International Journal of Scientific Research in Engineering and Management (IJSREM)
Volume: 10 Issue: 04 | April - 2026 SJIF Rating: 8.659 ISSN: 2582-3930

End-to-End Encrypted Messaging Application with Aadhaar-Based User Authentication: Design, Implementation, and Security Analysis

Sairaj Kad • Janak Keche • Jay Kangate • Ajay Dasalkar

PRN: 72231141K PRN: 72231195J PRN: 72231170C PRN: 72230902D

Department of Computer Engineering, Sinhgad College of Engineering (SCOE)
Savitribai Phule Pune University, Pune, Maharashtra, India — 411041
Academic Year 2025–26

Abstract

The widespread adoption of instant messaging for personal, academic, and professional communication has made the security and authenticity of digital conversations a paramount concern. While end-to-end encryption (E2EE) effectively prevents eavesdropping by ensuring that only the communicating parties can read message content, it does not address the fundamental question of identity — whether the person behind an account is genuinely who they claim to be. Existing platforms such as WhatsApp, Telegram, and Signal allow account creation with any phone number, creating fertile ground for fake profiles, impersonation attacks, social engineering, and coordinated spam. This paper presents the design, implementation, and evaluation of a secure web-based messaging application that closes this gap by integrating Aadhaar-based identity verification with robust end-to-end encryption. Aadhaar — India's government-administered biometric identity system managed by the Unique Identification Authority of India (UIDAI) — provides OTP-based authentication that cryptographically links every registered account to a verified, unique individual. Message payloads are encrypted at the sender's client using the recipient's public key and decrypted exclusively at the recipient's device, ensuring that intermediary servers handle only ciphertext. The system is implemented as a full-stack web application using React and JavaScript for the frontend, Spring Boot (Java) for the backend, and MySQL for persistent storage. Real-time message delivery is achieved via HTTP long-polling, avoiding the infrastructure complexity of WebSockets while maintaining sub-second perceived latency. Session security is enforced using JSON Web Tokens (JWT). Comprehensive testing — including unit, integration, system, and user-acceptance testing — validates all functional and non-functional requirements. The evaluation demonstrates zero server-side plaintext exposure, successful blocking of duplicate and fake registrations, and consistent message confidentiality. The proposed architecture is extensible to support forward secrecy, post-quantum cryptography, mobile platforms, and AI-driven threat detection, constituting a significant step toward trustworthy digital communication aligned with India's national digital identity infrastructure.

Index Terms — End-to-End Encryption (E2EE), Aadhaar Authentication, UIDAI, Secure Messaging, Asymmetric Cryptography, RSA, AES, JSON Web Token (JWT), Spring Boot, React.js, Long Polling, Key Management, Identity Verification, Digital Privacy.

I. INTRODUCTION

A. Background and Motivation

The proliferation of internet connectivity and near-ubiquitous personal computing has transformed how people communicate globally. Instant messaging — short, asynchronous or near-real-time text exchange — has become a core medium for interpersonal, academic, and commercial interaction. According to industry reports, over three billion people use some form of messaging application every month, generating hundreds of billions of messages daily. This scale makes messaging infrastructure a critical societal resource, demanding the highest standards of privacy, security, and trustworthiness.

Modern messaging systems must simultaneously satisfy three broad and sometimes competing demands: *confidentiality* — ensuring that only intended recipients can read message content; *authenticity* — guaranteeing that participants are genuinely who they claim to be; and *usability* — providing fast, reliable, and convenient interaction that does not burden users with complex security procedures. Achieving all three at scale, while maintaining a simple user experience, is the central engineering challenge addressed by this project.

End-to-end encryption (E2EE) has emerged as the industry-standard technique for achieving confidentiality. In an E2EE system, messages are encrypted on the sender's device using cryptographic keys that are never shared with the server, and decrypted only on the recipient's device. Even if an attacker intercepts the network traffic, or compromises the messaging server itself, they obtain only meaningless ciphertext. This provides strong protection against network eavesdropping, rogue server operators, and certain classes of government surveillance.

However, E2EE fundamentally does not solve the identity problem. A malicious actor who knows a target's phone number can create an account impersonating that person, or create dozens of fake accounts to conduct coordinated harassment campaigns. Social engineering attacks — where an attacker builds a false relationship with a victim before extracting sensitive information — are equally effective on encrypted platforms, because the encryption protects the content of messages but not the legitimacy of the identity behind them. The absence of verified identity is one of the root causes of spam, phishing, romance scams, and misinformation propagation on modern messaging platforms.

B. The Aadhaar Identity Infrastructure

India's Aadhaar system, administered by the Unique Identification Authority of India (UIDAI), is the world's largest biometric identification system. As of 2024, it has enrolled over 1.38 billion residents, assigning each a unique 12-digit Aadhaar number backed by biometric data (fingerprints and iris scans) and demographic information. UIDAI provides a standardized authentication API that allows authorized entities to verify an individual's identity using either biometric matching or OTP-based confirmation — where a one-time password is sent to the mobile number registered with the Aadhaar database.

