Residency Day 1: Deliverable 1

By: Shabnam Shaikh, Sakchham Sangroula,

Sandesh Pokharel, Nihar Turumelle,

Romika Souda.

Course: MSCS-632 Advanced Programming Languages

Instructor: Dr. Vanessa Cooper

University of The Cumberlands

Introduction

The proliferation of real-time communication systems has made chat applications a ubiquitous part of modern software ecosystems. This project presents a Simple Text-Based Chat Application implemented in two powerful systems programming languages Rust and Go each selected for its distinctive approach to concurrency, safety, and performance.

Rust, known for its guarantees around memory safety without a garbage collector, provides tools such as ownership, lifetimes, and the type system to enforce thread safety at compile time. It is particularly suited for low-level control and secure system design. In this implementation, Rust will demonstrate async concurrency, leveraging libraries like “tokio” to handle asynchronous message passing efficiently.

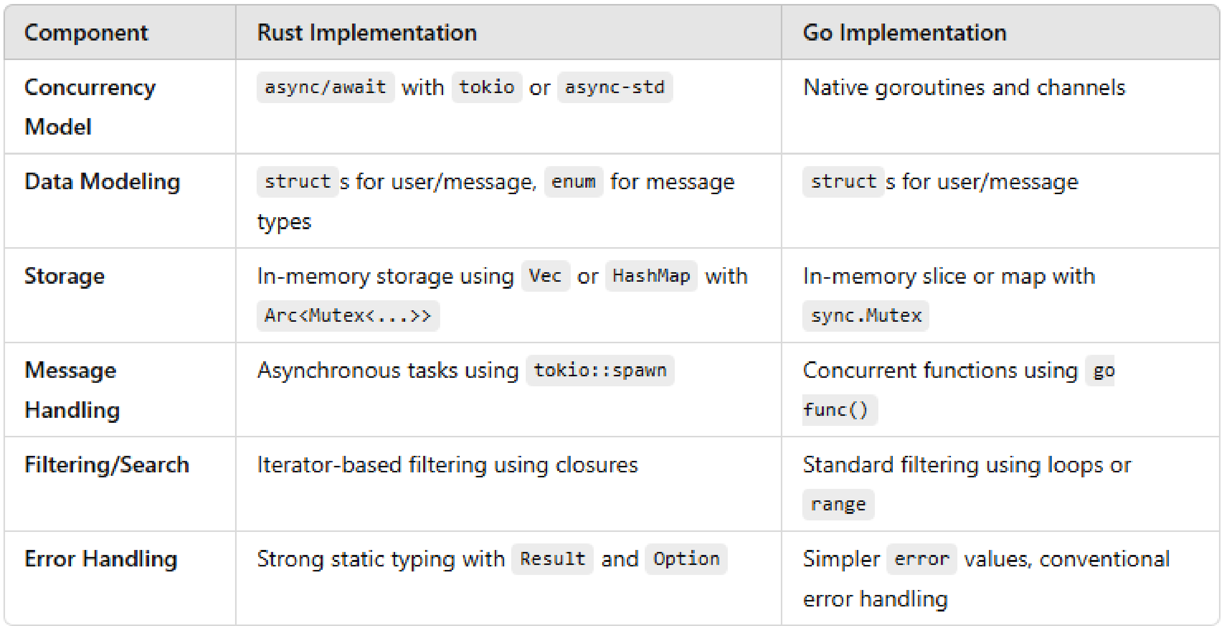
Conversely, Go (Golang) is designed for simplicity and speed in networked and concurrent applications. Its standout features goroutines and channels provide a lightweight and intuitive way to model concurrent workflows. This version of the application will utilize these tools to create performance and readable code for message handling.

The project will simulate a local chat environment with multiple users, message history, and filtering capabilities. Through this comparison, we aim to illustrate the design philosophies of both languages and the practical trade offs in performance, safety, and developer ergonomics.

Github link: https://github.com/sanspokharel26677/MSCS-632-Residency-Project

Application Design

The table below shows the overall structure and features for Rust and Go.



