MSCS-632: Residency Project: Deliverable 2 Report: Simple Chat Application: Core Functionality Implementation

---

Shabnam Shaikh, Sakchham Sangroula, Sandesh Pokharel, Nihar Turumelle, Romika Souda

University of the Cumberlands

MSCS-632 Advanced Programming Languages

Dr. Vanessa Cooper

April 05, 2025

1. Introduction

This document outlines the **core functionality implementation** for our chat application in two languages: **Rust** and **Go**. Both implementations are functional, meet the assignment's key requirements, and make use of language-specific features.

The purpose of this deliverable is to highlight how each language handles the same problem differently — from design to execution — using appropriate data structures, concurrency models, and memory management tools available in each ecosystem.

GitHub Repo: https://github.com/sanspokharel26677/MSCS-632-Residency-Project

2. Rust Implementation

Our Rust version is a **turn-based chat system** that allows users to take turns sending messages. The application uses a combination of structs, a message queue, and the tokio runtime for async behavior.

Structs and Memory Safety

Rust relies heavily on strong typing and memory safety. We defined a Message struct and used VecDeque to store chat history. Shared access is managed using Arc<Mutex<_>>.

```
struct Message {
    user_id: String,
    content: String,
    timestamp: String,
```

}

struct ChatServer {

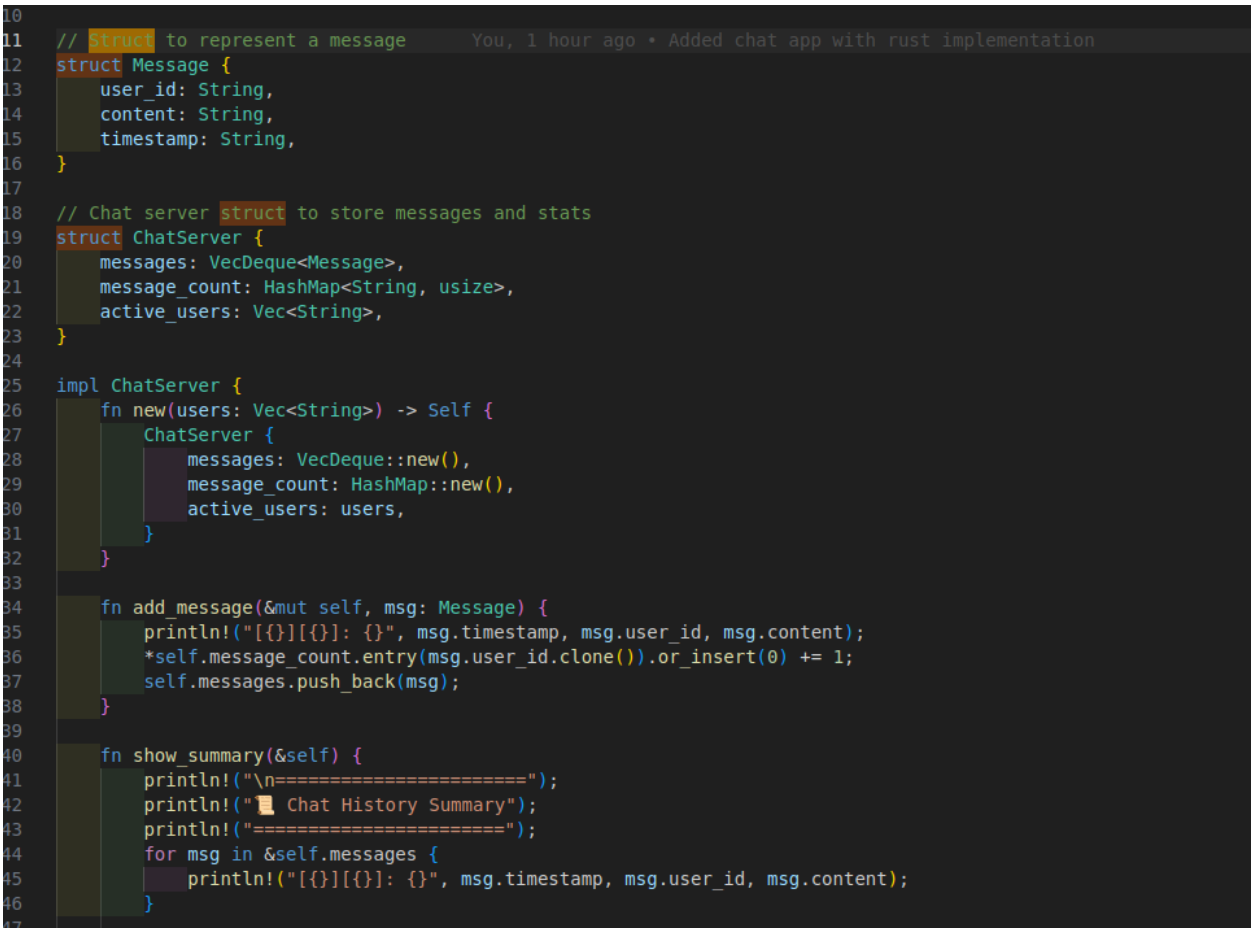
    messages: VecDeque<Message>,

    message_count: HashMap<String, usize>,

    active_users: Vec<String>,

}

📸 **Screenshot :** main.rs struct definitions and shared state initialization.



```rust

// Struct to represent a message      You, 1 hour ago • Added chat app with rust implementation
struct Message {
    user_id: String,
    content: String,
    timestamp: String,
}

// Chat server struct to store messages and stats
struct ChatServer {
    messages: VecDeque<Message>,
    message_count: HashMap<String, usize>,
    active_users: Vec<String>,
}

impl ChatServer {
    fn new(users: Vec<String>) -> Self {
        ChatServer {
            messages: VecDeque::new(),
            message_count: HashMap::new(),
            active_users: users,
        }
    }

    fn add_message(&mut self, msg: Message) {
        println!("[{}][{}]: {}", msg.timestamp, msg.user_id, msg.content);
        *self.message_count.entry(msg.user_id.clone()).or_insert(0) += 1;
        self.messages.push_back(msg);
    }

    fn show_summary(&self) {
        println!("\n======================");
        println!("📜 Chat History Summary");
        println!("======================");
        for msg in &self.messages {
            println!("[{}][{}]: {}", msg.timestamp, msg.user_id, msg.content);
        }
```

