Real-time Cryptographic Messaging System

by

Kavya Sri Punna

Anurag Gulishetty

This project is submitted to the Gannon University graduate faculty in

partial fulfillment for the degree Master of Science in Computer and Information Science.

Option: Cybersecurity & Information Assurance

Approved:

|  |  |  |  |
| --- | --- | --- | --- |
|  | | |  |
| *John. H. Coffman, MS* | | |  |
| Advising Professor in Charge of Research | | |  |
|  | |  |  | | |
|  | |  |  | | |

Gannon University

Erie, Pennsylvania 16541

May 2025

Acknowledgements

We are deeply grateful to Professor John H. Coffman for his mentorship and technical guidance throughout this project. His support helped shape the system’s design and strengthened the academic foundation of my work.

We would also like to thank my committee members—Mr. Jeremy Cannell, Dr. Mei-Huei Tang, and Deacon Stephen T. Frezza—for their constructive feedback, time, and expertise which helped refine and strengthen the project.

Additionally, we extend our sincere thanks to our families and close friends for their unwavering support and motivation throughout this academic journey. Finally, we acknowledge the contributions of the open-source community and the Gannon University CIS department, whose tools and resources made this secure communication system possible.

Table of Contents

[Acknowledgements ii](#_Toc8392258)

[Abstract v](#_Toc8392259)

[List of Figures vi](#_Toc8392260)

[List of Tables vii](#_Toc8392261)

[1. Introduction 1](#_Toc8392262)

[1.1 Overview 1](#_Toc8392263)

[1.2 Curriculum Scope 1](#_Toc8392264)

[1.3 Key Stakeholder Needs 1](#_Toc8392265)

[1.4 Product Perspective 1](#_Toc8392266)

[1.5 Product Position Statement 2](#_Toc8392267)

[1.6 Summary of Capabilities 2](#_Toc8392268)

[1.7 Alternatives and Competition 2](#_Toc8392269)

[1.8 Project Management Plan 3](#_Toc8392270)

[1.9 References 3](#_Toc8392271)

[2. Requirements Management 4](#_Toc8392272)

[2.1 Requirements Development Perspective 4](#_Toc8392273)

[2.2 Use Characteristics 4](#_Toc8392274)

[2.2.1 User Classes and Characteristics 4](#_Toc8392275)

[2.2.2 Use-Case Model Survey 4](#_Toc8392276)

[2.2.3 User Documentation 5](#_Toc8392277)

[2.3 Feature Attributes 5](#_Toc8392278)

[2.3.1 Name\_of\_Attribute-1 6](#_Toc8392279)

[2.3.2 Name\_of\_Attribute-2 6](#_Toc8392280)

[2.3.3 Name\_of\_Attribute-3 6](#_Toc8392281)

[2.4 Key System Features 6](#_Toc8392282)

[2.4.1 Key System Feature 1 6](#_Toc8392283)

[2.4.2 Key System Feature 2 (*and so on*) 7](#_Toc8392284)

[2.5 Key Design and Implementation Constraints 7](#_Toc8392285)

[2.5.1 Operating Environment 7](#_Toc8392286)

[2.6 Interface Requirements 7](#_Toc8392287)

[2.6.1 User Interfaces 7](#_Toc8392288)

[2.6.2 Hardware Interfaces 8](#_Toc8392289)

[2.6.3 Software Interfaces 8](#_Toc8392290)

[2.6.4 Communications Interfaces 8](#_Toc8392291)

[2.7 Nonfunctional Requirements 8](#_Toc8392292)

[2.7.1 Performance Requirements 8](#_Toc8392293)

[2.7.2 Security Requirements 9](#_Toc8392294)

[2.7.3 Software Quality Attributes 9](#_Toc8392295)

[2.7.4 Safety Requirements 9](#_Toc8392296)

[2.7.5 Other Requirements 9](#_Toc8392297)

[2.8 Assumptions and Dependencies 9](#_Toc8392298)

[3. Design 10](#_Toc8392299)

[3.1 Introduction 10](#_Toc8392301)

[3.1.1 Goals and Objectives of Design 10](#_Toc8392302)

[3.1.2 Statement of Software Scope 10](#_Toc8392303)

[3.1.3 Software Context 10](#_Toc8392304)

[3.1.4 Major Design Constraints 10](#_Toc8392305)

[3.2 Data Design 10](#_Toc8392306)

[3.2.1 Major Internal Software Data Structure 10](#_Toc8392307)

[3.2.2 Global Data Structure 10](#_Toc8392308)

[3.2.3 Temporary Data Structure 10](#_Toc8392309)

[3.2.4 Database Description 11](#_Toc8392310)

[3.3 Architectural and Component-Level Design 11](#_Toc8392311)

[3.3.1 Program Structure 11](#_Toc8392312)

[3.3.2 Key Software Components 11](#_Toc8392313)

[3.4 User Interface Design 12](#_Toc8392314)

[3.4.1 Description of the User Interface 12](#_Toc8392315)

[3.4.2 Interface Design Rules 12](#_Toc8392316)

[3.4.3 Components Available 12](#_Toc8392317)

[3.4.4 User Interface Design Description 12](#_Toc8392318)

[3.5 Restrictions, Limitations, and Constraints 12](#_Toc8392319)

[4. Verification and Validation 13](#_Toc8392320)

[4.1 Test items 13](#_Toc8392322)

[5. Conclusion 14](#_Toc8392323)

[6. Bibliography 15](#_Toc8392324)

[Appendix A: Glossary 16](#_Toc8392325)

[Appendix B: Use Case Analysis 17](#_Toc8392326)

[Appendix C: Analysis Models 17](#_Toc8392327)

[Appendix D: Design Models 17](#_Toc8392328)

[Appendix E: Testing Log and Summary Status 17](#_Toc8392329)

[Appendix F: Screen Captures 17](#_Toc8392330)

[Appendix G: Project File Repository Definitions 17](#_Toc8392331)

Abstract

With communication becoming ever more instantaneous and global, the need for privacy and secure exchange of information is critical. Traditional messaging platforms often struggle to maintain confidentiality and authenticity in the face of rising cyber threats.

This project presents a Real-time Cryptographic Messaging System (RCMS) developed using React.js for the frontend and Node.js + Express for the backend. It leverages AES-256 for message encryption, RSA-4096/ECC for key exchange, and SHA-256 for message integrity. WebSockets are used to enable real-time communication, and MongoDB ensures fast, scalable encrypted message storage.

The key challenges addressed include balancing encryption overhead with performance, integrating secure protocols without compromising user experience, and ensuring scalable data management. The final product offers a user-friendly yet secure environment for real-time communication, validating that encryption and speed can coexist.

