

DESIGN AND IMPLEMENTATION OF A SECURE INTRA-PLATFORM REAL-TIME MESSAGING SYSTEM FOR SOCIAL MEDIA APPLICATIONS

Edwin Oghenenyrovhwo Dodoh¹, Michael Oghale Ighofiomoni², Obruche Stanley Ewenode³

^{1,2,3} Department of Computer Engineering, Faculty of Engineering, Southern Delta University, Ozoro, Delta State, Nigeria

Article Info

Article History:

Published: 02 Oct 2026

Publication Issue:

Volume 03, Issue 10
October 2026

Page Number:

17-35

Corresponding Author:

Edwin Oghenenyrovhwo
Dodoh

Abstract:

Real-time messaging has become a central interaction layer in social media, yet persistent connections, multi-device sessions, third-party integrations and large user populations create security risks that conventional request-response controls do not fully address. This study designs, implements and evaluates a secure intra-platform messaging framework that combines persistent WebSocket communication with end-to-end encrypted message payloads, short-lived identity tokens, conversation-level authorization, replay resistance, rate limiting, privacy-preserving audit events and reliable delivery semantics. The work follows a design-science methodology comprising problem analysis, threat modelling, architectural design, prototype construction and controlled evaluation. A single-node prototype was implemented with an asynchronous WebSocket gateway, HTTPS fallback endpoint, JSON Web Token validation, origin checking, AES-256-GCM payload protection, message-size enforcement and connection-level flood control. Functional security tests examined unauthorized origins, expired and invalid tokens, missing authentication, excessive message rates and oversized payloads. Performance was measured at 25, 50, 100 and 200 concurrent users, with each client transmitting fifteen encrypted messages. The WebSocket design maintained mean round-trip latency between 0.83 and 8.12 milliseconds, compared with 5.91 to 54.42 milliseconds for HTTPS request-response messaging. At 200 users, WebSocket throughput reached 4,315.55 messages per second, while all seven security tests produced the expected rejection or acceptance behaviour. The findings indicate that a layered architecture can deliver low latency without treating transport encryption as a substitute for end-to-end confidentiality. The article contributes a reproducible architecture, logical data model, secure message lifecycle, implementation controls and evaluation framework for privacy-conscious social media messaging under realistic concurrent laboratory loads.

Keywords: secure messaging; social media; WebSocket; end-to-end encryption; zero trust; real-time systems; access control; message security

1. Introduction

Social media platforms no longer function only as publishing spaces. They are also communication infrastructures in which users expect private one-to-one conversations, group discussions, media exchange, presence information, delivery receipts, typing indicators and synchronized access across several devices. These expectations make messaging a high-frequency, stateful and security-sensitive subsystem. Unlike a conventional web page, a messaging service maintains long-lived connections, handles bursts of small events, retains delivery state and mediates interactions among users who may not know or trust one another. A design that focuses only on speed can expose conversations, while a design that focuses only on cryptography can become unusable, difficult to moderate or operationally unreliable.

The expansion of end-to-end encryption in consumer communication has raised the baseline expectation for confidentiality. However, transport encryption and end-to-end encryption solve different problems. Transport Layer Security protects data while it travels between a client and a server, but the server may still receive plaintext. End-to-end encryption protects message content from intermediary services by keeping decryption keys at the communicating endpoints. Bhuse (2023) observes that stronger cryptographic primitives have become central to social-media privacy, while Afzal et al. (2021) show that encrypted applications may still reveal observable network patterns and operational metadata. The implication is that a secure platform must address payload secrecy, metadata minimisation, authenticated identity, access control, endpoint compromise, abuse reporting, retention, deletion and availability as separate but interacting concerns.

Real-time transport introduces another layer of risk. WebSocket connections are persistent and bidirectional, which eliminates repeated connection establishment and supports low-latency interaction. The same persistence can also enable cross-site WebSocket hijacking, session abuse, unbounded message flooding, oversized frames and monitoring blind spots when security controls are applied only during the initial HTTP upgrade. OWASP Foundation (2025b) therefore recommends origin validation, explicit authentication, per-message authorization, size limits, rate controls and event logging for WebSocket services. These requirements are especially important in social-media systems because a valid user may still attempt to access a conversation, group or resource for which that user lacks current permission.

This article presents the design and implementation of a secure intra-platform real-time messaging system for social media applications. “Intra-platform” means that all participating users are registered identities within the same social platform and that the platform controls identity, conversation membership, routing and policy enforcement. The proposed framework uses a persistent secure WebSocket channel for low latency, client-side authenticated encryption for message confidentiality, short-lived access tokens, device and session controls, conversation-level authorization, ordered event delivery, ciphertext persistence, acknowledgement states and privacy-respecting moderation. A measured single-node prototype is used to demonstrate feasibility. The aim is not to claim Internet-scale production capacity from a laboratory test, but to provide an integrated and reproducible foundation that can be extended through horizontal gateway scaling, broker partitioning, distributed presence and regional data services.

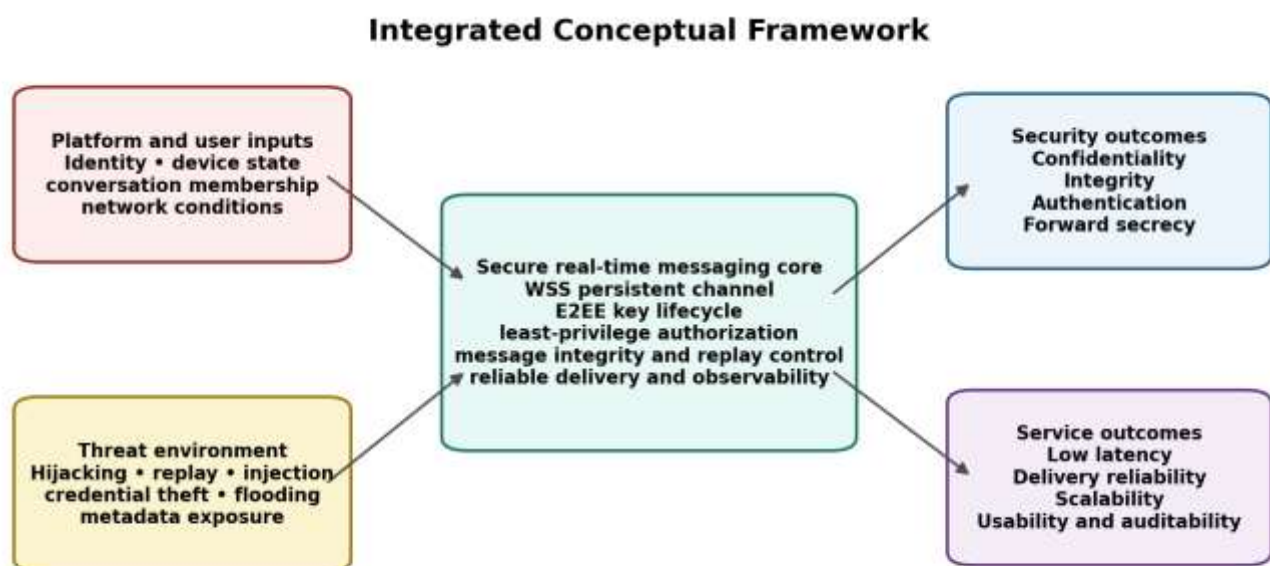


Figure 1.1. Integrated conceptual framework for secure intra-platform messaging.

1.1 Statement of the Problem

Many messaging implementations are assembled from independent components: a login API, a WebSocket library, a database table and a notification service. When these elements are connected without a coherent security model, the resulting platform may authenticate the socket but fail to authorize individual messages, store plaintext indefinitely,

expose tokens in URLs, accept connections from malicious origins, duplicate messages after reconnection or lose events during server failure. Persistent connections also complicate logout and revocation because a socket established under a valid session can remain active after the account state has changed.

A second problem is the false equivalence between WSS and end-to-end security. WSS encrypts the client-server channel, but it does not prevent the platform, a privileged administrator or a compromised message service from reading plaintext. Conversely, an end-to-end encrypted design can still leak who communicates with whom, when communication occurs, which devices are active and how large a message is. Kwon et al. (2020) demonstrate that metadata privacy at scale requires specialised architecture and measurable trade-offs. Secure messaging therefore cannot be reduced to selecting one encryption algorithm.

A third problem concerns the balance between privacy and platform safety. Complete server visibility simplifies content scanning but weakens confidentiality. Complete invisibility can make user-initiated abuse reports difficult to verify. Issa et al. (2022) show that cryptographic reporting mechanisms can preserve privacy for unreported messages while allowing accountable evidence for reported content. Existing prototype studies often examine one dimension encryption, network efficiency, user understanding or moderation without integrating these concerns into a deployable social-media workflow. This fragmentation motivates a system that treats identity, authorization, cryptography, reliability, observability and user safety as one design problem.

