User` → `TemplateMetadata`. Each user may have multiple templates.

These relationships facilitate reproducibility and submission workflow.

4.6.3 Summary of Data Model Enhancements

The improved data model:

Table 4.3: Summary of Data Model Enhancements in the Extended *PrivBench* System.

Enhancement	**Benefit**
Versioning of modules and results	Enables reproducibility and historical comparison across benchmark updates.
Queue tracking with timestamps	Improves transparency, enables detailed analysis, and simplifies debugging.
Reusable templates	Enhances user productivity by simplifying repeated submission workflows.
Many-to-many linking between versioned scores and modules	Ensures traceability of which modules contributed to each evaluation result.

4.7 User Interface and Usability Improvements

The main goal of the system improvements is to enhance the usability of *PrivBench* for researchers who submit their privatized outputs and administrators who manage the benchmark modules. The initial interface had basic submission and result pages, but it lacked control of submissions and transparency. Having specific improvements to help users interact with the platform efficiently was critical. Furthermore, users had no access to historical results, there was no way to reuse the submission metadata, and we had room for improvement in the task progress. Likewise, administrators needed a better interface that could facilitate easier module configurations and version management.

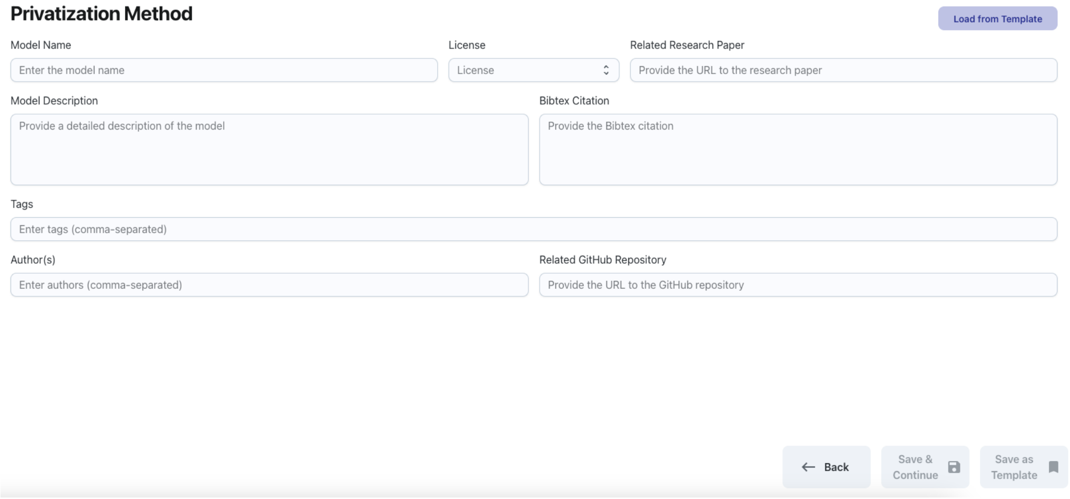
The frontend is extended with new components and views, such as improved status indicators, reusable metadata templates, and a results display that takes into account version information. We aim to improve the clarity of the benchmark flow and make it more intuitive with these features.

4.7.1 Enhanced Submission Workflow

The submission interface is extended to show users a clearer evaluation process. Users can now:

- Save and reuse commonly used configurations via Template Metadata.
- Cancel their submission after the evaluation starts.

This workflow helps users run experiments more efficiently, since they can reuse the same metadata (See Figure 4.5).



Privatization Method

Model Name: Enter the model name

License: License

Related Research Paper: Provide the URL to the research paper

Model Description: Provide a detailed description of the model

Bibtex Citation: Provide the Bibtex citation

Tags: Enter tags (comma-separated)

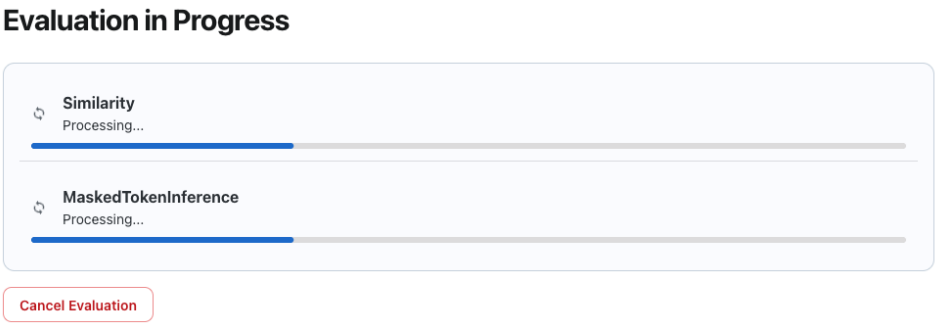
Author(s): Enter authors (comma-separated)

Related GitHub Repository: Provide the URL to the GitHub repository

Figure 4.5: Submission interface in the improved *PrivBench* platform, showing metadata fields.

4.7.2 Improvements in Progress and Status Visibility

We had a real-time progress update in the original platform. However, no information related to the queue was visible in the submission interface. Thanks to the `BenchmarkQueue` model and related implementation, users can now view their current position in the queue. Thus, we make the progress tracking more informative and increase the user experience (See Figure 4.6 and 4.7).

