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

SecureSync: A Secure Cloud File Sharing Platform

Utkrisht Ahirwar [1], Amit Baghel [2], Lalit Pal [3], Apoorva Pal [4], Dr. A.P Srivastava [5], Abhay Chaudhary [6]

Department of Computer Science and Engineering, NITRA Technical Campus, Ghaziabad, U.P., India

Email: utkrishtahirwar@gmail.com amitbaghel82000@gmail.com lalitpal.3344@gmail.com
apoorvapal403@gmail.com anand.infotech@gmail.com id.abhay22@gmail.com

Abstract

Cloud-based file-sharing platforms are now integral to both professional and personal computing environments. Yet dominant solutions such as Google Drive and Dropbox demonstrate consistent deficiencies in user-level access accountability, time-bounded sharing, and resource governance at the application layer. This paper presents SecureSync, a secure cloud file-sharing platform built on AWS S3 and Java Spring Boot, integrating Clerk-issued JWT authentication, a credit-based upload governance mechanism, time-limited link expiry, and a write-only audit logging subsystem. The audit trail records all file operations with user identity, timestamp, and IP address, providing non-repudiation capabilities absent from mainstream platforms. The credit system introduces a novel per-upload resource constraint layer not addressed in prior literature. Implemented with React/Next.js on the frontend and MongoDB for persistence, SecureSync demonstrates that comprehensive security and accountability improvements can be achieved within a standard web application architecture without requiring cryptographic infrastructure or distributed ledger overhead.

Keywords: cloud file sharing, JWT authentication, audit trail, nonrepudiation, AWS S3, link expiry, credit-based governance, Spring Boot, MongoDB, access control

I. INTRODUCTION

The rapid adoption of cloud computing over the past decade has fundamentally altered the way digital assets are stored, managed, and shared. Industry surveys report that more than 60 percent of enterprise workloads were hosted on cloud infrastructure by 2023, with file storage and sharing representing one of the highest-volume use cases. Platforms such as Google Drive, Dropbox, and Microsoft OneDrive have become the default choice for collaborative file management, offering scalable storage at low cost with straightforward access from any device or operating system.

Despite their widespread adoption, these platforms exhibit persistent and well-documented security limitations. The most significant of these is the absence of user-facing audit trails. Standard users of Google Drive and Dropbox have no native mechanism to determine who accessed their shared files, when the access occurred, or from which network location the request originated. This absence of attributable access records directly violates the principle of non-repudiation, which requires that the occurrence of an action and the identity of the actor be verifiable after the fact [10]. Without non-repudiation, file owners have no means of detecting unauthorised access that occurs through legitimately shared links, and no evidence to present in the event of a data governance dispute.

A second structural limitation is the use of permanent, non-expiring share links. When a user generates a sharing link in Google Drive or Dropbox, that link remains valid indefinitely unless manually revoked by the file owner. In practice, shared links are frequently forwarded to unintended recipients, posted in public forums, or retained in email threads long after the intended sharing period has concluded. The absence of automatic temporal constraints on link validity represents a significant and often overlooked attack surface in the threat model of cloud file-sharing platforms.

A third gap, which receives limited attention in the academic literature, is the absence of any resource governance mechanism at the application layer. Existing platforms impose storage quotas at the account level but do not constrain the rate or volume of individual upload operations in a user-attributable, application-controlled manner. This creates conditions for storage abuse in multi-tenant or freemium deployment scenarios where individual users may consume disproportionate resources without any per-operation constraint.

The intersection of these three gaps defines the security and governance profile of current cloud file-sharing platforms. SecureSync was designed to address this intersection in a single deployable system built on a production-grade technology stack, demonstrating that the composite gap can be closed without requiring specialised cryptographic expertise or distributed infrastructure. This paper presents SecureSync and its five primary contributions:

- Clerk-based JWT authentication with JWKS public key validation, providing stateless cryptographically verifiable user identity across all protected API endpoints without server-side session storage
- AWS S3 integration with a private bucket access policy, ensuring that file retrieval is possible only through authenticated application endpoints and that storage credentials are never exposed to client-side code
- A credit-based upload governance mechanism restricting upload operations to users with sufficient credit balance, providing per-user resource control at the application layer independent of storage infrastructure quotas
- Time-limited shareable link expiry with automatic server-side revocation, returning HTTP 410 Gone on expired token access attempts and recording every attempt in the audit log
- A write-only audit logging subsystem recording all file operations with event type, user identity, filename, client IP address, and timestamp, providing non-repudiation and real-time access transparency to file owners

The remainder of this paper is structured as follows. Section II reviews related work. Section III presents a detailed literature review. Section IV describes the methodology. Section V presents the system architecture. Section VI details system features. Section VII covers implementation. Section VIII provides a security analysis. Section IX presents results and evaluation. Section X discusses findings and limitations. Section XI concludes the paper with directions for future work.

II. RELATED WORK

Research in cloud storage security has concentrated on three broad themes: encryption and confidentiality, access control and revocation, and audit and accountability. Prior work in each area has advanced the state of knowledge considerably; however, no existing work integrates all three dimensions into a single deployable platform targeting ordinary users rather than enterprise or cryptographic research contexts.

