CSSE6400 - Software Architecture

Pigeon Messenger

(Source: Dall-E 3)

Team Members

Abdullah Badat (47022173)
Anshul Dadhwal (47386017)
Aryaman Tiwari (47540284)
Kaitlyn Lake (46426372)
Krisna Bou (46415983)
Kushagra Agrawal (47341360)
Sanchit Jain (47461688)

Table of Contents

Abstract

Encryption has become a quintessential requirement for any traffic travelling across the internet, and this takes higher precedence when it comes to private instant message (IM) applications. To this day, some instant messages are still sent over connections in plain text. Some platforms employ encryption on messages during transit, which is then decrypted and stored for it to be used in any manner by intermediaries like the server. However, as we shift to a decade where data privacy and user agency are paramount, most IM applications are shifting to a better-known industry standard - end-to-end encryption (E2EE). With E2EE, messages are encrypted by the sender and can only be decrypted by the receiver, which renders it impossible for any third-party user to gain access to the messaging.

This project develops an E2EE messenger called Pigeon. Pigeon involves two software systems: a client application that sends, receives, stores, encrypts and decrypts messages, and a server that routes encrypted messages between Pigeon clients. Pigeon has the non-functional requirements of reliability, security, scalability and availability. Security was considered the most important non-functional requirement. Both the server and client systems utilise a mostly monolithic, modular architecture to support the security and reliability of the system. While this architecture was the most appropriate to support the requirements of the Pigeon software landscape, it can also have negative impacts on the extensibility of the code. To test how well the system delivered the non-functional requirements, the following detailed testing plan was used. For security: Wireshark sniffing to demonstrate E2EE, Macaron to verify the security posture of software dependencies, unit tests for the client, and integration tests for the client and server. For reliability: GitHub actions automatic unit testing to ensure every push meets reliability standards, k6 load tests to check the reliability of message transfer during high volume traffic. For scalability: k6 load tests to supply high volume traffic and low volume traffic to scale up and then scale down the server. Finally, for availability: messaging over a long period and distance.

Changes

The proposal called for a variety of functional requirements and testing methods that the team decided to not implement for the Pigeon minimally viable product (MVP). As discussed in ADR3, the proposal specified that an MVP should include a minimal, clean and intuitive user interface. This interface was to include a registration page, login page, chat list page, contact page, chat page, and a settings and configurations page. Early on in the project, the team decided that in the interest of time, it would be wise to not implement a user interface as it does not directly contribute to the project's architecturally significant requirements of security, reliability, availability or scalability. This decision was made after work had already begun on the client-side code and the team's velocity was taken into account.

The most significant change was the change from a peer-to-peer (P2P) architecture to a client-server architecture, as described in ADR1. This change was made as a peer-to-peer architecture would be difficult to implement and may cause the team to run out of time for documentation, testing, video, etc. The pros and cons of this change in architecture will be discussed in the next section. The suggested tools to accomplish the P2P, E2EE system in the proposal, Tox and BitMessage, were not used as a result.

The proposal called for the encryption of the database on the client side, but the team decided that this was not an important feature given the time constraint. Additionally, encrypting the database would require user authentication which was not deemed critical or feasible for an MVP.

The proposal specified UI/UX user tests, spoofing tests and man-in-the-middle attack tests, but the team decided to remove these tests from the final test plan. As the graphical user interface requirement for the system was removed and significance was taken off of the frontend development due to the aforementioned reasons, the team decided that there was less reason for UI/UX user tests. Additionally, usability was not an architecturally significant requirement of this system, so UI/UX testing was not essential. The team decided to remove the spoofing and man-in-the-middle attack tests as they were deemed out of scope for the project. The main aspect of security that Pigeon emphasised was the E2EE and so the team focused on testing that.

Design Options & Trade-Offs

Software Landscape

Pigeon was initially meant to follow a P2P architecture that would allow for a more secure architecture. P2P architecture is ideal for reducing latency as the architecture involves direct communication between the users. Additionally, a client-server architecture is centralised making the server a single point of failure for the system, as opposed to that of a P2P architecture that is decentralised and has no single point of failure. Another tradeoff that comes with using a client-server architecture is that the overall performance takes a hit in extreme cases where multiple concurrent users are using the application at the same time. This is because bandwidth is handled mainly by the central server.

Unfortunately, though, purely P2P instant messaging technology is still highly experimental and is not widely documented or supported. Given the time constraints of the project, the likelihood of having a functional MVP that met all the significant functional and non-functional requirements and implemented P2P was deemed too low to risk implementing, so the well-documented and supported server-client architecture was chosen instead. A server-client implementation also has the benefit of being more maintainable and thus secure, as highly experimental technologies and toolsets like Tox and Bitmessage are untested and not well understood, increasing the risk of programming errors that could lead to security vulnerabilities.

The project called for secure, low-latency, instant messaging. The team had two options for connecting the client and server, long polling and web sockets, each with its pros and cons. Web sockets create a bidirectional communication channel between the client and server. One web socket connection would be able to be used for the entire client session as they can persist as long as necessary. In this way, they are more efficient for possibly long-lasting connections. Additionally, websockets are very low-latency which makes them suitable for real-time applications like instant messaging. Furthermore, web sockets scale efficiently which is an architecturally significant requirement of the system.

However, web sockets can be more complex to implement than long polling, especially since our team only had experience with HTTP-oriented connections like long polling. Long polling can accomplish near-real-time connection between the client and server by keeping the HTTP connection open until data is available. In this way, there is no continuous polling. Long polling is less scalable and performant than web sockets due to the open connections requiring more memory and CPU overhead. The team decided to forgo the scalability, performance, and efficiency benefits of web sockets to benefit

from the simplicity of implementing long polling as described in ADR2. This may have reduced the system's ability to deliver the architecturally significant requirement of scalability, however, the primary architecturally significant requirement of security was not significantly affected.

Client

A monolith architectural design pattern was chosen for the pigeon client, as it was deemed to achieve the best balance of security and usability, and the application is too small to require more than one or two deployable components. A monolithic architecture is easier to deploy because there is only one software component that requires set-up and integration with the system. Additionally, with a monolith, all dependencies, code and assets can be packaged and delivered to the target platform. This makes the application easier to use and thus supports the ASR of availability. A monolith can also increase security because all dataflows and assets are located internally and thus there are fewer data flows and a smaller attack surface exposed to an attacker. On the other hand, a monolith can reduce maintainability if it is not well defined and structured internally, as the internal logic and data flows are less understandable for a developer. This can potentially detract from security as developers are more likely to make mistakes that can lead to vulnerabilities.