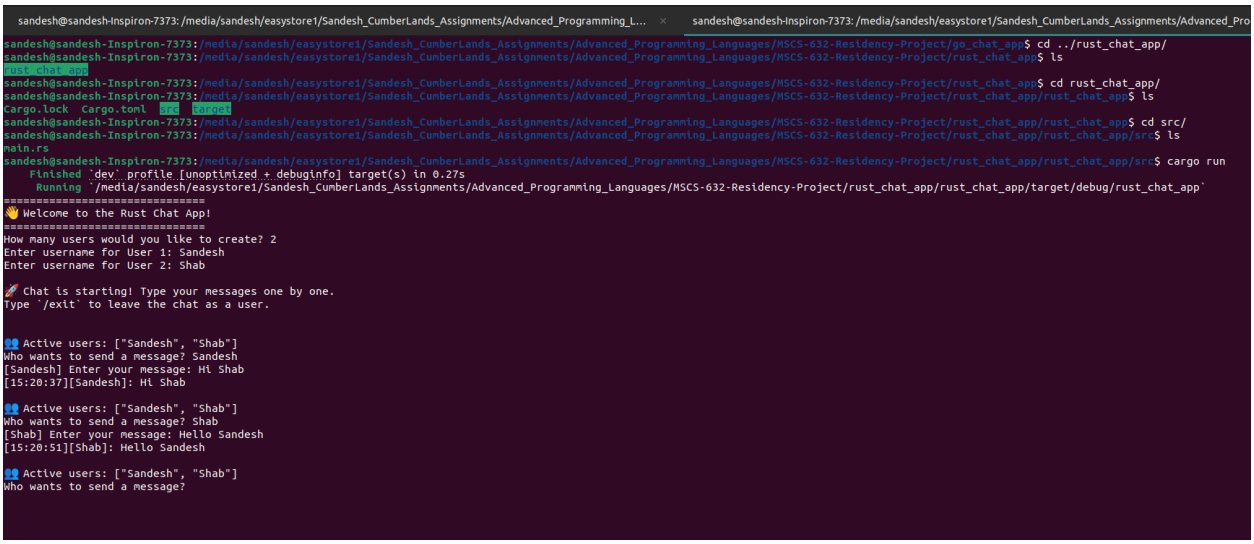
Async Concurrency

Rust's concurrency is handled using tokio and async fn. For example, the interactive user message loop is defined as an async function:

```
async fn start_user(id: &str, server: Arc<Mutex<ChatServer>>) {
    ...
    let mut srv = server.lock().unwrap();
    srv.add_message(msg);
    sleep(Duration::from_millis(200)).await;
}
```

This ensures non-blocking behavior while maintaining thread safety through ownership and borrowing rules.

📸 **Screenshot :** CLI in action with users taking turns to chat.



Message Handling Logic

Users are dynamically added and prompted for turn-based chat like:

Choose user [User1/User2]: User1

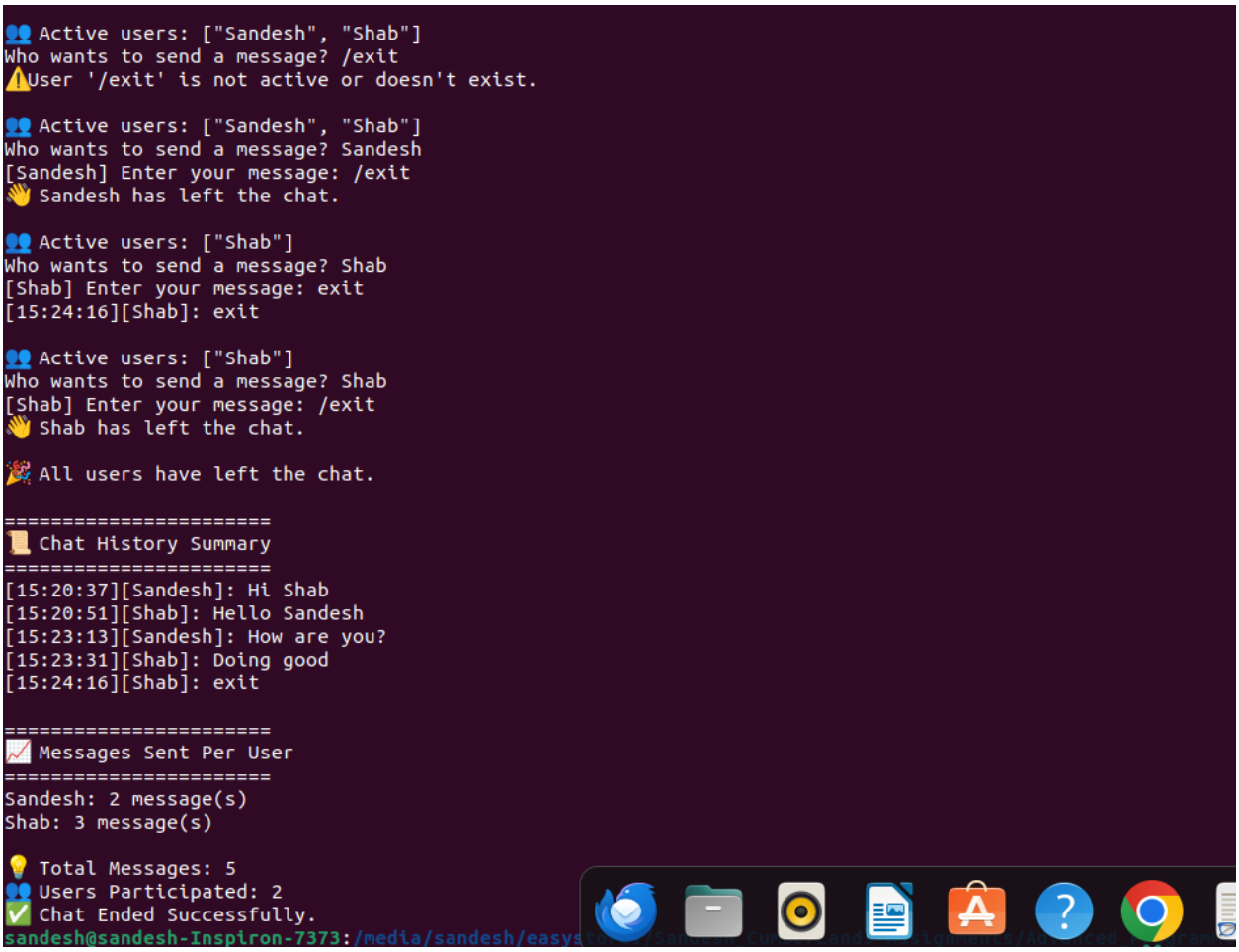
[User1] Enter message: Hello!