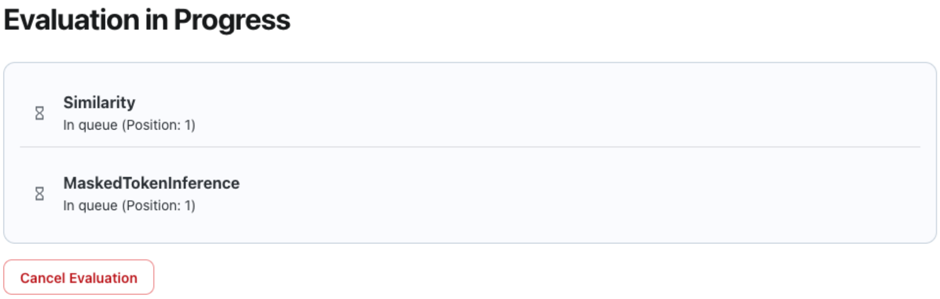


Evaluation in Progress

Similarity
Processing...

MaskedTokenInference
Processing...

Figure 4.6: Evaluation in progress view showing real-time status of module execution, allowing users to monitor evaluations as they run.



Evaluation in Progress

Similarity
In queue (Position: 1)

MaskedTokenInference
In queue (Position: 1)

Figure 4.7: Live submission status view showing real-time queue position and progress indicator.

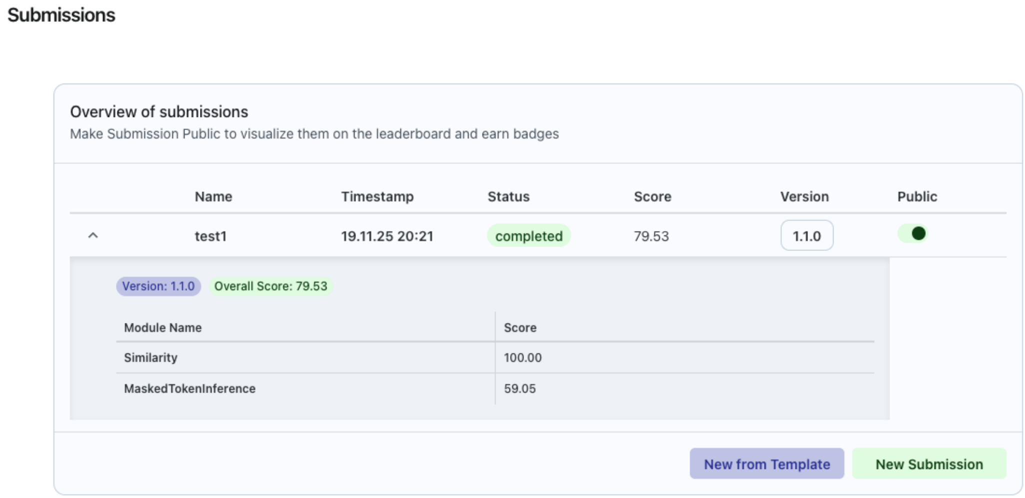
4.7.3 Version-Aware Result Visualization

Because the system now supports versioning of modules and submission results, we update the frontend to demonstrate this new capability. The results view now shows:

- The benchmark version used to evaluate each submission.
- The module set associated with that version.

- Historical scores across benchmark versions for the same submission.

This visibility is valuable for users who want to understand the context behind each evaluation. They can also compare their submissions across different version updates (See Figure 4.8).



Submissions

Overview of submissions

Make Submission Public to visualize them on the leaderboard and earn badges

	Name	Timestamp	Status	Score	Version	Public
^	test1	19.11.25 20:21	completed	79.53	1.1.0	●

Version: 1.1.0 Overall Score: 79.53

Module Name	Score
Similarity	100.00
MaskedTokenInference	59.05

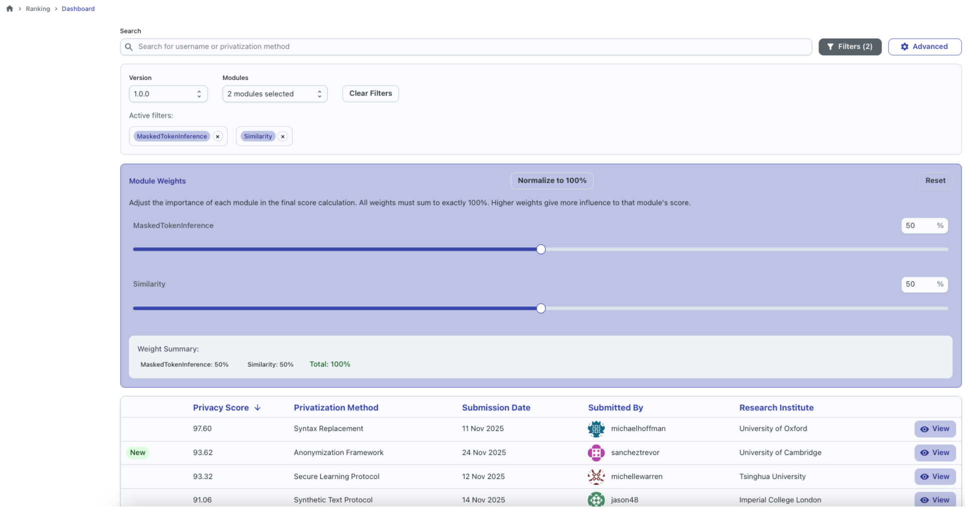
Figure 4.8: Version-aware results, displaying scores grouped by benchmark version.

4.7.4 Improved Leaderboard and Filtering Options

The leaderboard has also new features including:

- Filtering by module or benchmark version.
- Viewing per-version rankings to offer fair comparison.
- Advanced filtering, which allows users to enter weights for each module.

This granular filtering is useful for researchers who want to compare techniques under specific configurations (See Figure 4.9).



Search for username or privatization method

Version: 1.0.0

Modules: 2 modules selected

Active filters: MaskedTokenInference, Similarity

Adjust the importance of each module in the final score calculation. All weights must sum to exactly 100%. Higher weights give more influence to that module's score.

