Secure Chat System

Aaron Zhang
y.zhang347@newcastle.ac.uk

Haifa Alkhudair
h.alkhudair2@newcastle.ac.uk

Harshvardhan Patil
h.patil2@newcastle.ac.uk

Jiashu Chen
j.chen106@newcastle.ac.uk

Mrunal Kale
m.k.kale2@newcastle.ac.uk

Nayana Agrahara Dattatri
n.agrahara-dattatri2@newcastle.ac.uk

Wei Xie
w.xie11@newcastle.ac.uk

Yihang Cai
y.cai26@newcastle.ac.uk

***Abstract*—This project report demonstrates the development of a secure chat system designed to enhance the security and privacy of online communications. We use various technologies like FastAPI framework combined with WebSockets, PostgreSQL, Redis, ReactJS, docker compose, Prometheus to build this system. Various techniques are used to develop this like hashing, end to end encryption, secure file transfer, OAuth2, JWT tokens, botnet prevention. The performance of the system is evaluated by concurrently testing the system functions in a virtual machine environment. The results show that the system corresponds quickly and has good stability under low to moderate loads, but the average response time and error rate of the system are elevated in the case of high concurrent requests. In subsequent research, the system performance under high concurrency needs to be optimised and the load balancing mechanism needs to be optimised to cope with the constantly updated cyber threats.**

***Index Terms*—secure chat system, data security, privacy, concurrency performance, JWT, End-to-End encryption**

I. Introduction

In modern society, with the rapid development and wide acceptance of the Internet, there has been a shift in the way people communicate, and the use of online chatting systems has become a common way of communication nowadays [1]. These systems offer great convenience and make human-to-human communication more efficient. However, there are flaws in online chat systems and they are not as secure as one might think [2]. The digital environment can pose security risks to online chat and security issues for users, such as unauthorised account intrusion (hacking), exposure of sensitive information (information leakage) and deliberate manipulation of information (information tampering). These security breaches pose a threat to users and can have serious consequences, such as attackers analysing a user's personal data to find other important data, such as bank account information, and committing Internet crime - bank fraud; and may even give rise to legal liabilities and reputational damage to a business or organisation [3]. The security of personal information and communication channels is of paramount importance in a society where digital interactions are increasingly used. Therefore, it is necessary to develop a secure chat system. A secure chat system should integrate advanced security measures and become a weapon against common and emerging threats, ensuring that users can communicate freely without fear of threats to their personal information or informational data, and without fear of communications being intercepted or engaged in by unauthorised persons [4], thus the system is based on protecting the privacy of users and enhancing the security of the communication itself.

The main objective of this project is to design and develop a secure chat system that addresses the various flaws to digital communication platforms, therefore it must be equipped to protect user privacy and ensure data security by preventing unauthorised access, guarding against data leakage and maintaining the integrity of the information exchanged. A key component of this chat system is the implementation of basic user login and registration functionality, which serves as the foundation of a security framework that fundamentally prevents unauthorised access and creates a barrier that greatly reduces the risk of intrusion and misuse by requiring users to authenticate themselves. However, central to the security of the system is end-to-end encrypted communication, which ensures that the content of messages is visible only to the sender and receiver [5], thus preventing the content of communications from being intercepted or eavesdropped, and also JWT technology is required to ensure the integrity of the call content. This level of security is crucial in today's digital age, where the privacy of conversations is often compromised. At the same time, our project also focuses on user experience. We designed an intuitive and engaging user interface to improve the usability of the system, making it not only secure but also user-friendly. In short, the ultimate goal of this project is to develop a secure and easy-to-use chat system within a limited time frame.

This project aims to address the common security vulnerabilities that exist in some current chat systems, and is dedicated to enhancing privacy protection and data security in the field of digital communication, which is a direct response to the urgent need of an increasing number of users for private and secure communication channels. In short, by integrating encryption and authentication technologies as well as enhancing the user experience, this project provides a platform that meets the current user needs and offers a viable solution for more people seeking a secure way to communicate online.

II. Related Work

Nowadays, all of us are unfamiliar with the topic of chatapp, which is one of the most popular chat methods in today's context. The history of Chatapp can be traced back to the Bulletin Board System (BBS) in the late 1970s and the

Internet Relay Chat (IRC) in the 1980s [6] [7]. These examples implement the simple application of chatapp: communication using text messages. Today's chat applications have been very large, such as WhasApp and China's WeChat are both international and national mainstream applications, realizing many functions, not only limited to text chat, but also extended to file transfer and saving. Today's AI is also the mainstream of The Times, and robots can understand natural language and use it in different application scenarios, such as shopping, self-service and human customer service.

Technology in chat applications is also advancing, such as machine learning and deep learning. The application of ML and DL algorithms enables Chatapp to learn user preferences and behavior patterns to provide personalized recommendations and services [8]. In addition, real-time communication technology, for example, enables users to use chat apps without latency, optimizing the user experience.

During the development of a chat app, user experience (UX) and interaction design are key factors to ensure the success of the app [9]. These designs need to consider not only the functionality and efficiency of the application, but also the user's feelings, needs, and preferences. A good app should ensure the feedback efficiency and operability of the app under a clear and relatively simple design. After the release of an app, developers should always pay attention to user feedback so as to update and improve the functions and shortcomings of the app in real time, so as to ensure the popularity of the app and user recognition.

SMS applications on the Internet are one of the most common means of communication today [10]. The existence of Chatapp facilitates people's life, so that communication does not need to be through the telephone or paper letter format, but can be achieved on the Internet. WhatsApp, which exists today, is a good example. Through this app, we can send messages or calls to friends, or send some files. From the initial text system to the present intelligent dialogue system, chatapp has increased the user experience. Different functions also make Chatapp more popular and convenient for people's life.

However,in this context, chatapp has many challenges and drawbacks. Several key challenges include copyright protection, integrity verification, authentication, and access control [11] [12] [13]. If the system does not protect the user's privacy enough, it will lead to the privacy information being peeped or stolen, which will lead to the loss of the user. If the authentication is not strict enough, the access rights can easily be obtained by outsiders.