The project was successful in meeting its core objectives: enabling secure, real-time messaging with end-to-end encryption, robust backend performance, and a clean, intuitive frontend. Future improvements include group messaging, mobile app deployment, and transitioning toward decentralized peer-to-peer infrastructure.

# List of Figures

[Figure 1. Maybe a Context Diagram 1](#_Toc1377697)

[Figure 2. Maybe something else 5](#_Toc1377698)

[Figure 3. This is another Caption 5](#_Toc1377699)

< *A listing of the figures or graphs displayed in the document are itemized here, including the page number of the appearance of the graphic. The following format pattern should be used in the listing and should be used as the graphic’s caption. The text of the caption should be boldface, all capital letters, the caption should appear under the graphic, and the numbering should be enabled through the* /References/Insert-Table-of-Figures *option of MS Word. “N” refers to the page number where the figure appears.*

*The List of Figures for this template has been generated above.>*

# List of Tables

[Table 1: Position Statement 2](#_Toc8392414)

[TABLE 2: Benefits and Supporting Features 2](#_Toc8392415)

< A listing of the tables displayed in the document are itemized here, including the page number of the appearance of the table. The following format pattern should be used in the listing and should be duplicated as the table’s caption. The text of the caption should be boldface, all capital letters, the caption should appear under the graphic, and the numbering should be enabled through the /References/Insert-Table-of-Figures option of MS Word. “N” refers to the page number where the table appears.

*The List of Tables for this template has been generated above.>*

>

1. Introduction

Overview

In today's interconnected world, the demand for secure and real-time communication is more critical than ever. With the increasing number of data breaches, privacy concerns, and surveillance threats, traditional messaging platforms often fall short of meeting users’ security expectations.

The Real-time Cryptographic Messaging System (RCMS) we developed addresses this gap by offering a secure, fast, and scalable messaging solution. It utilizes advanced cryptographic techniques such as AES-256 for message encryption and RSA-4096 or ECC for secure key exchanges. The system operates on a client-server model and delivers messages instantly using WebSocket-based communication.

Our goal was to design a platform that not only ensures confidentiality and data integrity but also maintains a seamless user experience. With a responsive frontend built using React.js and a robust backend powered by Node.js and MongoDB, the system is geared toward real-time encrypted messaging while being user-friendly and scalable.

Curriculum Scope

This project directly applies to the core competencies acquired throughout our Master’s program in Cybersecurity & Information Assurance. From cryptographic principles and secure software design to full-stack development and systems integration, this project served as a capstone that brought together multiple aspects of our academic training. Specifically:

* Secure communication protocols
* Cryptography (symmetric, asymmetric, hashing)
* Client-server architecture design
* Full-stack development (React.js, Node.js, MongoDB)
* Real-time data transmission via WebSockets
* Secure data storage and privacy-by-design practices

Key Stakeholder Needs

Several challenges in modern communication platforms informed the design of this system. Stakeholders such as privacy-conscious users, cybersecurity professionals, and enterprise organizations require:

* End-to-end encrypted messaging that prevents unauthorized access
* Instant message delivery without delays
* Assurance of message integrity and authenticity
* Minimal exposure to metadata and data collection
* A user interface that is both secure and easy to use

Existing platforms often compromise speed for security or sacrifice privacy for performance. Our stakeholders needed a solution that could deliver all three: speed, security, and usability.

Product Perspective

The Real-time Cryptographic Messaging System (RCMS) is an independent and self-contained product, designed to deliver secure and instant communication through modern web technologies. While it can be integrated into larger organizational systems or enterprise communication platforms, it is fully functional as a standalone messaging solution.

The system is built on a client-server architecture and consists of four main components:

* Frontend (React.js): A user-friendly interface where users can register, log in, manage friends, and exchange encrypted messages.
* Backend (Node.js with Express.js): Handles core logic such as authentication, key management, message routing, and WebSocket connection setup.
* WebSockets: Ensures persistent, bidirectional communication between client and server for real-time delivery.
* Database (MongoDB): Stores encrypted message histories, user credentials, and contact lists in a secure and scalable manner.

The diagram below illustrates how these components interact with each other in real time:

A diagram of a computer system

AI-generated content may be incorrect.

Figure . Maybe a Context Diagram



Product Position Statement

The Real-time Cryptographic Messaging System is designed to serve users who prioritize both speed and privacy in digital communication. While many existing platforms offer some level of encryption or real-time performance, few achieve a reliable balance between both. This product fills that gap with its emphasis on security without sacrificing responsiveness or user experience.

The table below provides a clear summary of the product’s unique value proposition:

|  |  |
| --- | --- |
| For | Individuals and organizations that require secure, real-time digital communication |
| Who | Need to exchange sensitive messages without risking data breaches or surveillance |
| The Real-time Cryptographic Messaging System | is a Secure messaging application |
| That | Provides encrypted, instant communication with message integrity and privacy |
| Unlike | Traditional messaging apps with partial encryption or laggy performance |
|  | Delivers end-to-end encrypted messaging in real-time with an intuitive interface |

Table : Position Statement

This positioning clearly defines the system's intended audience, their needs, and how this product stands apart. Unlike conventional apps that often favor convenience over privacy, RCMS proves that secure communication can also be fast, scalable, and user-friendly.

Summary of Capabilities

The Real-time Cryptographic Messaging System offers a comprehensive blend of security, performance, and usability. Designed with modern communication needs in mind, the system delivers on its promise of secure, real-time messaging through a well-architected tech stack and thoughtful user experience.

Below is a summary of the core capabilities of the system, along with the features that support them:

**Real-time Cryptographic Messaging System**

|  |  |
| --- | --- |
| Benefit | Supporting Features |
| End-to-End Encrypted Communication | AES-256 message encryption, RSA-4096/ECC key exchange |
| Real-Time Message Delivery | Persistent WebSocket connections |
| Scalable and Secure Data Storage | Encrypted message storage using MongoDB |
| Seamless User Experience | Clean, intuitive React.js frontend |
| Data Integrity Assurance | SHA-256 message hashing to detect tampering |
| Secure User Authentication | Login and session management via Express and secure credential handling |
| Minimal Latency in Transmission | Event-driven architecture with bidirectional WebSocket messaging |
| Privacy by Design | Minimal data collection, encrypted storage, and encrypted transit protocols (TLS/SSL) |

TABLE : Benefits and Supporting Features

This feature set empowers users to communicate without fear of interception or delay, making it ideal for sensitive personal, professional, or organizational use.

Alternatives and Competition

Several messaging applications exist in today’s market, each offering varying degrees of security, speed, and usability. While some are widely adopted and easy to use, others focus more on encryption and security. However, there is often a trade-off between real-time performance and robust privacy features.