MaskedTokenInference: 50 %

Similarity: 50 %

Weight Summary:
MaskedTokenInference: 50% Similarity: 50% Total: 100%

	Privacy Score ↓	Privatization Method	Submission Date	Submitted By	Research Institute	
	97.60	Syntax Replacement	11 Nov 2025	michaelhoffman	University of Oxford	View
New	93.62	Anonymization Framework	24 Nov 2025	sancheztrevor	University of Cambridge	View
	93.32	Secure Learning Protocol	12 Nov 2025	michellewarren	Tsinghua University	View
	91.06	Synthetic Text Protocol	14 Nov 2025	jason48	Imperial College London	View

Figure 4.9: Leaderboard with filtering and advanced filtering options.

4.7.5 Administrative Interface for Module Updates

The module management panel, which administrators use, now has an improved interface. They are able to do the following through this interface:

- Add, edit, or delete evaluation modules.
- Register module updates with descriptions and change levels.
- Publish version increments through the `AppVersion` and `ModuleUpdate` models.

4.7.6 Summary

The improvements introduced in the UI enhance the usability of *PrivBench*. We now have a clearer process for submission, versioned benchmark results, and visibility into the submission queue. We think that these extended features make the system more approachable and powerful for researchers or new users.

4.8 Implementation Details and Technologies

In this section, we describe the key implementation details, frameworks, and libraries that are utilized in the *PrivBench* application. These technological foundations help us understand the architectural improvements mentioned in the earlier sections. Also, we summarize some of the practical decisions related to deployment and configurations.

The *PrivBench* platform is implemented as a containerized web application. The main components are a Flask backend, a React frontend, a PostgreSQL database, and a Celery task execution layer. Docker provides the execution environments for evaluation modules. The communication between those layers and components is handled by REST API and database interactions.

4.8.1 Backend Architecture and Frameworks

The backend is implemented using Flask. It is an easy-to-use Python-based backend framework that is compatible with NLP tooling. Flask's modular structure offers an advantage when trying to extend services or introduce new features, such as version control or queue management.

Key backend components are:

- Flask + SQLAlchemy for API endpoints and relational data management.
- Celery for asynchronous task scheduling.
- Redis as the message broker for Celery workers.
- Docker Software development kit (SDK) for Python for controlling evaluation module containers.

4.8.2 Frontend Implementation

The frontend is implemented using React. It provides a modular component-based interface, which is a useful property if we want to extend an application. The UI enhancements described in Section 4.7, including the submission queue, versioned results, and submission template metadata, are built with the help of:

- React Router for page structure.
- Axios for API requests.
- Material UI for styling.

The frontend integrates with the backend using REST API, and it uses status polling in addition to version-specific endpoints.

4.8.3 Asynchronous Task Execution with Celery

All evaluations are executed asynchronously thanks to Celery workers. They retrieve tasks from Redis and send them to the corresponding container for execution. The task pipeline uses task chaining when multiple modules must be evaluated. As a result, we have a strong evaluation pipeline that can handle concurrent submissions.

4.8.4 Container Management and Execution Environment

Evaluation modules are executed within the Docker containers, and this is managed by the Container Manager as we discussed earlier. This service uses the Docker Python SDK for its responsibilities.

The container manager service has specific tasks. It restarts module containers when there is an update. If there is a device specification such as CPU or GPU, it binds the device to the module. For GPU supported modules, the NVIDIA container runtime is used. Also, since logging and error reporting are critical in the production systems, it provides logging messages when necessary.

4.8.5 Database and Versioning Layer

We implement the database using PostgreSQL, as it is a reliable system with support for JavaScript Object Notation (JSON) fields. Furthermore, SQLAlchemy provides Object–relational mapping (ORM) functionalities, and this facilitates implementing queries and schemas. We have a reproducible and maintainable structure thanks to this combination of technologies.

4.8.6 Configuration and Environment Management

We have both development and production environments. The configuration is managed by `.env` files for environment variables, Docker Compose profiles, and runner scripts that handle database migrations and organize application start-up. Having this modular configuration allows us to keep development lightweight, and production remains even more stable.

4.8.7 Deployment Strategy and Production Readiness

The extended *PrivBench* platform is deployed as a multi-container application using Docker Compose. In the production deployment, we have a Celery worker pool for evaluations, a persistent Docker volume for storing module images, and also logging streams from backend, workers, and containers.

These enhancements keep the system available and suitable for multi-user support and continuous operations.

4.8.8 Summary of Technologies

Table 4.4: Summary of Technologies Used in the Extended *PrivBench* System.

Component	**Technology Stack**
Backend	Flask, SQLAlchemy
Task Execution	Celery, Redis
Containers and Execution	Docker, Docker SDK for Python, NVIDIA Container Runtime
Database	PostgreSQL
Frontend	React, Axios, Material UI
Deployment and Configuration	Docker Compose, Reverse Proxy (e.g., Nginx), Environment Variables (.env)

4.9 Summary

This chapter presents the design and implementation of the improved *PrivBench* platform. The extended architecture introduced an asynchronous queueing pipeline, container management service, and versioning. These features improve performance and system responsiveness and ensure that results remain comparable when the system gets updated over time.

The chapter also discusses updates that we implement in the user experience. It includes reusable metadata templates, result displays that contain versioning, and an updated admin panel. These improvements make the platform easier to use and more manageable for administrators. Finally, we explain the technologies that help build the system, including Flask, Celery, Docker, React, and PostgreSQL. Additionally, the deployment strategies that are required for reliable operation are mentioned.