1.2 Aim and Objectives of the Study

The aim of the study is to design, implement and evaluate a secure, low-latency intra-platform messaging framework suitable for social media applications.

The specific objectives are to:

1. develop a layered security architecture for identity, session, conversation and message protection; (2) define a ciphertext-centred data model and reliable message lifecycle;
2. implement a functional WebSocket prototype with token validation, origin checking, authenticated encryption, message-size controls and flood resistance;
3. evaluate functional security behaviour against representative misuse cases; and
4. compare the latency and throughput of persistent WebSocket messaging with an HTTPS request-response alternative under increasing concurrent load.

1.3 Contributions of the Study

The principal contribution is an integrated framework that connects secure design principles to an implementable message path. The article provides: a conceptual model; a zero-trust enforcement model; a modular service architecture; a logical database design; a message envelope and key-evolution model; a delivery sequence with acknowledgement and deduplication; a threat-control matrix; a measured prototype benchmark; and a set of implementation recommendations. The work also separates claims supported by the prototype from features intended for production expansion. In particular, the benchmark validates a single asynchronous gateway and does not substitute for multi-region chaos testing, independent cryptographic review or formal protocol verification.

2.0 Literature Review

The literature on secure messaging spans cryptographic protocols, distributed systems, usable security, privacy engineering and platform governance. This breadth reflects the fact that messaging security is not a single property. A system can provide confidentiality yet fail to preserve deletion, provide low latency yet permit session hijacking, or provide a technically sound verification ceremony that users do not understand. The review therefore examines concepts, theories and empirical studies relevant to a complete intra-platform design.

2.1 Conceptual Review

A real-time messaging system is an event-driven application that transmits messages and interaction states with sufficiently low delay for users to experience the exchange as immediate. In a social-media environment, the system usually supports direct messages, group conversations, media references, presence, receipts, reactions, edits, deletion requests and offline notification. “Real time” does not necessarily mean deterministic hard-real-time performance; it means that latency remains low and predictable enough for interactive communication.

WebSocket is appropriate for this pattern because it upgrades an HTTP connection into a full-duplex channel. After establishment, either endpoint can transmit frames without initiating a new request. This lowers protocol overhead and enables server-initiated delivery. The security consequence is that authorization must continue after connection establishment. A connection authenticated for a user does not automatically authorize every conversation identifier contained in subsequent frames. OWASP Foundation (2025b) identifies origin checks, message validation, per-action authorization, rate limiting and security logging as essential WebSocket controls.

Secure messaging requires confidentiality, integrity, authenticity, availability and privacy. Confidentiality prevents unauthorised reading. Integrity detects modification. Authenticity establishes the claimed sender or device. Availability preserves service during failures and abuse. Privacy extends beyond content to communication patterns, identifiers, retention and inference. Afzal et al. (2021) demonstrate that encrypted traffic can still reveal activity classes through packet patterns, illustrating why padding, metadata minimisation and cautious logging may be necessary for high-risk deployments.

End-to-end encryption moves message encryption and decryption to user devices. The delivery service stores and forwards ciphertext. Modern secure messaging also seeks forward secrecy, so compromise of a current key does not reveal earlier messages, and post-compromise security, so the conversation can recover after a compromised state is replaced. The Messaging Layer Security protocol standardises group key establishment and epoch changes for secure group communication (Barnes et al., 2023). Its architecture clarifies that a complete system still requires identity, delivery, storage, group membership and policy services around the cryptographic core (Beurdouche et al., 2025).

Identity and session management bind a platform account to one or more authenticated devices. Short-lived access tokens reduce exposure time, but tokens must be validated for signature, issuer, audience, time claims and permitted algorithms. Sheffer et al. (2020) warn against algorithm confusion and weak token verification. Contemporary OAuth guidance additionally recommends exact redirect validation, replay reduction and restricted token privileges (Lodderstedt et al., 2025). In a WebSocket design, token acquisition should occur through the authenticated application flow; the socket should avoid exposing bearer tokens in URLs where logs, analytics or browser history may capture them.

Authorization determines whether an authenticated identity may perform an action. In messaging, the protected resource is the conversation and the action may be send, receive, invite, remove, react, edit, delete or report. The authorization decision should consider current membership, role, block state, account status, device trust and policy restrictions. This resource-oriented approach follows zero-trust reasoning: no request is trusted solely because it originated inside the platform network or from an already-open socket (Rose et al., 2020).

Reliability is also a security and trust property. Each message should carry a unique identifier, conversation identifier, sender-device identifier, sequence or ordering value, creation time, protocol version and ciphertext. Idempotent processing prevents duplicates after retries. Delivery receipts should be separate events because a server acknowledgement, device delivery and user read state represent different facts. A broker or durable log can partition events by conversation, preserving local order while enabling horizontal processing. Data deletion must propagate across message stores, search indexes, caches and backups according to defined retention policy. Cohn-Gordon et al. (2020) show that high-level deletion intentions are often implemented incorrectly unless deletion is modelled as a first-class system operation.

Usability affects whether cryptographic guarantees are realised. Users may misunderstand what end-to-end encryption protects, overlook endpoint compromise or ignore identity verification. Akgul et al. (2021) found that brief in-workflow messages can improve some concepts in controlled settings, although longitudinal effects are difficult to sustain. The interface should therefore communicate encryption state, device changes and recovery consequences without creating repetitive warnings that users dismiss.

Moderation in an encrypted environment should be recipient-driven and evidence-limited. Instead of routine server inspection, a recipient may choose to submit a reported message, its cryptographic evidence and relevant context. Hecate illustrates how accountable reporting can coexist with default confidentiality and sender anonymity properties (Issa et al., 2022). Deniability research also shows that secure messaging can pursue stronger protections against compelled disclosure or device compromise, although such features increase implementation complexity (Chakraborti et al., 2023).

2.2 Theoretical Framework

The proposed design is anchored in Zero Trust Architecture and secure-by-design software engineering. Zero Trust Architecture rejects implicit trust based on network location. Access is granted to a specific resource after evaluating identity, device, session and policy context, and the decision is repeatedly revisited as context changes (Rose et al., 2020). For messaging, the conversation is the protected resource. The WebSocket is only a transport channel; every send, history request, membership change and receipt update remains subject to authorization.

The zero-trust decision can be represented as:

$$D = f(I, V, S, M, R, P) \in \{\text{allow, deny, step-up}\}$$

where I is identity assurance, V is device verification, S is session status, M is conversation membership, R is contextual risk and P is platform policy. “Step-up” can require reauthentication or multi-factor confirmation before high-impact actions such as exporting history, adding a new device or restoring encrypted backups.

Secure-by-design theory requires security to be incorporated into requirements, architecture, implementation, testing and maintenance rather than added after feature completion. NIST’s Secure Software Development Framework organises this responsibility around preparing the organisation, protecting software, producing well-secured software and responding to vulnerabilities (Souppaya et al., 2022). OWASP ASVS provides verifiable application-security requirements for authentication, session management, access control, validation, cryptography and logging (OWASP Foundation, 2021). Together, these perspectives support defence in depth: a failure in one control should not immediately expose the entire system.

The framework also applies the principle of minimised trust. The gateway is trusted to authenticate connections and route messages, but it is not trusted with plaintext or private conversation keys. The broker is trusted to preserve event order and delivery state, but not to interpret content. The database is trusted to persist ciphertext and metadata under access control, but a database disclosure should not directly reveal message text. Endpoints remain sensitive because they hold keys and plaintext; therefore device security, key storage, revocation and recovery are integral rather than peripheral controls.