In response to the user privacy issues mentioned above, developers need to ensure that chat systems respect user privacy and comply with laws and regulations when collecting and processing user data. Therefore, it is necessary to use data anonymization, differential privacy and other technologies to protect user data. For a chatapp, when the server processes user information, the received information should be encrypted. At this time, the server is only responsible for the transmission of information, and the end-to-end transmission needs to read the information through the secret key and decryption, which can ensure the security of user information and prevent data from being intercepted and eavesdropped during transmission. In addition, control and management of access should be reserved for authorized users, and validators can prevent unauthorized users from reading information. These are some countermeasures for user privacy.

Overall, although chatapp still faces many challenges and some potential risks that have not been solved yet, it is still the first choice for people to communicate in today's society, whether in study or work, it provides us with great convenience. The real-time communication of the App allows users to communicate with others at any time and place, improving communication efficiency. In addition, the versatility of App, such as photo sharing, geographical location sharing, file sharing, multimedia content transmission, etc., has greatly improved the efficiency of learning and work. Correspondingly, the popularity of apps has significantly reduced the cost of letters compared to the past.

III. Design and Implementation

A. *the design*

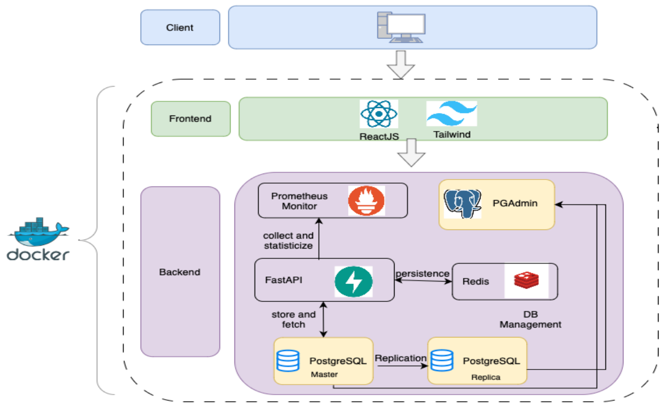


Fig. 1. System Architecture

In our approach, we designed the system to operate across three components: the Backend, the Frontend, and the Database. In the Backend, the language of choice is Python, and the framework of choice is FastAPI, providing a modern, fast (high-performance) web framework for building APIs and asynchronous web services. We chose FastAPI because of its ease of use, and compatibility with modern containerisation technologies such as Docker and Kubernetes. We chose Websocket, a protocol that provides a full-duplex communication channel over a single TCP connection. In our chat app, Websocket allows the server to send messages to the client in real-time and vice versa.

The system is developed using various services produced by Docker Compose, as detailed in the `'docker-compose.yml'` file within the codebase. These services include [database, backend, frontend, Redis, database backup, pgadmin, prometheus.]. Encapsulating each service into a Docker image ensures portability and consistency across different environments. To simplify deployment and ensure scalability and reliability, the entire application can be launched with a single command: `sudo docker compose up`. This command initiates Docker Compose, which deploys all services defined in the docker-compose.yml file. Therefore, the deployment process is independent of any specific IDE. Additionally, this deployment method requires no prerequisites other than having Docker installed, which makes it highly accessible.

On the database side, we chose PostgreSQL, which provides a powerful, scalable, and reliable database solution for our chat application that is suitable for such small projects. We also used Redis as a messages broker. In the Frontend, the languages used are HTML, CSS, JavaScript, and next.js, a modern web development framework based on React that provides server-side rendering and generation capabilities for static web pages. It improves performance, search engine optimisation and load times. Before the user attempts to connect to the chat service, the front-end application uses asynchronous to send authentication information (such as JWT) to the Backend, and the Backend verifies the information and allows the chat connection to be established upon successful verification.

To achieve a live chat, we established a socket connection. Once a user is authenticated, the client side will use the WebSocket protocol to establish a persistent connection with the server. WebSocket is a protocol that allows two-way communication between client and server, providing real-time messaging in chat applications. We used the Websocket library from FASTAPI to establish the websocket connection in the backend, and the frontend will typically use a Socket.IO-compatible JavaScript client library to create and manage WebSocket connections. In the setup, Next.js is used to build the user interface, ensuring a homogeneous front-end and back-end rendering, and facilitating interactions with back-end services. Leveraging WebSocket technology for real-time communication alongside the backend will deliver users a seamless live chat experience. Additionally, the front end will manage authentication, validate data, and potentially state management to maintain the chat application's responsiveness and user-friendliness.

Redis is an open-source, in-memory data structure storage system that can be used as a database, cache and messaging middleware. It supports many types of data structures such as strings, hashes, lists, sets, sorted sets, range queries, bitmaps, streams and geospatial index radius queries. In our system, we use Redis as a database to store the messages in case the recipient is offline. PostgreSQL is an open-source relational database management system. It is widely used for its reliability, scalability, and ease of use. To make the most of PostgreSQL, we use pgAdmin, a tool that provides various administration and management features. The system is monitored using Prometheus. Prometheus is a well-known open-source monitoring and alerting tool for containerised environments like Docker. It is built to achieve reliability and scalability in systems. In the front end, we used Tailwind, a utility-first framework that provides pre-designed, atomic utility classes to simplify the process of building web interfaces that it modern, responsive and time-efficient. Using these tools and techniques, we build our chat system to be efficient, reliable and easy to maintain. Docker provides ease of deployment and operation, and PostgreSQL, Redis and pgAdmin strongly support data caching and persistence. While Prometheus real-time monitoring and alerting for system metrics and performance data. This architecture ensures performance and stability when dealing with large numbers of users and messages.