5 Evaluation

5.1 Introduction

This chapter evaluates the extended *PrivBench* platform in terms of performance and usability. We aim to make an evaluation to understand whether the introduced architectural changes in Chapter 4 have measurable improvements. The assessment combines both quantitative measurements, such as module execution time and container latency, and qualitative interpretations we observed from the updated UI.

The new submission queue mechanism and the introduction of container reuse are the main focuses of the performance evaluation. We assess these features by comparing their previous structure with the newer one. The usability evaluation shows improvements in the submission workflow, including template metadata, real-time submission progress tracking, and the leaderboard, which has been extended with versioning.

We believe that these evaluations provide an analysis of the platform's usability as a production application. Additionally, we have a better understanding of its real-world use and suitability as a benchmarking environment.

5.2 Evaluation Setup

Experiments are conducted locally on a MacBook Air M1. This device is not a server machine, but it has the ability to represent a typical developer environment. Also, it can create realistic conditions and be sufficient for measuring the architectural improvements we discussed earlier. The evaluation setup includes details about the hardware, software environment, and metrics used.

5.2.1 Hardware Environment

All benchmarks are executed on a MacBook Air (M1, 2020) with the following specifications:

- **CPU:** Apple M1 (8-core CPU)
- **GPU:** Integrated 7-core/8-core GPU
- **RAM:** 8 GB unified memory
- **Storage:** 256 GB SSD

Since the M1 GPU does not support Compute Unified Device Architecture (CUDA) or NVIDIA's container runtime, all evaluation modules were executed on CPU.

5.2.2 Software Environment

The *PrivBench* platform is run locally in a containerized environment using the following software technologies:

- Docker Desktop for Mac
- Python 3.9
- Flask for backend logic
- React 18 for the frontend
- SQLAlchemy + PostgreSQL 14
- Celery with Redis as the message broker
- Docker SDK for Python for container management

5.2.3 Datasets and Metrics

All evaluations in this chapter are conducted using a lightweight dataset consisting of approximately 100 text rows. The dataset size allows us to focus on system architecture behavior such as queue management and container delays.

Metrics

To assess system performance and usability, we use the following metrics:

Performance Metrics

- **Execution Time:** Total time needed for a module to process the 100-row dataset.
- **Cold Start Time:** Time required to launch a fresh container for a module.
- **Warm Start Time:** Time when reusing an already running container.

Usability Metrics

- Responsiveness of real-time status updates
- Clarity of the versioned results UI
- Effectiveness of template-based submission reuse

5.3 Performance Evaluation

This section evaluates the performance benefits coming from the new architecture of *PrivBench* application. The main focus is on execution time and container management, which we explained as system improvements in Chapter 4. Additionally, all tests are performed locally on a MacBook Air M1 (8-core CPU, 8GB RAM) using a dataset of 100 text entries.

5.3.1 Experimental Setup

To understand the impact of the architectural changes we made, we execute the same benchmark tasks on both the old system and the new system. The crucial difference between the two systems is about container lifecycle management:

1. **Old System (Baseline):** In this architecture, for every benchmark module, the system creates a new Docker container. It then loads the required dependencies and models, processes the input, and destroys the container. In other words, for each module of a benchmark submission, we create and destroy a container during execution. Consequently, every run in this system represents a cold start.

2. **New System (Optimized):** This architecture uses the Container Manager. Here, containers are initialized once at application startup, and the existing running containers directly handle benchmark requests. Thus, benchmark runs in the new system represent warm starts. Furthermore, we eliminate the overhead of container creation and model loading thanks to this new architecture.

The evaluation uses two benchmark module types to represent different computational profiles:

- **Similarity:** This module combines lexical analysis (BLEU) with semantic embedding similarity. Moreover, it uses a lightweight Transformer model to create sentence embeddings. Although it requires some level of computation, it still runs relatively easily on a CPU, especially compared to Large Language Models (LLMs).

- **MaskedTokenInference:** This module utilizes a Masked Language Model, which is a larger and more complex model, to reconstruct redacted tokens. In other words, it uses a computationally intensive model, which requires a significant amount of time to load its model weights into memory.

5.3.2 Module Execution Time Analysis

Table 5.1 summarizes the runtime performance (in seconds) for both systems across 10 independent runs per module. We can observe from the results that there is a remarkable reduction in total processing time. For MaskedTokenInference, the mean execution time drops from 204.02s in the old system to 14.95s in the new system. In Similarity module, likewise, the mean execution time drops from 200.04s to 29.99s. Furthermore, the new system shows highly stable performance with minimal standard deviation.

Table 5.1: Statistical Overview per System and Module (n=10 runs each)

System	Module Name	Mean (s)	Median (s)	Min (s)	Max (s)
Old	MaskedTokenInference	204.02	123.80	108.70	815.52
New	MaskedTokenInference	14.95	15.16	11.80	18.01
Old	Similarity	200.04	96.52	72.62	929.48
New	Similarity	29.99	30.44	26.86	32.78

5.3.3 Impact of Container Reuse

By comparing the two systems, we can understand the container overhead better. As we defined in the setup, the old system has a cold start latency, whereas the new system represents a warm start. We can see that cold starts are significantly slower due to the initialization and loading overhead. Therefore, warm starts eliminate this cost and reduce the execution time. We can say that this results validate the container manager design even on a hardware without a GPU. It removes the initialization steps and introduces a more interactive experience.

