RESILIENT SECURITY: THREAT MODELING AND DEFENSIVE STRATEGIES FOR LARGE LANGUAGE MODELS PLATFORMS

SN: 24076607

ABSTRACT

This report presents a comprehensive approach to securing Large Language Model (LLM) platforms through threat modeling and defensive strategy development. We analyze a Flask-based AI dialogue system with user authentication, conversation management, and content moderation capabilities. Using STRIDE methodology, we identify and prioritize threats including session hijacking, brute force attacks, NoSQL injection, and DDoS attempts, demonstrating through practical simulations how adversaries could extract sensitive information. We then implement a multi-layered defense incorporating MFA, bcrypt password hashing, rate limiting, HTTPS encryption, and AI-driven content filtering to prevent storing prohibited content. The paper addresses regulatory compliance with GDPR[1], CRA, and PSTI frameworks while considering ethical implications of AI security systems. Finally, we evaluate enterprise scaling considerations and innovative approaches like contextual authentication and privacy-preserving techniques to ensure long-term viability of secure LLM platforms.[1]

Index Terms— Large Language Models, threat modeling, cybersecurity, authentication, privacy, STRIDE, defensive strategies, ethical AI

1. COURSEWORK 1: THREAT MODELING & ATTACK SIMULATION

1.1. Introduction and objectives

This project simulates a widely used AI dialogue model system developed using Flask (Python) and MongoDB, creating a user login, a registration interface and a LLM chat interface. The system stores the user account password and other related personal information in the MongoDB database, and the user's chat information is also stored in the associated database to protect and manage it. At the same time, machine learning is used to load the BERT model[2] to identify and delete banned messages in the chat to prevent the database from being contaminated with banned messages. The system will include the following elements, which will be analysed:

1. Front-end HTML forms: login, registration, AI chat interface, etc.
2. Back-end: a Flask-based server that manages user information and dialogue content.
3. Database: MongoDB (local instance) to store user data and messages.

First, when the webpage is opened, the interface is shown in the diagram below. Fig. 1 shows the system login interface and Fig. 2 shows the registration interface that the system retrieves.

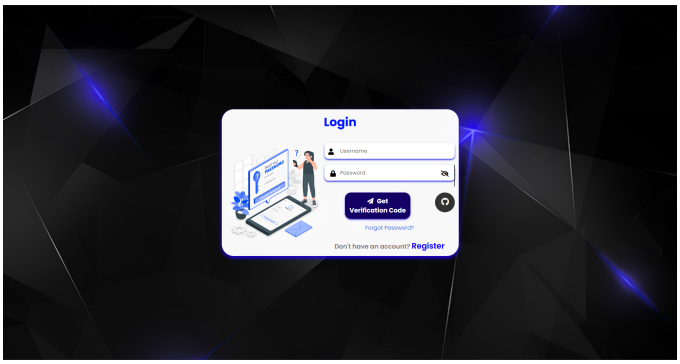


Fig. 1: Login interface of the system

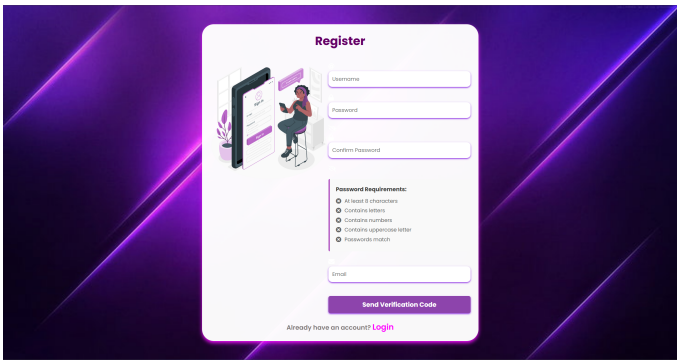


Fig. 2: Registration interface of the system

[1]The code is provided on GitHub: https://github.com/yushiran/ELEC0138Coursework_Group5 and the presentation video is available at: https://www.youtube.com/watch?v=dbHFTfhiuac.

After registration, a database message is generated in MongoDB containing all the user's personal data, including email address, password, etc. After logining, the user is redirected to the system's AI dialogue interface, where the same dialogue messages are machine-learned, classified and stored in the appropriate database.