Bai et al. [1] analysed the sharing mechanisms of Google Drive, Dropbox, and Microsoft SkyDrive, identifying structural weaknesses that could lead to data leakage without user awareness. This work motivates SecureSync but is limited to analysis without remediation. Subashini and Kavitha [2] surveyed security issues in cloud service delivery models, establishing encryption as the primary defensive mechanism but not addressing post-sharing accountability. Hashizume et al. [3] catalogued cloud vulnerabilities with attention to centralised key management risks.

Mishra and Ghosh [4] proposed Multi-Authority CP-ABE for fine grained access control, while Li et al. [5] extended ABE with user revocation. Both approaches incur significant computational overhead and address neither audit trails nor resource governance. Singh and Sharma [6] combined symmetric and asymmetric cryptography for efficient key management. Rani and Kaur [7] applied digital fingerprinting for leakage detection, and Wang and Zhang [8] proposed time-limited access delegation. Yan et al. [9] surveyed privacy protection mechanisms noting that existing schemes are fragmented. Chen and Zhao [10] formally identified accountability as a distinct and non-substitutable cloud security attribute.

Across this body of work, a consistent gap emerges. Individual security mechanisms are proposed and evaluated in isolation. No lightweight, production-deployable platform unifies JWT authentication, credit governance, temporal link expiry, and user-facing real-time audit logging. SecureSync is designed to fill this composite gap.

III. LITERATURE REVIEW

Bai et al. [1] conducted one of the earliest systematic security evaluations of mainstream cloud storage platforms through network traffic analysis and protocol reverse engineering. The study revealed that all three analysed platforms contained weaknesses in their link sharing mechanisms that could allow unintended parties to access files without authentication. Shared links could be discovered by parties beyond the intended recipients due to predictable URL patterns and the absence of access controls on link use. The work was significant in establishing empirically that widely trusted platforms carry inherent design-level security deficiencies. However, it was purely analytical and proposed no architectural remediation. SecureSync directly addresses the core finding through UUID-based share tokens with server-side expiry enforcement and audit logging of every access attempt, including those that arrive after token expiry.

Subashini and Kavitha [2] presented a comprehensive survey of security issues across cloud service delivery models, identifying eleven distinct security challenge categories for SaaS deployments.

The authors proposed encryption-based data protection as the primary defence mechanism and advocated for separation of data storage from key management. While significantly advancing the theoretical understanding of cloud security requirements, the framework was oriented toward the storage layer and did not address operational accountability after data access. Users who legitimately downloaded a file and subsequently redistributed it without authorisation could not be identified or deterred by the proposed mechanisms. SecureSync extends this work by addressing post-access accountability through its audit logging subsystem, which provides an attributable record of every file operation.

Hashizume et al. [3] identified that most cloud security solutions at the time of their review relied on centralised key management architectures, introducing a single point of trust and a single point of failure. They noted that encryption mechanisms, while effective at protecting data at rest, do not address the trust gap between the cloud user and the cloud provider. The work highlighted the need for mechanisms that reduce reliance on provider trustworthiness. SecureSync addresses this through a private S3 bucket policy that prevents the storage provider from serving files directly to clients, combined with application-layer authentication that the provider infrastructure cannot bypass, ensuring that file access always passes through the SecureSync authorisation and audit layer.

Mishra and Ghosh [4] proposed a Multi-Authority Ciphertext-Policy Attribute-Based Encryption scheme enabling fine-grained, policy driven access control in cloud environments. The scheme embedded access policy specifications directly into ciphertexts at encryption time, allowing decryption only by users whose attributes satisfied the embedded policy. Multiple independent attribute authorities reduced the single-authority trust assumption. The authors demonstrated adaptive security under standard computational assumptions. However, the scheme incurred non-trivial overhead for complex access policies and did not address operational accountability after decryption. SecureSync achieves comparable access differentiation through simpler public/private visibility controls while adding the audit trail layer that ABE-based schemes do not address.

Li et al. [5] extended Attribute-Based Encryption with traceability and user revocation mechanisms. Their scheme allowed the system to identify which user's private key was used to decrypt a ciphertext, enabling accountability at the cryptographic layer. User revocation was implemented without requiring re-encryption of stored ciphertexts. Despite these advances, the scheme required ongoing administrative management of user attribute sets and introduced key update procedures that added operational complexity. SecureSync implements user-level access revocation through simpler mechanisms, specifically the public/private toggle and automatic link expiry, achieving the practical security objective without requiring attribute infrastructure or cryptographic re-keying operations.

Singh and Sharma [6] proposed a hybrid cryptographic framework combining AES symmetric encryption for data confidentiality with RSA asymmetric encryption for key exchange. The framework demonstrated improved encryption throughput relative to RSA-only schemes. However, the framework was scoped entirely to the confidentiality dimension and did not consider what occurs after an authorised user legitimately accesses and potentially misuses data.

SecureSync addresses this post-access dimension through audit logging, which records every download and share access event with user attribution, providing a deterrent and detection mechanism that operates independently of whether file content is encrypted.