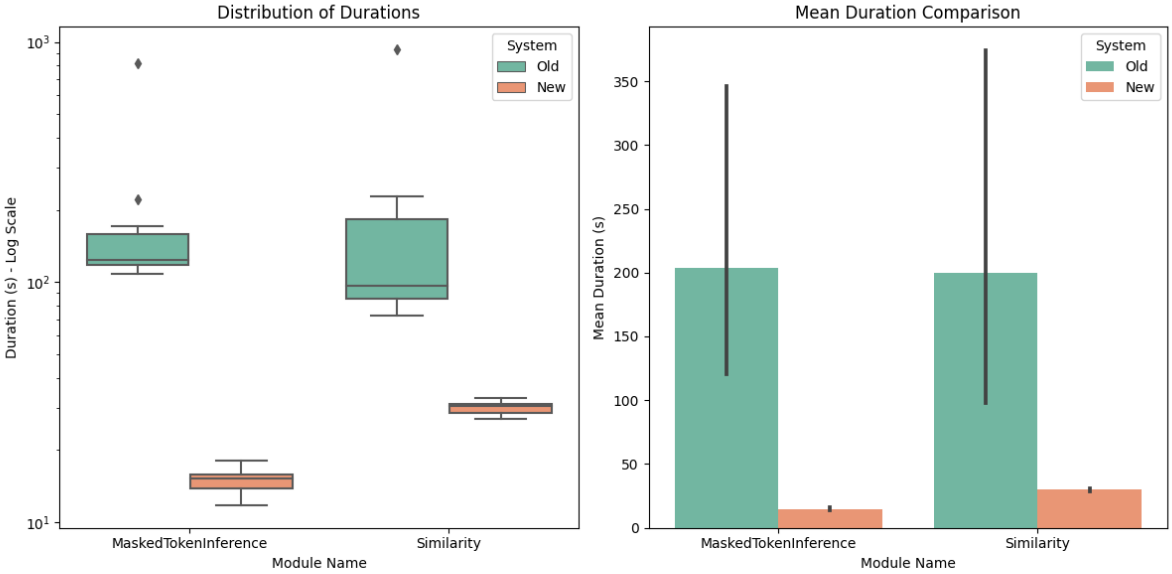


Figure 5.1: Performance comparison between the Old and New systems.

In Figure 5.1, the left panel displays the distribution of execution durations on a logarithmic scale. The right panel compares the mean execution time, displaying the significant speedup achieved by the new system.

5.3.4 Summary of Performance Results

We overcome the performance overhead of the system by redesigning the container architecture. The key findings are:

1. **Speedup:** Container reuse resulted in a 6x to 13x improvement in execution speed, depending on the module.
2. **Elimination of Cold Starts:** The previous latency was primarily related to initialization overhead. It has now been removed from the system.
3. **Predictability:** The new system offers more consistent and low-variance runtimes. This is important for a benchmark platform whose goal is to create comparable and reproducible results.

5.4 Usability Evaluation

This section evaluates the usability improvements introduced in the improved *PrivBench* system. We focus on how the new UI features affect the user experience and address the initial system's limitations.

5.4.1 User Experience Improvements

We focus on three key enhancements that directly impact the users:

- live queue updates, metadata templates, and the version-aware leaderboard.

Live Queue Updates

In the old system, users could see basic submission updates such as "pending", "processing", and "completed", but there was no information about the queue position. In other words, users did not have information about the number of tasks ahead of their submission.

During our evaluation, we verify that the system now displays the user's specific position in the queue, along with their status. This allows users to have a better view of the system's workload and their submission's status.

Template Metadata

Researchers often require repeated submissions, since they change or fine-tune their models. In the previous version, users had to manually enter metadata for every upload. Since this is a time-consuming process, we introduce template metadata. We observe that this feature simplifies the submission flow and, furthermore, allows researchers to reuse common data such as citations or authors.

Version-Aware Leaderboard

In the previous leaderboard, we did not have version filtering. Therefore, we were unable to select specific versions and view the results grouped by them. We now have an updated ranking page that integrates version information. Users can filter the results, interpret their rankings, and compare the scores. We confirm that users can compare their scores against others under the same conditions. Also, by displaying the version number explicitly, the platform provides sufficient context to understand historical results.

5.5 Summary

This chapter presents the evaluation of the extended PrivBench platform. We focus on performance and usability improvements. The new system provides substantial improvements in different dimensions.

Regarding performance, we show that introducing the Container Manager removes the cold-start latency overhead, which was the main issue in the previous system. Additionally, we achieve execution speedups ranging from approximately 6x to 13x by adopting the warm start approach. Thanks to this optimization, the new system can handle higher workloads.

In terms of usability, our architectural changes create a more efficient and clearer user experience. The new queue tracking feature allows users to see their position in the queue, and this improves the fairness of the benchmarking process. Also, reusable template metadata improves the efficiency of the submission flows, and a version-aware leaderboard helps users understand the performance of their method historically.

Overall, the evaluation shows that the new *PrivBench* system is a more scalable, reproducible, and user-friendly platform. Additionally, it addresses the major limitations identified in Chapter 3.

6 Discussion

This chapter discusses the results of the system redesign presented in Chapter 4 and the evaluation findings described in Chapter 5. We revisit the research questions introduced in Chapter 1 and give answers considering the architectural updates and improvements we have made. Moreover, we discuss the practical and theoretical implications of this work, provide an overview of the system's limitations, and outline potential ideas for future research.

6.1 Answering the Research Questions

The primary goal of this thesis is to enhance an existing text privatization benchmarking platform to meet the requirements of a standardized, reproducible, and high-performance evaluation system. Thus, we answer each research question individually.

RQ1: How can an existing modular evaluation framework for text privatization be extended to enable the standardized, reproducible, and interoperable measurement of privacy, utility, and performance?
To address this question, we focus on transforming the platform into a dynamic, version-controlled system.