B. Database Component

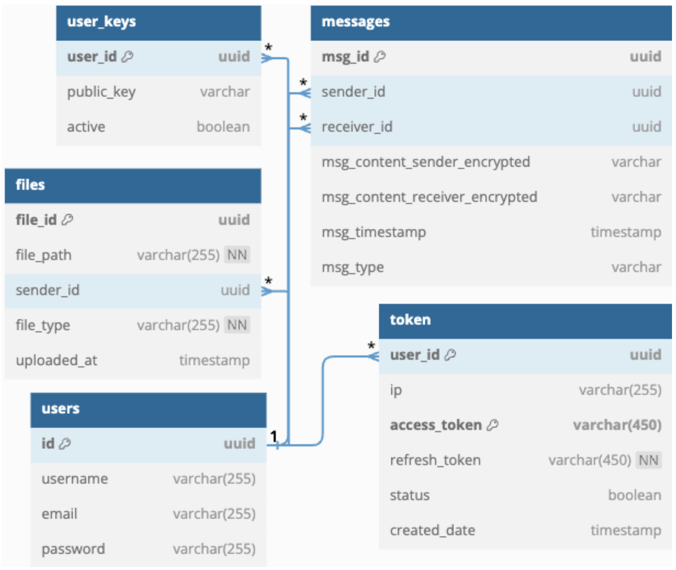


Fig. 2. ER Diagram

In this database schema, the Users Table serves as the central repository for user information within the system. Each user is uniquely identified by an ID and is associated with attributes such as their chosen username, email address, and hashed password for authentication. This table establishes connections with various other tables through relational links. For instance, the Token Table manages authentication tokens for user sessions, with each token being linked back to a specific user through the user_id attribute, thus establishing a many-to-one relationship between tokens and users. The Messages Table stores messages exchanged between users. It references the Users Table twice: once for the sender and once for the receiver of each message. This arrangement enables the system to associate each message with its respective sender and recipient. Similarly, the User Keys Table manages public encryption keys associated with users. Each key is linked to a particular user through the user_id attribute, facilitating secure communication within the system. Additionally, the Files Table stores information about files uploaded by users. Each file is linked to the user who uploaded it via the sender_id attribute, establishing a one-to-many relationship between users and files. This architecture enables efficient organization and retrieval of user-related data, ensuring seamless functionality and data integrity within the system.

C. Security Measures

For qualified chat software, the developer should consider the security measures implemented in the chat application. First of all, confidentiality: To keep user information confidential, we use private keys and public keys to encrypt the information sent by users. The private key and the public key are two keys in asymmetric encryption techniques that are usually used together to ensure the security of data transmission. This encryption method supports two main functions: encryption/decryption and signing/authentication. The private key is unique, held only by the key owner, and should not be shared or made public. A public key is publicly available and can be used by anyone to encrypt information or verify the digital signature generated by the corresponding private key. When a user sends a message to another user, the message is encrypted with the other user's public key at the server, and then the server stores the information in the database, and the other user transmits their public key to the server, and then uses their private key to decrypt the received message.

To ensure the integrity and authenticity of the system, we use the Jwt token. Jwt is used to securely transfer information between network application environments, and because this information is digitally signed, it can be verified and trusted. A JWT is actually an encoded and encrypted string consisting of three parts separated by periods: First, the header: contains the type of token (i.e. JWT) and the signature algorithm used. The second is the payload: it contains the declaration to be delivered, which can be the identity of the user, the issuer of the token, the expiration time, and so on. It can be public, private or registered. Finally, there is the signature: the first two parts are signed to verify that the message has not been tampered with during transmission. Because JWTs are designed to be self-contained, they are well suited for authentication and information exchange in cross-domain or microservice architectures.

Our project uses token authentication when users log in. Token authentication is a common authentication method in computer security and network communications, mainly used to confirm the legitimacy of a request or session. This approach is typically used in authentication and authorisation processes to ensure secure access for users or services. A token is an encrypted string that can contain data such as user identity information, privilege level and validity period. After a user successfully logs in, the server returns the generated token to the user. The token has a specific validity period and a new token must be obtained when it expires. If the token authentication is successful, the server provides appropriate services or data based on the information in the token. If the token authentication fails, the request is rejected.

D. Resilience Strategies

For backup, replication and recovery, our project uses PostgreSQL to build database storage information, back up the database and use scheduling scripts to trigger communication between the two databases. The process of backing up a database in PostgreSQL and triggering a backup periodically typically involves the following steps: setting up a backup policy, writing backup scripts, and setting up scheduled tasks. When writing backup scripts, you need to ensure that the execution of the backup scripts does not reveal sensitive information such as the database username and password. This sensitive information can be safely managed using environment variables or configuration files. Backup scripts record operations and any error information for troubleshooting and verifying the success of backup operations. They can be viewed in logs. In our project example, the database is backed up every day at 3am.

In terms of database communication, the project considers stream and logical replication, as well as data synchronisation using external tools. Stream replication allows one PostgreSQL server (master) to replicate data changes to one or more PostgreSQL servers (slaves) in real time. This method is mainly used to achieve high availability, disaster recovery and read load balancing. Logical replication allows more granular control and can selectively replicate specific tables or data changes. It is based on the concepts of publication and subscription and is suitable for database version upgrades, cross-database synchronisation and complex data distribution scenarios. In addition, tools such as pg-dump and pg-restore make it easy to backup and restore the database state to the same or different database instances. This is useful for data migration, backup and recovery, and disaster recovery plans.

The choice of synchronisation method depends on the specific business requirements, data consistency requirements, system architecture and operational complexity. For scenarios that require high availability and real-time data protection, stream replication is a good choice. If you need more flexible data synchronisation controls or are migrating databases between versions, logical replication will be more appropriate. For regular backup and recovery, using external tools such as pg-dump and pg-restore is a simple and effective option.

For monitoring purposes, this project chose Prometheus. Prometheus is an open source monitoring and alerting toolkit that is widely used to collect and process a variety of metrics, providing real-time alerts and data queries. Using Prometheus monitoring, developers can track and analyse application performance, resource usage, user activity and other key metrics in real time.

Because of the risk of personal accounts being compromised, we have taken a number of steps to prevent botnets: This is a feature called 'Botnet Detection', which is invoked every time a user logs in. It can query IP addresses in the database, and if a user has more than three IP addresses, or if a particular IP address is used by three or more users, the server will deny access.