Since maintainability is closely linked with security, the maintainability of the code was considered when designing the architecture. A potential internal structure for the monolith was an event-driven architecture, where frontend CLI actions performed by the user create an event. This can both reduce maintainability in the long term and support it in the short term. Coupling between internal components is reduced making functionality easier to add. However, this can lead to code becoming less readable and more difficult to debug, as internal dataflows are more difficult to track. This design pattern was not implemented because there are only a small number of simple components and extensibility is not an ASR, so it would have complicated the code more than it would have helped.

In terms of security, we have delivered on the encryption method mentioned in the initial proposal for using a powerful encryption method like 4096-bit RSA. However, we could have done a combination of asymmetric and symmetric encryption like RSA and AES to allow for faster encryption speeds with large messages and media files. The current implementation only suffices for small text messages. This is suitable for the current implementation, as support for large messages and media files is not a functional requirement for an MVP.

Server

The team chose a simple monolithic architecture running on an AWS ECS for the server but considered a variety of alternatives including a service-oriented architecture (SOA), a microservices architecture, a serverless architecture and an event-driven architecture (EDA). This decision was made to ensure we could deliver all our non-functional requirements and functional requirements on time. Monolithic architectures tend to lead to higher coupling than the microservices architecture or SOA or EDA as it doesn't separate functionality, however as the server logic was very simple, coupling was not a significant issue. Additionally, a monolithic architecture was more appropriate than a microservices architecture or SOA as the server was composed of very few separate services, so there was no benefit to deploying them out into different containers. Separating the services would increase the deployment overhead, increase the attack surface of the server - decrease security, needlessly increase the complexity of the server, and it would also decrease performance as the services would have to communicate over a web API as opposed to running in the same container. This ruled out the microservices architectures and SOA. The team decided that an EDA was not appropriate for instant messaging as there are no asynchronous or high response time processes that need to be run. If the system were to be extended and the system had to deal with uploading and processing large files, then some EOA influence would be useful.

A serverless architecture was also considered as it would simplify the deployment and management aspects of the server, but as we had the requirement of availability, we needed an architecture that would be able to run indefinitely. AWS lambda limits long-running processes and would not be able to support the availability requirement like an ECS would.

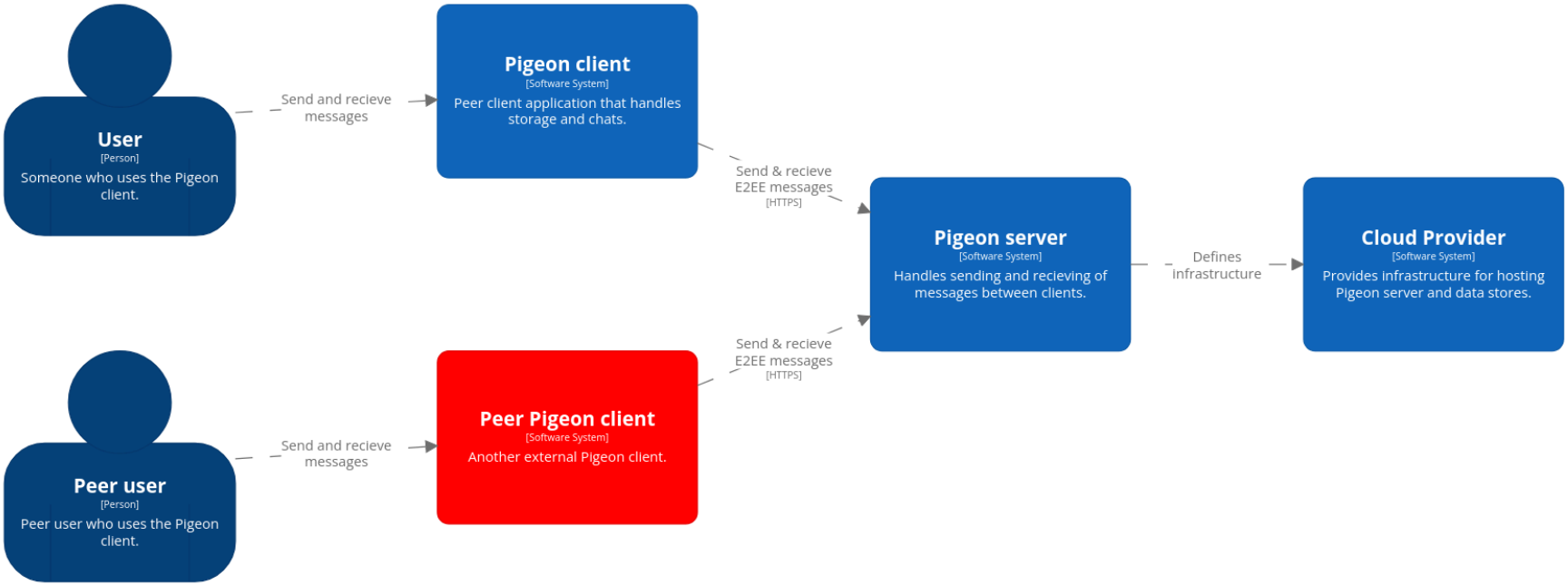
Monolithic architectures tend to be less extensible. However, as this project was just delivering an MVP (and was not intended to be used to develop a final release) and extensibility was not a non-functional requirement, this trade-off was accepted. Additionally, as the architecture did not prioritise extensibility, we did not implement any plugin interface, as you would see in a microkernel or plugin architecture.

The server needed a storage mechanism to hold the messages that had been sent by a sender for the recipient to collect. We decided on a database but also considered an in-memory solution like Redis. While an in-memory cache like Redis offers low latency and high-speed access which is important for instant messaging, it lacks the persistence and reliability of a database. Databases ensure the data is permanently stored and provide features like ACID compliance which ensure the robustness of our