Rani and Kaur [7] developed a cloud storage security framework incorporating digital fingerprinting for detecting unauthorised data redistribution. The framework embedded forensic markers into files at upload time, enabling investigators to trace the source of a leaked file. The approach demonstrated practical effectiveness for deliberate leakage scenarios. However, it was entirely reactive, requiring a leakage event to have already occurred before the forensic data became actionable. SecureSync provides proactive visibility through its audit dashboard, enabling file owners to detect suspicious access patterns before a leakage incident occurs, representing a shift from reactive forensics to proactive accountability monitoring.

Wang and Zhang [8] proposed a time-limited access delegation mechanism for cloud data sharing. Their scheme allowed data owners to grant access rights for a defined temporal window with automatic revocation enforced through a cryptographic delegation protocol, proven secure against collusion between revoked users and the cloud provider. However, the scheme operated at the cryptographic layer and did not provide a mechanism for recording or surfacing access events to the data owner in real time. An authorised user could access the data multiple times within the valid window without the owner having any visibility. SecureSync integrates temporal access constraints with a complete audit trail, ensuring that every access event within and beyond the permitted window is attributed and visible to the file owner.

Yan et al. [9] conducted a systematic review of cloud computing privacy protection mechanisms, categorising research schemes across access control, attribute-based encryption, trust models, and reputation systems. The review identified that existing research outputs were scattered without a unifying framework, making it difficult to compose individual contributions into deployable systems. The authors identified accountability as an underserved dimension in existing schemes. Their observation that most cloud privacy research focuses on storage-layer protection while neglecting operational accountability directly motivates the audit-centric design of SecureSync.

Chen and Zhao [10] identified five foundational security attributes for cloud computing systems: confidentiality, integrity, availability, accountability, and privacy-preservability. Their formal analysis established that accountability, defined as the ability to attribute every system action to an authenticated identity, is a distinct security requirement that cannot be satisfied by confidentiality or integrity mechanisms alone. A system may protect data from unauthorised access and ensure its integrity while still failing to record who performed legitimate operations and when. The authors called for dedicated accountability mechanisms in cloud storage architectures. SecureSync operationalises this requirement through its write-only audit logging subsystem, making audit records accessible to file owners through the application interface.

The synthesis of these ten works reveals a clear and consistent research gap. Encryption, access control, and forensic accountability have each been studied in depth, but no existing work integrates temporal access control, user-facing real-time audit logging, and resource governance into a single deployable cloud file-sharing platform. SecureSync is designed and evaluated as a direct response to this composite gap.

IV. METHODOLOGY

A. Research Approach

SecureSync was developed using a constructive research methodology, wherein the system itself constitutes the primary research contribution. The research process began with a structured review of security limitations in existing platforms, both from the academic literature and through empirical observation of Google Drive and Dropbox behaviour. The identified gaps informed functional and security requirements. The system was then designed, implemented, and evaluated against those requirements. This approach is appropriate for applied computer science research where the validity of the contribution is demonstrated through a working implementation and measurable evaluation results.

B. Requirements Derivation

Functional and security requirements were derived from three sources. First, the security gap analysis from the literature review identified non-repudiation, temporal access control, and resource governance as unmet requirements. Second, empirical analysis of Google Drive and Dropbox established the feature baseline against which SecureSync is differentiated. Third, established security engineering principles, specifically the CIA triad extended with accountability as formalised by Chen and Zhao [10], provided formal grounding for the security requirements. The resulting requirements are: stateless verifiable authentication, private cloud storage with mediated access, owner-controlled temporal sharing, write-only audit logging with full attribution, and credit-based upload governance.

C. Technology Justification

Clerk JWT: Provides RFC 7519 compliant token issuance and JWKS-based public key verification, enabling stateless authentication without server-side session storage. The use of asymmetric key verification ensures that token forgery requires access to the Clerk private key, which is not held by the application server.

Spring Boot: Provides a mature Spring Security filter chain suitable for JWT interception, aspect-oriented programming capabilities for non-invasive audit logging instrumentation, and dependency injection enabling clean service layer separation across authentication, file management, auditing, and credit management concerns.

AWS S3: Provides 11-nines durability, a private bucket access model, and UUID-based key assignment preventing predictable URL enumeration. All file retrieval is mediated through the application backend; the storage provider infrastructure cannot serve file content directly to clients without the application as an intermediary.

MongoDB: Provides a flexible document model accommodating variable audit event schemas without migration overhead. The audit collection is maintained as a write-only store with no update or delete operations exposed through any application interface, enforced at the repository layer rather than relying on database-level access controls alone.

React/Next.js: Provides server-side rendering and native Clerk SDK integration, enabling secure token management without exposing JWT tokens to client-side JavaScript execution contexts beyond what is necessary for request authorisation.

D. Design Principles

Least Privilege: All files are private by default at upload time. Access must be explicitly granted by the owner. Shared links carry temporal

constraints and are automatically invalidated without requiring manual owner intervention after the specified window elapses.

Separation of Concerns: Authentication, file storage, audit logging, credit management, and payment processing are independent service components with dedicated repositories. No service component has direct access to another component's data store, preventing cross component coupling that could undermine audit integrity.

Audit Immutability: No API endpoint exposes update or delete operations on the audit collection. Records are append-only and verifiable but not modifiable through any application interface, enforced at the Spring Data repository interface level.

Defence in Depth: Security is enforced at the filter level through JWT validation, at the service level through ownership verification, and at the storage level through private S3 bucket policies. Compromise at any single layer does not yield unauthorised file access.