Below are a few notable alternatives and how our system compares:

|  |  |  |
| --- | --- | --- |
| System | Strength | Weakness |
| Signal | Offers strong end-to-end encryption, open-source, highly trusted in the privacy community. | Can experience performance lag under poor network conditions; mobile-centric with limited extensibility for enterprise-level customization. |
| WhatsApp | Widespread adoption, simple interface, end-to-end encryption. | Owned by Meta, raising concerns about metadata collection and long-term privacy policies; lacks transparency in key handling mechanisms. |
| Telegram | Fast message delivery, cloud-based storage, rich feature set. | End-to-end encryption is optional and only for “Secret Chats,” not enabled by default; stores messages on the cloud by default. |
| Matrix/Element | Decentralized architecture, strong encryption, open standard. | Steeper learning curve, heavier setup requirements, and UI can be non-intuitive for new users. |

Project Management Plan

The development of the Real-time Cryptographic Messaging System followed a phased approach that aligned closely with standard software engineering practices. The core phases included problem identification, analysis, design, development, testing, and documentation. Each phase introduced its own learning curve and set of challenges, but also presented opportunities to deepen our technical expertise.

**Project Phases & Highlights:**

* Problem Identification: We began by analyzing real-world concerns around secure messaging, especially the growing tension between usability and privacy. Stakeholder interviews and research revealed gaps in existing systems, particularly the absence of seamless, encrypted, real-time communication in one unified platform.
* Analysis & Requirements Gathering: At this stage, we defined use cases, selected encryption standards, and scoped the architecture. A key decision was to use AES-256 for message encryption and RSA/ECC for key exchange, prioritizing both performance and proven cryptographic reliability.
* Design: We architected the system to separate concerns cleanly between frontend, backend, communication protocol, and database layers. Diagramming tools helped visualize data flow and interactions early on, reducing ambiguity in later phases.
* Implementation: Development was done iteratively using agile principles. Frontend (React.js) and backend (Node.js + Express) components were built in parallel, while WebSocket integration served as the backbone for real-time communication.
* Testing & Validation: Unit and integration testing were performed regularly, with emphasis on cryptographic integrity and latency checks. Encrypted message payloads were manually validated to ensure security goals were being met.
* Deployment & Debugging: MongoDB Atlas and secure socket layers (SSL/TLS) were deployed in staging environments. We also stress-tested the WebSocket connection to ensure the system could handle multiple real-time interactions.

**Project Phases & Highlights:**

* Balancing real-time speed with cryptographic overhead proved tricky. Optimizing encryption and decryption cycles without delaying message delivery required careful tuning.
* Ensuring seamless WebSocket reconnection on network drops demanded extra logic on both client and server sides.
* Designing an intuitive UI without exposing cryptographic complexity to users was both an art and a technical exercise.

Despite these challenges, each stage was successfully executed with active collaboration and regular code reviews. We learned to manage scope effectively and pivot when design assumptions didn’t align with technical feasibility.

References

<List **any other documents or Web addresses** to which this TeDPR refers. Include the “document-management/source-code control system” (DM-SCCS) address. These references may also include user interface style guides, contracts, standards, system requirements specifications, use case documents, or a vision and scope document. Provide enough information so that the reader could access a copy of each reference, including title, author, version number, date, and source or location.>

1. Requirements Management
2. **Do not delete… Bug in document**

Requirements Development Perspective

The Real-time Cryptographic Messaging System is a newly developed, standalone application that was conceptualized and implemented specifically to address the increasing demand for secure and real-time communication. It is not a follow-up to any existing product, nor is it a component of a larger software family. From its inception, the project was envisioned as a lightweight yet robust messaging platform that delivers both performance and end-to-end encryption in a single solution.

The requirements for this system emerged from a combination of cybersecurity research, user needs assessments, and technical feasibility evaluations. Existing platforms were studied to identify where they fell short, particularly in balancing privacy with speed. Stakeholder concerns highlighted the need for strong encryption standards, minimal latency, user-friendly interfaces, and scalable architecture. These needs drove the system's core requirements, such as using AES-256 for symmetric encryption, RSA-4096 or ECC for secure key exchange, and WebSockets for persistent real-time communication.

As the project progressed, certain areas were fully specified and implemented with confidence, such as the cryptographic core, the WebSocket messaging flow, and the basic user interface. However, some areas—like group messaging, file sharing, and offline message synchronization—were scoped out as future improvements and may require additional requirement specification in later versions.

Use Characteristics

User Classes and Characteristics

The Real-time Cryptographic Messaging System is designed for a range of user classes, with distinct needs and usage patterns. The primary user group consists of standard users, who use the system for secure and private personal or professional communication. These users are typically not technically inclined but highly value privacy and simplicity. They expect that encryption is handled seamlessly in the background and that their messages are delivered without noticeable delay.

Another class of users includes power users, such as developers, security-conscious individuals, or IT professionals. These users are more likely to examine how encryption is implemented and prefer transparency in system behavior. They may seek assurance that cryptographic standards like AES-256 and RSA/ECC are correctly applied and may be interested in auditability or advanced configuration settings.

Finally, there are administrative users, who manage the backend infrastructure and monitor server performance, logs, and data security. While this class does not interact with the front end regularly, their responsibilities are crucial for maintaining the integrity and operational uptime of the system. These users must ensure that encrypted data in the database is protected, WebSocket connections are stable, and server endpoints are secured with TLS.

Use-Case Model Survey

Several core use cases define the interaction between users and the system. One of the most fundamental use cases is user registration and login, where users create secure accounts and authenticate through encrypted credentials. After authentication, users can perform friend request management, allowing them to establish secure communication channels with approved contacts.

Once a connection is established, users engage in real-time encrypted messaging. This use case represents the heart of the system. Each message is encrypted on the sender’s end using AES-256 and transmitted via WebSocket. The server does not decrypt or store plaintext messages; instead, it forwards the encrypted payload to the recipient in real time.

Another critical use case is message integrity validation, which employs SHA-256 hashing to detect tampering or transmission errors. This ensures that what the sender sends is exactly what the receiver reads.

Finally, secure key exchange occurs through RSA-4096 or ECC, depending on configuration. These cryptographic protocols allow the two communicating parties to safely exchange keys without risking exposure to eavesdroppers.

Figure . Maybe something else

The use-case model in Figure 2 (to be included) will illustrate these interactions, depicting the main actors—users and the backend system—and the various actions they perform within the application.

Figure . This is another Caption

Figure 3 presents an architectural overview of the Real-time Cryptographic Messaging System. It illustrates the client-server interactions, encryption mechanisms, and real-time messaging pathways. The client, built with React.js, handles message encryption and key generation before sending data through secure WebSocket connections. On the server side, Node.js processes messages and routes them accordingly, while MongoDB stores all data in encrypted form. The architecture is designed to ensure secure, low-latency communication using end-to-end encryption and industry-standard cryptographic practices.