Integrating Aadhaar-level verification at the account onboarding step substantially raises the barrier against fake or multiple accounts. Since each Aadhaar number maps to exactly one real-world individual, a messaging application requiring Aadhaar verification during registration inherently limits each person to a single verified account, eliminating the sock-puppet problem at its root. This does not mean that the application needs to store or disclose users' Aadhaar numbers — the minimal-data principle requires that only a verification token or hashed reference be retained, adhering to both UIDAI regulations and the principles of privacy-preserving identity management.

C. Problem Statement

The core problem this project addresses is the absence of verified user identity in existing encrypted messaging platforms. Specifically, there is a need for a system that simultaneously guarantees: (1) message confidentiality through E2EE so that no intermediary — including the server — can read message content; and (2) identity authenticity through Aadhaar verification so that every participant is a genuine, unique, verified individual. The problem can be formally stated as:

"To design and develop a secure web-based messaging application that ensures verified user identity through Aadhaar-based OTP authentication while maintaining complete message confidentiality using asymmetric end-to-end encryption, deployed on a scalable and modular full-stack architecture."

D. Objectives

- Design a modular, layered system architecture integrating Aadhaar identity verification with asymmetric E2EE.
- Implement a Spring Boot backend that mediates UIDAI OTP verification, manages public key distribution, and relays ciphertext without ever accessing plaintext.
- Develop a React frontend that handles client-side key generation, encryption before transmission, and decryption after receipt.
- Achieve real-time messaging without WebSocket infrastructure using HTTP long-polling.
- Validate all functional and non-functional requirements through comprehensive unit, integration, system, and user-acceptance testing.
- Identify and mitigate key security threats including man-in-the-middle attacks, server compromise, and unauthorized access.

E. Scope and Limitations

The current scope encompasses web-based text messaging with Aadhaar OTP authentication and asymmetric E2EE. The system is designed for deployment in environments where UIDAI API access is available. Voice and video calling, offline message queuing (messages sent while the recipient is offline), multimedia file encryption, and mobile native applications are deferred to future work. The cryptographic implementation uses a simplified AES key-wrapping scheme suitable for academic demonstration; a production system should adopt RSA-OAEP key encapsulation with AES-256-GCM payload encryption and the Signal Protocol's Double Ratchet for forward secrecy.

II. BACKGROUND AND FUNDAMENTAL CONCEPTS

A. End-to-End Encryption (E2EE)

End-to-end encryption is a communication model in which messages are encrypted at the sender's device and decrypted only at the intended recipient's device. The critical property is that intermediate nodes — including network routers, ISPs, and the messaging server itself — handle only ciphertext and cannot recover plaintext without the recipient's private key. This contrasts with transport-layer encryption (e.g., TLS), which protects data in transit but leaves it accessible to the server in plaintext.

Modern E2EE systems typically use a hybrid cryptographic scheme: asymmetric cryptography (such as RSA or elliptic-curve Diffie-Hellman) for key exchange, and symmetric cryptography (such as AES-GCM) for bulk message encryption. This hybrid approach exploits the computational efficiency of symmetric encryption while using asymmetric cryptography for the mathematically secure exchange of symmetric session keys. The sender encrypts the symmetric key with the recipient's public key; only the recipient's private key can decrypt it to recover the session key and thus the message.

B. Public Key Infrastructure and Key Management

A Public Key Infrastructure (PKI) is the set of policies, procedures, hardware, software, and standards needed to create, manage, distribute, use, store, and revoke digital certificates and manage public-key encryption. In a messaging application, a lightweight PKI can be implemented where the messaging server acts as a public-key directory: it stores each user's public key, which other users can retrieve to encrypt messages.

Key management is a critical concern. Private keys must be generated on the client device and must never leave the device in plaintext. Public keys must be retrievable but also verifiable — a server that can substitute a user's public key with its own can mount a man-in-the-middle attack even against an E2EE system. Mitigation techniques include out-of-band fingerprint verification, key transparency logs, and certificate pinning. Forward secrecy — the property that compromise of long-term keys does not expose past messages — is achieved through ephemeral key exchange protocols such as the Double Ratchet algorithm used in the Signal Protocol.

C. Aadhaar Authentication Mechanisms

UIDAI provides two primary authentication mechanisms: (1) biometric authentication, where the user's fingerprint or iris scan is matched against the stored biometric template in the UIDAI database; and (2) OTP authentication, where a one-time password is sent to the mobile number registered with Aadhaar. For web-based applications, OTP authentication is the preferred mechanism because it does not require biometric capture hardware on the user's device.

UIDAI's authentication API follows a secure multi-step protocol: the application submits the user's Aadhaar number to the UIDAI sandbox or production endpoint; UIDAI sends an OTP to the registered mobile number; the user enters the OTP; and the application submits the OTP for verification. On success, UIDAI returns an authentication token confirming that the Aadhaar number belongs to a verified individual. Critically, UIDAI regulations prohibit applications from storing the raw Aadhaar number — only hashed or tokenized references may be retained, enforcing the minimal-data principle.

D. Real-Time Transport: Long Polling vs WebSockets

Real-time bidirectional communication in web applications can be achieved through several mechanisms. WebSockets establish a persistent TCP connection between client and server, enabling full-duplex communication with very low overhead per message. However, WebSockets require stateful server infrastructure, may be blocked by enterprise firewalls and reverse proxies, and add operational complexity.