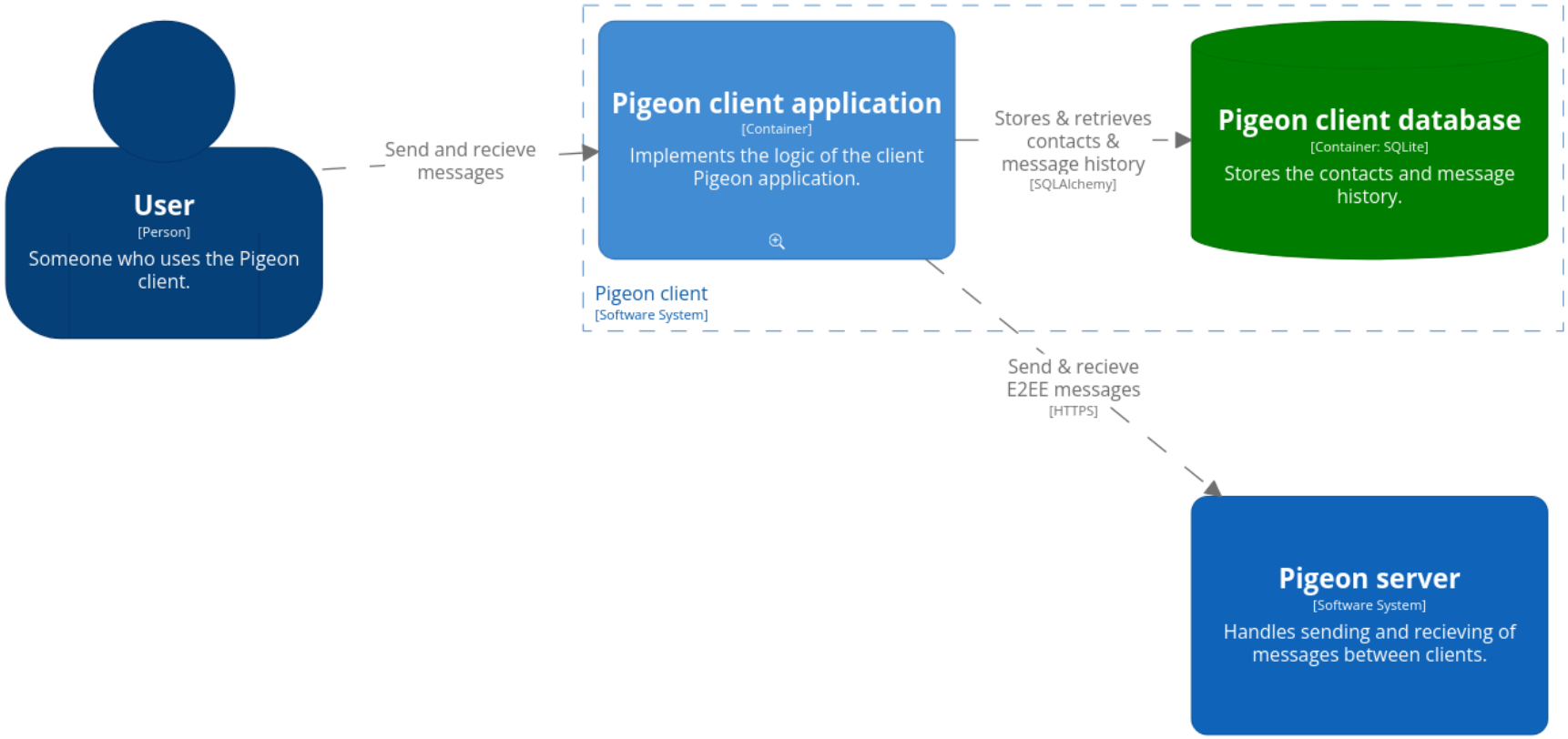
data. Even if the server goes down, the data will persist, unlike in an in-memory solution. Databases also have better security. However, they come with higher latency and complexity. We chose a database, because we prioritised reliability, availability, scalability, and security over speed, ensuring that messages are securely stored and retrievable even in the event of server restarts or failures. This approach helped us meet our non-functional requirements. We used PostgreSQL for our database over SQLite due to its support for scalability and robust transactional support, which are important for an instant messaging service.