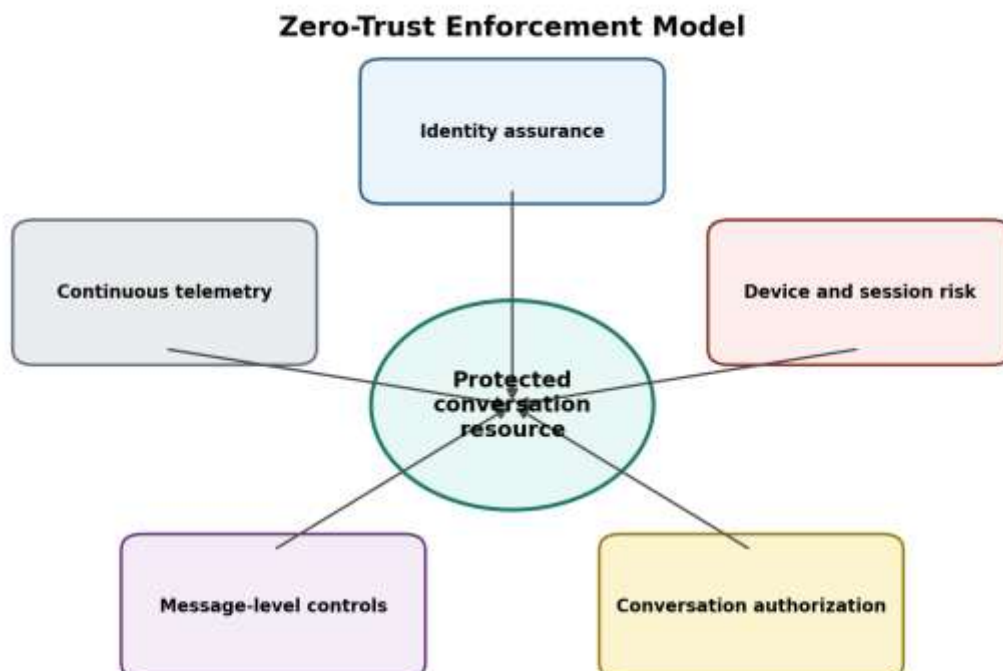


Figure 2.1. Zero-trust enforcement model for messaging actions.

2.3 Empirical Review

Table 2.1 summarises recent studies and standards that inform the proposed system. The evidence demonstrates mature work on individual components, but fewer studies provide a complete prototype that combines WebSocket hardening, end-to-end payloads, conversation authorization, reliable delivery and measurable concurrent performance.

Table 2.1. Empirical Review of Recent Secure Messaging Studies and Standards

Author(s)/Year	Focus/Method	Context	Principal Finding	Gap/Relevance
Kwon et al. (2020)	Designed XRD, a metadata-private messaging system using parallel mix networks.	Large-scale simulated/experimental system.	Demonstrated cryptographic metadata privacy and horizontal scale; reported substantial latency trade-offs.	Shows that content encryption alone does not hide communication relationships.
Cohn-Gordon et al. (2020)	Developed DELF for deletion correctness in online social networks.	Three deployed social-network environments.	Detected omissions and implementation errors in distributed deletion workflows.	Motivates explicit deletion events across messaging stores and replicas.
Akgul et al. (2021)	Studied in-workflow explanations of end-to-end encryption.	Survey and longitudinal secure-messaging study.	Short messages improved some concepts, but sustained behavioural effects were limited.	Supports simple, contextual encryption and device-change notices.
Afzal et al. (2021)	Analysed encrypted Signal network traffic for forensic artefacts.	Android Signal traffic traces.	Encrypted activity types remained distinguishable through network patterns.	Highlights metadata and traffic-analysis limits of E2EE.
Issa et al. (2022)	Proposed Hecate for abuse reporting with sealed sender.	Cryptographic construction and performance analysis.	Combined accountable reporting with privacy, deniability and secrecy properties.	Supports recipient-triggered reporting rather than routine plaintext access.
Alatawi & Saxena (2023)	Systematised E2EE authentication ceremonies.	Comparative analysis of secure messengers.	Identified usability and verification gaps across applications.	Informs optional high-assurance verification and transparent key-change design.
Bhuse (2023)	Reviewed cryptographic primitives used in social-media E2EE.	Comparative conceptual review.	Emphasised strong primitives and protocol transparency.	Supports use of reviewed protocols rather than proprietary cryptography.
Chakraborti et al. (2023)	Developed Wink for plausible deniability	Prototype with existing E2EE applications.	Reported negligible overhead while adding	Shows advanced security goals beyond

	under compromise.	device key		hidden communication.	deniable epochs,	baseline confidentiality.
Barnes et al. (2023)	Standardised group agreement.	MLS key	IETF specification.	protocol	Defined group epochs, forward secrecy and post-compromise security mechanisms.	Provides the preferred production basis for secure group conversations.
Beurdouche et al. (2025)	Defined the wider MLS architecture and security goals.		IETF architecture specification.		Clarified delivery-service, identity and application responsibilities around MLS.	Confirms that cryptography requires surrounding platform services.
Perkasa et al. (2025)	Implemented E2EE chat using Diffie–Hellman, AES-256 and microservices.		Prototype implementation.		Demonstrated feasibility of encrypted chat within a service architecture.	Provides implementation evidence but limited integrated threat and load evaluation.
Souppaya et al. (2022)	Specified secure software development practices.		NIST framework and mappings.		Connected organisational, development and vulnerability-response practices.	Guides secure lifecycle governance for the proposed system.

2.4 Gap in Literature

Recent studies have advanced individual dimensions of secure messaging, but they rarely unite them in an implementable social-media architecture. Kwon et al. (2020) prioritised metadata privacy and scale, whereas Cohn-Gordon et al. (2020) concentrated on deletion correctness. Akgul et al. (2021) examined users' understanding of end-to-end encryption, and Afzal et al. (2021) demonstrated that encrypted traffic still exposes forensic patterns. Issa et al. (2022) addressed accountable abuse reporting, while Alatawi and Saxena (2023) evaluated authentication ceremonies and Chakraborti et al. (2023) investigated deniability. The MLS specifications formalise group-key security, but do not prescribe an application stack, database model, WebSocket hardening strategy or measurable delivery pipeline (Barnes et al., 2023; Beurdouche et al., 2025). Consequently, evidence remains limited on how identity, encryption, authorization, reliability, moderation and low-latency transport can operate together within one intra-platform system. This study closes that gap through an integrated design, prototype implementation, security testing and concurrent-load evaluation.

3.0 Materials and Methods

The study adopted a design-science research methodology. Design science is appropriate because the research problem requires the construction and evaluation of an artefact rather than observation alone. The process comprised: problem identification; requirements definition; threat modelling; architecture and data design; prototype implementation; functional security testing; performance testing; and interpretation against the stated objectives. The artefact is the secure messaging framework and its executable single-node prototype.

3.1 System Requirements

Functional requirements included user authentication, persistent connection establishment, direct-message transmission, ciphertext relay, delivery acknowledgement, offline persistence, conversation membership checks, message deduplication, device registration and abuse reporting. Non-functional requirements included low latency, horizontal scalability, confidentiality, integrity, availability, revocation, privacy-preserving logging, observability and compatibility with web and mobile clients.

Security requirements were derived from the zero-trust model and OWASP verification guidance. The gateway must reject unauthorised origins, require authentication immediately after connection, validate token claims, restrict message size, enforce schema validation, rate-limit connections and messages, re-check authorization for each conversation action and close sessions after logout, expiry or risk escalation. Private keys must remain on endpoints or protected device keystores. Server stores must contain ciphertext and only the minimum routing metadata required for delivery.

Table 3.1. Core Functional and Security Requirements

Requirement Domain	Design Requirement	Proposed Mechanism
Authentication	Only registered users and approved devices may establish messaging sessions.	MFA-ready identity service; short-lived access token; device identifier.
Authorization	Every action must be checked against current conversation membership and role.	Resource-level policy decision at message processing time.
Confidentiality	Intermediary services should not read ordinary message content.	Client-side AEAD; private keys remain on endpoints.
Integrity and replay resistance	Modified or duplicated messages must be detected.	AEAD, unique IDs, sequence values and replay cache.
Reliability	Messages require durable acceptance, delivery and read-state semantics.	Broker partitioning, idempotency and separate receipt events.
Availability	Persistent connections must resist flooding and backpressure failure.	Quotas, rate limits, bounded frames, queues and circuit breakers.
Privacy and governance	Metadata, logs and retention must be minimised and controlled.	Ciphertext storage, pseudonymous logs and deletion workflow.
Safety	Users need reporting, blocking and moderation without universal plaintext access.	Recipient-triggered evidence and policy service.

3.2 Threat Model

The adversary model includes an unauthenticated Internet attacker, a malicious authenticated user, a compromised browser session, an attacker controlling a hostile website, a network observer, a database thief and a partially compromised internal service. The design does not claim to protect plaintext on an endpoint that is fully compromised while the user is actively reading messages. It also does not eliminate all metadata leakage because routing requires some conversation and delivery information.

Representative threats include credential theft, token replay, cross-site WebSocket hijacking, insecure direct object references, message injection, oversized frames, connection exhaustion, replayed ciphertext, duplicated delivery, unauthorised group access, database disclosure, key-directory substitution and sensitive logging. Controls are applied at several layers so that an attacker must bypass identity, session, authorization, cryptography and operational limits rather than a single perimeter.