User Documentation

The system includes user documentation designed to simplify onboarding and usage for both technical and non-technical audiences. A PDF Quick Start Guide is provided with every deployment. It outlines key steps such as account creation, adding friends, sending encrypted messages, and interpreting system feedback messages.

To supplement the guide, in-app contextual help is integrated into the user interface. This help feature provides short tooltips and explanations on forms, buttons, and error messages, making it easy for new users to understand each feature without needing external resources.

In future iterations, the system may also include a video tutorial, which will walk users through message encryption and how real-time delivery is handled using WebSockets. All documentation follows a non-technical, clean design that keeps user experience at the forefront

Feature Attributes

In the development of this system, each feature was analyzed and characterized based on key attributes to help guide prioritization, scope, and implementation planning. These attributes provide useful metadata about how features behave, their status, and their significance in the overall system.

Although this section is often optional in some projects, we have included three representative attributes that were most relevant to our decision-making process: Stability, Priority, and Implementation Status.

Stability

Stability refers to how likely a feature is to remain unchanged as the system evolves. In our case, foundational components such as the AES-256 message encryption and WebSocket-based real-time messaging are considered highly stable. These are core features that underpin the system's security and functionality, and there is minimal likelihood that they will require modification in future releases.

On the other hand, features like group messaging and encrypted file sharing—which are planned for future development—are currently marked as less stable. These features will depend on user feedback and further analysis of security implications before finalizing their implementation structure.

Priority

Priority is an attribute that helped us determine the order in which features were developed and tested. High-priority features such as user authentication, message encryption, and WebSocket communication were implemented first, as they form the critical backbone of the secure messaging system.

Medium-priority features included friend request management and real-time message integrity validation, both of which were essential for usability but could be developed after core messaging functions. Lower-priority features, like admin dashboards or visual encryption indicators in the UI, were either scoped for later or flagged for possible future versions.

By clearly assigning priorities, we were able to manage project scope effectively, ensuring that must-have functionality was delivered on time without sacrificing quality.

Implementation Status

Implementation status captures whether a feature has been completed, is still under development, or is planned for a future release. At the time of writing, all core messaging and security features have been fully implemented and tested, including the frontend/backend communication, message encryption, and storage handling.

Some features, such as enhanced error reporting, message delivery acknowledgments, and offline sync, are in development or exist in partially functional prototypes. Others, like group messaging and file attachments, are on the project roadmap but have not been implemented yet.

Documenting implementation status helped us maintain clarity and transparency throughout development and review cycles. It also laid a foundation for planning future iterations of the system.

Key System Features

The Real-time Cryptographic Messaging System includes several core features that enable secure, real-time communication between users. These features are modular yet interconnected, each serving a specific purpose within the larger cryptographic and messaging framework. In this section, we describe the key features, along with their corresponding classification attributes and functional requirements.

Encryption Real-Time Messaging

Description

At the heart of the system is its encrypted real-time messaging capability. This feature allows users to exchange messages securely and instantly. Messages are encrypted on the client side using the AES-256 algorithm before being transmitted over WebSockets to the server, which then routes the message to the recipient without ever accessing the plaintext. Upon arrival, the recipient decrypts the message locally using a previously exchanged secure key.

This feature ensures confidentiality, integrity, and minimal latency in communication, enabling users to hold private conversations without performance compromises.

Attribute Classification

This feature is classified as high priority, high stability, and fully implemented. It represents the central function of the system and is unlikely to change significantly across future versions.

Key Functional Requirements

REQ-1: The system shall allow users to send encrypted messages using AES-256 before data leaves the client device.

REQ-2: The system shall transmit messages over a persistent WebSocket connection to maintain real-time performance.

REQ-3: The system shall ensure that only the intended recipient can decrypt the message using a private key.  
REQ-4: The server shall act only as a message router and never store or access plaintext message content.

REQ-5: The system shall deliver encrypted messages with a latency not exceeding 200 milliseconds under standard network conditions.

Secure Key Exchange

Description

Before users can communicate securely, they must perform a secure key exchange. This feature enables users to generate and share cryptographic keys using either RSA-4096 or Elliptic Curve Cryptography (ECC). The server facilitates the key exchange but does not store or interpret any key material. Keys are shared securely and used only for decrypting messages on the client side.

This mechanism underpins the system’s end-to-end encryption model and ensures that even if communications are intercepted, the messages remain unreadable.

Attribute Classification

This feature is considered high priority, medium-to-high stability, and fully implemented in the current release. Any future adjustments may involve offering users the choice of additional algorithms or more complex key negotiation protocols.

Key Functional Requirements

REQ-1: The system shall generate public-private key pairs upon user registration or first-time login.

REQ-2: The system shall use RSA-4096 or ECC for secure transmission of session keys between users.

REQ-3: The server shall never store or have access to users’ private keys.

REQ-4: The system shall verify the integrity of key exchange messages using SHA-256 hashing.

REQ-5: In the event of a failed key exchange, the system shall alert users and request a retry without exposing key material.

Friend Request Management

Description

The friend request management feature provides the foundation for user-to-user communication. Users must first establish a trusted connection by sending and accepting friend requests before they are allowed to exchange messages. This model reduces the risk of spam, impersonation, or unauthorized communication.

Friend relationships are stored securely in the MongoDB database, and communication between users is only enabled when both parties have mutually accepted the connection.

Attribute Classification

This feature is classified as medium priority, high stability, and fully implemented. While enhancements may be made to improve UI/UX, the underlying logic is well-established.

Key Functional Requirements

REQ-1: The system shall allow users to send and receive friend requests.

REQ-2: The system shall prevent messaging between users until a friend request has been mutually accepted.

REQ-3: Friend list data shall be stored securely in MongoDB, linked to each user’s profile.

REQ-4: The system shall provide real-time updates when friend requests are accepted or declined.

REQ-5: The server shall log all request-related actions for audit and debugging purposes, without storing message content.

Key Design and Implementation Constraints

Every software system operates within a set of constraints that affect how it is designed, implemented, and deployed. The Real-time Cryptographic Messaging System is no exception. From encryption requirements to infrastructure dependencies, several factors guided the development process and shaped the final architecture.

First and foremost, the decision to use end-to-end encryption as a default mode for all communications introduced constraints related to data handling and performance. The system was required to ensure that no plaintext messages were ever stored or even accessible by the backend server. This dictated a strict separation of concerns, where the frontend client was responsible for all encryption and decryption operations. As a result, additional effort was required to manage secure key exchanges and prevent key leakage across the network.

Another design constraint was the use of WebSockets for real-time communication. While WebSockets provide fast, persistent connections that are ideal for instant messaging, they also require careful handling of connection lifecycles, error states, and reconnection logic. Building a stable WebSocket infrastructure meant ensuring consistent behavior across various devices and network conditions, especially in scenarios involving packet loss or server downtime.