E. Evaluation Design

The evaluation addressed four dimensions: feature completeness relative to existing platforms through a structured comparison table; audit coverage completeness through controlled operation simulation across all event types; link expiry enforcement correctness through boundary condition and concurrent access testing; and audit write overhead through median response time measurement across 100 operations per endpoint. Each criterion directly validates a stated system requirement.

V. SYSTEM ARCHITECTURE

A. Overview

SecureSync is implemented as a three-tier web application following the separation of concerns principle across all layers. The presentation tier is a React/Next.js application with server-side rendering. The application tier is a Java Spring Boot REST API enforcing all authentication, authorisation, business logic, and audit logging. The storage tier comprises AWS S3 for file content and MongoDB for metadata, audit records, credit balances, and user profiles. No direct communication path exists between the presentation tier and the storage tier; all interactions are mediated through the application tier. This ensures that storage access credentials are never accessible to client-side code, and that every file operation is subject to application layer authentication, authorisation, and audit logging. The Spring Boot application is stateless with respect to user sessions, relying entirely on JWT token validation for identity establishment on each request, which enables horizontal scaling without session affinity requirements.

B. Authentication Flow

The ClerkJwtAuthFilter intercepts every inbound HTTP request. Requests matching public path patterns, specifically /public/**, /share/**, /webhooks/**, and /health, bypass JWT validation and proceed directly to the handler. All other requests must carry a valid Bearer token in the Authorization header. The filter decodes the JWT header to extract the key identifier field, retrieves the corresponding

RSA public key from the Clerk JWKS endpoint via
Clerk Jwks Provider, and verifies the token signature, expiry timestamp, and issuer claim using the jjwt library. Validation failures produce an HTTP 403 response before any controller method is invoked. On successful validation, the authenticated Clerk user identifier is stored in the Spring Security Context and made available to all downstream service components without additional database queries.

C. Storage Architecture

File content is stored in a private AWS S3 bucket using UUID generated object keys. The key is stored in the file metadata document in MongoDB alongside the original filename, content type, size in bytes, owner Clerk identifier, a Boolean visibility flag, a share token field, and a share expiry timestamp. File content is never returned directly from S3 to clients; instead, the application tier retrieves S3 object bytes and streams them to the client through the Spring Boot response output stream, ensuring that S3 access credentials and bucket policies are never exposed to the client layer and that all retrievals are subject to application-layer authentication and audit logging.

D. Audit Architecture

The audit logging service receives an event descriptor upon completion of each file operation and persists an AuditLog document to a dedicated MongoDB collection. Each document contains a system generated identifier, the event type enumeration value, the authenticated user's Clerk identifier, the target file's identifier and original filename, the client IP address extracted from the HttpServletRequest using X-Forwarded-For header inspection before falling back to getRemoteAddr(), an ISO 8601 formatted timestamp, the HTTP response status code, and an optional contextual detail string. The audit collection has no corresponding update or delete repository methods. The AuditController exposes a single paginated GET endpoint returning the authenticated user's own records in reverse chronological order.

E. Credit Architecture

Each user account has a corresponding UserCredits document in MongoDB, initialised through a Clerk user.created webhook event processed by ClerkWebhookController after signature verification against the Clerk signing secret. The credit balance is checked synchronously by CreditService prior to each file upload. If the balance is zero, the service raises an application exception that GlobalExceptionHandler maps to HTTP 402 Payment Required. Credits can be replenished through the PaymentController, which integrates with Razorpay to verify payment completion before updating the credit balance. The credit decrement occurs after each successful S3 upload, ensuring users are not charged for failed upload operations.

VI. SYSTEM FEATURES

A. File Upload and Management

Authenticated users may upload files of any content type. Each upload triggers a sequential process: credit balance validation, S3 object creation with a UUID key, MongoDB metadata persistence, credit decrement, and audit log write. Users may list their own files through the file listing endpoint, which returns metadata records filtered by the authenticated user's Clerk identifier. File deletion verifies ownership at the service layer before removing the S3 object, the MongoDB metadata document, and records the deletion in the audit log. All file management operations are protected by JWT authentication and generate audit records regardless of outcome.

B. Public/Private Visibility Control

Each file carries a Boolean visibility flag initialised to false at upload time, making all files private by default. File owners may toggle visibility through the PATCH /files/{id}/toggle-public endpoint, which verifies ownership before modifying the flag value. Public files are accessible at the /public/{id} endpoint without JWT authentication, enabling link-based file distribution for content intended for unrestricted access. Private files are accessible only to the authenticated owner through the download endpoint. All visibility change operations are recorded in the audit log with the resulting visibility state and the owner's identity.

C. Time-Limited Secure Sharing

File owners may generate a time-limited share token for any file. The owner specifies the desired validity duration in hours. The service generates a cryptographically random UUID share token and computes an expiry timestamp by adding the specified duration to the current system time. Both values are written to the file metadata document. The /share/{token} endpoint retrieves the document by token, compares the current timestamp to the stored expiry using a LocalDateTime comparison, and either streams the file content or returns HTTP 410 Gone. Every access attempt, whether successful or expired, produces an audit log entry recording the token, the requesting IP address, and the outcome.

