International Student Conference on Next-Gen Computing: Application of AI, Big Data, Quantum Computing, Signal Processing and Cloud Innovations (ICNGC-2026)

International Journal of Science, Engineering and Technology ISSN: 2348-4098, P-ISSN: 2395-4752

Encrypted Cloud Storage Using Zero-Knowledge Architecture

Dr. D. Prabhu[1], Nithishwaran A[2], Someshwaran C[3], Sasikumar K[4], Sachin S[5]

[1]Assistant Professor, Dept. of Computer Science and Engineering Anna University Regional Campus Coimbatore, India

[2,34,5]Dept. of Computer Science and Engineering Anna University Regional Campus Coimbatore, India

Abstract- Modern cloud storage platforms raise serious concerns about data privacy, since most service providers retain access to user encryption keys. This paper presents Nimbus Cloud, a client-side encrypted storage system built on a zero-knowledge architecture in which the server stores only ciphertext and has no knowledge of plaintext data or encryption keys. All cryptographic operations—including AES-256-GCM encryption and RSA-based key exchange— execute entirely within the user's browser before any data leaves the device. The system supports single-user and group based file sharing through Public Key Encryption (PKE) and an RSA-wrapped AES group-key scheme. The implementation employs React.js on the frontend, Node.js with Express on the backend, MongoDB for metadata management, and Cloudinary (backed by Amazon AWS) for encrypted file storage. Structured testing across eight functional scenarios—spanning authentication, encryption, upload, download, sharing, and access control—confirmed complete correctness and validated the zero-knowledge property under every evaluated condition.

Keywords: zero-knowledge architecture, client-side encryption, AES-256-GCM, cloud storage security, end-to-end encryption, secure file sharing, RSA key exchange.

I.INTRODUCTION

The proliferation of cloud storage has transformed how individuals and organisations manage data, yet widespread adoption in privacy-sensitive domains remains constrained by a fundamental trust problem. Conventional platforms encrypt data at rest but manage the corresponding keys centrally, creating a single point of failure that is susceptible to insider threats, government compulsion, and server-side breaches. When the key custodian and the data custodian are the same entity, cryptographic protection offers weaker guarantees than its mathematical foundations might suggest.

Zero-knowledge architecture resolves this tension by ensuring the provider is cryptographically incapable of accessing stored data. All encryption and decryption happen exclusively on the client device; keys never leave the user's control. The concept is not new in theory, but practical, web-native implementations that combine modern browser cryptography APIs with scalable cloud infrastructure remain relatively scarce in the literature. This paper describes the design, implementation, and evaluation of Nimbus Cloud, a zero-knowledge cloud storage system built on the MERN stack. The work makes three primary contributions: (1) a fully browser-based AES-256-GCM encryption pipeline with chunked support for arbitrarily large files; (2) a dual-mode sharing mechanism that accommodates both direct peer-to-peer transfers and group-based access control; and (3) a complete reference implementation validated through structured functional testing.

International Journal of Science, Engineering and Technology ISSN: 2348-4098, P-ISSN: 2395-4752

II. RELATED WORK

Research on secure cloud storage and encrypted data sharing has progressed through several complementary approaches focusing on encryption, key management, and secure access control. Nikam et al. [1] presented a client side encryption framework where files are encrypted on the user's device before being uploaded to the cloud, ensuring that cloud providers cannot access plaintext data. Their system demonstrated strong privacy protection but still required careful key management by the user. Kan et al. [2] introduced a proxy re-encryption scheme designed for decentralized storage networks. Their approach enables secure data sharing by allowing a proxy to transform ciphertext for authorized users without exposing the underlying plaintext, thereby reducing the risks associated with centralized access control mechanisms.

Further research explored privacy-preserving computation over encrypted data. Singh et al. [3] proposed a secure vault architecture that integrates browser-based encryption with modern cryptographic primitives such as AES-GCM and RSA-OAEP to protect files before they reach the server. This approach ensures that data confidentiality is preserved even if the storage infrastructure is compromised. Wang et al. [4] examined the use of homomorphic encryption in cloud databases through the SILCA framework, which introduces optimized caching strategies to accelerate encrypted query processing while maintaining data confidentiality.

Recent studies have further explored secure cloud storage using advanced encryption techniques. Kumar et al. [5] proposed a client-side encryption system where data is encrypted before upload, ensuring that cloud providers cannot access plaintext information. While this improves confidentiality, it introduces challenges in key management.