- **Reproducibility through Versioning:** We introduce a versioning system to solve the reproducibility challenge using the `AppVersion`, `ModuleUpdate`, and `SubmissionVersionScore` entities along with their associated service logic. Having module configurations as versioned snapshots helps us to view the evolution of benchmark results. Even if the module logic has changed, a score generated today can be contextually compared to a score generated in the future.
- **Standardized Measurement:** The new data model has a standardized structure for each submission. We link every result to a specific `BenchmarkModule` and `AppVersion`. It creates a consistent "unit of measurement" for privacy and utility. Moreover, this interoperability is important for researchers to have valid comparisons under the same evaluation conditions.

RQ2: How can the platform be designed and engineered to fulfill key software quality attributes such as performance, maintainability, extensibility, and fairness in the benchmarking process?
This question addresses the engineering challenges of the platform, and our implementation results highlight three key design decisions:

- **Performance via Container Management:** The implementation of Container Manager is a critical factor in improving the application's performance. When we shift from "cold start", i.e., creating and destroying the containers, to a "warm start" reuse approach, it remarkably reduceds the execution time of submissions. Additionally, the evaluation shows that eliminating container initialization latency results in a significant speedup.
- **Fairness and Scalability:** The introduction of the queue mechanism and the `BenchmarkQueue` model focuses on the fairness attribute. We provide queue positions, so users can not block the application with many submissions. Thanks to this mechanism, the platform can scale under multi-user loads.
- **Maintainability:** We have a modular architecture. It is supported by independent Docker containers and a template-based metadata system. This increases extensibility because administrators can add or update modules without causing problems in the core application. Also, the separation of frontend, backend, and execution environments facilitates ongoing maintenance.

6.2 Implications of the Work

The improvements detailed in this thesis have important implications for the text privatization research.

6.2.1 Transition from Prototype to Production

The original PrivBench system was a functional proof-of-concept. Our enhancements that we presented make it a production-ready platform. Therefore, it is now an application that can be deployed as a public service that researchers can utilize.

6.2.2 Benchmarking with Historical Context

The addition of versioning introduces a time dimension to the *PrivBench* benchmark. Since researchers will have access to the latest version, they can follow the "state of the art" over time. This is a valuable property in the privacy domain, as regulations and methods are updated frequently. Thus, the platform now offers documentation that we can compare the models to their previous iterations.

6.2.3 Lowering the Barrier to Entry

As we improve the usability through metadata templates and real-time queue tracking, the platform has a lower technical barrier for researchers. They can focus on developing privacy models rather than dealing with the platform itself.

6.3 Limitations and Future Work

Although the extended platform achieves the main goals, we still have limitations in the current implementation that require future work.

6.3.1 Hardware and Environment Constraints

The performance evaluation in this thesis is conducted on a local machine (Apple M1). The results may differ on Linux servers or in different production clusters, since containerization might behave differently. Even though the platform has been successfully deployed in a Virtual Machine (VM) with an NVIDIA GPU, testing and benchmarking in that environment could not be completed within the scope of this work. Furthermore, the system architecture is designed to support GPU devices for complex models, but we could not quantify the specific performance gains from this hardware.

- *Future Work:* The next step is to conduct a full performance benchmark in the GPU-enabled VM environment. We can provide a more realistic metric after validating the scalability of the modules when a GPU is utilized.

6.3.2 Dataset and Module Logic Versioning

The new system versions benchmark modules and submission results. One drawback is that we do not yet include dataset versioning or versioning of module logic files, such as the algorithm file. As a result, reproducibility can still be an issue when a user attempts to retry a submission they previously made. The dataset or module logic file might be updated, so they may try to find different workarounds.

- *Future Work:* The data model should be extended to include dataset and module logic versioning. This would complete the reproducibility feature of the benchmarking platform, since it covers code, model, and the dataset.

6.3.3 Scalable Container Management and Worker Nodes

The current architecture relies on a single pool of Celery workers for task handling. In a more extreme scenario with high throughput, a single node might become a problem. Additionally, for larger scale benchmarking, a Kubernetes-based execution or a container manager with scaling capabilities can be a good idea.

- *Future Work:* We have an extensible architecture. Although it is not a straightforward task, one can implement distributed worker nodes across different physical machines. Alternatively, utilizing Kubernetes with different VMs can also help relax the application under high loads.

7 Conclusion

This thesis details the redesign and extension of *PrivBench*. The main goal of this thesis is to transform the platform from an initial functional prototype into a production ready benchmarking system. We address the limitations identified in the original system. Furthermore, we design and implement a new architecture that improves the platform's usability and reliability.

The extended system introduces a versioned evaluation pipeline. Thanks to this feature, we have reproducible comparisons across different states and can see the outdated results generated under various module configurations. In addition, the new versioning layer enables users to view which modules were used during evaluation, and this contributes to resolving the reproducibility problem in the original design.

In the new architecture, we achieve more stable and faster execution times, as we demonstrated in the performance measurements. The main contributors to these enhancements are the introduction of a submission queue and the container manager. They reduce the overhead and provide a more predictable runtime behavior. Although GPU execution could not be evaluated within the scope of this thesis, we have a system prepared to integrate GPU-enabled evaluation on the VM.

Beyond architectural and performance improvements, the new system offers a better user experience. In the updated interface, we have visible queue positions along with the task status for each submission. Therefore, users can understand evaluation progress in real time. Additional features such as reusable templates and a version-aware leaderboard simplify the submission workflow.

We have certain areas that are still open for work, even though the system addresses many of the initial shortcomings. Extending versioning to datasets and module source files would be a strong step to improve reproducibility further. Moreover, if we could validate the module execution with a GPU and evaluate performance on larger datasets, we would have a more comprehensive understanding of the system's capabilities.