After all users exit, a summary of messages and stats is printed.

server.filter_by_user("User1");

server.filter_by_keyword("hello");

📸 **Screenshot :** Chat summary showing filters and user message counts.



3. Go Implementation

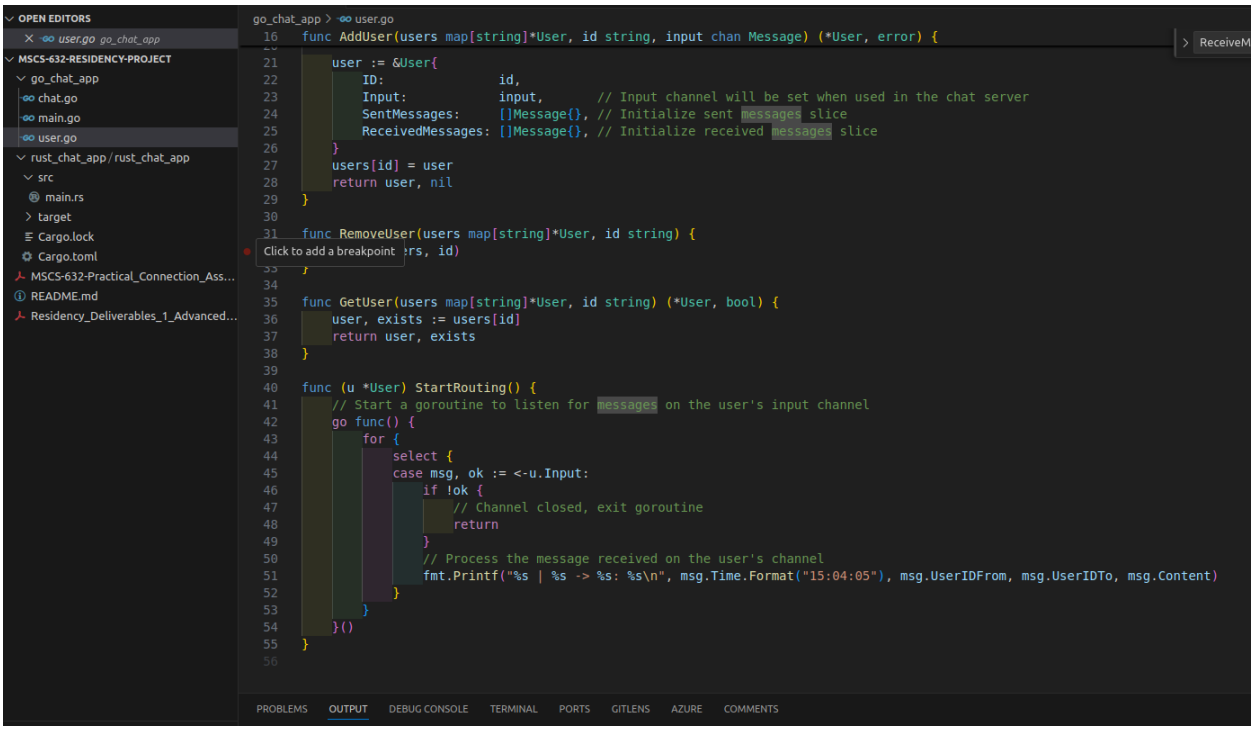
Our Go version is also a CLI-based application where users can be added dynamically using prompts. Messages are routed between users using **goroutines and channels**, showcasing Go's concurrency strengths.