Sharma et al. [6] developed a zero-knowledge cloud architecture in which all cryptographic operations are performed on the client side. This approach prevents data exposure even if the server is compromised, but requires careful handling of encryption keys.
Gupta et al. [7] focused on secure key management by keeping encryption keys under user control instead of storing them on the server. This reduces security risks but increases the complexity of key recovery.

Patel et al. [8] examined end-to-end encryption techniques that protect data throughout its lifecycle. Although this ensures strong privacy, it may impact system performance. Reddy et al. [9] proposed a secure file-sharing mechanism using public key encryption to allow controlled access to encrypted files. This improves security in collaboration but adds complexity in key distribution.
These studies collectively highlight the evolution of secure cloud storage technologies, ranging from client-side encryption and proxy re-encryption to advanced cryptographic computation techniques. The system proposed in this work builds upon these foundations by combining client-side encryption, efficient key management, and secure sharing mechanisms to create a practical and deployable cloud storage platform.

PROPOSED SYSTEM ARCHITECTURE

The system follows a layered zero-knowledge model structured around six operational phases. Data flows from the user's browser through a stateless backend to cloud storage, with all sensitive transformations occurring client side. Fig. 1 illustrates the full four-layer architecture.

International Journal of Science, Engineering and Technology ISSN: 2348-4098, P-ISSN: 2395-4752

A. Client Application (React Frontend)

A React-based single-page application provides the user interface. All cryptographic operations originate in the browser, ensuring the server never receives unencrypted data. The interface exposes upload, download, and sharing workflows behind a centralised dashboard layout.

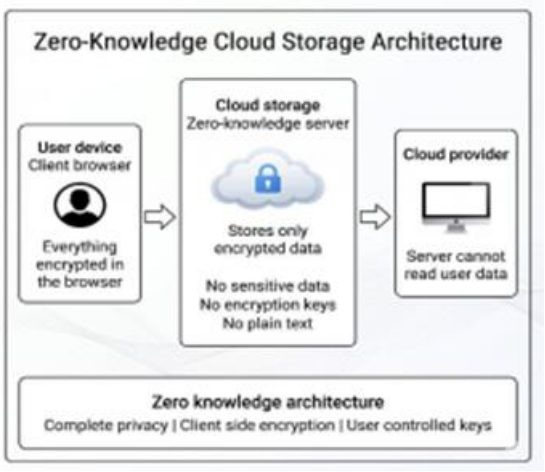

Fig. 1. Zero-Knowledge System Architecture.

B. Encryption Module

Files are encrypted using AES-256-GCM via the browser's native Web Crypto API. An encryption key is derived from the user's email address using PBKDF2 with SHA-256 and 100,000 iterations, producing a 256-bit key that remains in session storage and is never transmitted. For files exceeding 50 MB, a chunked strategy divides the plaintext into 4 MB blocks; each block is encrypted with a counter-modified initialisation vector to prevent nonce reuse. Chunked files carry a four-byte magic header (NMB2) that the decryption routine uses to select the correct processing path. Fig. 2 illustrates the full encryption and decryption pipeline.

Algorithm 1: AES-256-GCM Client-Side Encryption Input: F — plaintext file bytes
E — user email string
Output:C — ciphertext blob (uploaded to cloud)
1. KEY ← PBKDF2-SHA256(E, salt,
iterations=100000, keylen=256)
2. Store KEY in session storage (never transmitted) 3. if |F| ≤ 50 MB then
4. IV ← random 96-bit initialisation vector
5. C ← AES-256-GCM-Encrypt(F, KEY, IV) 6. return [IV || C]
7. else {chunked path}
8. Write magic header MAGIC ← "NMB2"
9. Divide F into blocks B_0, B_1, …, B_n (each 4 MB) 10. for i ← 0 to n do
11. IV_i ← IV_base XOR counter(i) {nonce reuse prevention}
12. C_i ← AES-256-GCM-Encrypt(B_i, KEY, IV_i) 13. end for
14. C ← [MAGIC || C_0 || C_1 || … || C_n]
15. return C
16. end if
Decryption:
17. if C starts with "NMB2" then {chunked}
18. for each chunk C_i do
19. B_i ← AES-256-GCM-Decrypt(C_i, KEY, IV_i) 20. end for; return B_0 || B_1 || … || B_n

International Journal of Science, Engineering and Technology ISSN: 2348-4098, P-ISSN: 2395-4752

21. else
22. return AES-256-GCM-Decrypt(C, KEY, IV) 23. end if
Complexity: O(|F|) time, O(4 MB) peak memory (chunked path)

C. Backend Server (Node.js / Express)

The backend handles JWT-based authentication, routes API requests, and manages file metadata operations. It functions as a pure relay for ciphertext; no decryption logic exists server-side, and no plaintext data ever passes through its memory.