In summary, the improved *PrivBench* system forms a good foundation for benchmarking text privatization techniques. It improves transparency and usability, and makes the platform an environment that supports research on NLP. While it provides a strong basis for future work in this rapidly changing domain, the system is now ready for further extension and real-world deployment.

List of Figures

List of Tables

Acronyms

API Application Programming Interface. 13, 41

BERT Bidirectional Encoder Representations from Transformers. 5, 11

BIG-Bench Beyond the Imitation Game Benchmark. 10

BLEU Bilingual Evaluation Understudy. 7, 11, 46

BLUE Biomedical Language Understanding Evaluation. 9, 11

CPU Central Processing Unit. 2, 20, 23, 32, 33, 42, 44–46

CRF Conditional Random Field. 5, 11

CUDA Compute Unified Device Architecture. 45

DL Deep Learning. 5

DP Differential Privacy. 7

GDPR General Data Protection Regulation. 4

GLUE General Language Understanding Evaluation. 9, 11

GPT Generative Pre-trained Transformer. 5, 11

GPU Graphics Processing Unit. 2, 20, 23, 32, 33, 42, 44, 45, 47, 52, 53

HELM Holistic Evaluation of Language Models. 8, 10, 11

HIPAA Health Insurance Portability and Accountability Act. 4

i2b2 Informatics for Integrating Biology and the Bedside. 9, 11

JSON JavaScript Object Notation. 42

LLM Large Language Model. 46

ML Machine Learning. 10

NER Named Entity Recognition. 5, 7, 14

NLP Natural Language Processing. 3, 5, 7, 9, 11, 33, 53

ORM Object–relational mapping. 42

REST Representational State Transfer. 13, 41

ROUGE Recall-Oriented Understudy for Gisting Evaluation. 7, 11

SDK Software development kit. 41, 42

SVM Support Vector Machine. 5, 11

TAB Text Anonymization Benchmark. 9, 11

UI User Interface. 3, 12, 13, 22–24, 28, 29, 32, 40, 41, 44, 45, 48

VM Virtual Machine. 52, 53

Bibliography

[1] N. Yadav, S. Pandey, A. Gupta, P. Dudani, S. Gupta, and K. Rangarajan. "Data Privacy in Healthcare: In the Era of Artificial Intelligence". In: *Indian Dermatology Online Journal* 14 (Oct. 2023), pp. 788–792. DOI: `10.4103/idoj.idoj_543_23`.

[2] I. Pilán, B. Manzanares-Salor, D. Sánchez, and P. Lison. "Truthful text sanitization guided by inference attacks". In: *Applied Soft Computing* 185 (2025), p. 114013. ISSN: 1568-4946. DOI: `https://doi.org/10.1016/j.asoc.2025.114013`. URL: `https://www.sciencedirect.com/science/article/pii/S1568494625013262`.

[3] S. Meisenbacher and F. Matthes. "Thinking Outside of the Differential Privacy Box: A Case Study in Text Privatization with Language Model Prompting". In: *Proceedings of the 2024 Conference on Empirical Methods in Natural Language Processing*. 2024, pp. 5656–5665.

[4] T. Deußer, L. Sparrenberg, A. Berger, M. Hahnbück, C. Bauckhage, and R. Sifa. "A Survey on Current Trends and Recent Advances in Text Anonymization". In: (Aug. 2025). DOI: `10.48550/arXiv.2508.21587`.

[5] J. Trienes, D. Trieschnigg, C. Seifert, and D. Hiemstra. "Comparing Rule-based, Feature-based and Deep Neural Methods for De-identification of Dutch Medical Records". In: *WSDM 2020 Workshop on Health Search and Data Mining (HSDM)*. Texas, USA: CEUR Workshop Proceedings, 2020, pp. 3–11.

[6] *Regulation (EU) 2016/679 (General Data Protection Regulation)*. Apr. 27, 2016. URL: `https://eur-lex.europa.eu/eli/reg/2016/679/oj` (visited on 11/01/2025).

[7] *Health Insurance Portability and Accountability Act of 1996*. Aug. 21, 1996. URL: `https://www.govinfo.gov/content/pkg/PLAW-104publ191/pdf/PLAW-104publ191.pdf` (visited on 11/01/2025).

[8] *NIST Privacy Framework: A Tool for Improving Privacy Through Enterprise Risk Management*. Tech. rep. National Institute of Standards and Technology, 2020. URL: `https://www.nist.gov/privacy-framework`.

[9] P. Lison, I. Pilán, D. Sánchez, M. Batet, and L. Øvrelid. "Anonymisation Models for Text Data: State of the Art, Challenges and Future Directions". In: *Proceedings of the 59th Annual Meeting of the Association for Computational Linguistics and the 11th International Joint Conference on Natural Language Processing (Volume 1: Long Papers)*. Association for Computational Linguistics, 2021, pp. 4188–4203. DOI: `10.18653/v1/2021.acl-long.323`.

[10] A. Shankar, A. Waldis, C. Bless, M. Rodriguez, and L. Mazzola. "PrivacyGLUE: A Benchmark Dataset for General Language Understanding in Privacy Policies". In: *Applied Sciences* 13.6 (2023), p. 3701. DOI: `10.3390/app13063701`.

[11] S. Krishna, R. Gupta, and C. Dupuy. "ADePT: Auto-encoder based Differentially Private Text Transformation". In: *Proceedings of the 16th Conference of the European Chapter of the Association for Computational Linguistics* (2021).