E. Innovative Components

Innovation lies at the heart of our secure chat system, where we introduce a groundbreaking feature: the ability to send hidden messages that remain invisible until the recipient chooses to display them, with the added functionality of making them invisible again. This is an innovative way for users to communicate securely, providing control over the visibility of their messages. Users can share confidential information or engage in sensitive discussions without privacy concerns. Imagine this scenario: Bob is a lawyer who must securely send sensitive case details to his client. He is aware of the potential risks of discussing such matters in public spaces. Therefore, Bob will send a message detailing confidential legal information, and he will decide to hide this message. Hence, Bob ensures that the sensitive content remains hidden from prying eyes until his client explicitly chooses to unveil it. Meanwhile, across town, Alice discusses a financial subject with a colleague in a bustling café. Knowing the topic's sensitivity, she chooses to send a hidden message to ensure discretion. In both instances, the recipients receive a message without content but a button shaped like an eye. Pressing this button will display the content in the chat, and pressing this button again will hide the content. This innovative feature empowers users like Bob and Alice to communicate securely in any environment, ensuring confidentiality and peace of mind.

F. *Implementation*

1) Register: Users can register to the application from the user interface. They provide their username, email, and password and confirm their password. Then, the public and private keys will be generated and stored on the client side. Moreover, the password will be hashed using "SH256" and sent along with the username, email, and public key to the server to be stored in the database. If the user registers successfully, a window containing the private and public keys will appear on the screen where the user can download the keys as a text file. An inherent security measure in the registration process is that the password will be hashed before it is sent to the server; since the hash is a one-way function, meaning the real value cannot be retrieved, the passwords will remain safe in the event of a server or database attack. Also, users' private keys will be stored only on the client side to guarantee the user's privacy.
2) Login: Users can log in from the user interface. users should provide their email and password; inside the client side, the password will be hashed, and the user credentials will be sent to the server to be compared to the email and hashed password in the database. If the user credentials are not valid, they will not be able to enter the system. Otherwise, if the user credentials are valid, the server will take the user's IP address and check if the user has more than three IPs or if more than three users used the IP from the token records in the database. This process is performed to reduce the system's misuse and detect botnets. Once the user passes the check, the server will create two JWT tokens. The first token is an authentication token with a 30-minute expiration time, and the other is a refresh token with a seven-day expiration time. The server will store the token and the IP address as token records in the database. Several security measures are in place to maintain system integrity. Firstly, the user will be kicked out of the system once the token has expired, and the user should log in to use the system. Another security measure is detecting botnets; this process is critical to protecting users from abuse. Also. It's worth mentioning that OAuth 2.0 is utilised in the authentication process to ensure secure access to the system. The implementation of OAuth 2.0 adds an additional layer of security by allowing for standardised authorisation flows, enhancing interoperability and security across various applications and services.
3) Change password: Users can change their password by clicking on the 'change password' button on the login page. On the client side, the user will ask to provide an email, the old password and the new password; both passwords are hashed using the 'SH256' hash function, and then they will be sent to the server. The server will check the email validity. After that, if the old password matches the stored password, then the user password will be modified to the new password and saved in the database; otherwise, in the case of invalid data, the process will not be performed.
4) Manage keys: Users can upload their private and public keys as text files from the "Manage Keys" option, accessible through the dropdown list that appears when clicking on the profile photo. This feature allows users to access the system from different browsers or computers. Once uploaded, users can encrypt and decrypt messages using their keys, ensuring that they can maintain the confidentiality and integrity of their communications across multiple devices.
5) Display users: If the user logs in successfully, the "get users" API will be called to retrieve all usernames in the system. Afterwards, "get active users" will return the usernames of the online users; therefore, the users can know who is online to start a chat. As a security aspect, the APIs will receive a token from the user; if the token is not valid, the server will not respond. Hence, only authentic users can use the system.
6) Send messages: This operation starts when the user clicks on a certain user to be the recipient. The server will retrieve the recipient's username and public key. Subsequently, the chat history will be retrieved and displayed on the user interface if any. Moreover, the WebSocket connection between the user and the server will be initialised. If the user wants to send a message to the recipient, they will enter it and decide whether it is a hidden message or not by pressing the hidden message button. The message will be encrypted using "JSEncrypt:" one version using the recipient's public key, and the other version using the user's public key. The client side will send the message to the server. The

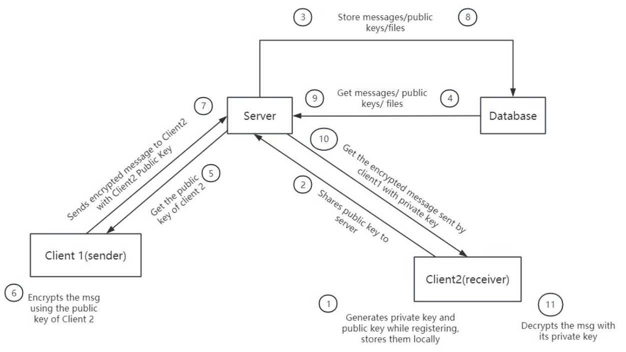

3
Store messages/public keys/files
8
Server
9
Get messages/ public keys/ files
4
Database
7
Sends encrypted message to Client2 with Client2 Public Key
5
Get the public key of client 2
2
Shares public key to server
10
Get the encrypted message sent by client1 with private key
Client 1(sender)
Client2(receiver)
6
Encrypts the msg using the public key of Client 2
1
Generates private key and public key while registering, stores them locally
11
Decrypts the msg with its private key

Fig. 3. Information transmission throughout the system

server will check the recipient's status, whether online or offline. If the recipient is online, the server will route the message to the recipient and store it in the database. For offline recipients, the messages will be stored in Redis and retrieved to the recipient once they are online. After the message arrives on the client side, it will be displayed in the user interface unless it's a hidden message. Hidden messages are only revealed when the user presses the 'eye' button within the message field. In sending messages, end-to-end encryption was used to guarantee the privacy of the chat, where only the sender and recipient can decrypt the message. Also, the encryption process occurs on the client side to increase security while transforming the messages through the network.

7) Delete messages: Each message in the chat has its own ID. If the user wants to delete a message, in the front end, the user will click on the delete option, and then the message ID will be sent to the server to delete the message from the database. This operation will be performed in real-time through web sockets; once the sender deletes the message, it will be removed from the recipient immediately.
8) Upload and download files: Users can upload files from the user interface, the user can upload a file, which will be encrypted using the recipient's public key and sent to the server. Once the server receives the file, an ID will be generated for the file to be used as an ID and a path to the file in the server. The ID will be stored along with the type, sender ID and path in a Files table in the database. The file's ID will be sent to the recipient using WebSocket, and the recipient will use the ID to request a download of the file to the server; when the user receives the encrypted file, they can decrypt the file using their private key. As a security aspect, the file name is randomised, so the file cannot be enumerated or brute-forced.
9) Logout: This API will activate if the user presses the 'log out' button. The user ID will be sent to the server to destroy the user's active token from the database. Also, the API will go through all the tokens in the tokens table in the database to check for expired tokens. If any, the system will forcibly log out these users with expired tokens and destroy their tokens from the database.