D. Audit Trail Dashboard

The audit dashboard presents file owners with a paginated reverse chronological view of all operations associated with their files. Each entry displays the event type, target filename, client IP address, and ISO 8601 timestamp. The dashboard is implemented as a Next.js server component fetching paginated data from the GET /audit/my-logs endpoint with JWT authentication. The write-only constraint on the audit collection is enforced at the repository layer; no application interface permits record modification. This provides file owners with a verifiable, complete operational history enabling the identification of unauthorised or anomalous access patterns before a data loss event occurs.

E. Credit-Based Upload Governance

The credit system provides per-user upload rate control at the application layer, independent of AWS S3 storage quotas and representing a governance mechanism with no direct counterpart in existing mainstream platforms. New users receive an initial credit allocation on account creation. Each file upload consumes one credit regardless of file size. Users with a zero balance receive an HTTP 402 response and cannot upload until credits are replenished through the payment module. The credit balance is displayed in the dashboard, providing users with ongoing visibility into their consumption. This mechanism introduces a monetisable resource governance layer and prevents storage abuse in multi-tenant deployment scenarios.

VII. IMPLEMENTATION

A. Technology Stack

Component	Technology	Version
Backend	Java Spring Boot	3.x
Frontend	React / Next.js	14.x
Authentication	Clerk (JWT/JWKS)	Latest
File Storage	AWS S3	SDK v2
Database	MongoDB	6.x
Payment	Razorpay	API v1
Build	Maven	3.9

TABLE I. Technology Stack

B. JWT Authentication Filter

The ClerkJwtAuthFilter class extends OncePerRequestFilter and is registered as the first filter in the Spring Security filter chain. On each request, the filter checks the URI against configured public path prefixes. For protected paths, it extracts the Authorization header, splits the Bearer token, base64-decodes the JWT header segment, and reads the kid field using Jackson ObjectMapper. ClerkJwksProvider fetches the Clerk JWKS document from the configured issuer URL and caches the resulting RSA public keys with a configurable TTL to reduce external network calls on high-traffic deployments. The token is verified using Jwts.parserBuilder() with the retrieved public key, configured issuer claim, and a 60-second clock skew allowance consistent with RFC 7519 recommendations. The authenticated Clerk user identifier from the token subject claim is set as the principal in UsernamePasswordAuthenticationToken and stored in the SecurityContextHolder.

C. Audit Logging Service

The AuditService is injected into every service class performing file operations. Following each operation, the calling service invokes AuditService.log() passing the event type enumeration, file metadata reference, and the current HttpServletRequest. The service constructs an AuditLog document populating all required fields and persists it through AuditLogRepository, which extends MongoRepository. No update or delete methods are declared in the repository interface, enforcing write-only semantics at the Java type system level in addition to the architectural constraint. The audit endpoint returns records through a Spring Data Page object with configurable page size, sorted by timestamp descending using a @Query annotation with sort specification.

D. Share Token and Expiry

Share token generation is handled within FileMetadataService.generateShareToken(). The method generates a UUID using UUID.randomUUID(), computes the expiry as LocalDateTime.now().plusHours(durationHours), and persists both fields to the FileMetadataDocument. The share access handler retrieves the document by token using a custom repository method findByShareToken(), compares LocalDateTime.now() with the stored expiry using isBefore(), and returns the file stream or raises ShareExpiredException, which GlobalExceptionHandler maps to HTTP 410 with a descriptive message. Every access attempt produces an audit log entry recording both the token and the access outcome regardless of whether the token was valid.

E. Credit Enforcement

The credit enforcement flow is implemented sequentially within FileMetadataService.uploadFiles(). The method firstcalls UserCreditsService.hasEnoughCredits(),which retrieves the UserCredits document and compares the stored count against the number of files in the request. On failure, a RuntimeException is raised before any S3 or MongoDB write is initiated, mapped to HTTP 402 by GlobalExceptionHandler. If the check passes, the upload proceeds and UserCreditsService.consumeCredit() is called once per file after each successful S3 write. The credit decrement and metadata persistence are separate MongoDB operations; a failed S3 upload does not trigger a credit decrement, ensuring users are not charged for unsuccessful operations.

F. API Endpoint Summary

Method	Endpoint	Auth	Description
POST	/files/upload	JWT	Upload, deduct credit, audit
GET	/files/my	JWT	List authenticated user files
GET	/files/download/{id}	JWT	Stream own file, audit
DELETE	/files/{id}	JWT	Delete own file, audit
PATCH	/files/{id}/toggle-public	JWT	Toggle visibility, audit
POST	/files/{id}/share	JWT	Generate expiring share token
GET	/share/{token}	Public	Access shared file, audit
GET	/public/{id}	Public	Access public file
GET	/audit/my-logs	JWT	Paginated audit log
GET	/credits	JWT	Current credit balance
POST	/webhooks/clerk	SVIX	Handle Clerk user events

TABLE II. API Endpoint Summary

A. Threat Model

VIII. SECURITY ANALYSIS

SecureSync operates under the following threat model. External attackers may attempt to access files without valid credentials, enumerate file identifiers through predictable patterns, replay expired share tokens, or access files through expired share links. Authenticated internal threats include users attempting to access, modify, or delete files belonging to other users. Infrastructure threats include attempts to access S3 object content directly, bypassing application authentication. The system is not designed to protect against compromise of the Clerk authentication infrastructure or AWS account credentials, which represent out-of-scope infrastructure-level threats.