Component	Rust Implementation	Go Implementation
Concurrency Model	`async/await` with `tokio` or `async-std`	Native goroutines and channels
Data Modeling	`struct` s for user/message, `enum` for message types	`struct` s for user/message
Storage	In-memory storage using `Vec` or `HashMap` with `Arc<Mutex<...>>`	In-memory slice or map with `sync.Mutex`
Message Handling	Asynchronous tasks using `tokio::spawn`	Concurrent functions using `go func()`
Filtering/Search	Iterator-based filtering using closures	Standard filtering using loops or `range`
Error Handling	Strong static typing with `Result` and `Option`	Simpler `error` values, conventional error handling

Below shows the component breakdown and language specific differences:

1. User Module
   a. Rust:
      - Struct: User {id: u32, name: String }
      - Thread-safe sharing with Arc

   b. Go:
      - Struct: type User struct { ID int; Name string }
      - Shared via references with sync.Mutex if needed
2. Message Module:
   a. Rust:
      - Struct: Message { sender_id: u32, content: String, timestamp: DateTime<Utc> }
      - Enum for message types (eg., Text, Command)

b. Go:

   - Struct: Message { SenderID int; Content string; Timestamp time.Time }

3. Chat Handler / Engine

   a. Rust:

      - Async fn for handling message send/receive
      - Shared state via Arc<Mutex<ChatState>>

   b. Go:

      - Channel based message passing (chan Message)
      - Goroutines for each simulated user.

4. Storage and Logging

   a. Rust:

      - In memory with Vec<Message> inside Mutex.

   b. Go:

      - In memory []Message, protected by sync.Mutex.

5. Message Filtering & Search

   a. Rust:

      - Functional filtering: .iter().filter(|msg| …)

   b. Go:

      - Procedural loop-based filtering.

6. Simulation of Multiple Users

   a. Rust:

      - Use tokio::spawn tasks to simulate user threads.

   b. Go:

      - Spawn goroutines per user with a central message dispatcher.

Major Differences

1. Concurrency:
   a. Rust provides compile-time safety with its ownership model, requiring explicit sharing of state and careful synchronization (Arc<Mutex<T>>).
   b. Go prioritizes runtime simplicity, allowing concurrent routines to share memory with simpler syntax but requiring vigilant management of race conditions.
2. Error Handling:
   a. Rust's pattern matching with Result types helps handle errors explicitly.
   b. Go uses the traditional if err != nil idiom which is verbose but familiar to many.
3. Tooling and Ecosystem:
   a. Rust uses Cargo and a growing async ecosystem (tokio, async-std).
   b. Go's standard library is rich in concurrency and network features, making external dependencies minimal.