There are several critical assets in the system and these are protected in the following to maintain privacy, data integrity and secure operation. The details are analysed below:

1. user login and registration information: This information can be exposed when accessing web pages, submitting web pages, and in the MongoDB database, so if not stored in MongoDB using the bcrypt hash algorithm, successful intrusion by outsiders could lead to impersonation or unauthorized access.

2. MongoDB databas: Data protection in the database is the most important part, which includes all legitimate personal data entered into the system. It also stores user conversations, including private data and illegal queries, which must be classified and monitored to avoid contamination of the local database.

3. Session identifiers: This type of data is stored in client cookies across a Flask session and, if intercepted, a malicious user can access user data without logging in.

4. System configuration: System configuration information is unlikely to be exposed, but it is a risk. The env file stores the key and connection string; once exposed, visitors can read the database data directly, compromising the encryption and security of the database.

5. HTTP communication channels: HTTP traffic is very easy to capture with Wireshark. Therefore, the use of HTTPS is also part of effective data management and security.

1.2. Threat model

Systems without built-in defences are exposed to a number of threats, all of which can have a serious impact on our websites and users, including privacy breaches and loss of profit.

Cyberattacks are the most common, immediate and obvious threat. Considering that users' names, emails, account names and passwords are directly or indirectly used and stored in our system, it is the most important task to prevent this information from being collected and exploited by attackers.

Insider threats are another key risk category. Since data is stored in MongoDB, developers and administrators with direct access to the database may accidentally or maliciously access or even leak user information.

Emerging risks, though less immediate, should be proactively considered in system design. Further research into encryption algorithms for long-term data is needed due to the ongoing development of quantum computing technology, which may lead to the obsolescence of today's encryption algorithms in the future.

1.3. Assess impact and prioritize threats

The most direct method of attack in cyberattacks is to brute-force a website's user login information and attempt to log in. In the absence of defensive measures, the attacker can use automated scripts to systematically guess user credentials, especially for users who use weak passwords and repeated passwords, the success rate of this attack is very high.

Furthermore, because we use MongoDB as the data storage tool in our system, we are vulnerable to NoSQL injection threats. If user input is embedded directly into the query object without rigorous cleaning or architectural enforcement, an attacker may be able to manipulate authentication logic or extract sensitive data. Unlike traditional SQL injection, NoSQL injection in document-based databases is typically less well known and therefore more easily overlooked by developers.

Nevertheless, the system is vulnerable to distributed denial-of-service (DDoS) attacks, especially via HTTP flooding. While a high volume of requests for login or registration endpoints does not compromise user information and privacy, it can exhaust server resources and cause legitimate users to temporarily lose access to the service.

Insider threats are inherently impactful due to the privileged nature of access although it is less common. In the absence of auditing or fine-grained access control, a single point of abuse can lead to a massive privacy breach.

Quantum computing presents a serious challenge to modern cryptographic assumptions. Many widely used encryption schemes, including RSA[3] and elliptic curve ciphers (ECC)[4], are theoretically vulnerable to quantum attacks, most notably through the Shor algorithm[5]. Once quantum computers reach a sufficient number of quantum bits (n) and error-correction capabilities, these cryptosystems are no longer reliable.

We used the STRIDE[6] threat classification model to classify and evaluate the threats identified in the system, specifically assessing the nature and severity of each threat to determine its priority:

- T1 - Plain Text Transfer (I: Information Leakage)

The project initially transmitted credentials via HTTP, which poses a serious risk according to the STRIDE model. As previously mentioned, it is very easy to perform packet sniffing (e.g. Wireshark) on the same network, which would compromise the privacy of all user input.

- T2 - SQL Injection (S: Spoofing and E: Privilege Escalation)

  SQL injection attempts allow an attacker to impersonate a valid user and this method will be very effective when the database is not encrypted. Although this can be partially mitigated using bcrypt password checking, it still allows unauthorised access to user sessions. Therefore, this remains a high priority.