HTTP long polling is a simpler alternative: the client sends a normal HTTP request that the server holds open until either a new message arrives or a timeout elapses. Upon receiving a response, the client immediately issues a new request, maintaining a continuous polling cycle. While slightly less efficient than WebSockets for very high message volumes, long polling works reliably through firewalls and proxies, requires no special server infrastructure, and is sufficient for the message rates typical of personal messaging applications. Spring Boot's DeferredResult mechanism provides native support for long-polling without blocking server threads.

E. JSON Web Tokens for Session Management

JSON Web Tokens (JWT) are an open standard (RFC 7519) for securely transmitting information between parties as a JSON object. A JWT consists of three Base64-encoded parts: a header specifying the algorithm, a payload containing claims (such as user ID and expiry), and a cryptographic signature.

The server signs the JWT with a secret key; any tampering with the payload invalidates the signature. JWTs are stateless — the server does not need to maintain a session store, making them highly scalable. In this system, a JWT is issued after successful Aadhaar verification and must accompany every subsequent API request, serving as proof of authenticated identity.

III. LITERATURE REVIEW

A. End-to-End Secure Mobile Chat (Akram & Ko, 2023)

Akram and Ko [1] propose a practical E2E architecture specifically for mobile chat applications, addressing the challenge of secure key exchange between users who may be offline at different times. Their key contributions include an offline-capable key-exchange design that pre-publishes key bundles for offline recipients — allowing senders to encrypt messages even when the recipient is not connected — a user-to-user authentication mechanism that avoids centralized trust for session authentication, and a local storage protection scheme that encrypts the message database on the device so that even physical access to the device does not expose historical conversations.

Their implementation uses a combination of ECDH key agreement and AES-GCM symmetric encryption, with ephemeral key bundles published to a server acting only as a key-distribution service. The paper demonstrates feasibility but acknowledges open questions around server scalability under high concurrent load and the extension to voice and video channels. Their emphasis on minimizing PII collection during registration and using ephemeral offline message stores deleted after delivery directly informs the design principles of our system.

Relevance: This paper provides the core architectural pattern — client-side key generation, offline key pre-publication, and ephemeral message stores — that is adapted in our design for the E2EE and long-polling message delivery subsystems.

B. SoK: E2EE Authentication Ceremonies (Alatawi & Saxena, 2023)

Alatawi and Saxena [2] present a systematization of knowledge (SoK) surveying popular E2EE messaging applications — including Signal, WhatsApp, Telegram, iMessage, and Viber — and their underlying cryptographic protocols: OTR (Off-the-Record), Signal's X3DH + Double Ratchet, ZRTP, and Telegram's MTProto. The survey's distinctive focus is on the authentication ceremony: the mechanism by which users verify that the public keys they are using genuinely belong to the intended party, rather than to an attacker who has substituted their own keys (a man-in-the-middle attack).

Their central findings are striking: most widely-deployed E2EE applications operate in 'opportunistic' E2EE mode, which is effective against passive eavesdroppers but remains vulnerable to active MitM attacks by a malicious server or network adversary. Authentication ceremonies — such as comparing key fingerprints or safety numbers — are universally present but consistently hidden in deep settings menus, labeled with confusing technical jargon, and rarely performed by ordinary users. The authors argue that correct cryptographic protocol design is necessary but not sufficient: usable, discoverable authentication ceremonies are an equally critical security control.

Relevance: This work defines the threat model against which our system is designed — particularly the MitM attack vector — and motivates the combination of Aadhaar-backed identity verification with key fingerprint exposure. Our design maps Aadhaar-verified identity to key fingerprints, creating an OOB (out-of-band) verification channel that reduces reliance on server honesty.

C. E2E Process Automation with Generative AI and IDP (Jeong et al., 2023)

Jeong, Kim, and Park [3] present a case study applying generative AI and Intelligent Document Processing (IDP) to automate corporate expense management workflows. They design a four-stage pipeline: OCR and IDP-based document recognition, policy-driven classification, generative-AI-assisted exception handling, and human-in-the-loop finalization with continuous learning. The study reports approximately 80% reduction in end-to-end processing time and significant accuracy improvements in exception handling over successive training iterations.

Although focused on document automation rather than messaging, the paper's governance and architectural lessons are broadly applicable. Key insights include: the importance of human-in-the-loop review for ambiguous cases in automated sensitive-data pipelines; the need for auditable decision logs when AI components make consequential decisions; and the value of a loosely-coupled, API-based architecture that allows individual components to be upgraded or replaced without disrupting the overall system.

Relevance: The automation and governance patterns from this work apply directly to the Aadhaar onboarding flow and potential future AI-based moderation in our system. Specifically, the human-in-the-loop model informs the design of anomaly detection for suspicious registration patterns, while the audit-trail principle motivates comprehensive logging of authentication events without storing sensitive personal data.

D. Cross-Paper Synthesis

Five common themes emerge from the literature review that directly shape our design:

1. **E2EE is necessary but not sufficient.** Correct cryptography must be paired with usable authentication ceremonies and verified identity to resist active

adversaries.