Task Assignment

Sakchham Sangroula – Go Program

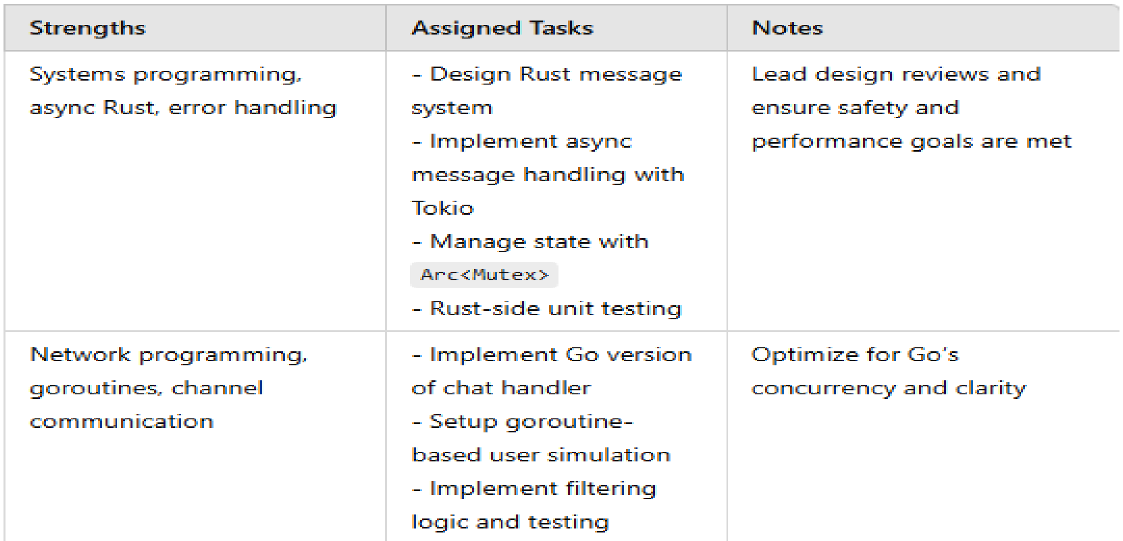
Shabnam Shaikh - Documentation

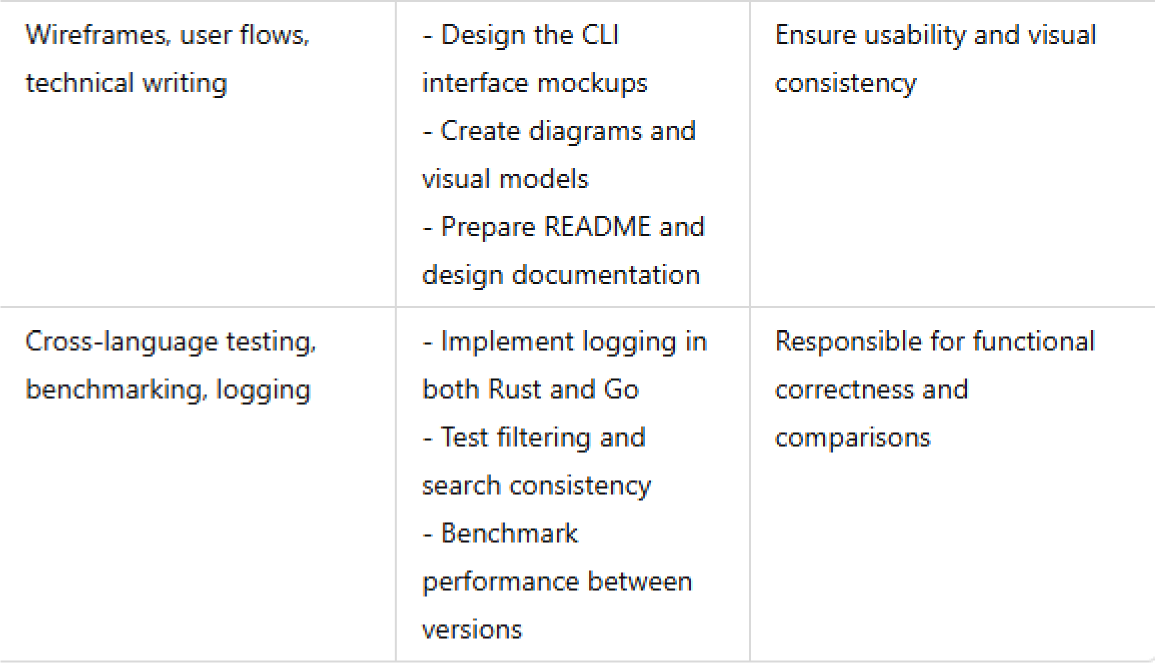
Sandesh Pokharel – Rust Program

Nihar Turumelle – Rust Program

Romika Souda – Documentation and planning

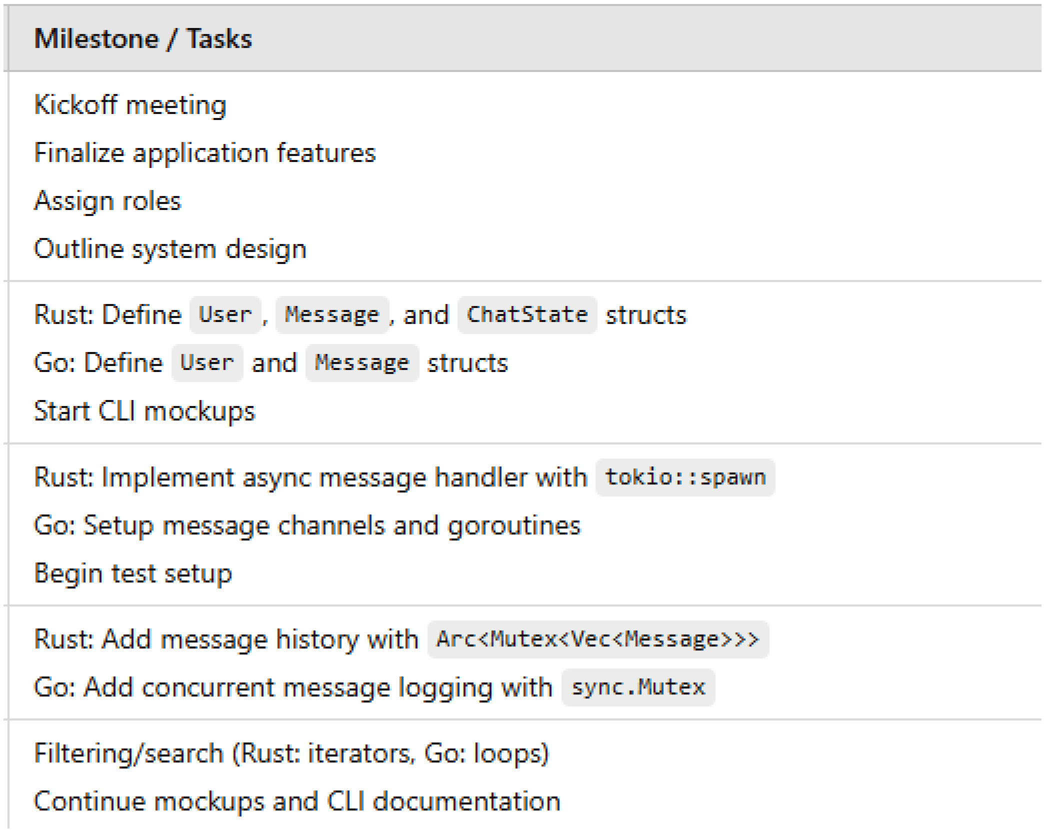
Residency Day 1: Deliverable 1



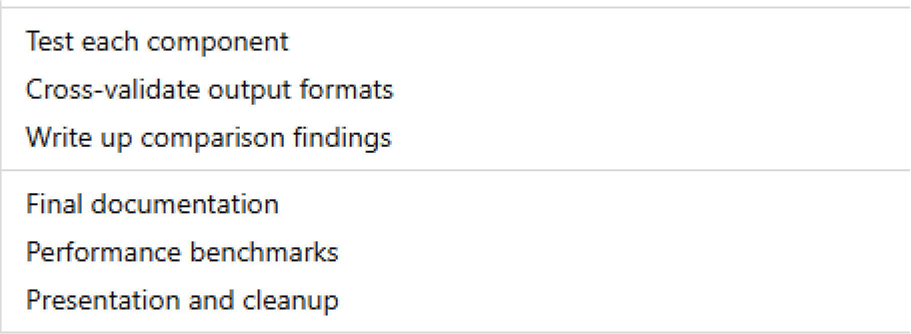


Strengths	Assigned Tasks	Notes
Systems programming, async Rust, error handling	- Design Rust message system - Implement async message handling with Tokio - Manage state with `Arc<Mutex>` - Rust-side unit testing	Lead design reviews and ensure safety and performance goals are met
Network programming, goroutines, channel communication	- Implement Go version of chat handler - Setup goroutine-based user simulation - Implement filtering logic and testing	Optimize for Go's concurrency and clarity
Wireframes, user flows, technical writing	- Design the CLI interface mockups - Create diagrams and visual models - Prepare README and design documentation	Ensure usability and visual consistency
Cross-language testing, benchmarking, logging	- Implement logging in both Rust and Go - Test filtering and search consistency - Benchmark performance between versions	Responsible for functional correctness and comparisons

Timeline



Milestone / Tasks
Kickoff meeting Finalize application features Assign roles Outline system design
Rust: Define `User`, `Message`, and `ChatState` structs Go: Define `User` and `Message` structs Start CLI mockups
Rust: Implement async message handler with `tokio::spawn` Go: Setup message channels and goroutines Begin test setup
Rust: Add message history with `Arc<Mutex<Vec<Message>>>` Go: Add concurrent message logging with `sync.Mutex`
Filtering/search (Rust: iterators, Go: loops) Continue mockups and CLI documentation



Test each component Cross-validate output formats Write up comparison findings
Final documentation Performance benchmarks Presentation and cleanup

Documentation of Design

1. Application Design Summary: Core Features (Both Versions)

    a. User simulation via threads (Rust: tasks, Go: goroutines).

    b. Message structure: sender_id, timestamp, content.

    c. History storage (in-memory).

    d. Message filtering by keyword or sender.

   e. Basic CLI interface for user input/output simulation.