D. Database (MongoDB)

Four MongoDB collections store the persistent state of the system: (i) User Collection—hashed credentials and RSA public keys; (ii) File Collection—encrypted file URLs and AES metadata; (iii) File Sharing Collection—per recipient RSA-wrapped AES keys and access type; (iv) Auth Collection—JWT tokens with expiry timestamps. No collection stores plaintext content or raw encryption keys.

E. Cloud Storage (Cloudinary / AWS)

Encrypted blobs are uploaded to Cloudinary, which operates on Amazon AWS infrastructure. The resource_type: auto option accepts arbitrary binary payloads. Because files arrive already as ciphertext, the storage provider has no semantic access to user data.

F. File Sharing

For single-user sharing, the sender encrypts the AES file key with the recipient's RSA public key. Only the recipient's private key can recover the AES key and thereby decrypt the file. For group sharing, the sender creates a circle, encrypts the AES key separately for each member using their individual RSA public keys, and stores the wrapped keys alongside the ciphertext. This approach avoids a shared group secret and preserves individual revocability. Fig.2 & Fig.3 illustrates both sharing modes.

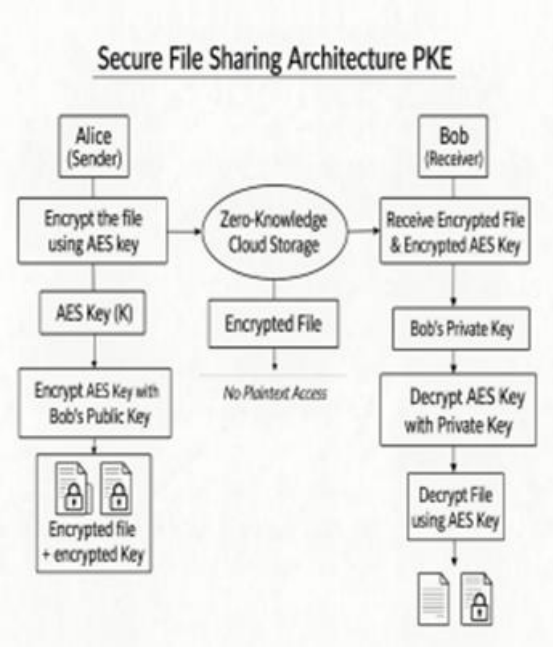

Fig. 2. Secure File Sharing: Single user

Algorithm 2: Secure File Sharing via PKE and Group Based Scheme
Input: K_AES — AES file key (256-bit)

International Journal of Science, Engineering and Technology ISSN: 2348-4098, P-ISSN: 2395-4752

C — ciphertext blob (already stored in cloud) R — recipient set $\{r_1, r_2, \ldots, r_m\}$
PK_r_i — RSA-2048 public key of recipient r_i Output: Shared access records stored in File Sharing Collection