2. **Asynchronous key exchange requires pre-publication.** Modern protocols like X3DH support offline message encryption by publishing key bundles in advance.
3. **Minimal data principle.** Collecting and storing only the data strictly necessary for operation reduces both privacy exposure and regulatory liability.
4. **Automation must preserve human oversight.** Automated identity verification pipelines must route ambiguous cases to human review with full auditability.
5. **Scalability and performance must be evaluated empirically.** Privacy-first designs incur overhead; benchmarking is necessary to validate practical viability.

IV. SYSTEM DESIGN AND ARCHITECTURE

A. High-Level Architecture

The system follows a three-tier client-server architecture with a clear separation of concerns between presentation, business logic, and data persistence. The three tiers are:

- **Presentation Tier (React Frontend):** Responsible for the user interface, user interactions, client-side key generation, local encryption/decryption of message payloads, and triggering long-polling requests for incoming messages.
- **Logic Tier (Spring Boot Backend):** Manages user registration and authentication, mediates UIDAI OTP verification, distributes public keys, relays encrypted messages, enforces access control via JWT, and implements the long-polling server endpoint.
- **Data Tier (MySQL Database):** Persists user metadata, hashed Aadhaar references, public keys, encrypted message records, and session tokens with appropriate encryption-at-rest and access controls.

A critical architectural invariant is that the Logic Tier handles only ciphertext — it never possesses any decryption key and therefore cannot read message content. This zero-knowledge server property is enforced architecturally: private keys are generated on the client and never transmitted to the server.

B. Component Architecture

The system is decomposed into six loosely coupled modules, each with a well-defined interface and single responsibility:

Module	Responsibility	Technology
Auth Module	Aadhaar OTP verification, JWT issuance and validation	Spring Boot, UIDAI API
Crypto Module	Client-side key generation, message encrypt/decrypt	React, crypto-js
Messaging Module	Long-poll server, message relay, ciphertext storage	Spring Boot DeferredResult
Key Store Module	Public key distribution and retrieval	Spring Boot REST, MySQL
User Profile Module	Profile management, contact discovery	React, Spring Boot
Database Module	Persistent data management, schema enforcement	MySQL, JPA/Hibernate

Table I: System Module Architecture

C. Aadhaar Authentication Flow

The Aadhaar-based registration and authentication flow proceeds through the following steps:

1. User navigates to the registration page and enters their Aadhaar number.
2. Frontend sends a POST request to the backend /api/auth/initiate endpoint with the Aadhaar number.
3. Backend forwards an OTP generation request to the UIDAI sandbox API.
4. UIDAI sends a one-time password to the mobile number registered with the Aadhaar database.
5. User enters the received OTP in the application frontend.
6. Frontend submits the OTP to the backend /api/auth/verify endpoint.
7. Backend verifies the OTP with UIDAI and, on success, generates a SHA-256 hash of the Aadhaar number for storage.
8. Backend generates a signed JWT token with the user's internal ID and an expiry timestamp.
9. Backend generates an RSA key pair on behalf of the client (or client generates it locally), stores the public key, and returns the JWT and public key metadata to the frontend.
10. All subsequent API calls include the JWT in the Authorization header; the backend validates the signature before processing any request.

D. End-to-End Encryption Design

The E2EE design follows the hybrid encryption model. Each registered user possesses an asymmetric key pair: a public key stored on the server and accessible to other users, and a private key stored exclusively on the client device (in browser local storage, protected by the user's session). The encryption protocol for sending a message from User A to User B is as follows:

1. User A's client requests User B's public key from the server's key-store endpoint.
2. A random AES session key is generated on User A's client for this message (or conversation session).
3. The message plaintext is encrypted with the AES session key using AES encryption.
4. The AES session key is encrypted (wrapped) with User B's RSA public key.
5. Both the encrypted message and the encrypted session key are transmitted to the server as a ciphertext bundle.

6. The server stores the ciphertext bundle in the messages table without any ability to decrypt it.
7. User B's client retrieves the ciphertext bundle via long polling.
8. User B's client decrypts the AES session key using their RSA private key.
9. User B's client decrypts the message using the recovered AES session key and displays the plaintext.

E. Long-Polling Message Delivery Design

The long-polling architecture uses Spring Boot's DeferredResult mechanism, which suspends the HTTP response without blocking a server thread. When a message is sent, the server checks whether the recipient has an active DeferredResult waiting; if so, it resolves the result immediately with the new message list. If no active poll exists for the recipient, the message is written to the database and will be retrieved on the recipient's next poll. The poll timeout is set to 5 seconds, after which an empty response is returned and the client immediately issues a new poll request. This creates a continuous, low-latency delivery channel with effectively no wasted server resources.

F. Security Architecture and Threat Model

The system is designed against a realistic threat model encompassing the following adversaries:

Threat	Adversary Capability	Mitigation
Network Eavesdropping	Passive interception of network traffic	E2EE + TLS transport encryption
Server Compromise	Full access to server storage and memory	Zero-knowledge server; only ciphertext stored
Fake Account Creation	Mass creation of accounts for spam/abuse	Aadhaar OTP verification limits 1 account per identity
Impersonation	Claiming to be another verified user	Aadhaar verification links account to real identity
MitM Key Substitution	Replacing recipient public key with attacker's	Key fingerprint display; Aadhaar-identity binding
Session Hijacking	Stealing and reusing authenticated sessions	Short-lived JWT tokens with HTTPS-only transmission
Unauthorized API Access	Calling APIs without valid credentials	JWT validation on every backend endpoint

Table II: Threat Model and Mitigations

V. IMPLEMENTATION

A. Technology Stack

The implementation uses a modern, open-source technology stack selected for maturity, community support, and fit with the security requirements of the system. Table III summarizes the complete stack.