Architecture Description

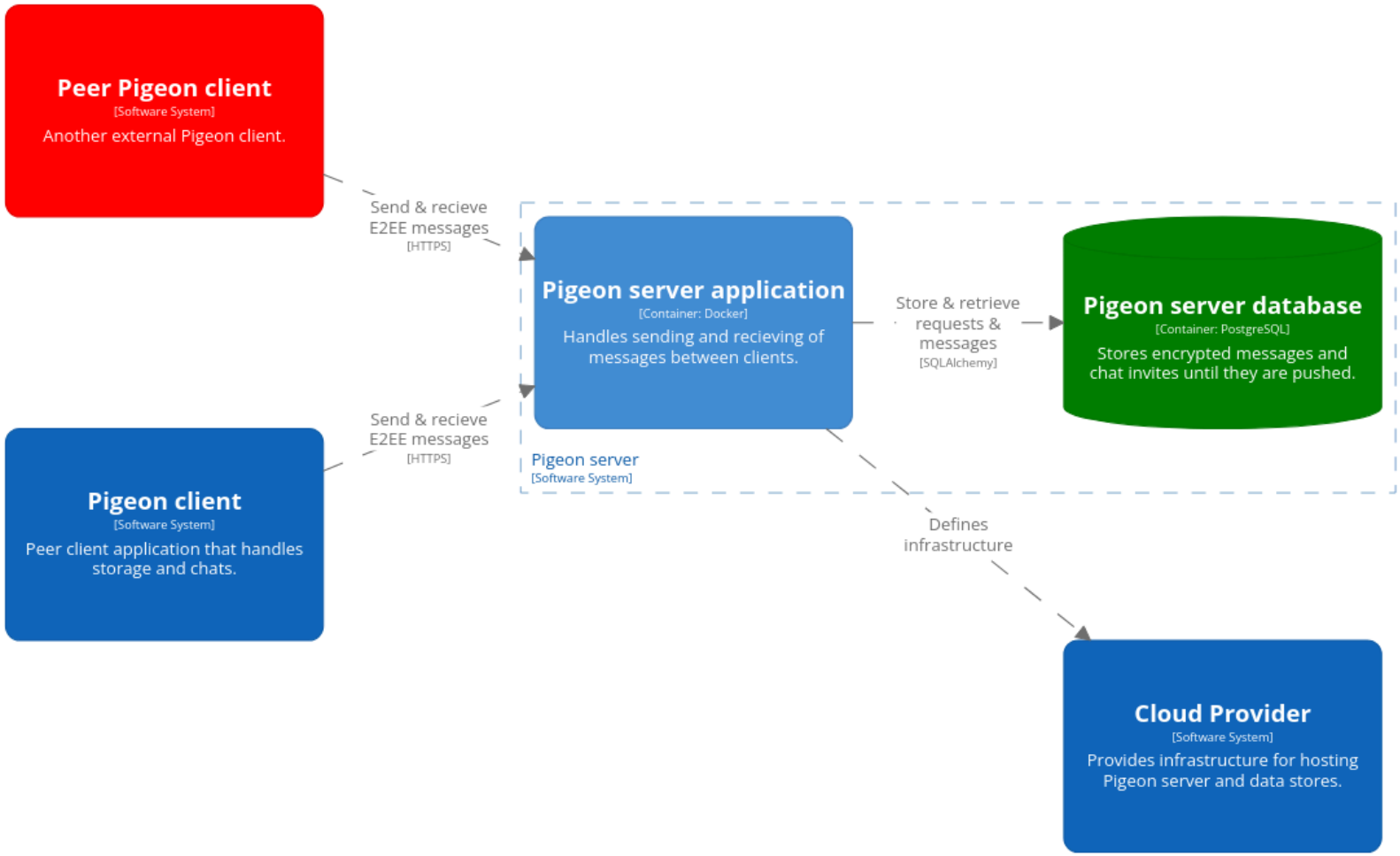
Context Diagram



Container Diagram

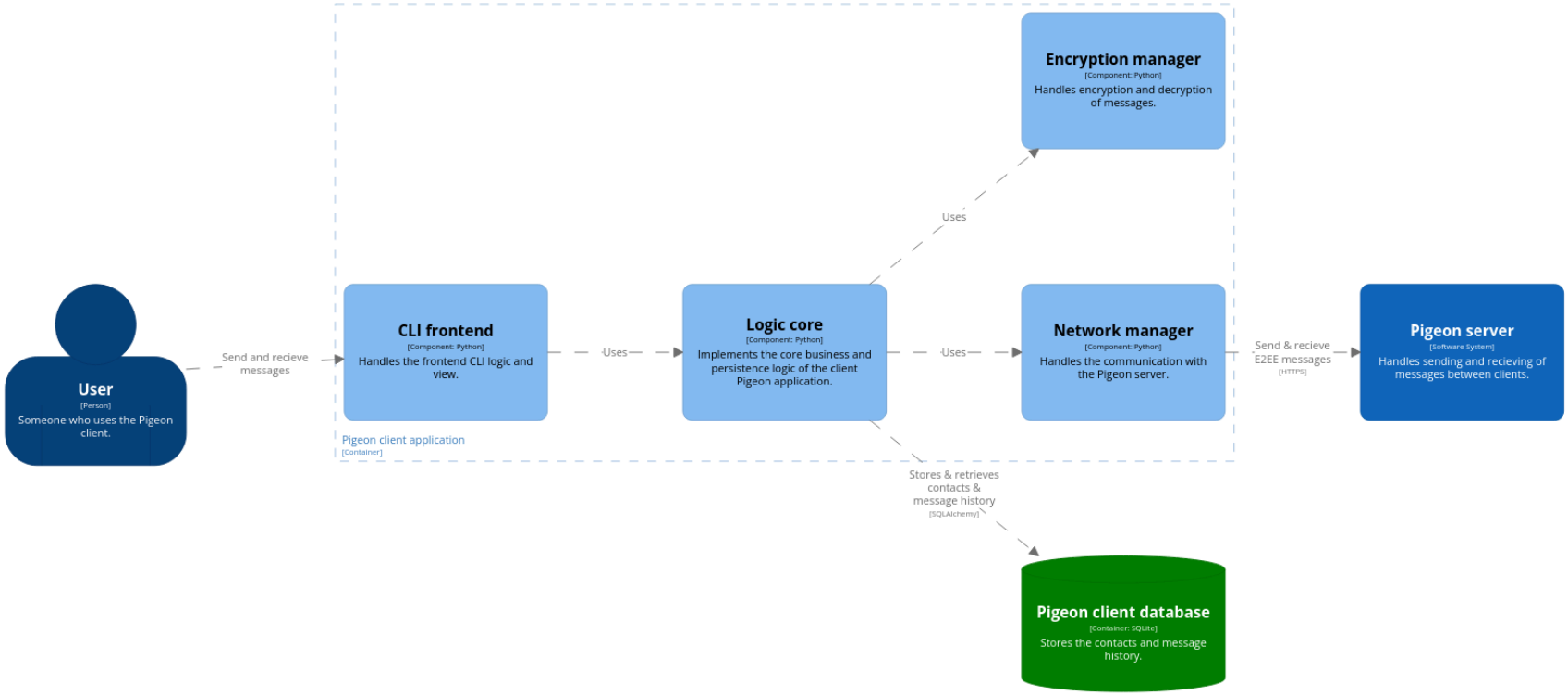


Pigeon client container diagram

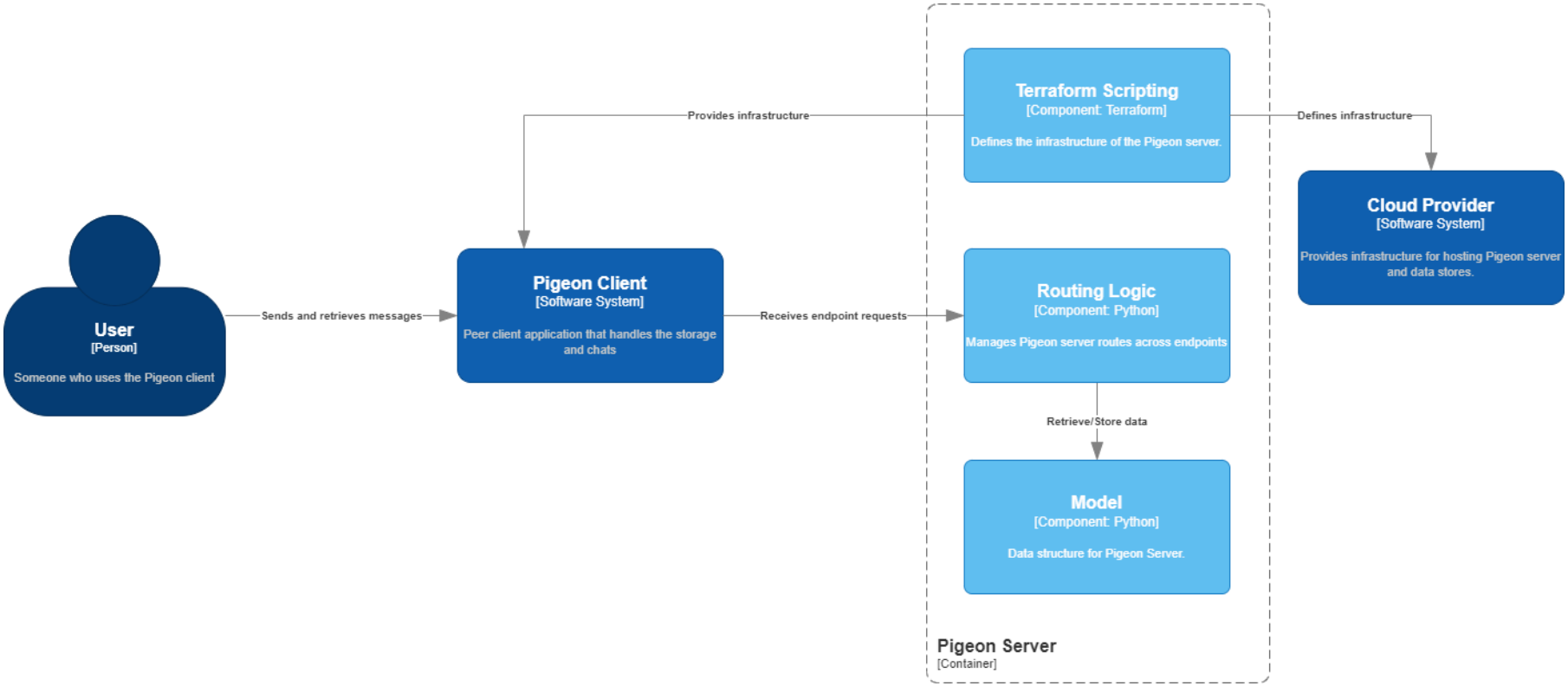


Pigeon server container diagram

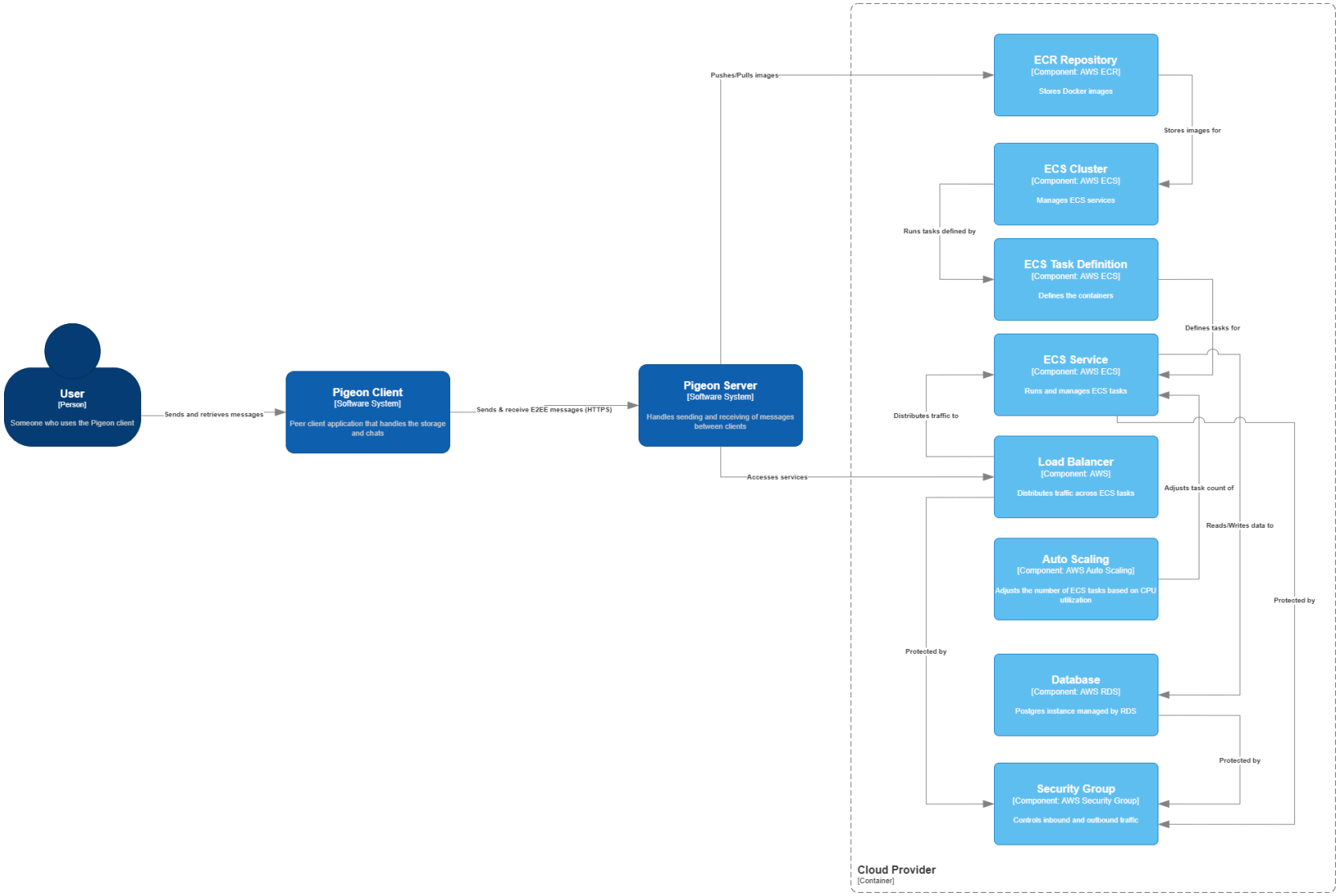
Component Diagram



Pigeon client component diagram



Pigeon server component diagram



Pigeon Cloud provider component diagram

Pigeon Client Description

The pigeon client is a single software system that consists of two containers, the pigeon client application itself developed in Python and the SQLite pigeon client database. The latter stores contact information, message histories, and encryption keys. The former consists of four main logical components: the CLI frontend which the user uses to control the application; the logical core which manages the jobs of networking, encryption and data persistence logic; the encryption manager which performs all security operations; and the network manager, which communicates with the pigeon server to send and receive messages and friend invitations. The pigeon client application and database are deployed in a single docker container to enhance ease of use. Thus, the pigeon client is a monolithic-like architecture with a defined internal structure that separates functionality into logical components.

Pigeon Server Description

The pigeon server consists of three main components including routing logic, model, and terraform scripting. The pigeon client sends an endpoint request to the routing logic component of the architecture. Here, the routing logic based on the endpoint request routes the request to retrieve and/or update data by interacting with the model which stores the data structure of the Pigeon server. However, for all this to work on the cloud platform, we need the terraform scripting component to define the infrastructure needed to perform these actions on the cloud platform i.e., AWS. Once the infrastructure is set, we have the ECS instances, databases, security groups, and the image available and running on the AWS account from where the user can access all the endpoints and receive the messages. It includes a load balancer at the front which points to an ECS in an autoscaling group. The ECS is connected to an RDS running PostgreSQL. The ECS has one service which pulls tasks uploaded to an ECR.