⚙️ Goroutines and Channels

Each user has a dedicated goroutine running ReceiveMessages(). All messages are routed via the ChatServer:

```
func (u *User) ReceiveMessages() {
    for {
        msg := <-u.Received
        fmt.Printf("[From %s to %s] %s\n", msg.SenderID, msg.ReceiverID, msg.Content)
    }
}
```

📸 **Screenshot :** Code showing goroutine setup and ReceiveMessages()



Data Structures

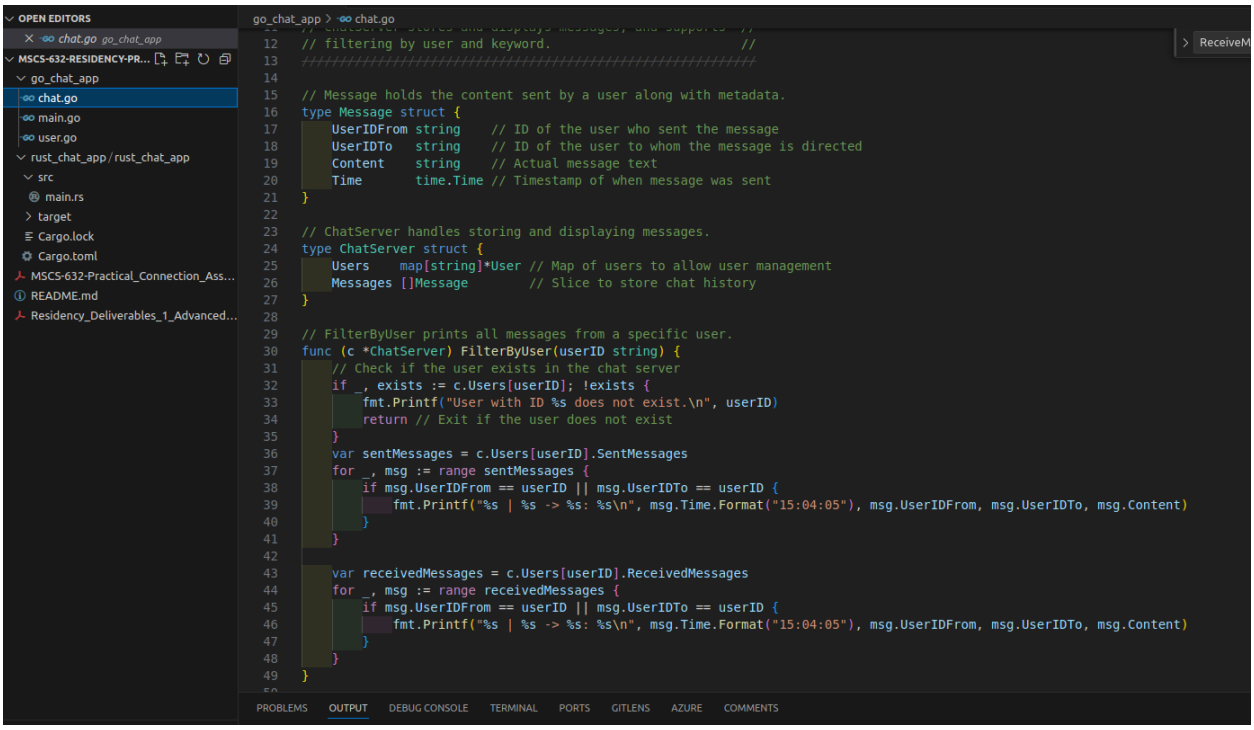
Go uses basic structs and slices, keeping things flexible and idiomatic:

```
type Message struct {
    SenderID   string
    ReceiverID string
    Content    string
}
```

ChatServer stores users and their message history:

```
type ChatServer struct {
    Users    map[string]*User
    Messages []Message
}
```

📸 **Screenshot :** Struct definitions from chat.go



Menu-Driven CLI

The Go version uses a command-based prompt:

1. Add User
2. Send Message
3. Filter Messages
4. Exit

Example send flow:

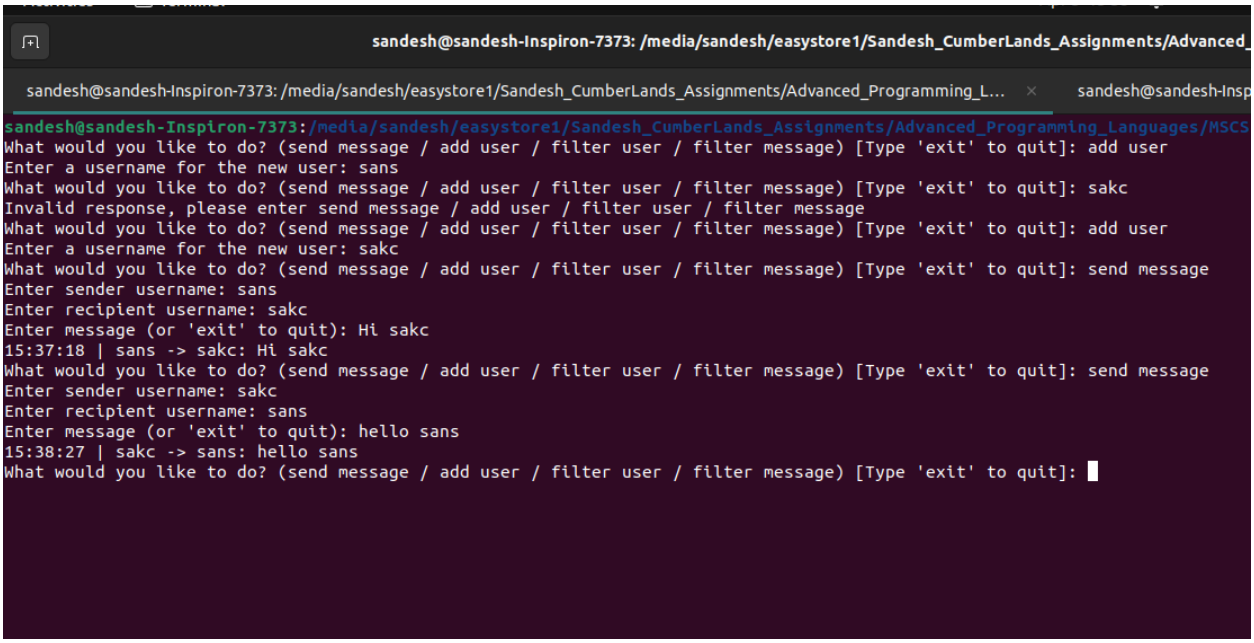
fmt.Print("Sender Username: ")

senderID := getInput()