Layer	Technology	Version	Rationale
Frontend UI	React.js + JavaScript	18.x	Component-based, efficient state management
Styling	CSS3, HTML5	—	Responsive, accessible design
Crypto (Client)	crypto-js	4.x	Browser-compatible AES/RSA
Backend	Spring Boot (Java)	3.x	REST APIs, DeferredResult, security
ORM	Spring Data JPA + Hibernate	6.x	Type-safe DB access, schema migration
Database	MySQL	8.0	ACID compliance, mature relational DB
Auth	UIDAI Sandbox + JWT	—	Government-grade identity verification
Version Control	Git + GitHub	—	Collaborative development, CI tracking
API Testing	Postman	10.x	REST endpoint validation
Unit Testing	JUnit 5 + Jest	—	Backend Java + frontend JS testing
Build Tool	Maven (backend), npm (frontend)	—	Dependency management and build automation

Table III: Technology Stack

B. Backend Implementation (Spring Boot)

1) Aadhaar Registration Endpoint:

The registration endpoint validates the Aadhaar OTP, hashes the Aadhaar number using SHA-256 before storage, stores the user's public key, and issues a JWT token. The raw Aadhaar number is never written to the database.

```
@PostMapping("/register")
public ResponseEntity<AuthResponse> register(
  @RequestBody RegistrationRequest req) {
  boolean otpValid =
    uidaiService.verifyOtp(req.getAadhaarNum(),
      req.getOtp());
  if (!otpValid) return ResponseEntity
    .status(HttpStatus.UNAUTHORIZED).build();
  String aadhaarHash = DigestUtils
    .sha256Hex(req.getAadhaarNum());
```

```
  User user = userService
    .createUser(req.getName(),
      aadhaarHash, req.getPublicKey());
  String jwt = jwtService.generateToken(user);
  return ResponseEntity.ok(
    new AuthResponse(jwt, user.getId()));
}
```

2) Long-Polling Message Endpoint:

The polling endpoint uses Spring's DeferredResult to suspend the response without blocking a thread, achieving efficient concurrent handling of many simultaneous long-poll connections.

```
@GetMapping("/messages/poll/{userId}")
public DeferredResult<ResponseEntity<
  List<MessageDTO>>> pollMessages(
    @PathVariable Long userId,
    @RequestParam Long lastMsgId) {
  DeferredResult<...> result =
    new DeferredResult<>(5000L);
  result.onTimeout(() -> result
    .setResult(ResponseEntity.ok(emptyList())))
;
  pollRegistry.register(userId, result,
lastMsgId);
  return result;
}
```

3) Message Send Endpoint:

When a message is sent, the backend stores the ciphertext and notifies any active long-poll DeferredResult for the recipient, enabling near-instant delivery.

```
@PostMapping("/messages/send")
public ResponseEntity<Void> sendMessage(
    @RequestBody MessageRequest req,
    @AuthenticationPrincipal UserDetails user) {
  Message msg = messageService.save(
    req.getSenderId(),
    req.getReceiverId(),
    req.getEncryptedPayload());
  pollRegistry.notifyUser(
    req.getReceiverId(), msg);
  return ResponseEntity.ok().build();
}
```

C

1) Client-Side Key Generation:

During registration, the browser generates an RSA key pair. The private key is stored in the browser's localStorage protected by the session; the public key is transmitted to the server for distribution.

```
async function generateKeyPair() {
  const keyPair = await
window.crypto.subtle.generateKey(
    { name: 'RSA-OAEP', modulusLength: 2048,
      publicExponent: new Uint8Array([1,0,1]),
      hash: 'SHA-256' },
    true, ['encrypt', 'decrypt']);
  const pubKey = await
exportPublicKey(keyPair.publicKey);
  const privKey = await
exportPrivateKey(keyPair.privateKey);
  localStorage.setItem('privKey', privKey);
  return pubKey;
}
```

2) Message Encryption and Decryption:

Messages are encrypted before leaving the device using the recipient's public key, and decrypted after retrieval using the sender's private key.

```
// Encrypt a message for the recipient
function encryptMessage(plaintext,
recipientPubKey) {
  const sessionKey =
CryptoJS.lib.WordArray.random(32);
  const encMsg = CryptoJS.AES.encrypt(
    plaintext,
sessionKey.toString()).toString();
  const encKey = rsaEncrypt(sessionKey,
recipientPubKey);
  return { encryptedMessage: encMsg,
           encryptedSessionKey: encKey };
}

// Decrypt a received ciphertext bundle
function decryptMessage(bundle, privateKey) {
  const sessionKey = rsaDecrypt(
    bundle.encryptedSessionKey, privateKey);
  const bytes = CryptoJS.AES.decrypt(
    bundle.encryptedMessage, sessionKey);
  return bytes.toString(CryptoJS.enc.Utf8);
}
```