- T3 - Forced login (D: Denial of Service)

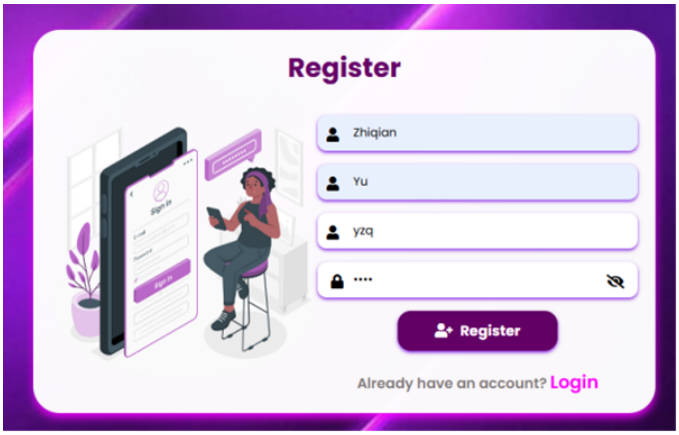
  Although a forced login scenario or DDOS network attack cannot significantly steal or decrypt databases and private customer information, it can also cause some disruption to the client or server and degrade performance. Therefore, they have medium to high priority.

- T4 - Session hijacking (S: Spoofing)

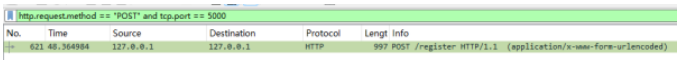
  If an attacker gets hold of a valid session cookie, they can bypass authentication. Although the impact is high, the likelihood is low.

- T5 - *.env* file leak (I + E)

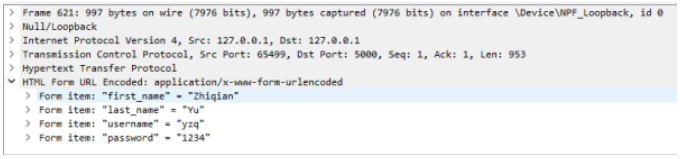
  Leaking *.env* files can lead to information disclosure and privilege escalation, especially if MONGOURI or AESKEY is compromised. However, depending on the misconfiguration, this is less likely to happen in a controlled deployment.



(a) Demo Registration



(b) Capture Information



(c) Information content

(d) AI dialogue Information content

Fig. 3: Wireshark's HTTP message capture

1.4. Data Sources and attacks set-up

1.4.1. Wireshark's HTTP message capture

We start with the registration information as an example as shown below. After registering the user information and clicking submit, we use Wireshark to add the filtered information header as shown in Figure X to the same network segment, which will show the record of all the requests submitted under that segment. So after clicking on that record we can get the relevant user information in Figure 3.

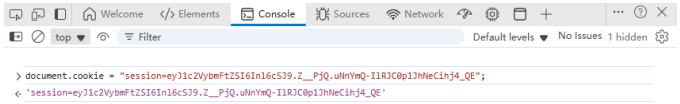
1.4.2. Session Hijacking

After the user has successfully logged in, there is a legitimate session cookie in the browser, at this point we can use Wireshark to get the value of this cookie through a packet grabber, inject the cookie into our own browser, and then visit the /secured page. This will allow us to access the /secured page without logging in, thus obtaining the user's privacy directly. As shown in Fig. 5, we have captured a login request in wireshark that contains a cookie message.

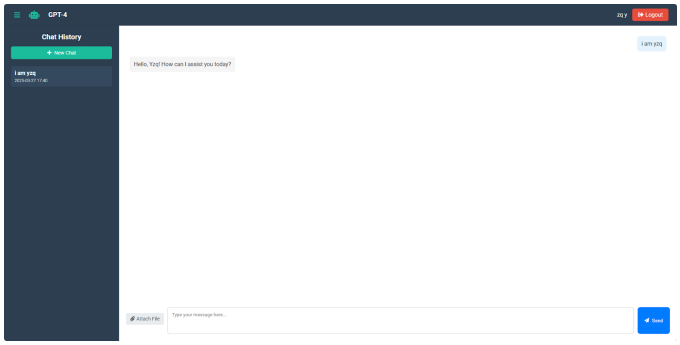
After getting this cookie information, open the login

Fig. 4: Information of Cookie

screen and enter the information shown in Fig. 5 in the Console, we will see that the system has recognised the content of the cookie. So we enter the page we need to go to in the web port which is http://127.0.0.1:5000/secured and we can see that we have successfully jumped to the user interface.



(a) Cookie content



(b) User interface

Fig. 5: Information of Cookie

Applications allow users to log in with a username and password. In the original design, there was no rate limiting or CAPTCHA[7] mechanism and the login endpoint was vulnerable to brute force attacks. We simulated an attacker using a multi-threaded Python script to systematically guess numeric passwords for known usernames. The script also allows the attacker to customize the length of the password to reduce the complexity of cracking if the user's account information is obtained by other means, such as the approximate length of the password or account name, etc. The script also allows the attacker to customize the length of the password to reduce the complexity of cracking. Once the correct password is recognized, the script terminates and saves the stolen credentials. For demonstration purposes, plain numbers are used as passwords and plain letters are used as user id.

