This work was done by both Ismael Cabrera-Hernandez and Andrew T. Bird.

1. Introduction

This report documents the complete development process of a modular, terminal-based client–server chat application. The project required the use of Python sockets, multi-client support, and optional augmentations such as message searching, deletion, colors, and TLS encryption. The work was completed using a modular architecture to allow incremental upgrades. This report contains the development process, design diagrams, prompts used during creation, TLS implementation details, and the final feature set.

2. Full List of Prompts Used

• I need to make a simple chat app that utilizes a client server model...
• Make it terminal based where a client can connect using: ChatClient.py [IP_ADDRESS_OF_SERVER]
• Make the framework modular so I can add features later.
• So what exactly does chatcore do?
• What do I put in __init__.py?
• Can you make the class diagram/improvement plan?
• We implemented TLS into the server allowing encryption of chats.
• Tested.
• Help me clean up the UI.
• Allow usernames to appear when sending messages.
• Allow color selection when joining the server.
• Add message IDs.
• Add a search command.
• Add a delete command.
• Add a refresh system.
• Fix client echo-loop.
• Fix TLS disconnection errors.
• Allow unique username+#id system.• Add server-side message history.
• Add dynamic commands and state management.

3. Development Process

The project began with a base framework generated using ChatGPT. The structure emphasized separation of concerns: protocol handling, message encoding, client state management, server broadcasting, and TLS configuration were isolated into their own modules. This made later expansions significantly easier. A GitHub repository was set up to collaborate and store consecutive improvements. The initial structure included ChatServer.py, ChatClient.py, and a chatcore/ package containing the protocol and handler-related modules. As development progressed, new features were introduced incrementally: - Username

registration - Color customization - Message history - Message IDs - Searching and deletion functionality - TLS encryption for secure communication

4. Architecture & Class Diagram

The architecture follows a classical client–server design. Every client establishes a secure TLS-wrapped connection to the chat server. The server maintains an internal message state, broadcasting incoming messages to all active clients. Commands are intercepted and processed by a shared command-handler subsystem. Below is the text-based class diagram used during planning:

+----------------------+ +------------------------+ | ChatClient | | ChatServer | |----------------------| |------------------------| | username | | clients[] | | color | | client_usernames{} | | start_client() | | client_ids{} | | listen() | | handle_client() | | send_message() | | broadcast() | +----------------------+ +------------------------+ | | | uses | uses v v +----------------------+ +------------------------+ | protocol | | state | | encode_message() | | add_message() | | decode_message() | | delete_message() | | format_message() | | search() | +----------------------+ +------------------------+ | | uses v +----------------------+ | commands | | cmd_delete() | | cmd_search() | | cmd_refresh() | | cmd_color() | +----------------------+

5. TLS Encryption Implementation

TLS support was added to both server and client. The process involved generating server.crt and server.key using OpenSSL. A server SSL context was created, wrapping each incoming socket connection. The client's connection was also wrapped in a TLS context, allowing encrypted communication without requiring certificate verification for development purposes. Server TLS example: tls_context = create_server_context(certfile="server.crt", keyfile="server.key") tls_conn = wrap_server_connection(conn, tls_context) Client side: tls_context = create_client_context(verify=False) tls_sock = wrap_client_socket(sock, tls_context)

6. Feature Set Implemented The final project includes all required features and several augmentations: - Multi-client chat -
Unique username#ID system - TLS-encrypted communication - Persistent message history - Message IDs - /search command - /delete command - /refresh command - /color command - Server-side system messages - Real-time updates on client join/leave

7. Meeting the Requirements

The project satisfies all primary requirements: 2+ clients, unique identifiers, message forwarding, proper disconnect handling, and modularity. For augmentations, four were required but six were successfully implemented: 1. Message searching 2. Message deletion 3. Color customization 4. TLS encryption 5. Message history 6. Client-list announcements

8. Conclusion

This project demonstrates a modular, extensible networked chat architecture supporting secure communication and multiple advanced features. The development process relied heavily on iterative refinement and modular design principles. The final implementation exceeds the augmentation requirements and provides a robust foundation for future enhancements such as private messages, nickname changes, or GUI extensions.