IV. Evaluation

Upon the project's completion, we conducted evaluations with following aspects: concurrency tests, overload tests, and error tests. Focusing on user registration, login, and message sending to evaluate the project through throughput, response time, latency, and other results, and based on the results of the test to improve the project. The testing was carried out in a virtual machine environment, with using JMeter tool to test the system, the system configurations are as follows: Ubuntu 22.04.4 run in the VirtualBox with 8GB of RAM and 3 CPU cores allocated.

A. Methodology

For user registration and login testing, we chose to use JMeter. First we created a CSV file with username, email, password and public key and registered them all to serve as 100 accounts for testing. After completing the registration, we can carry out the login test, here we tested the results of login 100 users and 200 users at the same time. First, we need to add a PerfMon Metrics Collector listener, which can be used to test CPU and memory response time; then, we add a thread group under the test plan, defining the number of users (number of threads), the number of loops, and so on. These are parameters that simulate the number of concurrent users and the frequency of their requests, e.g., in our system test, the login test for 100 users is set to 100 threads login at the same time and looping once, and for 200 users it is looping twice with 100 threads each time. Through testing, we can get throughput, response time, error rate, etc. to rate the system performance.

For concurrency testing of the WebSocket protocol, we conducted two types of tests: (1) two users sending a large number of messages to one user simultaneously, and (2) thousands of users sending messages concurrently. We employed different testing methods for these scenarios, utilizing the JMeter tool and WebSocket concurrency scripts in Python. For the first scenario, where two users send a high volume of messages to a single user, we used JMeter for testing. The test setup involved creating thread groups, configuring HTTP headers, establishing WebSocket connections, and controlling the number of iterations using loop controllers. We also incorporated WebSocket ping/pong checks to monitor connection status and closed the connections upon loop completion. The test results were evaluated based on metrics such as latency, throughput, and concurrent connections. In the second scenario, involving thousands of users sending messages simultaneously, we developed Python scripts to execute load tests on the FastAPI backend chat system. The scripts utilized thread pool to simulate multiple users establish a websocket connection with other users to conduct concurrent messaging tasks and established connections to the FastAPI server's WebSocket using randomly generated client IDs. The scripts measured the delay between receiving and sending messages, which was recorded for analysis. The main function of the script managed the concurrent messaging tasks, limiting the maximum number of consecutive connections to ensure system stability. The script calculated the average latency based on the collected data. These WebSocket evaluation tests aimed to assess the performance and scalability of the FastAPI backend chat system by simulating realistic usage scenarios. The tests provided valuable insights into identifying performance bottlenecks and determining the maximum number of concurrent users and messages that the system could efficiently handle.

B. Results

After thorough testing of the chat system, we have come up with an analytical report that demonstrates the system's performance in various aspects, followed by a detailed elaboration of the test results and an explanation of the results.

Firstly, there is the results of the concurrent registration test, we tested the throughput of the system per second when registering from 10 to 500 users at the same time. As can be seen from the Figure12, when the number of users is small (0 to about 100 users), the throughput increases significantly with the number of users; and as the number of users approaches 500, the throughput levels off.

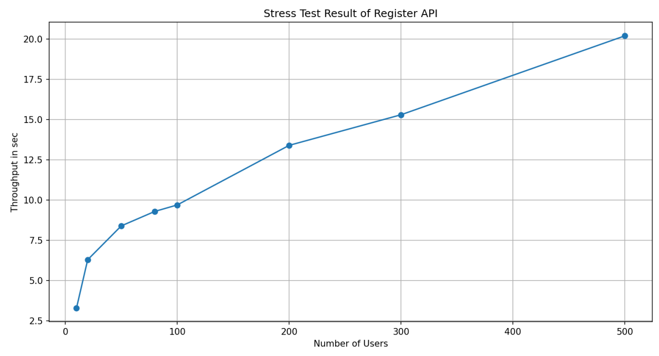


Fig. 4. stress test result of register API

The following shows the results of a concurrent test with 100 users login. First, an overall performance test report table is presented, which is shwon in Table I. This table includes data such as the number of requests being executed, the error rate, the average response time for all requests, and the system throughput. From the table we can see that with 100 concurrent user logins, the average response time is around 5 seconds, and the number of request failures is 0, which implies that the system is stable under the test conditions. The throughput rate was 10.47, indicating that approximately 10 requests were completed per second.

Moreover, the resources usage is also monitored during the processing, which is shown in Figure 5, recording CPU and memory usage separately, with the vertical axis showing the values of the performance metrics and the horizontal axis showing the data points for each second since the start of the

Requests		HTTP Request
Executions	Samples	100
	Fail	0
	Error%	0.00%
Response Times(ms)	Average	5764.26
	90th pct	9247.80
	99th pct	9539.55
Throughput	Transactions/s	10.47

TABLE I
OVERALL PERFORMANCE TEST RESULTS FOR 100 CONCURRENT USER LOGINS

test, in terms of runtime. From the graph, it suggests that the memory usage remained high throughout the test period, which is due to the high concurrency demand caused by 100 users login at the same time; the CPU usage fluctuates, indicating that the CPU load has different peaks at different points in time, which the fluctuating peaks is due to the fact that the system is processing the log-in requests, and falls back to the low point when waiting for the next batch of requests to be processed.