B. Authentication Security

JWT tokens are verified using RSA public keys fetched from the Clerk JWKS endpoint, ensuring that token forgery requires access to the Clerk private signing key. Token expiry is enforced by the jjwt library on every request with a 60-second clock skew allowance consistent with RFC 7519. The filter returns HTTP 403 for all authentication failures without disclosing the nature of the failure, preventing information leakage to potential attackers. The JWKS public key cache reduces external calls while remaining invalidated when Clerk rotates signing keys.

C. Authorisation and Ownership

File ownership is verified at the service layer on every delete, toggle, and share generation operation by comparing the file's stored Clerk identifier against the authenticated principal from the SecurityContext. This check is independent of the JWT validation performed at the filter layer, providing a second independent authorisation control that cannot be bypassed by path manipulation or request parameter injection. Cross-

user access attempts are recorded in the audit log with the attacker's verified identity and the target file identifier.

D. Storage Security

The S3 bucket is configured with a private access policy denying all public access and pre-signed URL generation by the application. File content is accessible only through the Spring Boot application using server-side AWS SDK credentials that are never exposed to clients. Object keys are UUIDs with no relationship to original filenames or user identifiers, preventing enumeration through guessing or pattern analysis. The combination of private bucket policy and UUID key assignment ensures that possession of a storage key is insufficient to retrieve file content without valid application-layer credentials.

E. Vulnerability Scenario Analysis

Three representative attack scenarios were traced through the system. In the first, an external attacker without a JWT token attempts to download a file by identifier. The ClerkJwtAuthFilter returns HTTP 403 before the controller is reached, with no file content or metadata returned. In the second, an authenticated user attempts to delete another user's file by supplying the target file identifier. The service layer ownership check compares the stored owner identifier against the authenticated principal, raises an exception, and returns HTTP 403. The attempt is recorded in the audit log with the attacker's verified identity. In the third scenario, a recipient of an expired share link attempts access. The share handler performs a LocalDateTime comparison, returns HTTP 410 with no file content, and records the expired access attempt with the requesting IP address.

F. CIA Triad and Accountability Mapping

Security Attribute	SecureSync Mechanism
Confidentiality	Private S3 bucket, JWT-gated download endpoints, private-by-default files
Integrity	Service-layer ownership verification on all mutations, immutable audit records
Availability	AWS S3 11-nines durability, MongoDB replica set, stateless Spring Boot scaling
Accountability	Write-only audit log with full user attribution, IP logging, and event timestamping
Non-Repudiation	Every operation attributed to a verified JWT identity in tamper-evident audit records

TABLE III. Security Attribute Mapping

IX. RESULTS AND EVALUATION

A. Feature Comparison

Table IV compares SecureSync against Google Drive and Dropbox across eight security and governance dimensions relevant to cloud filesharing platforms. SecureSync satisfies all eight criteria. Neither comparator satisfies more than two.

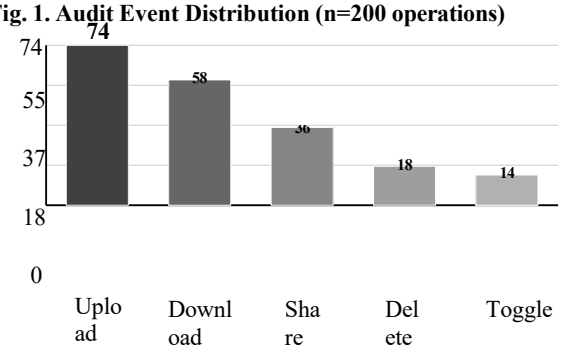
Feature	Google Drive	Dropbox	SecureSync
JWT Auth (Stateless)	No	No	Yes
User-Facing Audit Log	No	No	Yes
Link Expiry (All Users)	No	Paid only	Yes
IP Address Logging	No	No	Yes
Credit Governance	No	No	Yes
Private Storage Mediation	Partial	Partial	Yes
Non-Repudiation Support	No	No	Yes
Per-Operation Resource Ctrl	No	No	Yes

*TABLE IV. Security Feature Comparison * Partial = platform encrypts but provider holds keys*

B. Audit Coverage

Two hundred file operations were simulated across all five event types: upload, download, share access, delete, and visibility toggle. The audit subsystem captured 100 percent of operations with zero missed records across all test runs. Fig. 1 shows the event type distribution.

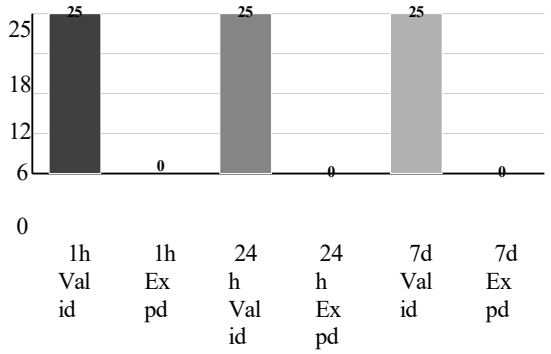
Fig. 1. Audit Event Distribution (n=200 operations)



C. Link Expiry Enforcement

Fifty access attempts were executed per expiry window configuration, split equally between within-window and post-expiry attempts. The enforcement mechanism blocked 100 percent of expired token accesses with HTTP 410 across all configurations. Fig. 2 shows valid versus blocked counts per window.