[12] A. Kovačević, B. Bašaragin, N. Milošević, and G. Nenadić. "De-identification of clinical free text using natural language processing: A systematic review of current approaches". In: *Artificial Intelligence in Medicine* 151 (2024), p. 102845. ISSN: 0933-3657. DOI: `https://doi.org/10.1016/j.artmed.2024.102845`. URL: `https://www.sciencedirect.com/science/article/pii/S0933365724000873`.

[13] R. Staab, M. Vero, M. Balunović, and M. Vechev. *Large Language Models are Advanced Anonymizers*. 2025. arXiv: `2402.13846 [cs.AI]`. URL: `https://arxiv.org/abs/2402.13846`.

[14] K. El Emam, L. Mosquera, and R. Hoptroff. *Practical Synthetic Data Generation: Balancing Privacy and the Broad Value of Data*. O'Reilly Media, 2020.

[15] C. Dwork. "Differential Privacy". In: *Proceedings of the 33rd International Colloquium on Automata, Languages and Programming (ICALP)*. Springer, 2006, pp. 1–12.

[16] K. Papineni, S. Roukos, T. Ward, and W.-J. Zhu. "BLEU: a Method for Automatic Evaluation of Machine Translation". In: *Proceedings of ACL*. 2002, pp. 311–318.

[17] C.-Y. Lin. "ROUGE: A Package for Automatic Evaluation of Summaries". In: *ACL Workshop on Text Summarization Branches Out*. 2004, pp. 74–81.

[18] T. Zhang, V. Kishore, F. Wu, K. Q. Weinberger, and Y. Artzi. "BERTScore: Evaluating Text Generation with BERT". In: *International Conference on Learning Representations (ICLR)*. 2019.

[19] C. Dwork and A. Roth. "The Algorithmic Foundations of Differential Privacy". In: *Foundations and Trends in Theoretical Computer Science* 9.3–4 (2014), pp. 211–407. DOI: `10.1561/0400000042`.

[20] R. Schwartz, J. Dodge, N. A. Smith, and O. Etzioni. "Green AI". In: *Communications of the ACM* 63.12 (2020), pp. 54–63. DOI: `10.1145/3381831`.

[21] P. Liang et al. "HELM: Holistic Evaluation of Language Models". In: *Stanford Center for Research on Foundation Models* (2022). URL: `https://crfm.stanford.edu/helm`.

[22] A. Wang, A. Singh, J. Michael, F. Hill, O. Levy, and S. R. Bowman. "GLUE: A Multi-Task Benchmark and Analysis Platform for NLP". In: *Proceedings of the International Conference on Learning Representations (ICLR)*. 2018. URL: `https://openreview.net/forum?id=rJ4km2R5t7`.

[23] A. Wang, Y. Pruksachatkun, N. Nangia, A. Singh, J. Michael, F. Hill, O. Levy, and S. R. Bowman. "SuperGLUE: A Stickier Benchmark for General-Purpose Language Understanding Systems". In: *Advances in Neural Information Processing Systems (NeurIPS)*. 2019, pp. 3266–3280.

[24] Y. Peng, S. Yan, and Z. Lu. "Transfer Learning in Biomedical Natural Language Processing: An Evaluation of BERT and ELMo on Ten Benchmarking Datasets". In: *Proceedings of the 18th BioNLP Workshop and Shared Task*. 2019, pp. 58–65. DOI: `10.18653/v1/W19-5006`.

[25] P. Rajpurkar, J. Zhang, K. Lopyrev, and P. Liang. "SQuAD: 100,000 Questions for Machine Comprehension of Text". In: *Proceedings of the 2016 Conference on Empirical Methods in Natural Language Processing (EMNLP)*. 2016, pp. 2383–2392. DOI: `10.18653/v1/D16-1264`.

[26] O. Uzuner, Y. Luo, and P. Szolovits. "Evaluating the State-of-the-Art in Automatic De-identification". In: *Journal of the American Medical Informatics Association* 14.5 (2007), pp. 550–563. DOI: `10.1197/jamia.M2444`.

[27] I. Pilán, P. Lison, L. Øvrelid, A. Papadopoulou, D. Sánchez, and M. Batet. "The Text Anonymization Benchmark (TAB): A Dedicated Corpus and Evaluation Framework for Text Anonymization". In: *Computational Linguistics* (2022). Pre-print available at arXiv:2202.00443.

[28] A. Srivastava et al. "Beyond the Imitation Game Benchmark (BIG-Bench)". In: *arXiv preprint arXiv:2206.04615* (2022). URL: `https://arxiv.org/abs/2206.04615`.

[29] G. Loiseau, D. Sileo, D. Riquet, M. Meyer, and M. Tommasi. "Tau-Eval: A Unified Evaluation Framework for Useful and Private Text Anonymization". In: *Proceedings of the 2025 Conference on Empirical Methods in Natural Language Processing: System Demonstrations*. Association for Computational Linguistics, 2025, pp. 216–227. URL: `https://aclanthology.org/2025.emnlp-demos.16`.

[30] W. Huang, Y. Wang, and C. Chen. "Privacy Evaluation Benchmarks for NLP Models". In: *arXiv preprint arXiv:2409.15868* (2024). URL: `https://arxiv.org/abs/2409.15868`.

[31] Y. Sun, V. Schlegel, S. Nandakumar, I. Zahid, Y. Wu, Y. Wu, H. Li, J. Zhang, W. Del-Pinto, G. Nenadic, S. K. Lam, and A. A. Bharath. "SynBench: A Benchmark for Differentially Private Text Generation". In: *arXiv preprint arXiv:2509.14594* (2025). URL: `https://arxiv.org/abs/2509.14594`.