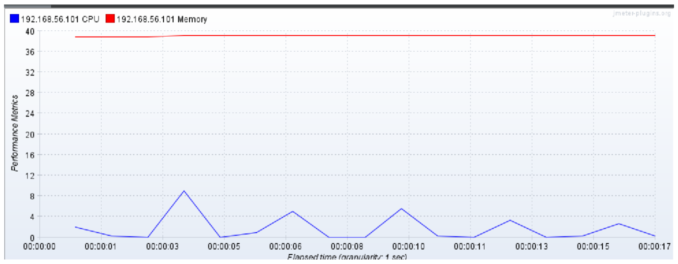


Fig. 5. CPU and memory usage

The third part of the evaluation focuses on the relationship between the number of requests per second and the median response time. In Figure 6, the horizontal axis represents the number of global requests per second, while the vertical axis represents the median response time. The data points on the graph indicate the median response time derived from the test at the given number of requests per second. As evident from Figure 6, the response time increases as the number of requests grows. When the number of requests is low, the system responds quickly, resulting in low response times. However, a significant increase in response time is observed when the number of requests falls between 6 and 8, indicating that the system begins to experience higher stress during this period. Interestingly, the maximum response time is recorded when the number of requests reaches 12, after which it decreases. This phenomenon is likely attributed to the variability inherent in the testing process.

To provide a comprehensive evaluation of the system's performance under stress conditions, we also conducted a test with 200 concurrent user logins. As Table II presents an overall performance test, it shows that when 200 users log in concurrently, 10 requests fail, which means that 5 percent of the requests are unsuccessful in concurrent situations, which can be attributed to either insufficient system resources or insufficient virtual machine resources in the current situation;

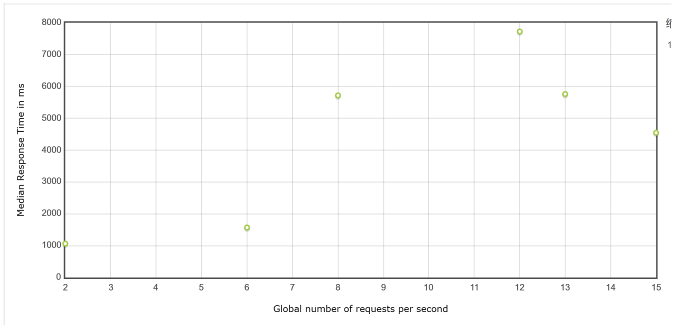


Fig. 6. Request and response times

requests		HTTP request
executions	samples	200
	fail	10
	error%	5.00%
response times(ms)	average	18755.62
	90th pct	38013.90
	95th pct	43372.57
throughput	transactions/s	4.552

TABLE II
PERFORMANCE TEST RESULTS FOR 200 CONCURRENT USER LOGINS

the average response time is about 18 seconds, which is relatively high, indicating that the system has a high latency in processing requests, and the system throughput is 2.49 times per second.

Figure 7 presents a graph illustrating CPU and memory performance metrics during the concurrent login test with 200 users, from which it can be seen that memory usage has remained at near-full levels during the test, indicating that the system memory resources was in a high-pressure state; CPU usage fluctuates, which is the same as when dealing with 100 users login concurrently.

The corresponding response time percentile graph is shown in Figure 8, which indicates what percent of requests were completed with their corresponding response time. There is a large jump in response time between 50 percent and 60 percent, showing that the system was starting to experience significant delays in processing requests in this segment; after this point, the response time increases steadily as the percentile increases.

To give a comprehensive performance, uploading files function was tested with 200 concurrent users, which overall results

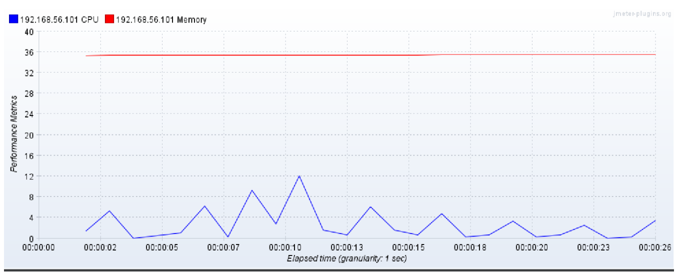


Fig. 7. CPU and memory usage (200 users)

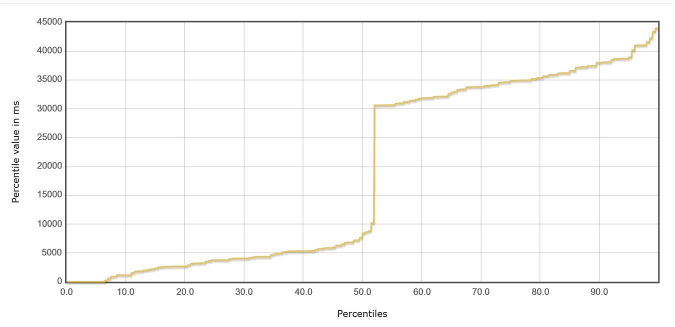


Fig. 8. response time percentiles

is shown in Table III. The table concludes that the system is able to complete most of the requests when the user uses the file upload function, but there is still a 10 percent failure rate with a long response time. We found that error rates has the similar results to the values in the login concurrent tests, leading us to conclude that inadequate memory causes these results.

request		HTTP request
executions	samples	200
	fail	20
	error%	10.00%
response times(ms)	average	18755.62
	90th pct	38013.90
	99th pct	43372.57
throughput	transactions/s	4.52

TABLE III
OVERALL PERFORMANCE TEST RESULTS FOR 200 CONCURRENT USERS UPLOADING FILES

Table IV presents the results of the WebSocket test, first focusing on the performance data of the WebSocket connection during the test period, including the open link and single write operations. The table summarizes the performance results for the WebSocket connection, suggesting that all 4 requests successfully open the link with an average response time of 8.5 milliseconds, but there are 649 single write failures out of 8000 operations were observed, with nearly zero response time, which can be caused by a extremely fast write operations or login an incorrect value.