Table 3.2. Threat–Control Matrix

Threat	Attack Description	Primary Controls
--------	--------------------	------------------

Cross-site WebSocket hijacking	Malicious site initiates an authenticated socket using ambient credentials.	Strict Origin allow-list; SameSite cookies; explicit socket authentication.
Token theft or replay	Stolen bearer token is reused from another device or context.	Short expiry; audience/issuer checks; device binding; revocation; sender-constrained tokens where possible.
Broken object authorization	Authenticated user targets a conversation they do not belong to.	Per-message membership and role checks; deny by default.
Message injection	Payload carries malformed schema, script or database input.	Schema validation; prepared storage operations; safe rendering; content encoding.
Replay and duplication	Captured ciphertext or retried frame is processed repeatedly.	Unique message IDs; sequence windows; idempotent consumers; replay cache.
Connection/message flooding	Attacker consumes sockets, CPU, memory or broker capacity.	Connection quotas; token bucket; maximum frame size; backpressure; circuit breakers.
Database disclosure	Attacker obtains message store or backup.	Ciphertext-only content; encrypted storage; least privilege; key separation.
Key substitution	Directory replaces recipient public key.	Signed device keys; transparency log; user verification; change notifications.
Sensitive logging	Tokens or plaintext are written to operational logs.	Structured allow-listed events; redaction; restricted retention and access.
Endpoint compromise	Malware reads plaintext and keys on a device.	Secure keystore; screen/app protection; device revocation; key evolution; risk signals.

3.3 Proposed System Architecture

The architecture separates the social-media application from specialised messaging services. Clients obtain an authenticated platform session through the identity service. A short-lived access token is then presented during WebSocket establishment through a protected cookie, subprotocol exchange or immediate authentication frame; the prototype uses the first application frame to avoid token exposure in the URL. The API gateway terminates TLS, validates the expected origin and forwards accepted upgrades to the WebSocket gateway.

The WebSocket gateway maintains a connection registry that maps authenticated device identifiers to active sockets. It validates message envelopes, applies rate limits, checks conversation membership and publishes authorised events to a broker. Partitioning by conversation preserves relative event order. The persistence service stores ciphertext, message identifiers, sequence numbers and delivery status. Active recipients receive events through their registered connections; offline recipients receive a generic push notification that reveals no plaintext.

A public-key directory distributes device public keys and signed pre-key material but never stores private keys. Clients establish or update a conversation secret and derive message keys locally. For group conversations, a production implementation can adopt MLS epochs and membership commits. A moderation service accepts only user-initiated reports containing the selected plaintext, evidence and necessary context. An audit service records connection, authentication, authorization and security events without recording message bodies or bearer tokens, consistent with OWASP logging guidance (OWASP Foundation, 2025b).

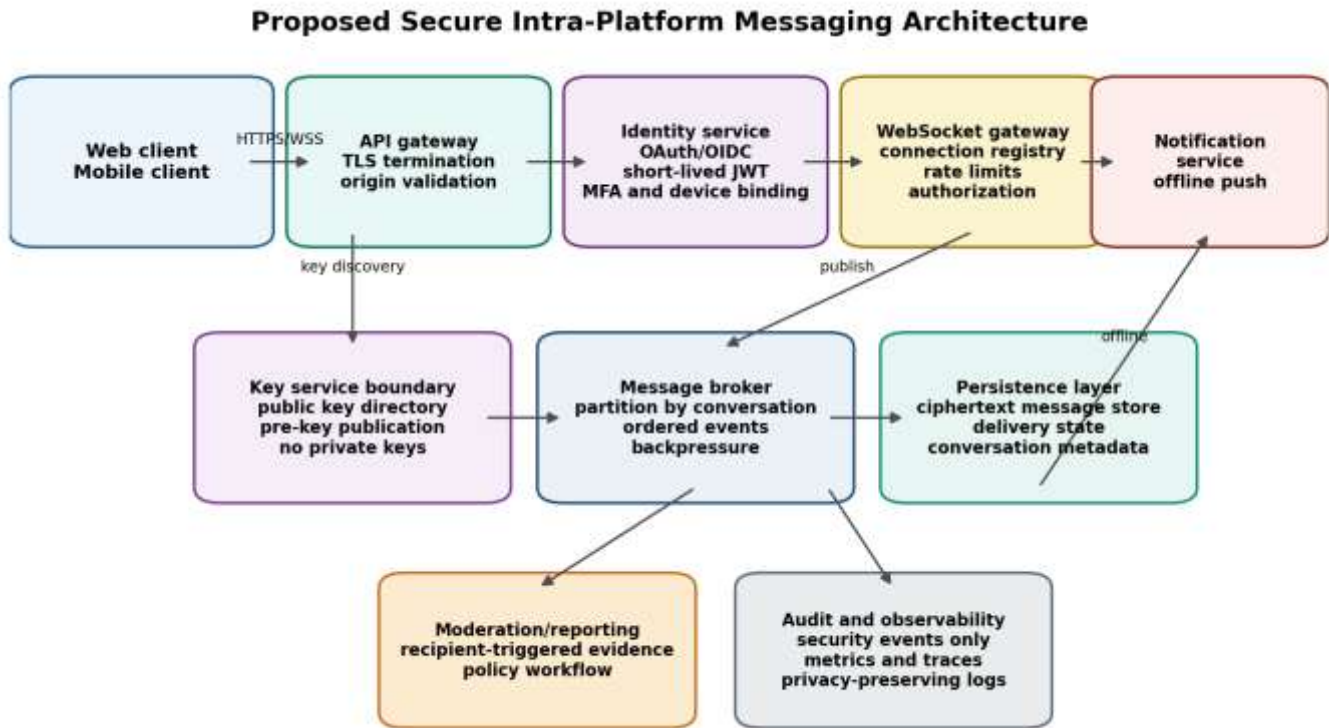


Figure 3.1. Proposed secure intra-platform messaging architecture.

3.4 Cryptographic and Message Model

Each logical message is divided into routing metadata and an encrypted payload. The routing envelope contains a protocol version, message identifier, conversation identifier, sender-device identifier, client timestamp and ciphertext length. Sensitive application content—text, reply references, media keys and optional reactions—is encoded and encrypted on the sender device using an authenticated-encryption algorithm.

For a message m at key step t :

$$c = \text{AEAD_Encrypt}(K_t, N_t, m, \text{AAD}) \quad (1)$$

where K_t is the current message key, N_t is a unique nonce and AAD binds non-secret routing fields such as protocol version, conversation identifier and message identifier. The recipient computes:

$$m = \text{AEAD_Decrypt}(K_t, N_t, c, \text{AAD}) \quad (2)$$

Decryption fails if ciphertext or authenticated metadata has been modified. The prototype uses AES-256-GCM because it provides confidentiality and integrity in a widely available implementation. A production two-party protocol should derive K_t through a reviewed ratchet rather than reusing a static key. A simplified evolution relation is:

$$K_{(t+1)} = \text{HKDF}(K_t, \text{context} || \text{message_id} || \text{epoch}) \quad (3)$$

Old keys are erased after the allowed out-of-order window. Group messaging should use MLS to manage membership changes and epoch secrets, which provides forward secrecy and post-compromise security goals for dynamic groups (Barnes et al., 2023; Beurdouche et al., 2025).

Replay protection combines unique message identifiers, per-sender sequence values and a bounded cache of recently processed identifiers. The server deduplicates routing events, while clients reject ciphertext already accepted. Key-directory responses should be authenticated and transparency-monitored so that a malicious service cannot silently substitute device keys. Identity verification should be available for high-risk users without forcing every ordinary conversation through a burdensome ceremony, reflecting the usability concerns identified by Alatawi and Saxena (2023).

3.5 Logical Data Model

The data model is ciphertext-centric. A user can own several devices, and a conversation can contain several users through membership records. Messages reference a conversation and sender device, while receipts reference a message and recipient device. This distinction supports multi-device delivery without pretending that an account-level receipt identifies which device actually received the event.

The message table stores immutable core fields and append-only state transitions. Editing is represented as a new encrypted event that supersedes a previous message rather than overwriting history invisibly. Deletion uses tombstone events and a background erasure workflow across primary storage, replicas, indexes, caches and backups. The system should expose a deletion status that distinguishes immediate user-interface removal from completed backend erasure. This approach reflects the deletion-correctness concerns documented by Cohn-Gordon et al. (2020).