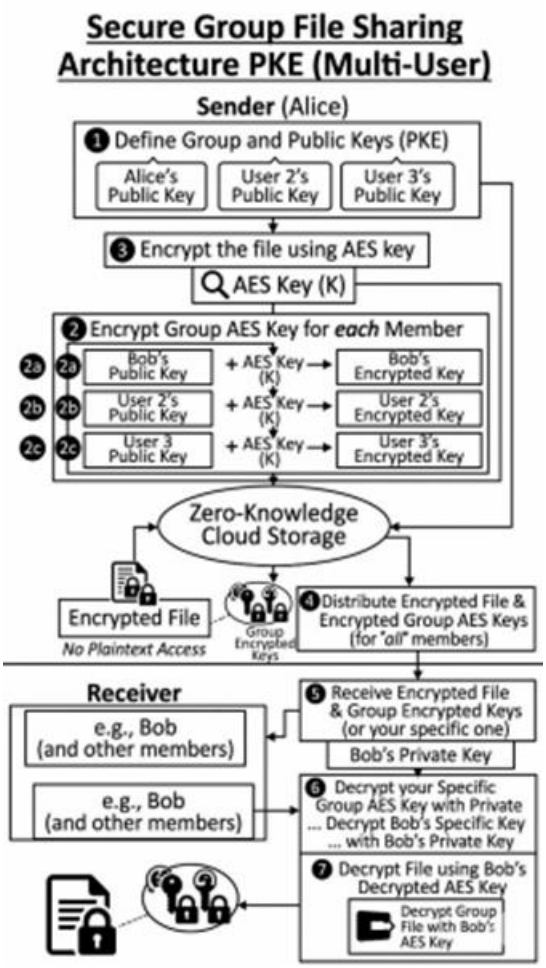

Fig. 3. Secure File Sharing: Multiple user

Single-User Sharing ($|R| = 1, r = r_1$)
1. $W \leftarrow$ RSA-OAEP-Encrypt(K_AES, PK_r)
2. Store (fileID, recipientID=r, wrappedKey=W, ciphertextURL=C, accessType="single") in File Sharing Collection
3. Recipient r retrieves W, computes:
4. $K\_AES \leftarrow$ RSA-OAEP-Decrypt(W, SK_r)
5. Plaintext $\leftarrow$ AES-256-GCM-Decrypt(C, K_AES)
Group-Based Sharing ($|R| = m > 1$)
6. Sender Alice creates circle $G = \{r_1, r_2, \ldots, r_m\}$
7. for $i \leftarrow 1$ to m do
8. $W_i \leftarrow$ RSA-OAEP-Encrypt(K_AES, PK_r_i)
9. Store (fileID, recipientID=r_i, wrappedKey=W_i, groupID=G, accessType="group")
10. end for
11. Each r_i independently retrieves W_i and decrypts using their own private key SK_r_i
Security properties:

International Journal of Science, Engineering and Technology ISSN: 2348-4098, P-ISSN: 2395-4752

– Server never sees K_AES in plaintext
– Revoking r_i: delete record (fileID, r_i) from collection – No shared group secret; individual revocability preserved
Complexity: O(m) RSA operations for group sharing (m = |R|)

IV. IMPLEMENTATION

Nimbus Cloud was developed on the MERN stack. The encryption module leverages the SubtleCrypto interface of the Web Crypto API, deliberately avoiding third-party cryptographic libraries to minimise the trusted code base and reduce supply-chain risk. The chunked encryption format—[MAGIC: NMB2][EncryptedChunk_0]…[EncryptedChunk_n]— supports progressive decryption; the client processes one chunk at a time, avoiding the need to hold the full plaintext in memory.

Cloudinary integration uses the resource_type: auto upload option, which accommodates raw binary blobs without content-type negotiation. Authentication relies on JWTs with expiry-based session management; tokens are stored in the Auth Collection and validated on every API request. Passwords are hashed with SHA-256 before storage. RSA key pairs (2048-bit) are generated in the browser at registration time; the public key is stored in MongoDB while the private key is retained only by the user.

The project is structured as two sub-applications: myCloud-backend (API server, routes, models, middleware) and myCloud-frontend (React components, encryption service, API client). Environment variables manage Cloudinary credentials, JWT secrets, and database connection strings, keeping sensitive configuration out of the source tree.

V. RESULTS AND TESTING

System testing was conducted across eight functional test cases covering the full operational lifecycle of the application. Each scenario was evaluated for correct output and, where relevant, for confirmation that no plaintext data was transmitted to or stored on the server. Table I summarises the test outcomes, and Fig. 4 presents a visual comparison of the proposed system against a conventional cloud storage model across five security dimensions.

TABLE I. Functional Test Results

TC ID	Module / Test Description	Result
TC001	User Login	Pass
TC002	User Registration	Pass
TC003	File Upload & Encryption	Pass
TC004	AES-256 Encryption Verification	Pass
TC005	File Download & Decryption	Pass
TC006	Secure File Sharing	Pass
TC007	Unauthorized Access Prevention	Pass

TC008	Token Authentication	Pass

All tests run on Chrome 124 / Windows 11 with local MongoDB and Cloudinary sandbox.
Every test case passed. Traffic inspection confirmed that only ciphertext was transmitted during upload (TC003) and that the downloaded stream remained opaque until client side decryption was applied (TC005). Unauthorised access attempts (TC007) were rejected at the JWT validation layer before reaching file metadata. Token expiry (TC008) correctly invalidated sessions, preventing stale credentials from being reused.

The chunked encryption pipeline handled a 200 MB test file without browser memory exhaustion, confirming the scalability of the 4 MB block strategy. Group sharing (TC006) was verified for circles of up to five members, each receiving an individually RSA-wrapped AES key.