...

server.SendMessage(senderID, receiverID, content)

📸 **Screenshot :** Terminal showing Go CLI in action with commands

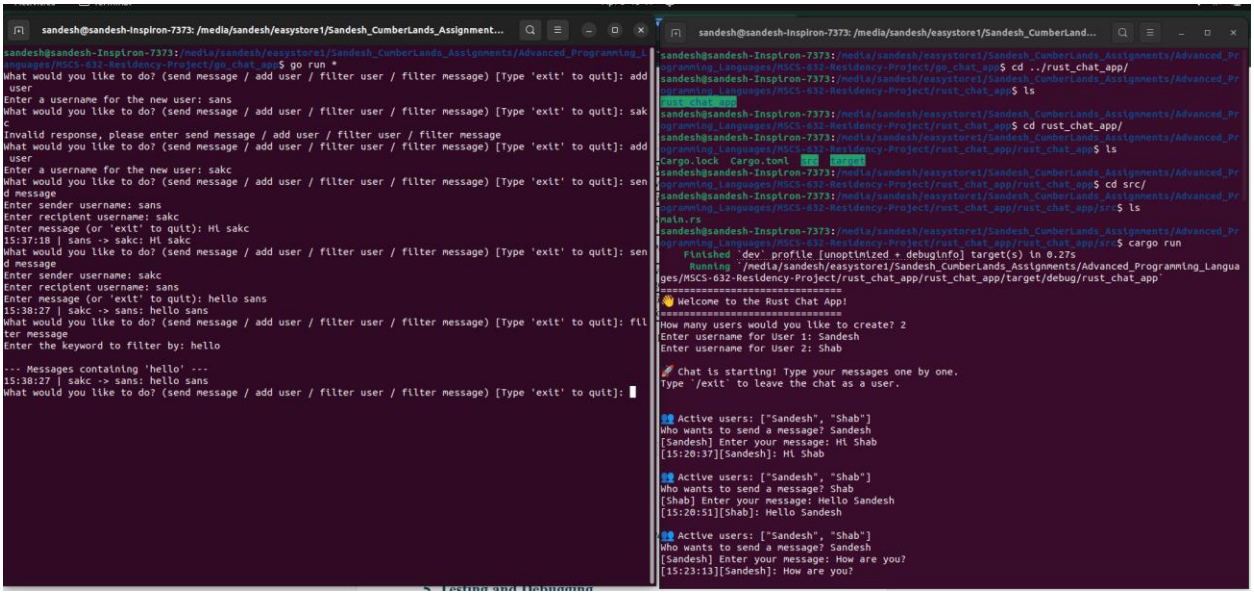


4. Comparison Summary

Feature	Rust	Go
User Creation	At the start, predefined	Dynamically via prompt
Message Flow	Turn-based	Menu-based CLI
Concurrency Model	tokio async + Arc<Mutex>	Goroutines + channels

Data Structures	Structs + VecDeque, HashMap	Structs + slices + maps
Message History	Global queue with summary	Per-user and global history
CLI Experience	Turns + summaries + filters	Menu system with live receiver goroutines
Language-Specific Focus	Memory safety, async handling	Goroutines, channels, performance focus