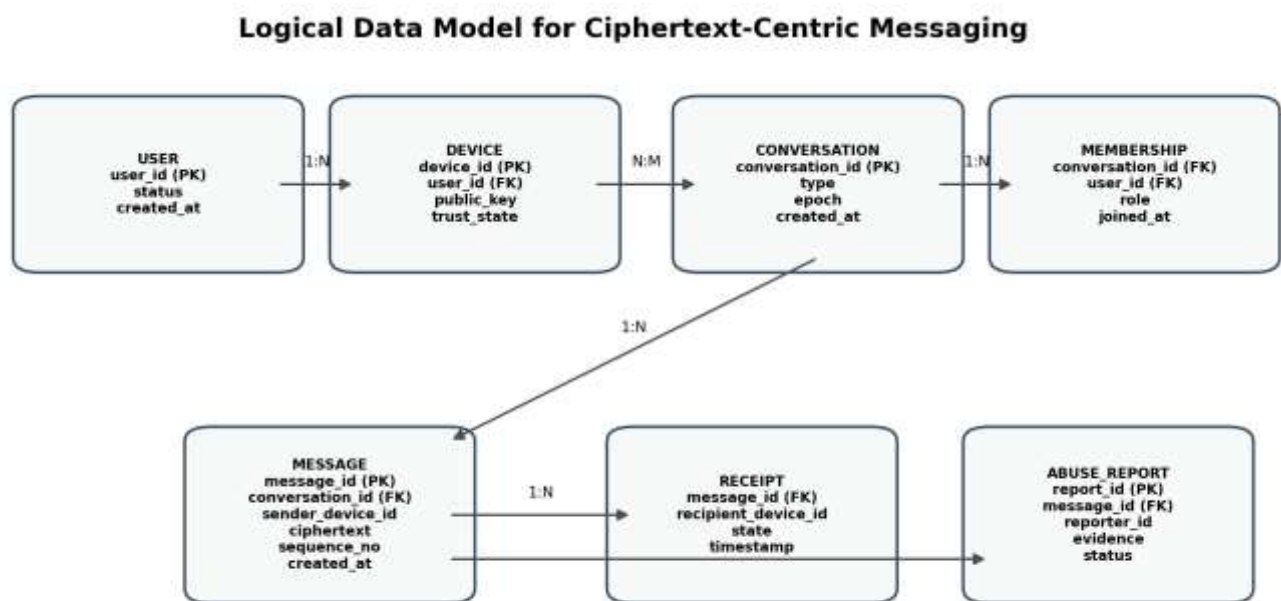


Figure 3.2. Logical data model for ciphertext-centred messaging.

Table 3.3. Principal Data Entities

Entity	Selected Fields	Purpose
User	user_id, status, created_at	Platform identity and account state.
Device	device_id, user_id, public_key, trust_state	Device-specific key and session binding.
Conversation	conversation_id, type, epoch, created_at	Direct/group container and cryptographic epoch.
Membership	conversation_id, user_id, role, joined_at	Authorisation relationship and role.
Message	message_id, conversation_id, sender_device_id, ciphertext, sequence_no	Immutable encrypted message event.
Receipt	message_id, recipient_device_id, state, timestamp	Accepted, delivered or read transition.

Abuse report	report_id, message_id, reporter_id, evidence, status	Recipient-submitted evidence.	moderation
--------------	--	-------------------------------	------------

3.6 Message Processing Algorithm

The send algorithm operates as follows. First, the client confirms an authenticated, non-expired session and a current conversation key. Second, it creates a globally unique message identifier and advances the local key state. Third, it encrypts the payload with authenticated metadata. Fourth, it transmits the envelope over WSS. Fifth, the gateway validates schema, size, rate and session state. Sixth, the authorization service verifies that the sender is an active conversation member and is not blocked from sending. Seventh, the gateway performs deduplication and publishes the ciphertext event to the conversation partition. Eighth, the store commits the event and the gateway returns a server-accepted acknowledgement. Ninth, recipient devices verify, decrypt and return delivery receipts. Finally, the sender receives receipt updates as independent events.

The distinction among “accepted”, “delivered” and “read” prevents misleading status. An accepted message is durably recorded by the service. A delivered message reached at least one recipient device. A read message was intentionally marked as displayed by the recipient client, subject to the recipient’s privacy settings. Failed or expired delivery is represented explicitly and can be retried with idempotency rather than creating a new logical message.

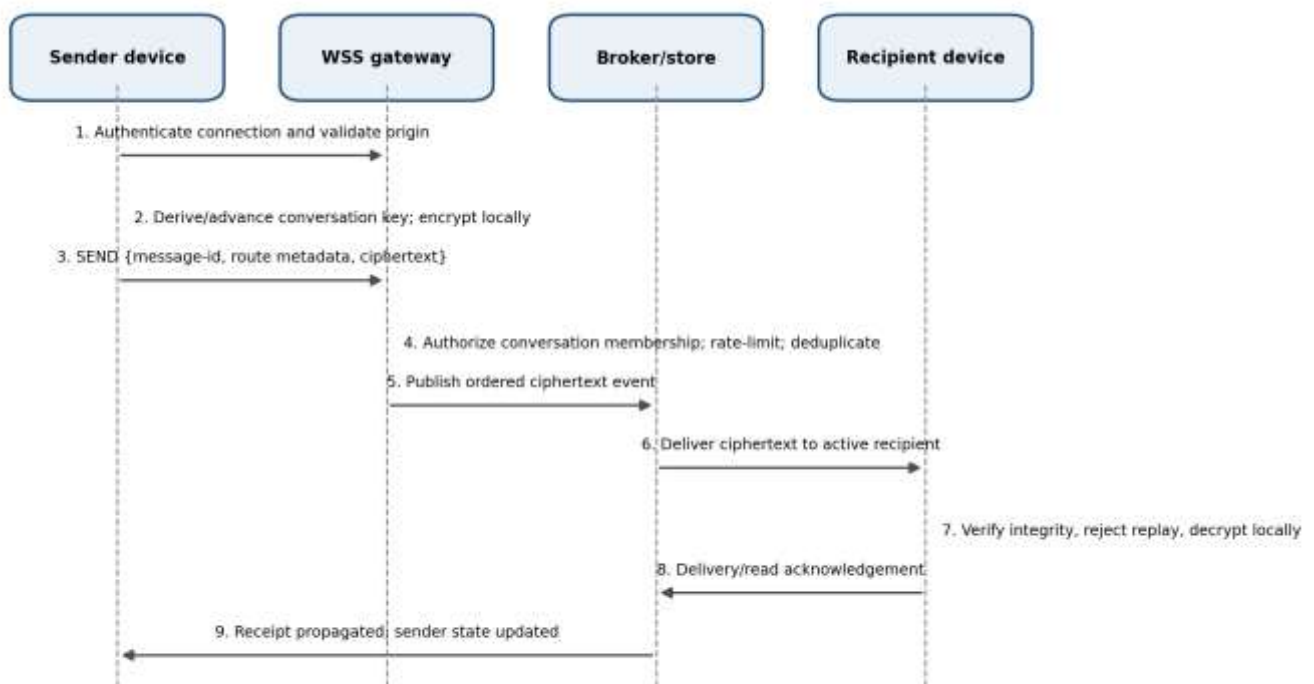


Figure 3.3. Secure message lifecycle and delivery sequence.

3.7 Prototype Implementation

The prototype was implemented in Python using the asynchronous aiohttp framework. The server exposed a WSS endpoint and an HTTPS request-response endpoint for comparison. A locally generated TLS certificate protected both transports. JSON Web Tokens used an explicit HS256 algorithm and were validated for signature, issuer, audience and expiration. Although a production platform would normally use asymmetric signing and managed key rotation, symmetric signing was sufficient for the controlled single-service benchmark because the experiment examined transport and message-processing overhead rather than identity-provider distribution.

During the WebSocket handshake, the server required the origin <https://social.example>. After upgrade, the first application frame carried the access token. Invalid or expired tokens caused a policy-violation closure. The connection accepted binary ciphertext frames up to one megabyte and enforced a maximum of 100 messages per second per

connection. Payloads were generated as 256-byte plaintext values and encrypted client-side with AES-256-GCM using a 96-bit nonce and authenticated protocol context. The server remained ciphertext-blind and echoed the frame to model relay. The HTTPS comparison used the same payload encryption and token checks but sent every message as a separate POST over a persistent client session.

The local environment used Linux, five logical CPU cores and approximately 5.9 GB of memory. Tests were run on the loopback interface to isolate application and protocol overhead from Internet variability. Consequently, the absolute latency values should not be interpreted as wide-area performance. The relative pattern is useful because both alternatives were tested in the same process environment, with the same encrypted payload size and comparable validation logic.

Table 3.4. Prototype and Benchmark Environment

Parameter	Specification
Operating environment	Linux 6.18, loopback network
Compute	5 logical CPU cores; approximately 5.9 GB RAM
Server framework	Python aiohttp asynchronous server
Transport conditions	WSS versus HTTPS with local TLS
Application protection	JWT validation; origin allow-list; AES-256-GCM; 1 MB maximum frame
Load levels	25, 50, 100 and 200 concurrent users
Workload	15 encrypted 256-byte messages per user

3.8 Evaluation Procedure

Performance tests used four concurrency levels: 25, 50, 100 and 200 users. Each user sent fifteen encrypted messages, producing 375, 750, 1,500 and 3,000 messages per protocol condition. Round-trip latency was measured from the client send operation to receipt of the returned ciphertext. Mean, median, 95th percentile, 99th percentile and aggregate throughput were calculated.