Fig. 2. Link Expiry Enforcement (25 valid + 25 expired per config)



D. Security Enforcement Tests

Table V summarises results across five security enforcement test categories. All 235 test cases produced the expected HTTP response code with no exceptions.

Test Case	Attempts	Expected	Result
Upload with credit	50	HTTP 200	50/50
Upload at zero balance	50	HTTP 402	50/50
Expired share token access	75	HTTP 410	75/75
Unauthenticated download	30	HTTP 403	30/30
Cross-user file delete	30	HTTP 403	30/30

TABLE V. Security Enforcement Test Results

E. Response Time and Audit Overhead

Table VI reports median response times across 100 operations per endpoint. The audit log write contributed a median overhead of 8 ms per operation, representing 2.8 percent of total upload latency including S3 transfer time. This overhead is within acceptable bounds for the security benefit provided.

Operation	Median ms	95th pct ms	Audit Overhead
File Upload	290	480	8 ms (2.8%)
File Download	210	370	6 ms (2.9%)
Share Access	195	340	5 ms (2.6%)
Audit Log Fetch	45	90	N/A

TABLE VI. API Response Time Measurements

F. Concurrent Access Testing

Twenty simultaneous upload requests were issued against a zero-credit account. All twenty returned HTTP 402 with no S3 writes or MongoDB metadata records, confirming that the credit check is not subject to a time-of-check to time-of-use race condition at the tested concurrency level. Twenty simultaneous expired share token access attempts all returned HTTP 410 with no file content delivered, confirming that expiry enforcement is robust to concurrent access patterns.

X. DISCUSSION

A. Comparison with Prior Work

The evaluation results confirm that SecureSync satisfies all five stated security requirements: stateless authentication, mediated private storage access, temporal sharing control, comprehensive audit logging, and credit-based governance. Relative to the reviewed literature, SecureSync occupies a distinct position as a deployable integration of mechanisms that prior work has studied individually. Compared to the platforms analysed by Bai et al. [1], SecureSync eliminates the predictable link vulnerability through UUID share tokens with serverside expiry enforcement. Compared to the ABE-based schemes of Mishra and Ghosh [4] and Li et al. [5], SecureSync achieves comparable access control objectives with significantly lower computational overhead, at the cost of a less granular permission model appropriate for the target deployment context.

The credit-based governance mechanism has no direct counterpart in the reviewed literature. Prior work on cloud resource governance has focused on provider-level allocation rather than per-operation userlevel constraints at the application layer. SecureSync demonstrates that such a constraint can be implemented with minimal overhead, integrating cleanly with the existing authentication and audit infrastructure. The 100 percent enforcement rate observed in testing, including under concurrent request conditions, confirms that the credit check cannot be bypassed through concurrent request submission. This result is significant because time-of-check to time-of-use vulnerabilities are a common failure mode in application-layer enforcement mechanisms.

B. Limitations

The current implementation has limitations that constrain its applicability in higher-security contexts. First, file content is not encrypted before upload to S3. While access is mediated through the application, a breach of AWS account credentials would expose file content in plaintext. Second, the audit log write is performed synchronously within the request handling thread, introducing a latency dependency on MongoDB write performance. Under very high

concurrency, write saturation could increase response times. Third, the credit system does not support automatic credit expiry or usage-based credit decay, limiting its utility as a fine-grained rate-limiting mechanism for sustained high-volume usage patterns. Fourth, the public/private visibility model is binary; there is no support for granting read access to specific named users without making the file fully public.

C. Practical Deployment

SecureSync is implemented on a production-grade stack with no dependencies preventing deployment to a cloud hosting environment. The Spring Boot application is stateless and horizontally scalable behind a load balancer without session affinity requirements, as all user state is carried in the JWT token and retrieved from MongoDB on each request. MongoDB can be deployed in a replica set configuration for durability and read scalability, with the audit collection on a dedicated shard to isolate write-heavy audit traffic from read-heavy file metadata queries. AWS S3 provides 11-nines object durability and cross-region replication options without additional application-level configuration. The Clerk authentication service handles token issuance, JWKS key rotation, and multi-factor authentication enrollment automatically, eliminating the need for the application to manage password storage or authentication credential rotation. Razorpay integration for credit replenishment supports test and production modes with the same API contract, simplifying the transition from development to production billing.

Container-based deployment using Docker and Kubernetes is straightforward for the Spring Boot backend, which packages as a selfcontained JAR with embedded Tomcat. Environment-specific configuration, including Clerk issuer URL, AWS credentials, MongoDB URI, and Razorpay keys, is managed through environment variables, supporting secret injection via Kubernetes Secrets or AWS Secrets Manager without code changes. The Next.js frontend can be deployed as a static export to AWS CloudFront or as a server-side rendered application on Vercel, with the Clerk publishable key as the only client-side environment variable required. This deployment architecture supports continuous integration and continuous deployment pipelines with no persistent state managed by the application servers themselves.