Technology choices also introduced constraints. The system was built using Node.js and Express on the backend, React.js on the frontend, and MongoDB as the database. These tools were chosen for their performance, scalability, and ease of integration—but they also influenced architectural patterns and limited our options in terms of synchronous operations, transaction handling, and data modeling. For example, MongoDB’s document-based storage model required us to serialize encrypted payloads in a way that was compatible with BSON and could still support searchability for future extensions.

Security compliance was a major constraint throughout development. All data in transit had to be secured using TLS/SSL encryption, and user credentials had to be stored using strong, salted hashing functions. These policies were non-negotiable and dictated how authentication flows, session management, and password resets were implemented.

Finally, time and scope limitations influenced feature depth. While the roadmap includes features like group messaging and file sharing, these were postponed to ensure the delivery of a secure and stable core system within the given timeframe. Prioritizing foundational encryption and real-time messaging allowed the project to meet its primary objectives while setting the stage for future enhancements.

Operating Environment

The Real-time Cryptographic Messaging System is designed to run in a modern web-based environment, leveraging cloud infrastructure and cross-platform technologies to ensure flexibility and accessibility. The client application is built using React.js, and is intended to operate seamlessly in all major web browsers including Chrome, Firefox, and Edge. It is also compatible with desktop-class web runtimes like Electron, should a standalone desktop version be developed in the future.

On the backend, the system is powered by Node.js (v18+) with the Express framework, hosted in a secure server environment. It is optimized to run on Linux-based distributions (e.g., Ubuntu Server 22.04), though it remains portable to Windows Server and macOS environments with minor configuration changes. Backend operations require support for WebSocket connections, TLS certificates, and asynchronous request handling.

The database layer relies on MongoDB, preferably hosted on MongoDB Atlas or similar cloud-based services that offer automatic scaling, encrypted storage, and replica sets for high availability. All sensitive data stored in MongoDB is encrypted prior to insertion, ensuring that even in the event of unauthorized access, message contents remain unreadable.

The full stack requires Node Package Manager (NPM) for dependency management, and build tools such as Webpack and Babel for frontend bundling. The environment must support HTTPS traffic and allow persistent WebSocket connections over secure channels (WSS). Additionally, hosting infrastructure must provide TLS/SSL certificates for all server endpoints and enforce HTTPS redirection for client-side applications.

For development and testing purposes, the system was deployed using local Docker containers and later migrated to a cloud testbed. In a production setting, the solution is ideally deployed using NGINX as a reverse proxy, Let’s Encrypt for SSL certificates, and PM2 for managing Node.js services in a clustered environment.

Interface Requirements

User Interfaces

The Real-time Cryptographic Messaging System includes a clean, intuitive user interface built with React.js. The interface is designed to reduce complexity for users while ensuring all communication remains secure. Upon accessing the system, users are greeted with a login or registration screen, both of which provide immediate feedback for success or errors, and enforce strong password policies.

Once logged in, users can view a dashboard displaying their friend list, active conversations, and system notifications. The messaging interface is modeled after familiar chat layouts, with text input fields, message timestamps, and delivery indicators. While encryption occurs in the background, the user is subtly informed that their conversation is secure via lock icons and tooltip indicators.

The system adheres to UI design principles such as minimalism, clarity, and responsiveness. All interactive elements—like buttons, dropdowns, and modals—are accessible via both mouse and keyboard inputs. Visual feedback and error states (e.g., failed message delivery) are clearly displayed, making the experience seamless even in cases of network interruptions.

Though the interface is web-based, it has been styled with mobile responsiveness in mind using CSS Flexbox/Grid and media queries, making it functional across desktops, tablets, and smartphones.

Hardware Interfaces

The system is entirely software-driven and does not directly interface with any physical hardware components outside of the user’s standard input/output devices (e.g., keyboard, display, network interface). No custom sensors, embedded modules, or hardware-level encryption chips are required to operate this system.

Software Interfaces

The system is composed of modular software components that interact through well-defined APIs and communication protocols. The frontend React application interfaces with the backend Node.js server using RESTful APIs for operations such as registration, login, friend requests, and fetching message history. Real-time messaging is handled through WebSocket connections, which allow for bi-directional communication without requiring repeated HTTP requests.

The backend integrates with MongoDB for persistent data storage. Collections include encrypted message data, user profiles, and friend request statuses. The server uses Mongoose (an ODM library) to enforce schema structures and data validations.

For cryptographic operations, the backend relies on established libraries such as crypto, bcrypt, and jsonwebtoken for hashing, password storage, and session handling. These components work together to provide secure and extensible architecture.

In a full production deployment, the backend is expected to run behind an NGINX reverse proxy, which handles HTTPS connections and routes requests securely to backend services.

Communications Interfaces

The system depends on secure and reliable communication channels to support encrypted, real-time message delivery. All REST API requests and WebSocket connections are transmitted over TLS/SSL, ensuring data in transit is protected from interception or tampering.

The messaging protocol itself uses WebSocket (WSS), which establishes a persistent connection between the client and the server. This protocol is ideal for real-time communication and reduces latency compared to traditional polling mechanisms.

Messages transmitted through this channel are AES-256 encrypted on the client side and include SHA-256 hash signatures for validation upon receipt. If the hash does not match the original, the message is rejected, and an integrity alert is generated.

The system is built to comply with modern security practices and will not operate unless secure protocols (HTTPS/WSS) are in place. Additionally, timeouts and heartbeat mechanisms are implemented to detect dropped connections and ensure reliability in message delivery.

Nonfunctional Requirements

Nonfunctional requirements define the quality standards and constraints under which the Real-time Cryptographic Messaging System must operate. These requirements cover aspects like performance, security, software quality, and operational safety—ensuring that the product is reliable, usable, and compliant with industry best practices.

Performance Requirements

Performance was a critical consideration throughout the design of the system. Since the core promise of the application is secure real-time communication, message delivery latency had to be minimal. Under typical usage conditions and average internet bandwidth, the system is designed to deliver encrypted messages with an average latency of less than 200 milliseconds.

The system also supports concurrent user sessions with efficient resource management. WebSocket connections are kept persistent, allowing users to remain connected for long durations without significant memory overhead. The backend architecture is event-driven, minimizing blocking operations and improving throughput.

MongoDB was chosen for its high read/write speed and ability to scale horizontally. Indexing strategies and schema optimization ensure that fetching conversation histories or managing friend requests does not introduce noticeable delays even as the user base grows.

Security Requirements

Security is the foundation of this system and influences every design and implementation decision. All messages are encrypted using AES-256 before transmission. Keys are exchanged securely using RSA-4096 or ECC, depending on the configured cryptographic method. Once received, messages are decrypted locally on the recipient’s device.

The system enforces end-to-end encryption by design. No plaintext messages are ever stored or processed on the backend. Key management is decentralized, meaning the server cannot decrypt or access private messages.