3) Long-Polling Client Loop:

The client maintains a continuous polling loop that immediately issues a new request upon receiving a response, ensuring no window exists during which messages could be missed.

```
async function startPolling(userId, lastMsgId) {
  let lastId = lastMsgId;
  while (true) {
    try {
      const res = await fetch(
        `/api/messages/poll/${userId}?lastMsgId
=${lastId}`,
        { headers: { Authorization: `Bearer
${jwt}` } });
      const msgs = await res.json();
      if (msgs.length > 0) {
        msgs.forEach(m => displayDecrypted(m));
        lastId = msgs[msgs.length-1].id;
      }
    } catch (e) {
      await sleep(1000); // back-off on error
    }
  }
}
```

D. Database Schema and Design

The relational database schema is designed to store only the minimum necessary data, in compliance with UIDAI regulations. Raw Aadhaar numbers are never stored. The three primary tables are:

```
CREATE TABLE users (
  id            BIGINT AUTO_INCREMENT PRIMARY
KEY,
  name          VARCHAR(100) NOT NULL,
  aadhaar_hash  VARCHAR(64) UNIQUE NOT NULL,
  public_key    TEXT NOT NULL,
  created_at    DATETIME DEFAULT
CURRENT_TIMESTAMP
);

CREATE TABLE messages (
  id                BIGINT AUTO_INCREMENT
PRIMARY KEY,
  sender_id         BIGINT NOT NULL,
  receiver_id       BIGINT NOT NULL,
  encrypted_payload TEXT NOT NULL,
  encrypted_key     TEXT NOT NULL,
  timestamp         DATETIME DEFAULT NOW(),
  FOREIGN KEY (sender_id) REFERENCES users(id),
  FOREIGN KEY (receiver_id) REFERENCES users(id)
);

CREATE TABLE sessions (
  id         BIGINT AUTO_INCREMENT PRIMARY KEY,
  user_id    BIGINT NOT NULL,
  jwt_token VARCHAR(512) NOT NULL,
  issued_at DATETIME NOT NULL,
  expires_at DATETIME NOT NULL,
  revoked    BOOLEAN DEFAULT FALSE
);
```

VI. TESTING AND EVALUATION

A. Testing Strategy

A bottom-up testing strategy was followed, beginning with unit testing of individual components, progressing through integration testing of component interfaces, system testing of the complete application, and finally user-acceptance testing (UAT) with real users. Security-specific tests validated resistance to unauthorized API access, message tampering, and duplicate Aadhaar registration. All tests were automated where possible using JUnit 5 (backend) and Jest (frontend), with API-level tests maintained in Postman collections for regression testing.

B. Unit Testing

Unit tests were written for all critical backend service methods, including Aadhaar verification, JWT generation and validation, message persistence, and the long-polling registry. Frontend unit tests covered the encryption, decryption, and key generation functions. Representative test cases are shown below.

```
@Test
void testEncryptDecrypt_RoundTrip_Succeeds() {
  String original = "Hello, Secure World!";
  KeyPair kp = cryptoUtil.generateRSAKeyPair();
  String enc = cryptoUtil.encrypt(original,
kp.getPublic());
  String dec = cryptoUtil.decrypt(enc,
kp.getPrivate());
  assertEquals(original, dec);
}

@Test
void testJWT_ExpiredToken_ThrowsException() {
  String expiredJwt =
jwtService.generateExpiredToken(user);
  assertThrows(ExpiredJwtException.class,
    () ->
jwtService.validateToken(expiredJwt));
}
```

C. Integration Testing

Integration tests validated the end-to-end data flow across component boundaries: the Aadhaar verification request-response cycle between the frontend, backend, and UIDAI mock; the encrypted message storage and retrieval cycle between the messaging module and MySQL; and the long-polling delivery cycle between the React client, Spring Boot endpoint, and the polling registry. A full end-to-end integration test simulated: User A sending an encrypted message, the backend storing the ciphertext, User B's long-poll being resolved, User B's client decrypting the ciphertext, and verifying that the decrypted text matches the original plaintext.

D. System Testing

System testing evaluated the complete application against both functional and non-functional requirements in a local environment with the frontend running on localhost:3000, the backend on localhost:8080, and MySQL on a local instance. The full set of system test cases is summarized in Table IV.

TC ID	Module	Test Description	Input	Expected Output	Result
TC-01	Auth	Valid Aadhaar OTP registration	Valid Aadhaar + OTP	Account created, JWT issued	Pass ✓
TC-02	Auth	Invalid OTP rejection	Wrong OTP string	HTTP 401 Unauthorized	Pass ✓
TC-03	Auth	Duplicate Aadhaar blocked	Same Aadhaar twice	HTTP 409 Conflict	Pass ✓
TC-04	Crypto	E2EE round-trip correctness	Plaintext message	Decrypted text == original	Pass ✓
TC-05	Crypto	Server cannot read plaintext	Inspect DB after send	Only ciphertext in DB	Pass ✓

```
@Test
void
testAadhaarVerification_ValidOtp_ReturnsTrue()
{
  boolean result =
    uidaiService.verifyOtp("123456789012",
"456789");
  assertTrue(result);
}
```