Critique & Evaluation

Pigeon has four main architecturally significant requirements (ASRs) for it to be a successful application. They are as follows:

- Security
- Reliability
- Availability
- Scalability

The monolith architecture of the client and the server aids in security because there are fewer individually deployed containers, so less exposed data flows between them. The only data flow from the client side is interaction with the server API over the internet. This reduces the attack surface of the system. On the other hand, the monolith architecture makes the code less maintainable because data flows between internal components are harder to trace, and there is a higher risk of coupling. This means there is a higher risk of software bugs that can lead to vulnerabilities or errors, potentially detracting from security and reliability. Additionally, the monolith design of the client allows the client to be packaged and deployed in a single Docker container, making it easy to use, set up, and accessible on a wide range of platforms, supporting availability.

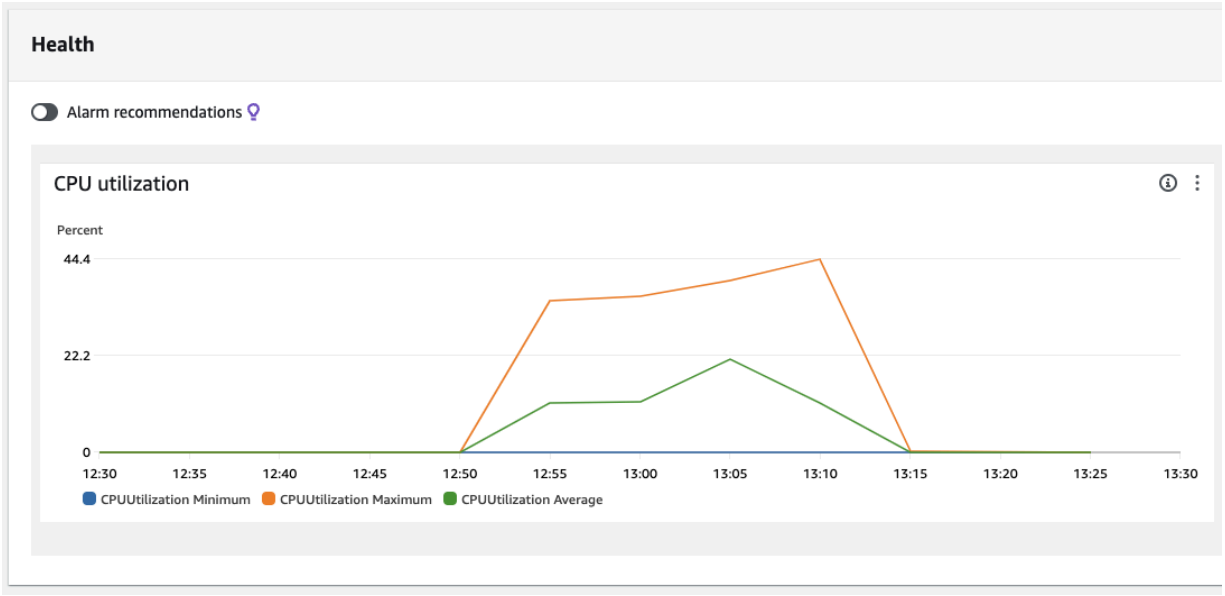
Macaron was used to verify the supply-chain level security of the software as described in the Evaluation section. Two out of ten tests are being passed, which implies more could be done to support the supply-chain level software security. A perfect Macaron result is not expected for an MVP, but to improve this aspect of security, the results suggest the development pipeline could be established in more depth and the build definitions and configurations could be verified.

The client and server further deliver on the functional and non-functional requirements for security and privacy by storing the minimal amount of information about a user and their actions as possible. The only identifying information for a user the client stores are those necessary for functionality: a UUIDv4 string to identify the user, their list of contacts, and their message history. The server only stores messages and message requests in transit; as soon as the destined user pulls the delivered messages and requests, the server deletes them from its database. Thus, the system is secure and private.

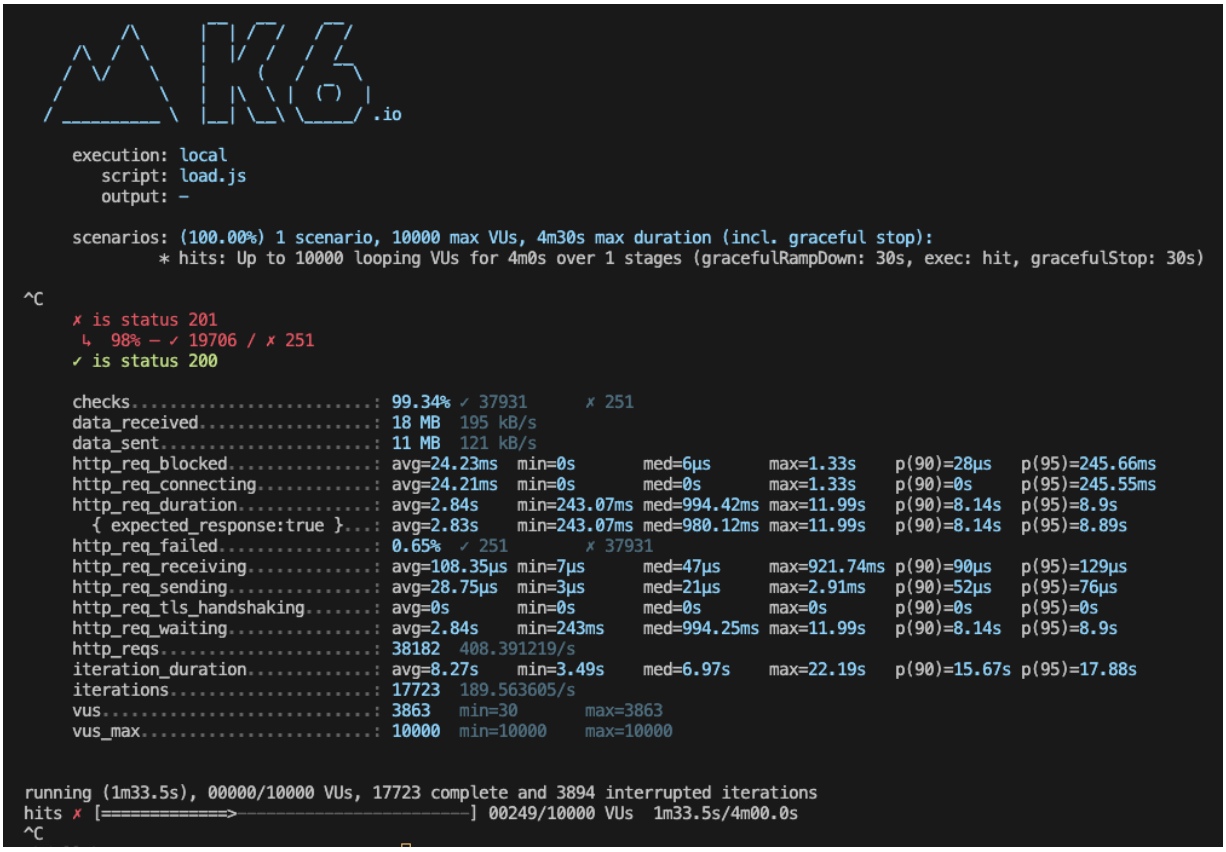
Since the client and server are implemented as separate monolithic software systems, the architecture lets the server be deployed and operate independently from the client.