In addition, DDoS attacks are designed as decoys to cause confusion and create opportunities before executing attacks that actually threaten the security and privacy of user information. The initial Web services do not enforce any form of request limitation or IP blacklisting. An attacker can exhaust server resources by issuing a large number of seemingly legitimate HTTP GET requests. In the designed DDoS attack script, four types of attacks are included: HTTP Flood, UDP Flood, SYN Flood, and IP Fragment Flood.

2. COURSEWORK 2: SECURITY & PRIVACY DEFENSE STRATEGY

2.1. Security (or Privacy or both) Interaction/visualisation/actuation system

We propose a multi-layered, defense-in-depth architecture that integrates preventive, detective, and recovery mechanisms to mitigate the threats identified in our system.

For access control and authentication, we enforce

Fig. 6: Brute force attack Python script demo

Multi-Factor Authentication (MFA) using TOTP or device-based tokens and adopt a zero-trust architecture where each access request is continuously verified rather than implicitly trusted. User credentials are stored using bcrypt hashing to prevent SQL injection and accidental leaks by developers who can access the database. The rate limiting is also implemented to prevent brute-force attempts.

In terms of data security, all communication between client and server is secured via HTTPS, eliminating the risk of plain-text interception during login or session activity. Additionally, cookies are secured using flags such as HttpOnly, Secure, and SameSite=Strict to mitigate the risk of session hijacking through packet sniffing tools like Wireshark. To further ensure data integrity and legal compliance, we also implemented AI-driven content filtering that analyzes user input in real time. If violent, criminal, or otherwise illegal language is detected, the system will automatically block the content and prevent it from being written into the database. This preemptive measure helps us avoid storing unlawful data and enhances compliance with data protection regulations.

For backend integrity, we performed input validation to mitigate NoSQL injection attacks, ensuring user inputs are sanitized before being processed in MongoDB queries. We also implemented basic session binding, where a session is invalidated if the IP address does not match the original login IP, providing an extra layer of protection against session hijack scenarios.

These implementations significantly enhance the overall resilience of our system against practical threats such as interception, brute-force login, and injection-based exploits.

2.2. Prototype Implementation and Demonstration

This section illustrates the demonstration and implementation of the prototype we developed. This prototype demonstrates how real-world threats, such as plain text data transmission, unauthorized access, database breaches, and data pollution, can be mitigated using a layered security approach. The following subsections detail the practical setup and visual demonstration of these implementations, supported by code snippets, screenshots, and system behavior after applying each countermeasure.

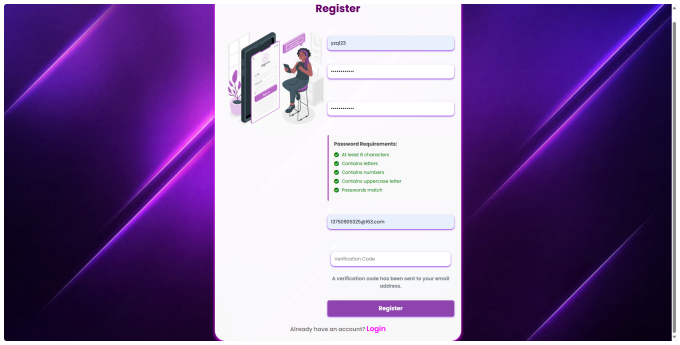
2.2.1. Enable TLS Encryption to Prevent Data Leakage

First, because Wireshark may try to capture sensitive information (e.g. usernames, passwords, chats, etc.) in clear text, we implemented TLS encrypted communication for our web application, upgrading it from the original HTTP protocol to HTTPS, and this section documents the whole process from certificate generation to the Flask project's access to TLS. First, we generate a self-signed certificate and a private key locally using OpenSSL: cert.pem: server-side public key certificate and key.pem: private key. Then, in the Flask project, we enable TLS using the ssl_context parameter, simply by specifying the path to cert.pem and key.pem at runtime. With this encryption, even if attackers use Wireshark to capture packets on the same network, they will not be able to obtain valid sensitive information. The data transmission process is changed from plain text to an encrypted stream, which significantly improves the security of user authentication, chat, and other operations.