TC ID	Module	Test Description	Input	Expected Output	Result
TC-06	Messaging	Long-poll delivery latency	Send message from User A	User B receives within 1s	Pass ✓
TC-07	Messaging	Message persistence on reload	Refresh browser after chat	Chat history reloads correctly	Pass ✓
TC-08	Security	API access without JWT	Request with no token	HTTP 403 Forbidden	Pass ✓
TC-09	Security	API access with expired JWT	Expired token in header	HTTP 401 Unauthorized	Pass ✓
TC-10	Security	Message tampering detection	Modified ciphertext in DB	Decryption fails gracefully	Pass ✓
TC-11	Profile	Profile update persistence	Updated name and status	Changes reflected on reload	Pass ✓
TC-12	System	Concurrent user load (10 users)	10 simultaneous chat sessions	All messages delivered correctly	Pass ✓

Table IV: System Test Case Results (All 12/12 Passed)

E. User Acceptance Testing (UAT)

User acceptance testing was conducted with a group of eight participants, including both technical and non-technical users, who performed a structured set of tasks: account registration with Aadhaar OTP, login, sending messages to another user, verifying message confidentiality, and reviewing chat history. Participant feedback was collected using structured questionnaires assessing usability, responsiveness, and perceived security.

All participants successfully completed the registration flow without assistance. Seven of eight rated the interface as intuitive or very intuitive. Message delivery was perceived as instantaneous by all participants. The Aadhaar OTP step was identified as adding some friction to registration, but all participants acknowledged its importance for security. No participant was able to access another user's messages through any tested attack vector.

VII. RESULTS AND DISCUSSION

A. Security Analysis

The implemented system achieves all stated security objectives. The zero-knowledge server property holds by construction: the server stores only ciphertext, and no decryption key is ever transmitted to or stored on the server. Even a complete server compromise would yield an attacker only encrypted blobs with no practical means of decryption given the strength of the underlying RSA and AES algorithms.

The Aadhaar identity binding eliminates the fake account problem at the registration layer. Since each Aadhaar number can be registered at most once (enforced by the unique constraint on the aadhaar_hash column), mass account creation is prevented without requiring CAPTCHA or phone verification that can be bypassed with virtual numbers. The use of a cryptographic hash (SHA-256) of the Aadhaar number as the stored reference ensures that raw Aadhaar numbers cannot be recovered from the database even in the event of a data breach.

JWT-based session management prevents session hijacking through token expiry and HTTPS-only transmission. The server validates the JWT signature on every request, ensuring that tampered or forged tokens are rejected. Token revocation on logout is implemented by marking the session record as revoked in the sessions table.

B. Performance Analysis

Under simulated load of ten concurrent users conducting simultaneous chat sessions, all messages were delivered correctly with no loss or duplication. Perceived message delivery latency was consistently below one second for users on the same local network. The DeferredResult-based long-polling approach maintained low server-side resource consumption, with suspended requests consuming no active threads during the waiting period.

The Aadhaar OTP verification step introduces a dependency on UIDAI API latency, which in the sandbox environment averaged approximately 800ms. In a production deployment, this latency occurs only at registration and is not in the critical path for message sending or receiving.

C. Comparison with Existing Systems

Feature	WhatsApp	Signal	Telegram	Proposed System
E2EE	Yes	Yes	Optional	Yes ✓
Verified Identity	Phone only	Phone only	Phone only	Aadhaar (Govt.) ✓
Fake Account Prevention	Partial	Partial	Partial	Strong ✓
Zero-Knowledge Server	Yes	Yes	No	Yes ✓
Forward Secrecy	Yes	Yes	Partial	Planned
Open Source	No	Yes	Partial	Yes ✓
India Identity Integration	No	No	No	Yes (Aadhaar) ✓

Table V: Feature Comparison with Existing Messaging Platforms

D. Limitations and Known Issues

The current implementation has several limitations that should be addressed before production deployment:

- **Simplified cryptography:** The AES key-wrapping scheme is a simplification of full RSA-OAEP + AES-256-GCM. Production deployment should use the Web Crypto API with proper key encapsulation.
- **No forward secrecy:** Compromise of a user's long-term private key could expose all past messages. The Signal Protocol's Double Ratchet should be integrated for production.
- **No offline message queuing:** Messages sent while the recipient is offline are stored but only delivered when the recipient next connects and polls.
- **Single-device per user:** The current design does not support key synchronization across multiple devices for the same user.
- **UIDAI sandbox dependency:** Testing uses UIDAI's sandbox environment; production deployment requires approved AUA/KUA entity registration with UIDAI.

VIII. CONCLUSION

This paper presented the design, implementation, and evaluation of an end-to-end encrypted messaging application with Aadhaar-based user authentication — a system that addresses the twin and often-overlooked challenges of message confidentiality and user identity authenticity in digital communication.

By integrating India's UIDAI Aadhaar infrastructure at the account onboarding stage, the system ensures that every registered user is a genuinely verified, unique individual, eliminating the fake account, impersonation, and mass spam problems that afflict authentication-light platforms. By implementing asymmetric E2EE with a zero-knowledge server architecture, the system ensures that message content remains confidential even against a fully compromised server. The full-stack implementation — React frontend, Spring Boot backend, MySQL database — demonstrates that these strong security guarantees are achievable within a modern, maintainable, and extensible web application architecture.