📸 **Screenshot :** Side-by-side CLI views of Rust and Go versions

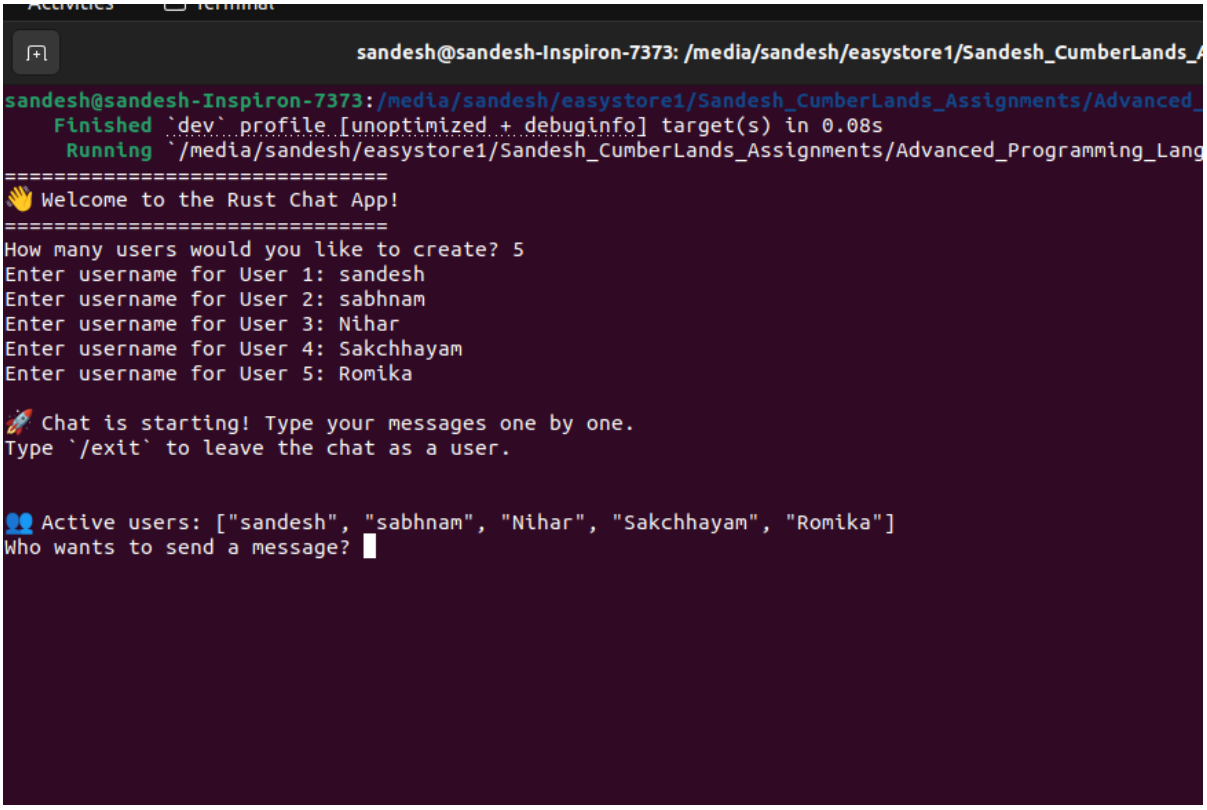


5. Testing and Debugging

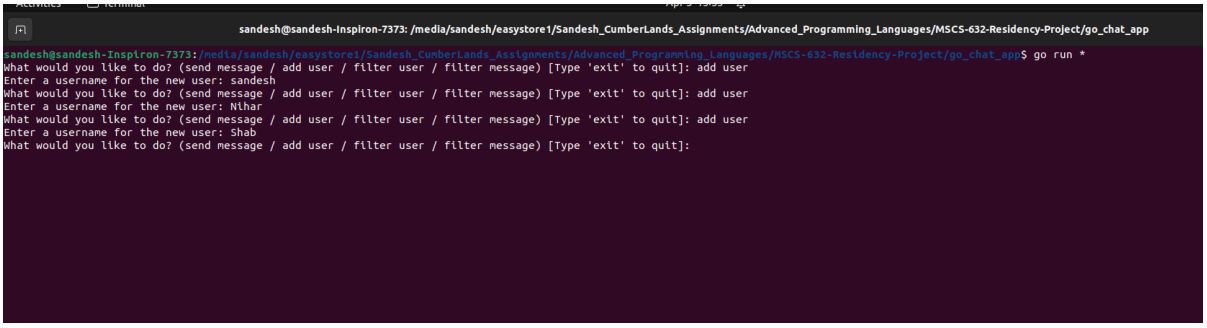
Both applications were tested with:

- ***Multiple users***

  <u>Rust Impl:</u>

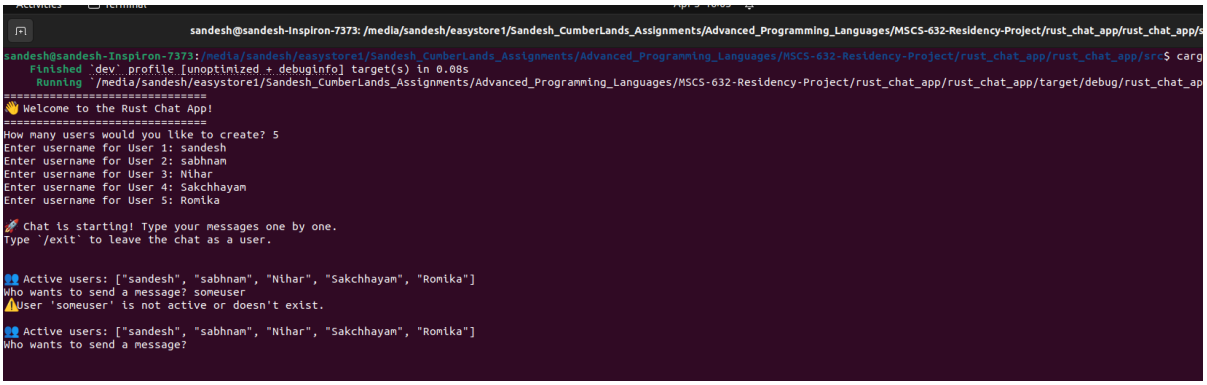


  <u>Go impl:</u>

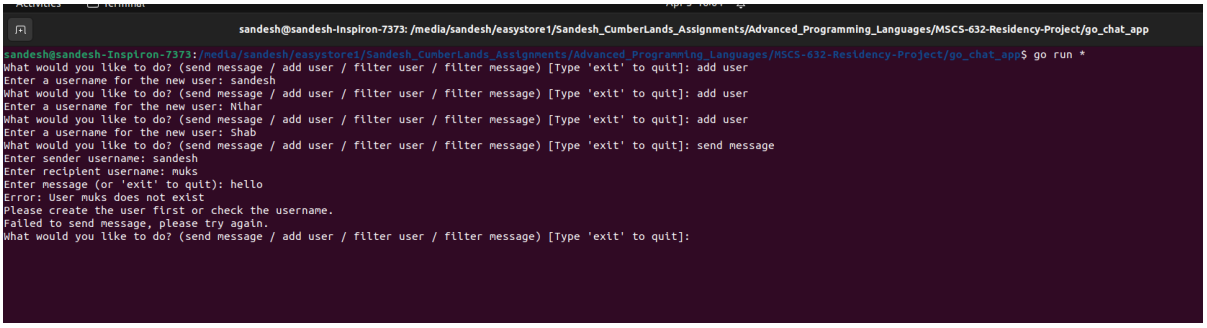


- ***Edge cases (empty messages, invalid usernames)***

  <u>Rust Impl:</u>

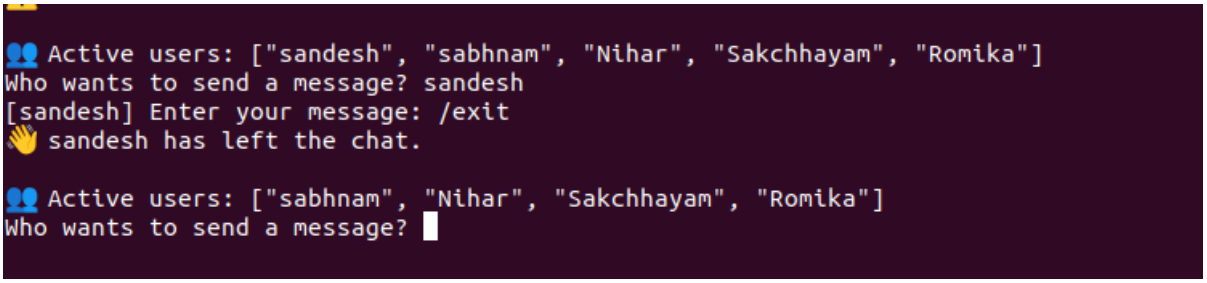


<u>Go Impl:</u>

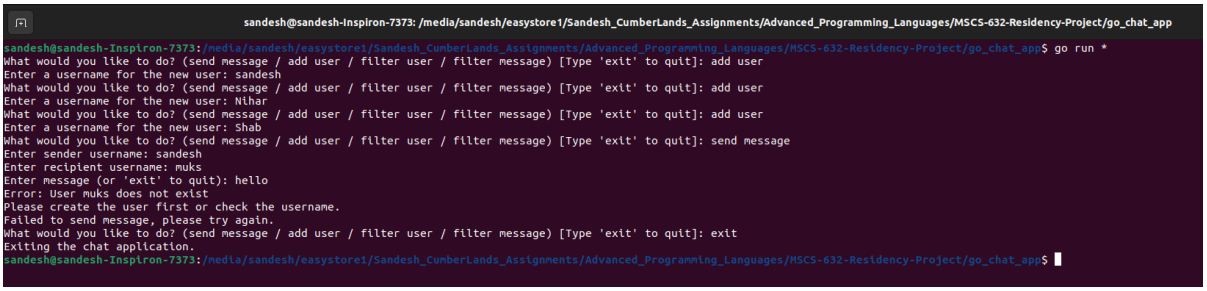


- ***User exit conditions***

<u>Rust Impl:</u>

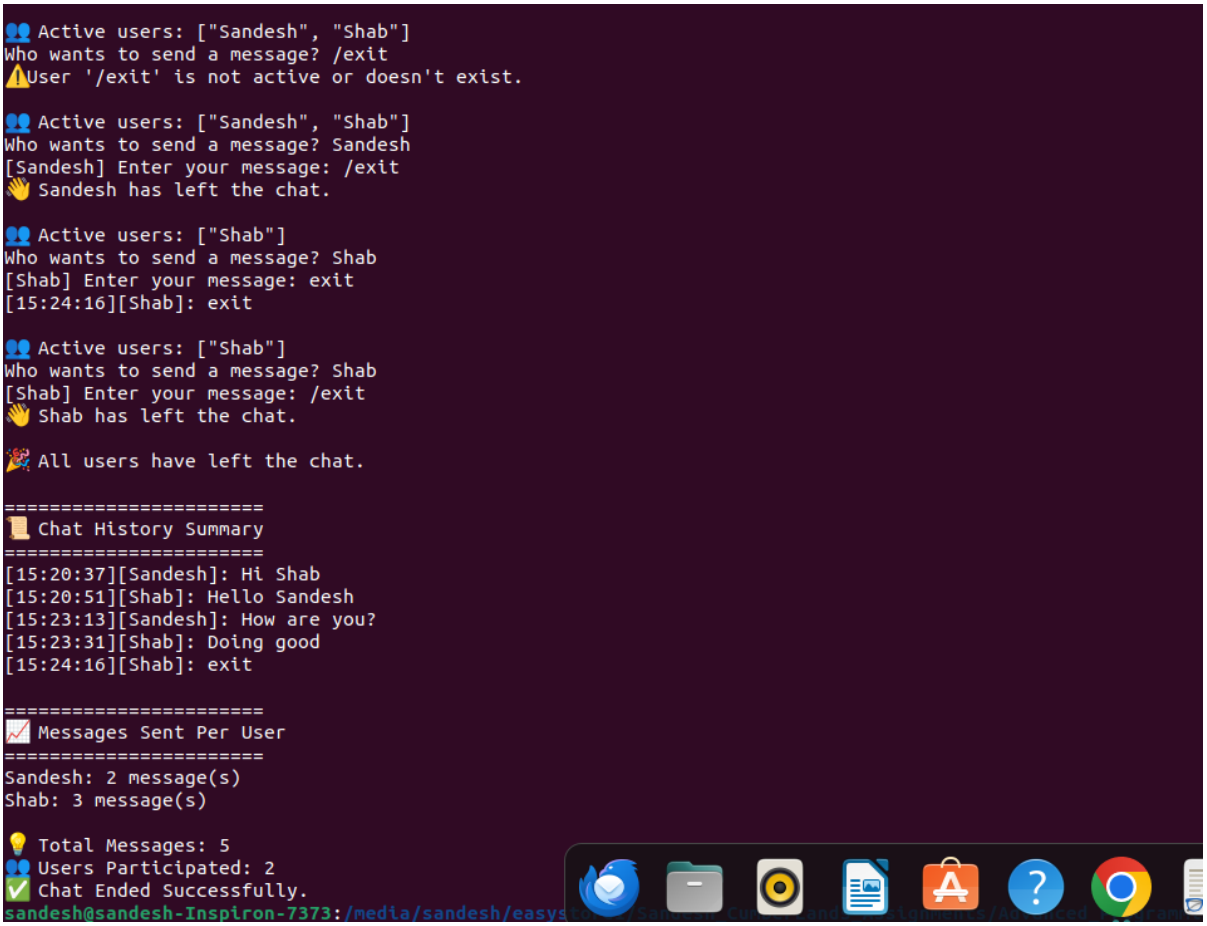


<u>Go Impl:</u>

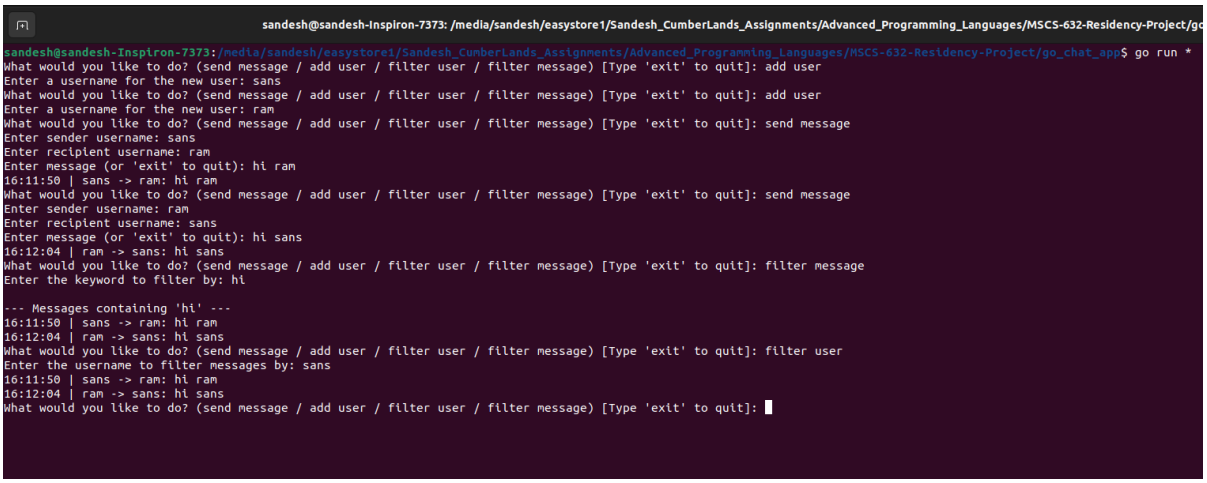


- ***Chat history filtering***

  <u>Rust Impl:</u>



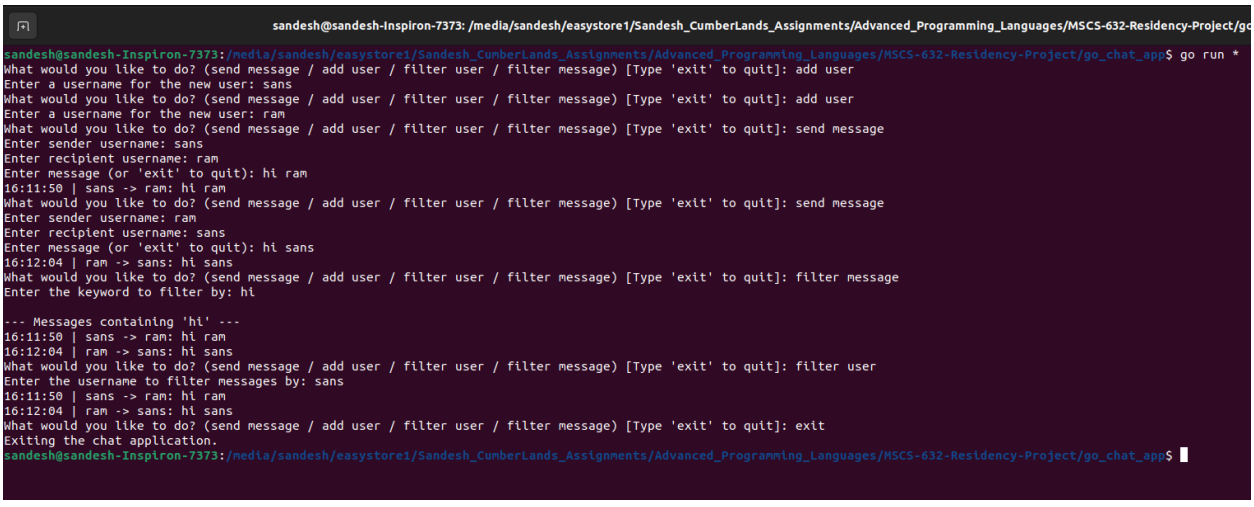
  <u>Go Impl:</u>



Debugging tools like println! (Rust) and fmt.Println (Go) helped in identifying flow issues and confirming data integrity. Both versions handle errors gracefully with simple condition checks.

📸 **Screenshot :** Error handling or test run showing clean exit

Go impl:



Rust impl:

6. GitHub Repository

All code is available at:

🔗 **https://github.com/sanspokharel26677/MSCS-632-Residency-Project**

Folders:

- /rust_chat_app – Turn-based chat with summary reporting
- /go_chat_app – Dynamic CLI chat with live routing

7. Conclusion

Both implementations meet the project requirements and show how different programming paradigms (Rust's safety-first async vs Go's goroutine-based concurrency) can be applied to solve the same problem.

This side-by-side exercise helped us deeply understand language-specific idioms, best practices, and strengths.