2.2.2. Implementing Multi-Factor Authentication (MFA)

Next, we add the authentication scheme in the main interface, that is, we add the email verification code as a double guarantee, to prevent hackers from breaking in violently. To cooperate with the email authentication code function, we modify the specific requirements during registration, besides the simple username and password, we add the registration of additional email information. The specific information is shown in the Fig. 7 below.

(a) Email verification login page



(b) Email verification registration page

Fig. 7: Email verification

2.2.3. Implementing GitHub OAuth Authentication

To enhance both security and user experience, we implemented GitHub OAuth authentication as an alternative login method. OAuth (Open Authorization) is an open standard protocol that allows secure authorization without exposing the user's credentials to the application. Instead, it uses token-based authentication where users authorize third-party applications to access their information without sharing passwords.

This implementation provides several security benefits:

- Eliminates password transmission between our application and the user
- Leverages GitHub's robust security infrastructure for authentication
- Reduces the risk of credential stuffing attacks
- Provides automatic MFA if the user has enabled it on their GitHub account

The OAuth flow works as follows:

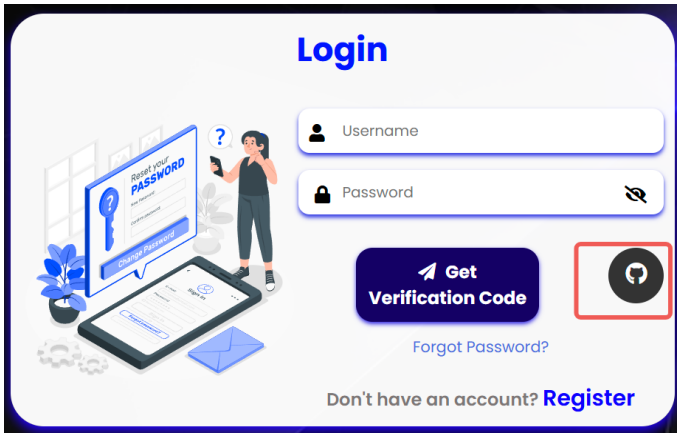
1. The user clicks "Sign in with GitHub" on our login page
2. They are redirected to GitHub's authorization page
3. After approving access, GitHub redirects back to our application with a temporary code
4. Our server exchanges this code for an access token
5. The token is used to retrieve the user's GitHub profile information
6. We create or authenticate the user based on their GitHub identity

Figure 8 shows the GitHub OAuth login option we implemented:

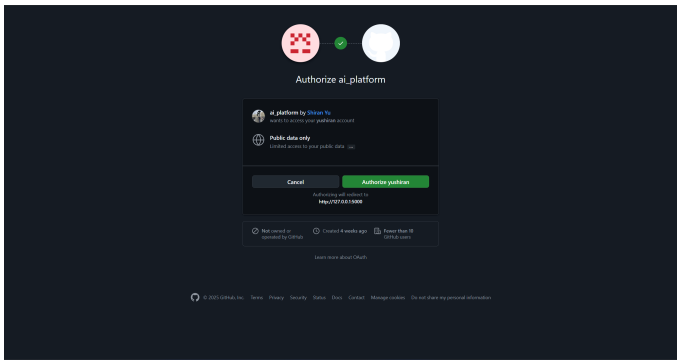
This OAuth implementation balances security and convenience, allowing users to authenticate without creating new credentials while still maintaining strong security protections through GitHub's authentication infrastructure.

2.2.4. Implementing Rate Limiting and CAPTCHA

At the same time we are registering information into the database, considering the use of the MongoDB database itself has a very strong privacy protection mechanism, so we do not consider for the time being the processing of this database. But in this particular case, after grabbing the .env file, we also have the opportunity to enter the database through the URI. Therefore, when we store the relevant information, as shown in the Fig. 9 below, we encrypt the password with a hash algorithm, which can effectively protect the user's information even if it is illegally invaded.