To protect user credentials, passwords are hashed using bcrypt with salt, and all authentication requests are processed over HTTPS. Session tokens are implemented using JWTs (JSON Web Tokens) with expiration times to prevent hijacking.

In addition, communication channels—both HTTP and WebSocket—are secured with TLS 1.2 or higher, and the application enforces HTTPS-only access in production deployments.

Software Quality Attributes

The system emphasizes several key quality attributes to ensure both usability and maintainability:

* Reliability: Extensive testing ensures message delivery remains consistent, even across network interruptions.
* Maintainability: The codebase follows modular principles and clean architecture patterns, allowing for easy future updates and bug fixes.
* Usability: The React-based frontend prioritizes simplicity and clarity, ensuring even non-technical users can navigate the app comfortably.
* Portability: The application stack (Node.js, React, MongoDB) is fully cross-platform and can be deployed on various operating systems with minimal adjustments.
* Scalability: The infrastructure is cloud-compatible and supports scaling both vertically and horizontally, accommodating growth in user base and message traffic.

These attributes collectively contribute to a robust, user-centric system that is both secure and practical for real-world use.

Safety Requirements

Although the application does not involve physical systems where safety risks are typical, data safety and system integrity are prioritized. All sensitive data is encrypted in transit and at rest, minimizing risk even in the event of a system breach.

Error handling mechanisms ensure that failed or tampered messages do not disrupt the user experience or compromise the system’s security. Users are notified in case of integrity check failures, and any suspicious activity (such as repeated failed decryption attempts) can trigger alerts or automatic session invalidation.

The system is also designed to prevent common attack vectors, such as SQL injection (though not applicable to NoSQL), cross-site scripting (XSS), and cross-site request forgery (CSRF), through secure coding practices and appropriate middleware.

Other Requirements

The application was developed with internationalization and accessibility in mind. Although the current release is English-only, it is structured to support multilingual text rendering and right-to-left layout compatibility in future iterations.

In terms of legal and ethical considerations, the system is compliant with standard data protection guidelines such as GDPR principles, emphasizing minimal data retention and user consent. It is also structured to enable future enhancements such as audit logging or compliance reporting without redesigning the core architecture.

Assumptions and Dependencies

The development and functionality of the Real-time Cryptographic Messaging System are based on several assumptions and are dependent on certain environmental and third-party factors. These assumptions guided our implementation decisions and defined the boundaries within which the system was expected to operate reliably.

One key assumption is that users will access the system through modern, standards-compliant web browsers such as Chrome, Firefox, or Edge. The React.js frontend is optimized for these environments, and while mobile browser compatibility is considered, the primary interface is designed with desktop interactions in mind.

Another assumption is that network conditions will be generally stable, allowing for persistent WebSocket connections. While reconnection logic is built into the system, long-term offline usage or severely unstable networks may degrade the user experience. The real-time nature of the system depends on uninterrupted access to the internet, particularly for maintaining live chat sessions.

The application also assumes the presence of a secure server infrastructure. The backend relies on TLS certificates for encrypted data transport and must be hosted on platforms that support HTTPS and WebSocket Secure (WSS) protocols. This includes access to valid SSL certificates—either commercial or via providers like Let's Encrypt.

Dependencies include core technologies like Node.js, Express, MongoDB, React, and WebSocket libraries. The system also depends on open-source cryptographic libraries for AES, RSA/ECC, bcrypt, and JWT handling. Any significant changes in these libraries or deprecated APIs could necessitate updates or patches to maintain compatibility and security.

Lastly, the system assumes a trusted deployment environment. For example, MongoDB is assumed to run with secure access controls, firewall protection, and encrypted storage settings enabled. If deployed in a shared or unmanaged hosting environment, additional security hardening may be required to prevent data leaks or unauthorized access.

Being aware of these assumptions and dependencies allows us to better prepare for future upgrades and to anticipate the potential limitations of the system under atypical usage conditions.

1. Design

Introduction

This section presents the design architecture and internal structure of the Real-time Cryptographic Messaging System. It includes the logic behind data handling, software modules, system interfaces, and user interaction flows. Each component was intentionally designed to balance performance, security, and maintainability.

From the beginning, we knew this system needed to perform end-to-end encryption at scale while remaining responsive and easy to use. Designing such a system involved navigating a number of technical trade-offs—such as where to handle encryption (client vs. server), how to manage persistent connections, and how to store messages securely without compromising availability. Every major design decision was informed by these constraints, as well as our commitment to privacy-by-design principles.

This section outlines how we arrived at a modular, layered architecture built with a clear separation of concerns between the frontend, backend, and data storage layers. It also documents key patterns used to reduce technical debt and enable future growth.

Goals and Objectives of Design

The primary goal of this project was to deliver a secure real-time messaging application that protects users’ data from unauthorized access and tampering while maintaining the responsiveness and usability of mainstream chat platforms. Specifically, we aimed to:

* Implement true end-to-end encryption, ensuring that only the sender and receiver could access message content.
* Enable real-time message delivery using persistent WebSocket connections, eliminating the need for polling or manual refresh.
* Use modular and scalable architecture that separates user interface, application logic, and data storage.
* Design an intuitive, non-technical user experience that hides the complexity of encryption while still promoting transparency and trust.
* Prioritize security-first practices in data handling, key management, and transport protocols.

Each of these goals shaped the selection of tools, the breakdown of responsibilities between system layers, and the structure of communication flows between users and services.

Statement of Software Scope

The Real-time Cryptographic Messaging System is a web-based application that allows registered users to communicate securely and instantly. The system covers the full cycle of secure communication: user authentication, key exchange, friend management, encrypted messaging, and message integrity verification.

The scope includes both the frontend client, where users interact with the system, and the backend server, which routes encrypted data and manages persistent user relationships. The backend connects to a MongoDB database that stores encrypted messages, friend lists, and authentication records.

Core inputs include user-submitted data (e.g., messages, friend requests, login credentials), and cryptographic material (keys, hashes). Processing includes AES encryption/decryption, RSA/ECC key exchange, hash validation, and WebSocket routing. The outputs are real-time message updates, friend status changes, and system feedback messages (e.g., message delivery confirmation, error handling).

The current version is limited to one-on-one encrypted chats, but is designed to be extended to include group messaging and encrypted file sharing in future versions.

Software Context

The system exists as a secure alternative to common messaging platforms, especially those that lack true end-to-end encryption or rely on centralized message processing. It’s designed to operate in both academic and real-world settings—usable by security-conscious individuals, researchers, or small organizations seeking private communication tools.

The broader context of the software includes:

A modern cloud infrastructure that enables secure deployment using HTTPS and TLS.

Open-source cryptographic libraries and communication protocols that integrate seamlessly with our tech stack.