The latency of message i was:

$$L_i = t_{\text{receive},i} - t_{\text{send},i}$$

and throughput was:

$$T = N / \Delta t$$

where N is the number of completed message round trips and Δt is the elapsed wall-clock time for the condition. The comparison was descriptive because each condition was run as an engineering benchmark rather than a randomised population sample.

Functional security tests assessed seven cases: a valid origin and token, a cross-site origin, an expired token, missing authentication, a message flood above the configured threshold, an oversized HTTPS payload and an invalid bearer token. A test passed when the system accepted the legitimate case or rejected the malicious/invalid case using the intended status or WebSocket close behaviour.

4.0 Results

The implementation satisfied the core functional path: connection establishment, token authentication, encrypted frame exchange, origin rejection, rate control and size enforcement. The results are presented as prototype evidence rather than a claim that all production threats have been eliminated.

4.1 Functional Security Test Results

All seven functional tests produced the expected result. A connection from the approved origin with a valid token authenticated successfully. A hostile origin was rejected before WebSocket establishment with HTTP 403. Expired and missing authentication states caused the WebSocket to close. A client that transmitted more than 100 frames within one second was closed with policy code 1008. A payload larger than one megabyte received HTTP 413, and an invalid bearer token received HTTP 401.

These outcomes demonstrate that persistent connections can enforce controls at both handshake and message-processing stages. Origin checking reduced cross-site hijacking risk; token validation reduced unauthenticated access; rate limits constrained a single connection's ability to monopolise event-loop capacity; and size limits prevented unrestricted allocation. The tests do not replace penetration testing, fuzzing or formal verification, but they confirm that the proposed safeguards were implemented rather than discussed only at a conceptual level.

Table 4.1. Functional Security Test Outcomes

Test	Passed	Observed Result
Valid origin and token	True	Authenticated
Cross-site WebSocket attempt	True	HTTP 403
Expired JWT	True	Close frame CLOSE
Missing authentication	True	Close frame CLOSE
Message flood above 100/s	True	Close frame CLOSE, code=1008
Oversized HTTP payload	True	HTTP 413
Invalid bearer token	True	HTTP 401

4.2 Performance Results

Mean WebSocket latency increased gradually from 0.83 ms at 25 users to 8.12 ms at 200 users. HTTPS mean latency rose from 5.91 ms to 54.42 ms across the same load levels. The difference became more pronounced as concurrency increased because WebSocket reused an already-established bidirectional channel and avoided repeated HTTP request parsing and response handling.

Tail latency showed an even clearer distinction. At 200 users, the WebSocket 95th percentile was 8.75 ms, whereas the HTTPS 95th percentile was 308.90 ms. The HTTPS distribution contained bursts caused by concurrent request scheduling, even though connection pooling was enabled. WebSocket throughput reached 4,315.55 messages per second at 200 users, compared with 3,623.88 messages per second for HTTPS. At 25 users, HTTPS recorded higher aggregate throughput because many concurrent requests completed in a short burst, but its individual latency was substantially worse. Throughput alone would therefore have hidden the interactive delay experienced by users.

The performance results support persistent WebSocket delivery for interactive messaging. They do not imply that WebSocket is automatically secure or that the measured single-node throughput will be sustained with durable brokers, multi-region replication, push notifications, key-directory queries and full production telemetry. Each added control has cost. The design goal is to preserve acceptable latency while applying necessary controls, not to maximise a synthetic message count by removing them.

Table 4.2. Encrypted Messaging Performance by Concurrent Load

Protocol	Concurrent users	Mean latency (ms)	Median latency (ms)	P95 latency (ms)	Throughput (msg/s)
----------	------------------	-------------------	---------------------	------------------	--------------------

WSS	25	0.83	0.81	0.93	3047.20
HTTPS	25	5.91	3.71	35.44	4142.47
WSS	50	1.72	1.69	1.99	4479.91
HTTPS	50	11.09	7.06	67.66	4423.48
WSS	100	3.69	3.66	4.23	4512.20
HTTPS	100	25.20	16.03	145.77	3887.84
WSS	200	8.12	8.12	8.75	4315.55
HTTPS	200	54.42	35.35	308.90	3623.88

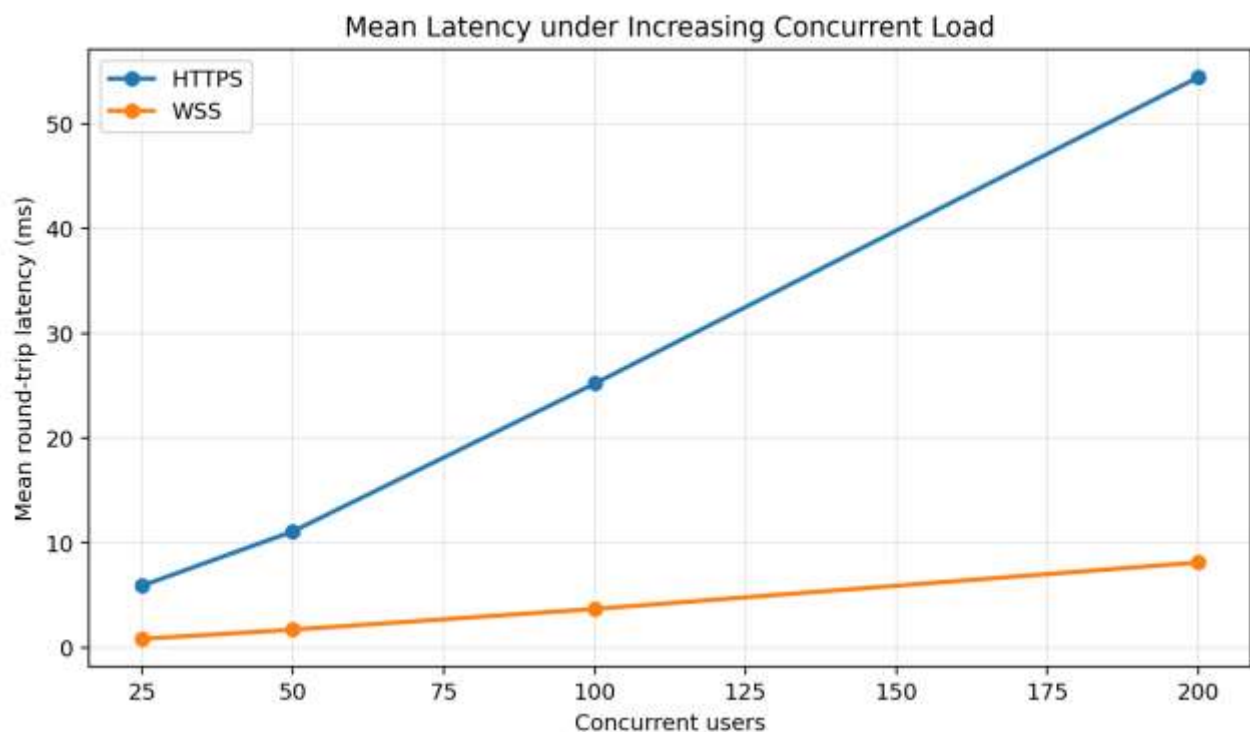


Figure 4.1. Mean round-trip latency under increasing concurrent load.

4.3 Comparative Interpretation

At 200 concurrent users, the mean WebSocket latency was approximately 85 percent lower than the HTTPS comparison. The persistent channel also maintained a narrow relationship between median and 95th-percentile latency, indicating more predictable interactive behaviour in the tested environment. Predictability matters because users experience occasional long delays more negatively than a small increase in average delay.

The benchmark also shows that application-layer encryption did not prevent low latency for small messages. AES-GCM operations were performed on every request and response, yet the WebSocket path remained below 10 ms mean latency at the highest tested condition. This finding supports a layered approach: encryption, authentication and origin checking need not be sacrificed to achieve responsive messaging. However, key agreement, secure hardware, attachment encryption and multi-device fan-out will add processing and should be measured independently.