(a) Github OAuth button



(b) Github OAuth page

Fig. 8: GitHub OAuth

```
Username: wxm
Password Hash: $2b$12$Re8hCnMiKQedi3wvMtMaVei7YcQ/tXqsVOcu/8x1ekbD50jmNx0Zq
First Name: xiaomin
Last Name: wang
--------------------------------------
Username: xm
Password Hash: $2b$12$rVO4ly0qh5o2XlWYr4rRluxN4e69gvLUNPP5XSloAK8CFe2POcsLy
First Name: min
Last Name: xiao
--------------------------------------
Username: tongxue
Password Hash: $2b$12$2NLrqxYDga4mUBEEEmpucezpuqBs7fBIrNlWOZHGXj/ZfoxoSy0Vq
First Name: hua
Last Name: li
```

Fig. 9: Partially encrypted information

2.2.5. AI-based Mechanism to Prevent Sensitive Data Pollution

To implement an effective mechanism for detecting and filtering sensitive or toxic user input to prevent them from entering the database and polluting the content, we developed a lightweight text classification model capable of distinguishing between safe and harmful content. The model was trained using a labeled dataset containing example sentences that were manually categorized as either benign or containing inappropriate language, such as violent, abusive, or otherwise prohibited expressions.

1. Train Dataset

   The train dataset is from the Toxic Comment Classification Challenge on Kaggle[2]. The dataset consists of approximately 160,000 text samples, each annotated with six binary labels indicating different types of harmful content: toxic, severe_toxic, obscene, threat, insult, and identity_hate. Each label is represented as either 0 or 1, signifying the absence or presence of the respective trait in the text. To simplify the classification task into a binary setting, we consolidated these six labels into a single unified label. Specifically, if a sample is marked with a value of 1 in any of the six categories, we consider it harmful and assign it a final label of 1; otherwise, the label is 0. This preprocessing step allows the model to focus on general toxic language detection rather than distinguishing between different subtypes of harmful content, which is more suitable for our goal of preventing any sensitive or offensive text from entering the database.

2. Model

   The model used is a pre-trained Bert-base-uncase model[3] [8]. It consists of 12 transformer layers (also called encoder blocks), 768 hidden units per layer, and 12 attention heads, totaling approximately 110 million parameters. As a deep bidirectional model, it considers both left and right context in all layers, making it highly effective for capturing the meaning and nuances of natural language. By using this model as a feature extractor or fine-tuning it for binary classification, we are able to significantly improve the model's ability to detect toxic and sensitive content.

3. Result

   Due to the large size of the training dataset and the limitations of available computational resources, we limited the model training to 3 epochs. During training, the model parameters were updated using the Adam optimizer with weight decay, and binary cross-entropy was used as the loss function. Table 1 shows the model performance over epochs. The model achieved over 96% accuracy, with F1 scores

[2]The dataset is provided on https://www.kaggle.com/c/jigsaw-toxic-comment-classification-challenge/data?select=train.csv.zip

[3]The model is provided on https://huggingface.co/google-bert/bert-base-uncased

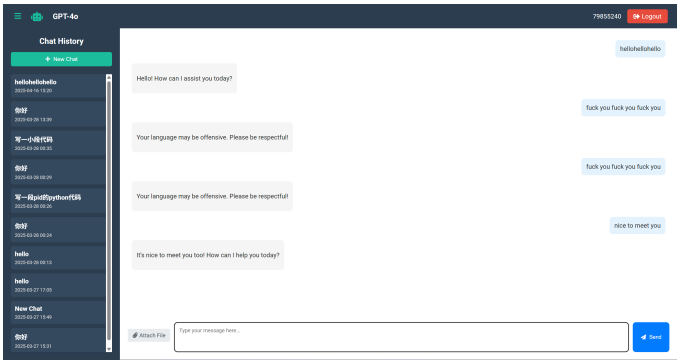
remained stable, suggesting that the model had learned meaningful patterns without significant overfitting.

Table 1: Model performance over epochs

Epoch	Training Loss	Accuracy	F1 Score
1	0.182700	0.955500	0.820721
2	0.072100	0.960000	0.817352
3	0.021700	0.960500	0.819222

After training, the model was uploaded to the Hugging Face Model Hub[4], which allows the model to be directly loaded via the transformers library without any additional configuration. Finally, we integrated it into the Flask backend.

As shown in the Fig. 10, although we present four dialogues in the dialogue box, the AI system detects that some of the dialogues contain sensitive words and therefore initiates a warning. And we can also clearly see in the database that the content stored in the database does not contain the sensitive content that was warned.