Compatibility with various user devices and operating systems, ensuring the application can reach users wherever they are.

This system can also be deployed in internal corporate environments or federated networks where centralized control is undesirable or where compliance with data protection laws is required.

Major Design Constraints

The project operated under a number of technical and practical constraints that influenced design decisions. Chief among them was the requirement to implement end-to-end encryption on the client side, which removed the possibility of processing or indexing plaintext messages on the server. This limited options for features like content-based notifications or search, which are typical in unencrypted messaging systems.

Another constraint was real-time communication, which required the use of WebSockets instead of traditional HTTP request-response patterns. Ensuring low latency and reliability under variable network conditions required extra care in connection management and event handling.

There were also constraints on storage and data access. Since MongoDB stores only encrypted messages, no traditional filtering or content analytics can be performed on message bodies, reinforcing the security-first approach but introducing challenges in user experience customization.

Finally, time and scope limitations required us to defer certain advanced features like multi-device sync, secure media sharing, and full audit logging for future versions.

Data Design

The Real-time Cryptographic Messaging System’s data architecture was designed with a strong emphasis on security, scalability, and simplicity. This section outlines how the system structures and handles data both in-memory and in persistent storage. The separation between internal, global, and temporary data structures ensures modularity, easy debugging, and optimal performance across all system layers.

Major Internal Software Data Structure

The system processes messages using internal objects that encapsulate key identifiers, encrypted content, and metadata. These message objects are used during runtime and then passed into MongoDB for persistent storage. The internal structure reflects the database schema, ensuring consistent serialization and deserialization.

An example of the internal `MessageObject` includes the following fields:

```json

{

"\_id": "67eea02341c2428b04e2633c",

"senderId": "67d158bd5839a1e68237384a",

"receiverId": "67d158865839a1e682373843",

"text": "835f4a7194b7c91bac7c8c0aa95beecf:6638e0cd53bed50daea5a8814d176e61b736f…",

"createdAt": "2025-04-03T14:50:11.335+00:00",

"updatedAt": "2025-04-03T14:50:11.335+00:00",

"\_\_v": 0

}

```

The `text` field contains the encrypted message payload, which includes a colon-separated format: an AES-encrypted string combined with a corresponding integrity hash (e.g., SHA-256). Both sender and receiver IDs are stored as ObjectId references to ensure referential integrity.

Global Data Structure

The primary global data structure in the system is the \*\*User schema\*\*, which defines essential user attributes and is accessible throughout the backend for operations like authentication, message routing, and friend management.

Below is the actual schema used in the application:

```javascript

const userSchema = new mongoose.Schema(

{

email: {

type: String,

required: true,

unique: true,

},

fullName: {

type: String,

required: true,

},

password: {

type: String,

required: true,

minlength: 6,

},

profilePic: {

type: String,

default: "",

},

},

{ timestamps: true }

);

```

This schema ensures that each user is uniquely identified by their email address and stores only essential information for login and identification. Passwords are hashed before storage using `bcrypt`, and profile pictures (if uploaded) are stored as file paths or URLs. The `timestamps` option automatically manages `createdAt` and `updatedAt` fields.

Temporary Data Structure

Temporary data structures are used during transient operations that require sensitive, short-lived data handling. These include one-time tokens for session validation, nonce values during key exchange, and WebSocket connection states.

For instance, when a user initiates a key exchange, a temporary in-memory structure holds a cryptographic nonce and public key while awaiting a response from the peer client. Once the exchange is completed, this data is immediately discarded. This approach minimizes the time sensitive information exists in memory, reducing the attack surface in case of compromise.

Another use of temporary data structures occurs during user authentication. After a successful login, a signed JSON Web Token (JWT) is created and sent to the frontend. This token is stored client-side and used for session verification, but is not stored server-side, keeping the backend stateless and reducing session management overhead.

These temporary objects help the system maintain secure state transitions while preserving performance and avoiding unnecessary persistence of ephemeral data.

Database Description

The system uses MongoDB as the backend database, managed through Mongoose, a popular object data modeling (ODM) library. MongoDB’s document-oriented structure provides flexibility for evolving schemas while ensuring high performance and scalability.

There are three primary collections in the system:

**3.2.4.1. Users Collection**

This collection stores essential user information. Each user record contains:

* Email (unique identifier)
* Full name
* Hashed password (using bcrypt)
* Optional profile picture path
* Automatic timestamps (`createdAt`, `updatedAt`)

This information supports authentication, account management, and UI personalization.

**3.2.4.2. Messages Collection**

All messages are encrypted before being saved. Each message document includes:

* `\_id`: MongoDB-generated unique identifier
* `senderId` and `receiverId`: ObjectIds referencing users
* `text`: A colon-separated string combining AES-encrypted content and SHA-256 hash
* `createdAt` and `updatedAt`: Timestamps for logging and sorting
* `\_\_v`: A versioning field automatically managed by Mongoose

The database never stores or accesses plaintext message content, preserving end-to-end encryption.

**3.2.4.3. Friends Collection**

This collection tracks the relationship status between users—pending, accepted, or rejected. Each record connects two user IDs and logs timestamps to support mutual friend verification.

MongoDB’s schema-less structure is constrained using Mongoose models to enforce field types and required attributes. Indexes are applied to fields like `senderId`, `receiverId`, and `createdAt` to optimize query performance for message history and conversation loading.

The full Entity-Relationship Diagram (ERD) and relational mapping schema are provided in Appendix D, visualizing the logical connections among users, messages, and friends.

Architectural and Component-Level Design

Program Structure

The Real-time Cryptographic Messaging System follows a modular, component-based architecture using a client-server model. It is structured into three primary layers:

* Client Layer (Frontend): Built using React.js, this layer is responsible for user interaction, real-time message display, and client-side encryption/decryption of messages. It connects to the backend via both HTTP (for RESTful APIs) and WebSocket (for real-time messaging).
* Server Layer (Backend): The backend, developed using Node.js and Express.js, handles routing, user authentication, session management, and real-time communication using WebSocket protocols. The server enforces all security protocols and connects users based on authentication tokens.
* Data Layer (Database): MongoDB acts as the system’s persistent data store. It houses all encrypted message data, user profiles, and friend relationships. Data is stored in structured collections and accessed securely via Mongoose models.

Architecture diagram

A pictorial representation of the architecture is presented.

Key Software Components

<A detailed description of key software components contained within the architecture is presented. The section is repeated for each of n components.>

Processing Narrative for <component name>

<Replace the <component name> in the title with the name of the component in the system. Provide a processing narrative for the component. Describe its role in the system.>

<Continue adding individual “Processing Narrative” sections for each key software component.>

<Address the following points in a degree appropriate to your project.>

* Component n interface description.

A detailed description of the input and output interfaces for the component is presented.