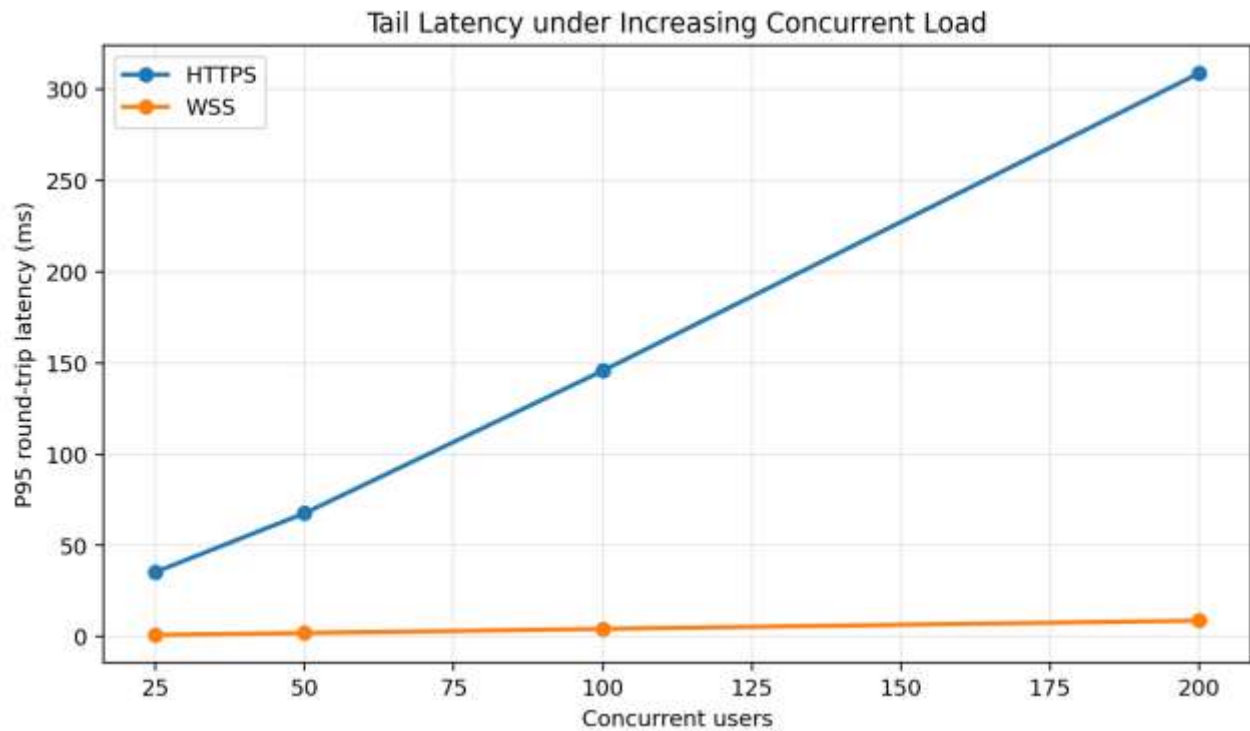


Figure 4.2. Ninety-fifth percentile latency under increasing concurrent load.

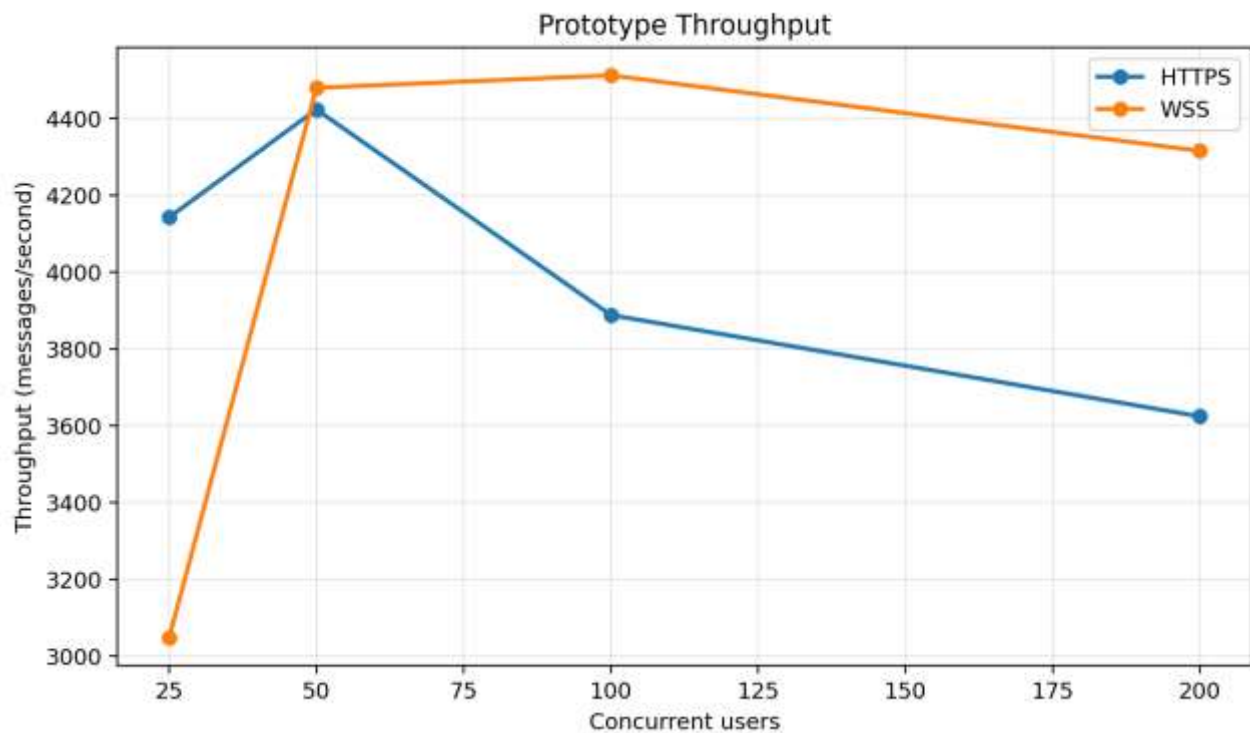


Figure 4.3. Aggregate prototype throughput under increasing concurrent load.

5.0 Discussion

The results answer the research questions in several ways. First, secure intra-platform messaging requires controls beyond TLS. TLS protects the connection, but the architecture still needs origin validation, token verification, conversation authorization, payload encryption, replay controls, rate limits, size limits, secure persistence, logging discipline, deletion workflows and endpoint key management. Treating any single control as sufficient creates an avoidable trust concentration.

Second, end-to-end confidentiality can coexist with server-side routing and reliable delivery when the envelope separates non-secret routing fields from ciphertext content. The server requires enough metadata to identify the conversation, order events, determine recipients and manage receipts, but it does not need message text. Authenticated additional data binds the ciphertext to important routing fields, preventing undetected substitution. This separation is consistent with the direction of MLS, which standardises secure group state while leaving application delivery architecture to implementers (Barnes et al., 2023; Beurdouche et al., 2025).

Third, the measured prototype confirms the latency advantage of a persistent channel under local concurrent load. The advantage is not merely fewer bytes; it is the ability to maintain connection state, push events immediately and avoid repeated request lifecycle overhead. Nevertheless, persistent state creates operational complexity. A load balancer must route or rediscover user connections, a connection registry must expire stale entries, and logout or token revocation must propagate to live sockets. Horizontal scaling therefore requires shared presence state or broker-based fan-out rather than assuming one user remains attached to one server.

Fourth, privacy and moderation should not be framed as an all-or-nothing choice. Routine plaintext inspection is incompatible with strong end-to-end confidentiality, but recipient-triggered reporting can reveal a selected message and cryptographic evidence. Hecate demonstrates that research can preserve confidentiality for unreported messages while strengthening accountability for reported abuse (Issa et al., 2022). A practical platform can begin with user-controlled reporting, metadata-based abuse detection, rate analysis, block controls and account-level investigation, while avoiding secret server decryption keys.

The literature also cautions against overclaiming. Afzal et al. (2021) show that encrypted traffic can remain classifiable through observable patterns. Kwon et al. (2020) demonstrate that strong metadata privacy may impose substantial latency and infrastructure costs. Chakraborti et al. (2023) pursue deniability against more powerful compromise scenarios than ordinary commercial systems. These studies indicate that “secure” is contextual. A mainstream social platform may optimise for low-latency confidentiality and abuse reporting, while a high-risk activist platform may prioritise metadata resistance and deniability even at higher operational cost.

Usability remains critical. Device-verification prompts that are too frequent will be ignored, while invisible key changes may expose users to substitution attacks. Akgul et al. (2021) found that educational interventions do not automatically produce lasting understanding. The proposed design therefore uses progressive disclosure: ordinary users see a clear encryption state and device-change notice; high-risk users can verify fingerprints or QR codes; and risky recovery actions trigger stronger warnings and step-up authentication.