(a) Chat interface after anti-pollution


```
_id: ObjectId('67ffca8f29f100445dee20f8')
username : 79855240
model : "gpt-4o"
▾ messages : Array (4)
  ▾ 0: Object
      role : "user"
      content : "hellohellohello"
      timestamp : 2025-04-16T15:19:58.623+00:00
  ▾ 1: Object
      role : "assistant"
      content : "Hello! How can I assist you today?"
      timestamp : 2025-04-16T15:19:58.623+00:00
  ▾ 2: Object
      role : "user"
      content : "nice to meet you"
      timestamp : 2025-04-16T15:20:20.631+00:00
  ▾ 3: Object
      role : "assistant"
      content : "It's nice to meet you too! How can I help you today?"
      timestamp : 2025-04-16T15:20:20.631+00:00
created_at : 2025-04-16T15:19:43.264+00:00
updated_at : 2025-04-16T15:20:20.631+00:00
```


(b) Database information after anti-pollution

Fig. 10: AI dialogue system after anti-pollution

2.3. Regulation and Ethical considerations

Our system's security architecture must comply with major data protection and cybersecurity regulations while addressing critical ethical considerations around AI implementation.

2.3.1. Regulatory Compliance

The system has been designed to meet requirements from three key regulatory frameworks:

General Data Protection Regulation (GDPR): Personal data protection is ensured through bcrypt password hashing, secure MongoDB storage patterns, and Flask session management. User consent and control mechanisms are implemented via time-limited verification codes and email validation for authentication events. Additionally, the system adheres to data minimization principles by collecting only essential information required for functionality.

Cyber Resilience Act (CRA): The system incorporates security-by-design principles through comprehensive input validation and anti-brute force mechanisms, including account lockouts. Vulnerability handling is addressed with multi-factor authentication and AI-driven content filtering. Regular security audits are conducted to identify and mitigate potential weaknesses, ensuring a robust and resilient security framework.

The Product Security and Telecommunications Infrastructure Act (PSTI) compliance is achieved through secure communication via HTTPS/TLS implementation and encrypted authentication tokens, alongside robust data security measures in transit and storage, including validated file upload handling and secure binary storage.

2.3.2. Ethical Considerations

Several ethical challenges were identified and addressed:

Surveillance and Privacy Risks: To address surveillance and privacy risks, we implemented data retention policies that automatically anonymize older conversations, ensuring that sensitive user data is not retained longer than necessary. Transparent user controls for data management were added, allowing users to manage their data effectively. Additionally, clear privacy notices were created to explain data collection practices, ensuring users are fully informed about how their data is handled.

Security vs. Usability Balance: Balancing security and usability was achieved by providing clear guidance

[4]The model is provided on https://huggingface.co/ZheZHEZHE020106/zt-harmful_language_detecting

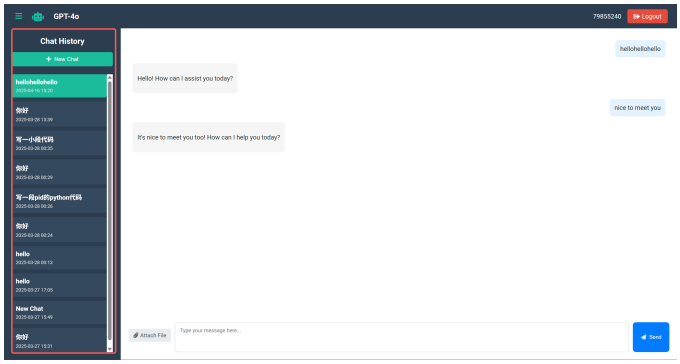


Fig. 11: Transparent user controls

on security requirements, helping users understand and comply with necessary measures. Progressive security mechanisms were implemented based on the sensitivity of operations, ensuring that security measures are proportional to the risk involved. Alternative authentication methods were also offered to enhance accessibility, ensuring that all users can interact with the system securely and conveniently.

The system achieves a balance between robust security and ethical responsibility, ensuring both compliance with regulatory requirements and respect for user privacy and autonomy. These considerations are particularly important in AI systems where automated decisions impact user experience and data handling.

2.4. Scalability, Innovation & Enterprise Considerations

2.4.1. Enterprise Scalability

To ensure our system scales effectively for enterprise adoption, we propose a multi-layered approach to infrastructure development. For technical infrastructure, containerization through Docker and orchestration via Kubernetes would allow horizontal scaling based on demand and consistent deployment across environments. Our MongoDB implementation would evolve to incorporate sharding for distributed storage, replica sets for high availability, and connection pooling to handle thousands of concurrent users.