The server also only has a deployed database and application logic container, which allows the server to scale easily to different loads while effectively and safely reading and writing from the database. Load testing using Javascript k6 was conducted on the server to verify this. As seen below, the CPU utilisation increases as the load increases. It crosses the 15% autoscaling benchmark after 12:50 and then increases the number of server instances running until a maximum of 5 are up as seen below. As seen in the results of the k6 load testing below, this allows the server to maintain 98% successful connections. Thus, the design of the server supports both scalability and availability.

Maximum number of server instances running under k6 load testing



CPU utilisation under k6 load testing



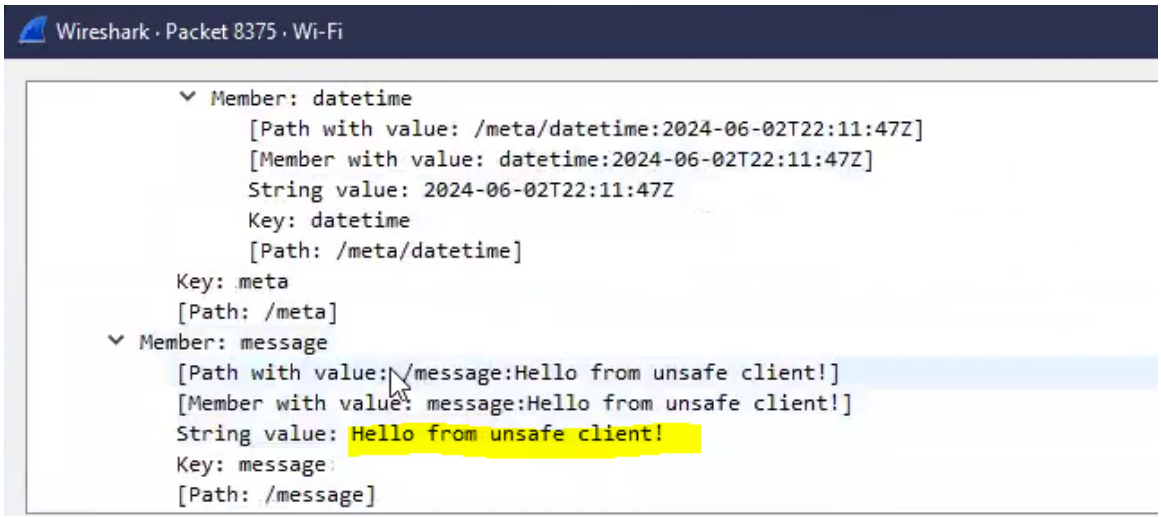
k6 load testing results

The unit tests demonstrate the client system is robust and reliable. Reliability requires that the software system not be error-prone and maintain the security, privacy, and integrity of users and their messages. The client system is critical to ensure reliability, as it handles all encryption storage and executes user commands. As such, the client system was fully tested, including all internal components and functions and interaction with the server where they were feasible to test within the time frame with unit tests. These tests are described in the Evaluation section. As seen, all unit tests pass, showing that the software is functional and well-integrated, proving the reliability of the client and server. Manual integration evaluations were also conducted using the manual test plan described in the evaluation, which was likewise successful.

Long polling was used to interface with the server API. We could have used web sockets instead of long polling to connect the client and the server on the pigeon application for better performance. However, significant server loads would be required for the performance difference of long polling versus web sockets to be perceived, and these kinds of loads were not expected on an MVP. Performance optimisation can constitute future work.

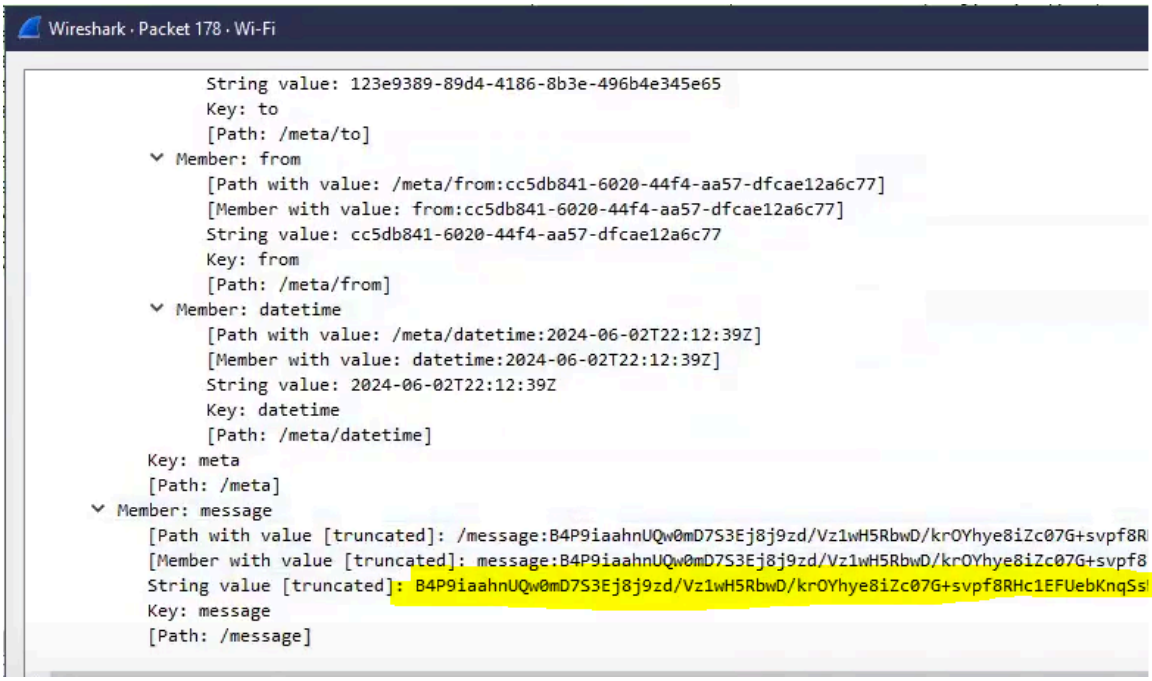
With the core feature of encryption of messages between users, primary testing of this was done in the form of unit tests and Wireshark. Wireshark is an open-source packet analyzer software tool that allows us to view various data across networks. It also allows us to analyse how the network performs and helps diagnose any abnormalities in the platform's behaviour, i.e., connectivity issues, packet losses, and slower-than-usual performance. Additionally, using the analysis from Wireshark, we can make informed decisions for improving the existing deployed system of Pigeon.