Comprehensive testing across unit, integration, system, and user-acceptance layers confirmed that all twelve system test cases pass, all security properties hold under tested attack scenarios, and usability is sufficient for non-technical users. The proposed system represents a meaningful contribution toward trustworthy digital communication infrastructure that aligns with India's growing emphasis on digital identity, cybersecurity, and data sovereignty.

IX. FUTURE SCOPE

Several directions for future work have been identified:

#	Enhancement	Description
1	Signal Protocol Integration	Adopt X3DH key agreement and Double Ratchet for forward secrecy and break-in recovery.
2	Mobile Applications	Develop cross-platform iOS/Android clients using React Native with push notifications and offline message queuing.
3	Voice & Video E2EE	Integrate WebRTC-based voice/video calling with DTLS-SRTP encryption, authenticated with Aadhaar identity.
4	Post-Quantum Cryptography	Migrate key exchange to NIST-standardized post-quantum algorithms (CRYSTALS-Kyber) to resist quantum adversaries.
5	AI-Based Threat Detection	Integrate ML models for spam, phishing, and abuse detection on metadata (not message content) with human-in-the-loop review.
6	Blockchain Identity	Decentralized identity verification using blockchain for tamper-resistant, server-independent identity attestation.
7	Cloud Deployment & Scaling	Deploy on AWS/GCP with horizontal scaling, Redis-backed poll registry, and CDN-fronted React bundle.
8	Group Messaging E2EE	Extend to group conversations using the Signal Protocol's Sender Keys mechanism for efficient group E2EE.
9	Encrypted Media Sharing	Add client-side encrypted image, document, and video sharing with server-side storage of only encrypted blobs.
10	Multi-Device Support	Implement key synchronization across multiple devices per user using device-linking protocols similar to Signal's linked devices feature.

Table VI: Future Enhancement Roadmap

REFERENCES

[1] J. Akram and K. Ko, "End-to-End Secure and Privacy Preserving Mobile Chat Application," *Proc. IEEE International Conference on Consumer Electronics (ICCE)*, IEEE Xplore, 2023.

[2] H. Alatawi and N. Saxena, "SoK: An Analysis of End-to-End Encryption and Authentication Ceremonies in Secure Messaging Systems," *Proc. 16th ACM Conf. Security and Privacy in Wireless and Mobile Networks (WiSec '23)*, ACM, New York, 2023. DOI: 10.1145/3558482.3581773.

[3] M. Jeong, D. Kim, and S. Park, "End-to-End Process Automation Leveraging Generative AI and IDP-Based Automation Agent," *IEEE International Conference on Artificial Intelligence and Knowledge Engineering*, IEEE Xplore, 2023.

[4] UIDAI, "Aadhaar Authentication API Specification v2.5," Unique Identification Authority of India, Government of India, New Delhi, 2024. [Online]. Available: https://uidai.gov .in/en/ecosystem/authentication-devices-documents/api-and-s dk.html

[5] W. Diffie and M. Hellman, "New Directions in Cryptography," *IEEE Transactions on Information Theory*, vol. 22, no. 6, pp. 644–654, Nov. 1976. DOI: 10.1109/TIT.1976.1055638.

[6] E. Rescorla, "The Transport Layer Security (TLS) Protocol Version 1.3," IETF RFC 8446, Internet Engineering Task Force, Aug. 2018. [Online]. Available: https://www.rfc-editor.org/rfc/rfc8446.

[7] M. Marlinspike and T. Perrin, "The Double Ratchet Algorithm," Open Whisper Systems, 2016. [Online]. Available: https://signal.org/docs/specifications/doubleratchet/.

[8] M. Marlinspike, "The X3DH Key Agreement Protocol," Open Whisper Systems, 2016. [Online]. Available: https://signal.org/docs/specifications/x3dh/.

[9] M. Jones, J. Bradley, and N. Sakimura, "JSON Web Token (JWT)," IETF RFC 7519, May 2015. [Online]. Available: https://www.rfc-editor.org/rfc/rfc7519.

[10] R. Rivest, A. Shamir, and L. Adleman, "A Method for Obtaining Digital Signatures and Public-Key Cryptosystems," *Communications of the ACM*, vol. 21, no. 2, pp. 120–126, 1978.

[11] National Institute of Standards and Technology (NIST), "Recommendation for Block Cipher Modes of Operation: Galois/Counter Mode (GCM) and GMAC," NIST SP 800-38D, 2007.

[12] Spring Framework Documentation, "Spring Boot 3.x Reference Guide — Asynchronous Requests," VMware Inc., 2024. [Online]. Available: https://docs.spring.io/spring-boot/d ocs/current/reference/html/.

[13] React Documentation, "React 18 Official Guide: Hooks and State Management," Meta Platforms Inc., 2024. [Online]. Available: https://react.dev.

[14] MySQL Documentation, "MySQL 8.0 Reference Manual — Security and Encryption Functions," Oracle Corporation, 2024. [Online]. Available: https://dev.mysql.com/doc/refman/8.0/en/.

[15] Government of India, "Personal Data Protection Bill and UIDAI Data Governance Framework," Ministry of Electronics and Information Technology, New Delhi, 2023.