* Sub-Component n.m processing detail

A detailed algorithmic description for each sub-component within the component n is presented.

Repeat section; describe for each of the m sub-components of component n.

Optionally, choose to add the following elements, also.

* + **Interface description:** A description of sub-component m inputs and outputs is presented.
  + **Algorithmic model:** The pseudocode listing for sub-component m is presented.
  + **Restrictions/limitations:** The external environment and/or infrastructure that must exist for sub-component m to operate correctly is provided.
  + **Local data structures:** The data structures used within sub-component m are presented.
  + **Performance issues:** Information on topics that may affect the run-time performance, security, or computational accuracy of this sub-component are presented.
  + **Design constraints:** Attributes of the overall software design (including data structures, OS features, I/O, and interoperable systems) that constrain the design of this sub-component are presented.>
* External machine interfaces

Interfaces to other machines (computers or devices) are described.

* External system interfaces

Interfaces to other systems, products, or networks are described.>

User Interface Design

*<A description of the user interface design of the software is presented. In the following sections>*

Description of the User Interface

*<A detailed description of user interface including screen images or prototype is presented.*

Screen images

*<Discussion of the interface from the user's point of view.>*

Objects and actions

*<All screen objects and actions are identified. >*

Interface Design Rules

*<Conventions and standards used for designing/implementing the user interface are stated.>*

Components Available

*<GUI components available for implementation are noted.>*

User Interface Design Description

*<The user interface development system is described.>*

Restrictions, Limitations, and Constraints

*<Special design issues which impact the design or implementation of the software are noted here.>*

1. Verification and Validation

Test items

*<Summarize the software items (programs, modules, classes) and software features tested. The need for each item and its version/revision history may be included. Also specify characteristics of their transmittal media that impact hardware requirements or indicate the need for logical or physical transformations before testing can begin (e.g., programs must be transferred from tape to disk). Items that are to be specifically excluded from testing may be identified. >*

***<****In a narrative sections, focus on the key tested features. Address the points below.>*

***Features tested***

*Identify all software features and combinations of software features tested. Provide reference to requirements for each feature.*

***Features not tested***

*Identify all features and significant combinations of features that were not tested and the reasons.*

***Item pass/fail criteria***

*Specify the criteria used to determine whether each test item has passed or failed testing.*

***Testing tasks***

*Identify the set of tasks necessary to prepare for and perform testing. Identify all intertask dependencies and any special skills required.*

***Environmental needs***

*Specify both the necessary and desired properties of the test environment. This specification should contain the physical characteristics of the facilities including the hardware, the communications and system software, the mode of usage (e.g., stand-alone), and any other software or supplies needed to support the test.*

*Also specify the level of security that must be provided for the test facilities, system software, and proprietary components such as software, data, and hardware.*

*Identify special test tools needed. Identify any other testing needs (e.g., publications or office space). Identify the source for all needs that are not currently available to the test group.*

***Responsibilities***

*Identify the groups/person managed, designed, prepared, executed the test tasks.*

*These groups may include the developers, testers, operations staff, user representatives, technical support staff, data administration staff, and quality support staff.*

***Test Result Summary***

*Summarize the results of testing.*

*In an appendix, for each feature/combination of features tested, provide test cases used and the results of test cases, whether there are issues raised from the test result and the resolution of issues.>*

1. Conclusion

<The **Conclusion** should constitute at least four paragraphs. More are acceptable, but the material suggested is to be addressed.

<Paragraph: Identify the result of the research effort as either “successful” or “requiring further work.” Indicate major tests and their results to support the conclusion.

<Paragraph: Compare the final state of the work with the original intent sought when the project was proposed. Include how the project could be extended into further versions.

<Paragraph: Discuss the ethical elements of the product. Include such items as privacy aspects, stewardship needs, security concerns, data access and integrity perspectives. Describe how your product attended to these ethical elements. Include points where ethical features remain to be addressed.

<Paragraph: Describe realization or potential for realization of benefits to the organization. Indicate how the product enables these benefits.

*< Each appendix should appear. Each appendix should begin on a new page.*

*The text of an appendix should be in Calibri, 11 pt. font, double-spaced.*

*Even if no content should be included in an appendix, the appendix page should still appear. For instance, the project may not have a human interface component (Appendix F). Then the appendix title appears with the text: Not applicable.>*

1. Bibliography

|  |  |
| --- | --- |
| [1] | J. Doe, "Strange papers are worth citing," in *IEEE Conference on Strange Report Writing*, TimbukTu, 1995. |

< Using **IEEE citation format**, list all sources referenced to develop the work or explicitly cited in the writing. Online support for the format is available, EndNote supports it as does RefWorks. Note that you should reset the Style of the “Bibliography” section to “Heading 1” to ensure that it works with the TOC.>

Use the "Insert Citation" button to add citations to this document.

Appendix A: Glossary

< Define and alphabetize all the terms necessary to properly interpret the writing, including abbreviations, acronyms, and initialisms with their expanded terms and meanings. The reader should find any term – particularly abbreviations – used in the writing in the list. The format of each term should be similar to the following. Place sufficient tabs between the list of abbreviations and the terms so that the colons align; double space between abbreviations.

ACM : Association for Computing Machinery

: A professional organization of the computing field. ACM professionals provide guidance to standards-committees and to government in defining and understanding computing technology.

CIS : Computer and Information Science

: Department conferring graduate degree

TCP/IP : Transmission Control Protocol / Internet Protocol

: Communication protocol

>

Appendix B: Use Case Analysis

<Include use case model(s) and use case descriptions for the major, exemplary use cases developed in the project . Each separate figure and item presented in the appendix should be consecutively captioned as Figure B.N where “N” will increment beginning with 1.>

Appendix C: Analysis Models

<Include any pertinent analysis models such as data flow diagrams, class diagrams, state-transition diagrams, or entity-relationship diagrams required for a complete understanding of the research. Each separate figure and item presented in the appendix should be consecutively captioned as Figure C.N where “N” will increment beginning with 1>

Appendix D: Design Models

< Include any pertinent design representations such as relational schemas, data dictionary, physical data flow diagrams, or navigation sequence diagrams Each separate figure and item presented in the appendix should be consecutively captioned as Figure D.N where “N” will increment beginning with 1.>

Appendix E: Testing Log and Summary Status

< Include any pertinent testing artifacts such as detailed test cases and test results. Each separate figure and item presented in the appendix should be consecutively captioned as Figure E.N where “N” will increment beginning with 1.>

Appendix F: Screen Captures

< Include screen images if the human interface components. Each separate figure and item presented in the appendix should be consecutively captioned as Figure F.N where “N” will increment beginning with 1.>

Appendix G: Project File Repository Definitions

< Include a listing of the directories and file designations used by the document management or source-code control system in order to reference the documentation and/or source code of the project.>