2. Rust Specific Design Challenges:

   a. Borrow checker complexity when sharing mutable state.

   b. Arc<Mutex<T>> to allow safe concurrent access.

   c. Asynchronous programming using tokio::main and .await syntax.

3. Go-Specific Design Challenges:

   a. Synchronization of shared message history using sunc.Mutex.

   b. Preventing race conditions with proper locking.

   c. Clear channel communication design to avoid deadlocks.

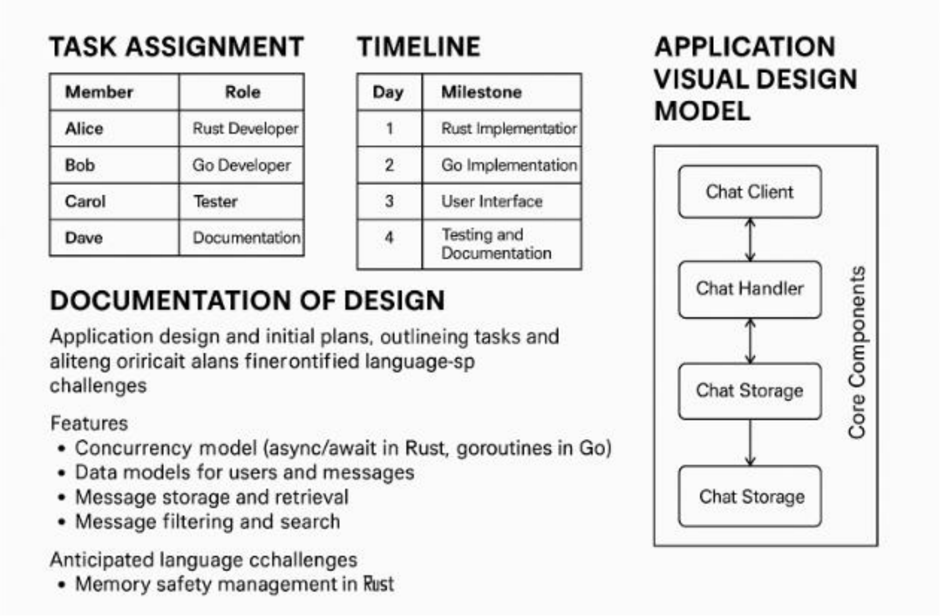
4. Application Visual Design Model

   a. Simulated users are represented as entities that send/receive messages.

   b. A central dispatcher routes messages.

   c. A message store maintains chat history.

   d. CLI interface allows message viewing and filtering.



TASK ASSIGNMENT

Member	Role
Alice	Rust Developer
Bob	Go Developer
Carol	Tester
Dave	Documentation

TIMELINE

Day	Milestone
1	Rust Implementatior
2	Go Implementation
3	User Interface
4	Testing and Documentation

APPLICATION VISUAL DESIGN MODEL

Chat Client

Chat Handler

Chat Storage

Chat Storage

Core Components

DOCUMENTATION OF DESIGN

Application design and initial plans, outlineing tasks and aliteng oriricait alans finerontified language-sp challenges

Features

- Concurrency model (async/await in Rust, goroutines in Go)
- Data models for users and messages
- Message storage and retrieval
- Message filtering and search

Anticipated language cchallenges

- Memory safety management in Rust

References

1. Klabnik, S., & Nichols, C. (2018). *The Rust Programming Language*. No Starch Press. https://doc.rust-lang.org/book/
2. Google. (2024). *Effective Go*. https://golang.org/doc/effective_go.html
3. Tokio.rs. (2025). *Asynchronous Programming in Rust with Tokio*. https://tokio.rs/
4. Go by Example. (2025). *Goroutines and Channels*. https://gobyexample.com/
5. Rust async book. (2025). *Async Programming in Rust*. https://rust-lang.github.io/async-book/
6. Cox, R. (2012). *Go Concurrency Patterns: Pipelines and Cancellation*. https://blog.golang.org/pipelines