Performance at scale would be achieved through strategic implementation of Redis or Memcached for session management, response caching for frequent AI queries, CDN integration, and appropriate API rate limiting. This infrastructure would support essential enterprise authentication integration requirements, including:

- SAML for Single Sign-On integration
- LDAP/Active Directory for enterprise user management
- Expanded OAuth provider support
- Advanced MFA protocols for organizational security policies

For operational stability at scale, we would implement comprehensive monitoring through an ELK stack or similar solution, real-time metrics with Prometheus/Grafana, custom health dashboards, and automated alerting for system issues. Additionally, CI/CD pipelines would ensure reliable deployment with automated testing, canary deployments, and infrastructure-as-code practices.

2.4.2. Innovative Security Approaches

Our platform implements several innovative approaches that differentiate it from traditional security methods:

AI-Powered Content Security: Our existing ML-based content moderation can be extended to detect potential security threats such as social engineering attempts and data exfiltration in user prompts, creating an intelligent defensive layer that evolves with threats.

Privacy-Preserving AI: We propose implementing federated learning techniques to improve AI security models without exposing sensitive user conversation data, and exploring homomorphic encryption to allow AI processing on encrypted data while maintaining privacy.

2.4.3. Long-term Viability and Cost Management

The long-term sustainability of our security architecture depends on balanced resource allocation and strategic planning. For cost optimization, we would implement:

- Tiered usage plans based on organizational needs
- Automatic scaling during varying usage periods
- Token/query budgeting with department-specific tracking
- Advanced caching strategies to reduce redundant AI calls

Future-proofing strategies include designing for quantum resistance by implementing post-quantum cryptographic algorithms where possible, establishing a component replacement strategy for rapid integration of emerging security technologies, and maintaining compliance flexibility through modular policy engines that adapt to regulatory changes.

Through these comprehensive approaches to scalability, innovation, and long-term planning, our platform can evolve from a demonstration project into an enterprise-ready system capable of supporting thousands of users while maintaining security, performance, and regulatory compliance.

3. REFERENCES

[1] "Eu general data protection regulation (gdpr): Regulation (eu) 2016/679 of the european parliament and of the council of 27 april 2016 on the protection of natural persons with regard to the processing of personal data and on the free movement of such data, and repealing directive 95/46/ec (general data protection regulation)," 2016, OJ 2016 L 119/1.

[2] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova, "Bert: Pre-training of deep bidirectional transformers for language understanding," in North American Chapter of the Association for Computational Linguistics, 2019.

[3] Ronald Rivest, Adi Shamir, and Len Adelman, A Method for Obtaining Digital Signatures and Public-Key Cryptosystems (1978), pp. 463–474, 02 2021.

[4] Neal Koblitz, "Elliptic curve cryptosystems," Mathematics of Computation, vol. 48, no. 177, pp. 203–209, 1987.

[5] P.W. Shor, "Algorithms for quantum computation: discrete logarithms and factoring," in Proceedings 35th Annual Symposium on Foundations of Computer Science, 1994, pp. 124–134.

[6] Microsoft, "The stride threat model," 2009, Accessed: 2025-04-19.

[7] Luis von Ahn, Manuel Blum, Nicholas J. Hopper, and John Langford, "Captcha: Using hard ai problems for security," in Advances in Cryptology — EUROCRYPT 2003, Eli Biham, Ed., Berlin, Heidelberg, 2003, pp. 294–311, Springer Berlin Heidelberg.

[8] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova, "Bert: Pre-training of deep bidirectional transformers for language understanding," arXiv preprint arXiv:1810.04805, 2018.

4. EXPLAINING CONTRIBUTION

I served as the primary developer for our secure LLM platform, handling both front-end and back-end development. I designed intuitive user interfaces using HTML/CSS/JavaScript and built the core Flask application architecture. My work included implementing MongoDB schema design for user data and conversations, and integrating multiple security measures: MFA with email verification, TLS/HTTPS encryption, GitHub OAuth, password hashing, secure session management, input validation, and rate limiting. I managed system integration between the AI response system, content moderation service, and authentication mechanisms. Additionally, I established the GitHub repository, coordinated team contributions through code reviews, and documented the system architecture and security features for the final report, all while balancing security with usability.