Wireshark allows us to filter through through specific packets. Our implementation of long polling through HTTP GET and POST requests can be seen in the image below:



Unencrypted message

With the encryption disabled, and messages being sent insecurely across the network we can see the JSON contents of the message sent across clients. However, with the encryption in place, sending the message again produces a more complex and unreadable message to decipher:



Encrypted message

With this, we can ensure that outside entities cannot intercept messages between clients, and ensure security for clients.

Test plan

For testing our primary ASRs, we decided to do the following:

- Unit tests
- Macaron
- Wireshark
- K6 Load Testing

For testing our scalability, we used k6 load testing. K6 load testing provides a very efficient simulation of how well the pigeon scales when used by concurrent users. This is especially important for understanding the performance and any points of failure when placed under heavy loads. Additionally, since k6 load testing is written in JavaScript, it gives us a lot of flexibility in the future to make API calls or handle cookies with ease.

We used Macaron to check the software supply-chain levels. We created a clone of the repository to run and see the SLSA levels of the project. This is especially important given the rise of supply-chain attacks in recent times where the malicious code is injected in the form of compromised packages or builds that lead to the bad actors gaining access to sensitive information. Currently, we are only passing the basic checks of Macaron which are as follows:

mcn_build_script_1	Check if the target repo has a valid build script.	• Scripted Build - SLSA Level 1
mcn_version_control_system_1	Check whether the target repo uses a version control system.	• Version controlled - SLSA Level 2

(Source: pigeon.html)

This, however, is a part that should be taken care of in the future if we plan on making the project secure and robust. Some of the future works involved in achieving this would be to produce provenance of our artefacts that would later be published on a package registry. The full Macaron report can be found in the Pigeon repository.

An automated unit and integration test suite were developed to verify the reliability of the client and server-client interaction. Instructions on how to manually run these automated tests can be found in the Pigeon repository, and the automated unit tests

were run as part of a GitHub Actions Workflow. These were run with every push and merged pull request to the remote repository. As seen in the test suites described below, all internal components of the client architecture were tested except the CLI front end. Tests for it were not implemented due to it being the least important component and because there was not enough time to implement automated tests for it. Instead, the CLI was manually tested through a test plan.

Unit test	Component tested	Description	Status
test_valid_uid	Network manager	The send_message can recognise valid UUIDs	PASS
test_invalid_uid	Network manager	The send_message can recognise invalid UUIDs	PASS
test_valid_json	Network manager	The send_message can recognise valid JSON data	PASS
test_add_contact	Logic core	The client can save contacts and their public keys persistently	PASS
test_send_encryption	Logic core, Encryption manager	The client can send encrypted messages and decrypt them on receipt	PASS
test_view_contacts	Logic core	The client can retrieve stored contacts	PASS
test_add_contact	Logic core	The client can save contacts and their public keys persistently	PASS
test_select_contact	Logic core	The client can retrieve chat history in order of messages sent and received	PASS
test_invite_not_sent	Logic core	The logic core does not send an invite	PASS

Unit tests

Integration test	Description	Status
test_recieve_message	The network manager can send and receive messages	PASS
test_send_invite	The network manager and encryption manager can send and receive message requests	PASS
test_invite_sent	The logic core, network manager and encryption manager can send and receive message requests	PASS
test_invite_accept	The logic core, network manager and encryption manager can invite and accept contact requests	PASS

Integration tests

A full automated integration test of the client and server was not possible to implement given the time constraints. This instead can and was tested manually using a detailed integration test plan:

1. Deploy the Pigeon server to the cloud provider of choice.
2. Run two client applications with >= 40km of physical distance between them.
3. One client adds another client to be a contact.
4. Once both clients have been registered as contacts, >= 5 messages should be sent by either user in random order.

If all steps in the manual evaluation are successful, it implies that the client and server interaction is fully functional.

As mentioned in the previous section, Wireshark tests were done to simulate external entities' capabilities to intercept messages between clients. However, with encryptions in place, we could see the encryption hashing in place of the message, ensuring the message's security and secrecy between two clients.

Reflection

This section outlines the key lessons learned and potential improvements for future projects based on our experiences throughout the project and developing software.

Monolithic Software Systems

Maintaining and developing monolithic software systems was more difficult than anticipated due to the higher coupling between internal components. This tight coupling caused inefficiencies as team members had to wait for others to complete their code before proceeding with their tasks. In future projects, we would address this by modularizing the internal architecture more effectively. This approach would reduce dependencies between components, allowing team members to work more independently and improving overall productivity.

Designing Architecture

Designing the architecture for an application proved challenging because we did not fully understand the technologies that made it possible, such as encryption techniques and client-side API interactions. To mitigate this, we developed architecture diagrams early in the process and kept the system as simple as possible. This strategy helped us achieve an MVP within the project's narrow time frame. Moving forward, we would invest more time in learning and understanding the key technologies before starting the design phase, leading to a more robust and efficient architecture.

Automating Integration Testing

Automating integration testing was another hurdle we faced. Initially, we explored creating a GitHub Actions workflow similar to our cloud infrastructure testing suite. However, the complexity of managing the AWS environment and Docker containers led us to choose a manual testing plan to demonstrate integrated functionality. We discovered that establishing a Continuous Integration (CI) pipeline in the beginning would have allowed us to verify and test our code regularly, and efficiently, increasing productivity and reducing the likelihood of errors. In future projects, we would prioritise setting up automated testing and CI pipelines to enhance our development workflow and ensure higher code quality.

Time Management

Developing this software system within a month posed a significant challenge of managing our contributing time since our team members had assignments from other courses as well throughout this time frame. However, our collaborative spirit and effective communication through Discord servers allowed us to overcome these constraints. Regular meetings, division of tasks, and problem-solving helped us to stay on track and deliver an MVP despite the tight schedule. In future projects, we would continue to leverage these strategies, ensuring that time management remains a strength even under demanding circumstances.