The total number of samples executed during the test was 116,008, with an overall failure rate of 8.16%. The open connection operation had a 0% failure rate, while the single write operation had an 8.11% failure rate. In terms of response times, the average response time for the entire test was 9.95 milliseconds. The 90th percentile response time was 44 milliseconds, and the 99th percentile response time was 50 milliseconds. The open connection operation had an average response time of 8.50 milliseconds, with both the 90th and 99th percentile response times at 10 milliseconds. The single write operation had an average response time of 0.02 milliseconds, with the 90th percentile response time at 0 milliseconds and the 99th percentile response time at 1 millisecond. The throughput for the open connection operation was 400 transactions per second, while the throughput for the single write operation was 0 transactions per second.

request		total	open connection	single write sampler
executions	samples	116008	4	8000
	fail	1306	0	649
	error%	8.16%	0.00%	8.11%
response times(ms)	average	9.95	8.50	0.02
	90th pct	44.00	10.00	0
	99th pct	50.00	10.00	1
throughput	transactions/s	0.00	400	0

TABLE IV
PERFORMANCE TEST RESULT FOR WEBSOCKET

Besides testing with single user sends multiple messages with WebSocket, we also conduct multiple users send single message to give full view of latency and response time, which the result is displayed in Figure 9.

The following figure shows the relationship between latency and the number of connections when multiple users are sending a large number of messages at the same time. From the figure, we can see that when the number of concurrent connections is low (around 0 to 2000) the latency stays low and stable, indicating that the system does not significantly affect the performance when dealing with low loads; the latency fluctuates more after the number of concurrent connections exceeds 2000, implying that the system is under pressure to process the requests as more users are added; and the latency is at a higher level when the number of concurrent connections is close to 10,000 stabilises, indicating that the upper limit of the system's processing load may have been reached. The fluctuations are due to the thread pool in python script, we set the "`MAX_Connection`" hyper parameter so that the program will not crash, this would lead to the time waiting when the thread pool is fully occupied.

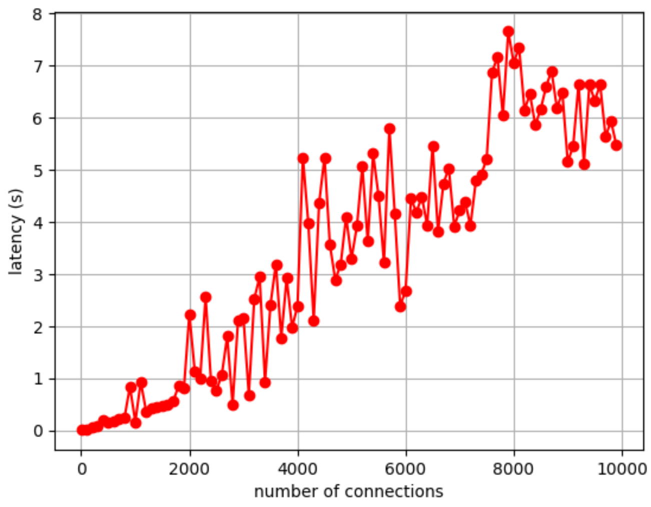


Fig. 9. latency and number of connection

C. Discussion

Overall we conducted system evaluation based on two aspects, which are basic http requests including registration, login, and file upload; Websocket requests including messaging. Measuring their concurrency tolerance, resources usage, error rate, and latency.

Regarding http based evaluation, the system demonstrated excellent performance under low to moderate concurrent user loads, exhibiting low latency and swift response times. This indicates that the system is well-optimized for handling typical usage scenarios, ensuring a smooth user experience. In the websocket based testing, it shows qualified performances on both multi-connections and multi-messages cases, in which the multi-connections experiment shows the resilience of the system cooperating with redis to handle large amount of connections, while the multi-messages experiment presents how efficient the server to pass the messages as the intermediary. These two complementary experiments greatly reflects the performances of real-time message communication, as it guarantees instant message delivery and robust tolerance on high concurrency.

However, the challenge in the evaluation process is that we have limited hardware resources to give the precise stress evaluation results on the software level. In cases, the system memory was fully occupied, so that affecting the results precision. In the future work we will focus on enhancing the hardware quality and deploy kubernetes cluster and a software-based load balancer to distribute the traffic to obtain lower the error rates and latency.

In conclusion, despite the challenges, the overall evaluation results of the secure chat system are promising in terms of the low latency and concurrency handling. It turns out all the functions inside the system are The system can handle the typical usage scenarios to present its security and resilience. In the future implementation, by integrating load balance technologies, continuously monitoring and optimizing the system, this chat application can obtain more robust and provide more privacy protection.

V. Conclusion

In this project, we designed and implemented a secure chat application based on security and resilience. By integrating security practices such as end-to-end encryption, JWT authentication, and botnet detection, the application satisfies the confidentiality, integrity, and user authenticity. Additionally, we also implemented message self-destruction and secure message persistent to provide more security options for users.

Focusing on the system resilience, besides the multi-session communication features, database auto replication and system monitor are implemented to prevent from data tampering and anomaly traffic incoming, making the system more resilient. Moreover, all the services we build are worked inside the docker, cooperating with each other using docker compose, which would be easily scaled up as the user increasing.

After implementation, we conducted precise evaluation in terms of security and resilience aspects. Despite some challenges on hardware resource limitations, the overall evaluation results are promising, indicating that system's ability is able to handle the typical business scenarios.

In conclusion, our secure chat system offers a range of secure features and resilient performances for users to socialize with others while mastering their privacy and keeping their conversations secure in daily communications.

REFERENCES

[1] C. Johansen, A. Mujaj, H. Arshad, and J. Noll, "The Snowden Phone: A Comparative Survey of Secure Instant Messaging Mobile Applications," Security and communication networks, vol. 2021, pp. 1–30, 2021, doi: 10.1155/2021/9965573

[2] S. Behera, A. Kanth, A. A. Suresh, CVR Ashwin, and J. R. Prathuri, "Chat Application Using Homomorphic Encryption," ITM Web of Conferences, vol. 50, p. 01011, 2022, doi: 10.1051/itmconf/20225001011

[3] S. Rathore, P. K. Sharma, V. Loia, Y.-S. Jeong, and J. H. Park, "Social network security: Issues, challenges, threats, and solutions," Information sciences, vol. 421, pp. 43–69, 2017, doi: 10.1016/j.ins.2017.08.063

[4] Adebayo, A. O., Oyindolapo, K., Chiemela, O., and Toluwalase, O. "Creation of a Secure Chat System", G.J. E.D.T.,Vol.3, no.1, pp. 62-71, 2014

[5] V. Suma, Z. Baig, S. Kolandapalayam Shanmugam, and P. Lorenz, "Unique Authentication System in End-To-End Encrypted Chat Application," in Inventive Systems and Control, Singapore: Springer, 2022, pp. 65–75. doi: 10.1007/978-981-19-1012-8_5