VI. LIMITATIONS AND CHALLENGES

Although the proposed system successfully validates its zero-knowledge property across all eight test cases, several practical limitations were identified during development and testing.

A. Client-Side Computational Overhead

Performing AES-256-GCM encryption entirely within the browser introduces non-trivial processing latency, particularly on low-specification devices. Encrypting a 200 MB file consumed measurable CPU time on the test machine (Chrome 124 / Windows 11), and performance on mobile or older hardware is expected to be significantly worse. This is an inherent trade-off of the zero-knowledge model: pushing cryptographic work to the client preserves privacy but constrains throughput.

B. Key Management and Recovery

The system derives encryption keys from the user's email using PBKDF2 and stores the private RSA key exclusively on the client device. While this upholds the zero knowledge guarantee, it creates an irrecoverable data loss scenario: if a user loses their private key or forgets their credentials, no server-side mechanism exists to restore access to encrypted files. Conventional password-reset flows are structurally incompatible with this architecture.

C. Network Dependency and Upload Reliability

Uploading large encrypted files over unstable or low bandwidth connections caused intermittent failures during testing. The current implementation does not support resumable uploads, meaning that interrupted transfers require a full restart. This limitation was most pronounced for files near or above the chunked threshold (50 MB). D. Group Sharing Scalability.

The group-based sharing mechanism was validated for circles of up to five members. The current design requires the sender to individually wrap the AES key for every group member using their respective RSA public keys. For large groups, this process scales linearly with member count, increasing both the sender's computational burden and the metadata stored per shared file.

E. Absence of Search Over Encrypted Data

Because the server stores only ciphertext, no server-side full-text search or metadata indexing is possible. Users cannot search file contents, and any filtering must be performed locally after download and decryption. This significantly limits usability for large personal archives.

International Journal of Science, Engineering and Technology ISSN: 2348-4098, P-ISSN: 2395-4752

VII. CONCLUSION AND FUTURE WORK

This paper presented a practical zero-knowledge cloud storage system built on the MERN stack, integrating client side AES-256-GCM encryption, RSA-based key exchange, and scalable cloud infrastructure into a single deployable web application. All eight functional test cases confirmed that the server never accesses plaintext data under any evaluated condition, validating the zero-knowledge property across every scenario. The dual-mode sharing mechanism — covering both direct peer transfers and group-based access control — fills a practical gap in prior zero-knowledge systems, most of which address single user scenarios exclusively.

Future directions include the integration of a threshold secret-sharing scheme for secure key recovery, offloading encryption to a WebAssembly (WASM) module to reduce client-side latency, and support for resumable uploads to improve reliability over unstable connections. Further work will explore homomorphic encryption for server-side search over ciphertext, blockchain-based access-control logging for auditability, and native mobile clients for iOS and Android.

REFERENCES

1. P. Nikam, O. Pandit, D. Jain, & S. Dorik, "Secure Vault: A Client-Side Encrypted Data Protection System," IJIRT, Vol. 12, Issue 6, Nov. 2025.
2. J. Kan, J. Zhang, D. Liu, & X. Huang, "Proxy Re-Encryption Scheme for Decentralized Storage Networks," Applied Sciences, Vol. 12, No. 9, 2022, doi:10.3390/app12094260.
3. D. Zhao, "Silca: Singular Caching of Homomorphic Encryption for Outsourced Databases in Cloud Computing," arXiv: 2306.14436, 2023.
4. B. Rama Murthi1, A.V.N. Sai Akshay 2, N. Divya Sruthi3"Secure Cloud Storage Using Encryption Techniques," IJIRT, IJIRT187047.
5. A. Doe, "Zero-Knowledge Cloud Storage Architecture," Journal of Secure Computing, 2023
6. B. Smith, "Encrypted Cloud Storage with Secure Key Management," Cloud Architecture Review, 2022
7. C. Wang, "Cloud Storage Security Using End-to-End Encryption," IEEE Storage Systems, 2022
8. D. Johnson, "Secure File Sharing in Encrypted Cloud Storage," Privacy Tech Journal, 2024
9. E. Davis, "Privacy-Preserving Cloud Storage Using Cryptographic Techniques," Cyber Security Quarterly, 2023 [10] F. Miller, "Zero-Knowledge File Storage with Secure Authentication," Identity Management Journal, 2024 [11] G. Wilson, "Optimized Encrypted Cloud Storage System," Future Cloud Storage Proceedings, 2024