The framework has implications for data governance. Ciphertext reduces the consequences of a content database breach, but it does not remove obligations for metadata protection, retention limitation or deletion. Membership records, timestamps, IP addresses and device identifiers can be sensitive. Logs should record event categories and pseudonymous identifiers, not complete message bodies or access tokens. OWASP Foundation (2025b) explicitly recommends logging security-relevant WebSocket events while excluding sensitive data. Deletion must be engineered across stores rather than represented only as a user-interface flag, as shown by Cohn-Gordon et al. (2020).

From a software-engineering perspective, the system should be built through secure lifecycle practices. Threat models and abuse cases should be versioned with code; cryptographic dependencies should be pinned and inventoried; token and authorization tests should run automatically; and changes to the message envelope should be backward-compatible and fuzz-tested. NIST SSDF emphasises that vulnerability reduction depends on organisational practices throughout development, not only on final penetration testing (Souppaya et al., 2022). Independent cryptographic review is especially important because seemingly small choices in nonce generation, key reuse, identity binding or recovery can invalidate otherwise strong algorithms.

The prototype’s limitations are deliberate and must guide interpretation. It used one server process, one host, loopback networking, a self-signed certificate, symmetric JWT signing and an echo-style relay. It did not include a distributed broker, durable multi-node database, actual mobile secure enclave, attachment service, MLS library, push-notification provider or Byzantine key-transparency log. It also did not simulate packet loss, mobile radio transitions, regional latency, server restart or millions of idle connections. Production evaluation should therefore add load tests with long-lived connections, broker failure, database failover, reconnection storms, adversarial fuzzing, key-rotation experiments and independent security review.

5.1 Recommended Production Deployment Model

A production deployment should place stateless or lightly stateful WebSocket gateways behind a load balancer that supports connection draining and health-aware routing. Gateways should register active device connections in a distributed presence store with short leases. Authorised events should be published to a durable broker partitioned by conversation or another ordering key. Consumers should persist ciphertext before issuing a durable acceptance receipt. Cross-region replication should preserve a home region for conversation ordering or use an explicit conflict-resolution model.

Identity tokens should be short-lived, audience-restricted and signed with rotated asymmetric keys. Refresh tokens should remain outside the WebSocket channel and be protected against replay. Session revocation events should close active sockets promptly. Device public keys should be signed, versioned and exposed through a transparency mechanism. Private keys should be held in platform-supported secure storage, and encrypted backup recovery should require a user-controlled secret or hardware-bound recovery process rather than a universal platform key.

Operational controls should include per-account, per-device, per-IP and per-conversation rate limits; maximum connection and frame sizes; backpressure; circuit breakers; schema versioning; idempotency; malware scanning for uploaded attachments before end-to-end wrapping where appropriate; and generic push notifications. Metrics should measure connection count, authentication failures, send-to-accept latency, accept-to-deliver latency, dropped frames, broker lag, retry rates and unusual authorization failures without collecting plaintext.

6.0 Conclusion

The study designed and implemented a secure intra-platform real-time messaging framework for social media applications. The framework combines persistent WebSocket communication with end-to-end encrypted payloads, short-lived identity tokens, conversation-level authorization, replay and duplication controls, ciphertext persistence, reliable acknowledgements, privacy-preserving logging and recipient-driven reporting. Its theoretical foundation integrates zero-trust access decisions with secure-by-design development.

The measured prototype showed that these baseline protections can coexist with responsive performance. WebSocket mean round-trip latency remained between 0.83 and 8.12 ms across 25–200 concurrent local users, while the HTTPS comparison reached 54.42 ms at the highest load. All seven functional security tests produced the intended result. These findings support the use of WSS as the real-time transport, provided it is not mistaken for end-to-end encryption and is protected throughout the connection lifecycle.

The contribution is therefore not a claim that one prototype solves every secure-messaging problem. It is a coherent engineering blueprint that connects identity, cryptography, authorization, reliability, data governance, moderation and performance. Future work should implement a reviewed ratcheting or MLS library, deploy distributed brokers and multi-region gateways, evaluate mobile energy consumption, test metadata-minimisation techniques, add key transparency, conduct formal protocol analysis and perform independent penetration testing with real social-media workloads.

6.1 Recommendations

Social-media developers should adopt end-to-end payload encryption as a separate layer above WSS; enforce authorization for every conversation action rather than only at connection time; use short-lived, audience-restricted tokens and immediate socket revocation; validate WebSocket origins and message schemas; apply bounded frame sizes, connection quotas and multi-dimensional rate limits; store ciphertext instead of plaintext wherever product requirements permit; design idempotent delivery with explicit accepted, delivered and read states; build deletion as a cross-store workflow; provide transparent device and recovery notices; enable recipient-controlled abuse reporting; minimise metadata and sensitive logs; and subject cryptographic and session designs to independent review before production deployment.

References

1. Afzal, A., Hussain, M., Saleem, S., & Shahzad, M. K. (2021). Encrypted network traffic analysis of secure instant messaging application: A case study of Signal Messenger app. *Applied Sciences*, 11(17), 7789. <https://doi.org/10.3390/app11177789>

2. Akgul, O., Bai, W., Das, S., & Mazurek, M. L. (2021). Evaluating in-workflow messages for improving mental models of end-to-end encryption. In 30th USENIX Security Symposium (pp. 447–464). USENIX Association.
3. Alatawi, M., & Saxena, N. (2023). SoK: An analysis of end-to-end encryption and authentication ceremonies in secure messaging systems. In Proceedings of the 16th ACM Conference on Security and Privacy in Wireless and Mobile Networks (pp. 187–201). <https://doi.org/10.1145/3558482.3581773>
4. Barnes, R., Beurdouche, B., Robert, R., Millican, J., Omara, E., & Cohn-Gordon, K. (2023). The Messaging Layer Security (MLS) protocol (RFC 9420). Internet Engineering Task Force. <https://doi.org/10.17487/RFC9420>
5. Beurdouche, B., Rescorla, E., Omara, E., Inguva, S., & Duric, A. (2025). The Messaging Layer Security (MLS) architecture (RFC 9750). Internet Engineering Task Force. <https://doi.org/10.17487/RFC9750>
6. Bhuse, V. (2023). Review of end-to-end encryption for social media. *International Conference on Cyber Warfare and Security*, 18(1), 35–37. <https://doi.org/10.34190/iccws.18.1.1017>
7. Chakraborti, A., Suci, D., & Sion, R. (2023). Wink: Deniable secure messaging. In 32nd USENIX Security Symposium (pp. 1271–1288). USENIX Association.
8. Cohn-Gordon, K., Damaskinos, G., Neto, D., Cordova, J., Reitz, B., Strahs, B., Obenshain, D., Pearce, P., & Papagiannis, I. (2020). DELF: Safeguarding deletion correctness in online social networks. In 29th USENIX Security Symposium. USENIX Association.
9. Issa, R., Alhaddad, N., & Varia, M. (2022). Hecate: Abuse reporting in secure messengers with sealed sender. In 31st USENIX Security Symposium (pp. 2335–2352). USENIX Association.
10. Kwon, A., Lu, D., & Devadas, S. (2020). XRD: Scalable messaging system with cryptographic privacy. In 17th USENIX Symposium on Networked Systems Design and Implementation (pp. 759–776). USENIX Association.
11. Lodderstedt, T., Bradley, J., Labunets, A., & Fett, D. (2025). Best current practice for OAuth 2.0 security (RFC 9700). Internet Engineering Task Force. <https://doi.org/10.17487/RFC9700>
12. OWASP Foundation. (2021). OWASP Application Security Verification Standard 4.0.3.
13. OWASP Foundation. (2025b). WebSocket security cheat sheet. OWASP Cheat Sheet Series.
14. Perkasa, M. J. P., Sari, R. F., & Harwahu, R. (2025). Implementation of secure end-to-end encrypted chat application using Diffie–Hellman key exchange and AES-256. *Engineering Proceedings*, 107(1), 98. <https://doi.org/10.3390/engproc2025107098>
15. Rose, S., Borchert, O., Mitchell, S., & Connelly, S. (2020). Zero trust architecture (NIST SP 800-207). National Institute of Standards and Technology. <https://doi.org/10.6028/NIST.SP.800-207>
16. Sheffer, Y., Hardt, D., & Jones, M. (2020). JSON Web Token best current practices (RFC 8725). Internet Engineering Task Force. <https://doi.org/10.17487/RFC8725>
17. Souppaya, M., Scarfone, K., & Dodson, D. (2022). Secure Software Development Framework (SSDF) version 1.1: Recommendations for mitigating the risk of software vulnerabilities (NIST SP 800-218). National Institute of Standards and Technology. <https://doi.org/10.6028/NIST.SP.800-218>