[6] Brewster, K., & Ruberg, B. (2020). SURVIVORS: Archiving the history of bulletin board systems and the AIDS crisis. First Monday, 25(10). https://doi.org/10.5210/fm.v25i10.10290

[7] Mayuuf, H. H., & Al-Ghizzy, M. J. D. (2022). Stylistic Features of Internet Relay Chat IRC as a Medium of Computer-Mediated Communication. International Journal of Linguistics, Literature and Translation, 5(12), 161–167. https://doi.org/10.32996/ijllt.2022.5.12.20

[8] Plantak Vukovac, D., Horvat, A., Čižmešija, A. (2021). Usability and User Experience of a Chat Application with Integrated Educational Chatbot Functionalities. In: Zaphiris, P., Ioannou, A. (eds) Learning and Collaboration Technologies: Games and Virtual Environments for Learning. HCII 2021. Lecture Notes in Computer Science(), vol 12785. Springer, Cham. https://doi.org/10.1007/978-3-030-77943-6_14

[9] Khetani, V. ., Gandhi, Y. ., Bhattacharya, S. ., Ajani, S. N. ., & Limkar, S. . (2023). Cross-Domain Analysis of ML and DL: Evaluating their Impact in Diverse Domains. International Journal of Intelligent Systems and Applications in Engineering, 11(7s), 253–262. Retrieved from https://www.ijisae.org/index.php/IJISAE/article/view/2951

[10] MobileHCI '13: Proceedings of the 15th international conference on Human-computer interaction with mobile devices and servicesAugust 2013Pages 352–361 https://doi.org/10.1145/2493190.2493225

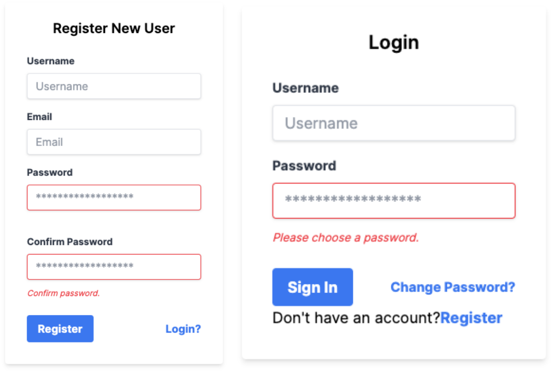
[11] Waleed.F. "DEVELOPING ACHAT SERVER",Notre Dame UniversityFaculty of Natural and Applied Sciences Department of computer Science, 2000.

[12] Singh, P., Agarwal, S. A self recoverable dual watermarking scheme for copyright protection and integrity verification. Multimed Tools Appl 76, 6389–6428 (2017). https://doi.org/10.1007/s11042-015-3198-9

[13] Sarkar, A., Singh, B.K. A review on performance,security and various biometric template protection schemes for biometric authentication systems. Multimed Tools Appl 79, 27721–27776 (2020). https://doi.org/10.1007/s11042-020-09197-7

APPENDIX

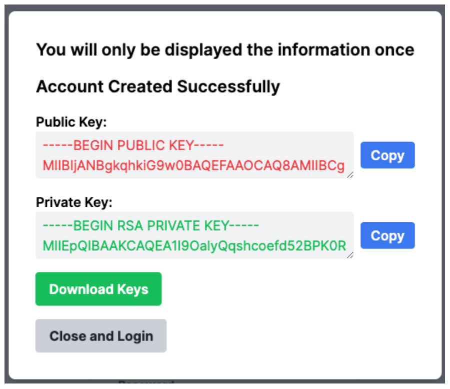
Name	Contribution
Jiashu Chen	13.4
Harshvardhan Patil	13.4
Mrunal Kale	12
Nayana Agrahara Dattatri	12.1
Wei Xie	12
Yihang Cai	12
Aaron Zhang	12.1
Haifa Alkhudair	13
Total	100



Confirm password.

Please choose a password.

Fig. 10. Register and Login Interface


You will only be displayed the information once

Account Created Successfully

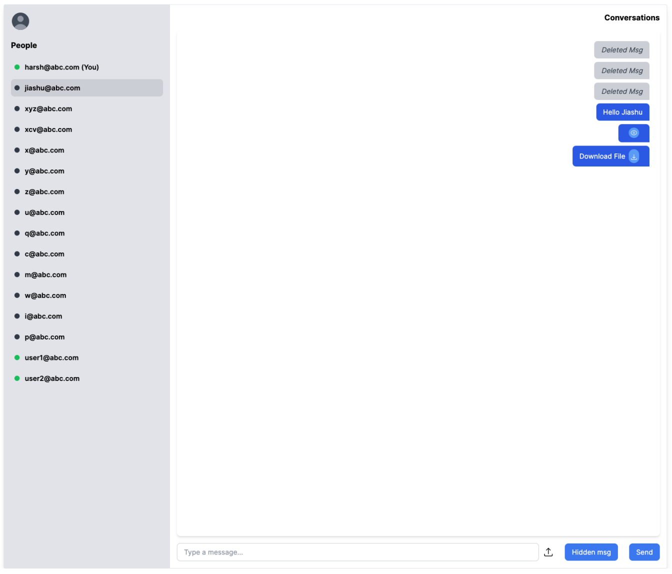
Public Key:

-----BEGIN PUBLIC KEY-----
MIIBIjANBgkqhkiG9w0BAQEFAAOCAQ8AMIIBCg

Private Key:

-----BEGIN RSA PRIVATE KEY-----
MIIEpQIBAAKCAQEA1l9OalyQqshcoefd52BPK0R

Fig. 11. Display keys Interface



harsh@abc.com (You)
jiashu@abc.com
xyz@abc.com
xcv@abc.com
x@abc.com
y@abc.com
z@abc.com
u@abc.com
q@abc.com
c@abc.com
m@abc.com
w@abc.com
i@abc.com
p@abc.com
user1@abc.com
user2@abc.com

Deleted Msg
Deleted Msg
Deleted Msg
Hello Jiashu

Type a message...

Fig. 12. Chat Interface