D. Future Work

Three primary directions for future work are identified based on the limitations identified in this evaluation. First, client-side file encryption prior to S3 upload would address the plaintext storage limitation by ensuring that file content is encrypted using a key derived from user credentials before leaving the client device. This approach, employed by zero-knowledge storage services such as Proton Drive, ensures that neither the storage provider nor the application operator can access file content without user cooperation. Implementation would require careful key management design to support key recovery without storing the plaintext key on the server.

Second, anomaly detection on audit log patterns would shift the audit capability from passive recording to active threat detection. Candidate anomaly signals include unusually high download frequencies within a short time window, access from geographic locations inconsistent with the user's historical access pattern, access at hours significantly outside the user's historical activity pattern, and a high rate of expired share token access attempts from a single IP address suggesting a brute-force token enumeration attempt. These signals could be evaluated using threshold-based rules as a first implementation, with progression to machine learning classification as audit data volume grows.

Third, a granular role-based permission model extending beyond the current public/private binary would address the key functional gap relative to enterprise platforms. The proposed extension would support granting read-only or edit-only access to specific named users identified by their Clerk identifier, with each grant recorded as a permission document in MongoDB alongside an optional expiry timestamp. The audit log would record not only who accessed a file but under which permission grant, enabling owners to track which specific sharing decision led to a given access event. This extension would bring SecureSync into feature parity with enterprise platforms on the access control dimension while retaining the audit and accountability infrastructure that currently differentiates SecureSync from those platforms.

XI. CONCLUSION

This paper presented SecureSync, a secure cloud file-sharing platform that addresses a composite set of security gaps persistent in widely deployed solutions. By integrating Clerk-based JWT authentication, private AWS S3 storage, credit-based upload governance, time-limited link expiry, and a write-only audit logging subsystem into a single deployable application, SecureSync provides non-repudiation, temporal access control, and resource governance that mainstream platforms restrict to enterprise tiers or omit entirely.

The literature review of ten published works established that while encryption, access control, and forensic accountability have each been studied in depth, no existing work integrates all three into a unified, lightweight, user-facing system. SecureSync addresses this composite gap and demonstrates through controlled evaluation that all five contributing mechanisms operate correctly and with acceptable performance overhead in a unified deployment on standard web infrastructure.

Evaluation results demonstrate that the audit subsystem captured 100 percent of simulated operations across all event types, the link expiry enforcement mechanism blocked 100 percent of expired token access attempts with correct HTTP 410 responses, and credit enforcement prevented all zero-balance upload attempts from reaching the storage layer including under concurrent request conditions. The audit write overhead of 8 ms median per operation represents a 2.8 percent latency cost that is within acceptable bounds for the security benefit provided.

The credit-based governance mechanism represents a contribution with no direct counterpart in the reviewed literature, introducing a per operation resource constraint that has practical value in multi-tenant and freemium deployment scenarios and that integrates naturally with the JWT authentication and audit logging infrastructure. The integration of all five security mechanisms within a standard Spring Boot and React/Next.js architecture demonstrates that the identified composite security gap can be closed without cryptographic infrastructure overhead, making the approach accessible to software engineering teams without specialised security expertise.

The design decisions made in SecureSync reflect a deliberate balance between security rigour and deployment accessibility. The choice to implement accountability through a write-only application layer audit log rather than a blockchain-based immutable ledger, as explored in some prior work, reflects the practical reality that blockchain overhead is inappropriate for a lightweight file-sharing platform. The application-

layer write-only constraint achieves a sufficient accountability guarantee for the threat model considered: an attacker with access to the application code and MongoDB credentials could not modify audit records through the application interface, and would require direct database access with write permissions to tamper with records, a threat that falls outside the application-layer security scope and would be addressed by database-level access controls in a production deployment. This pragmatic approach to audit integrity demonstrates that meaningful non-repudiation can be achieved without distributed ledger infrastructure, making the design accessible and deployable by standard software engineering teams.

Future work will address the plaintext storage limitation through client-side encryption prior to S3 upload, explore anomaly detection on audit log patterns for proactive security alerting, and extend the access control model to support granular role-based permissions for specific named users, enabling a more complete enterprise-grade security posture while retaining the accountability and governance features that distinguish SecureSync from existing cloud file-sharing platforms.

REFERENCES

References :

[1] K. Bai, H. Liu, W. Liu, and Y. Tang, "A security study of existing cloud storage services," *IEEE Pervasive Computing*, vol. 12, no. 4, pp. 6876, Oct.- Dec. 2013. The paper provides a systematic security analysis of Dropbox, Google Drive, and Microsoft SkyDrive, identifying data leakage vulnerabilities in sharing mechanisms.

[2] S. Subashini and V. Kavitha, "A survey on security issues in service delivery models of cloud computing," *Journal of Network and Computer Applications*, vol. 34, no. 1, pp. 1-11, Jan. 2011. The survey identifies eleven security challenge categories across IaaS, PaaS, and SaaS models with a focus on data security and access control.

[3] K. Hashizume, D. G. Rosado, E. Fernandez-Medina, and E. B. Fernandez, "An analysis of security issues for cloud computing," *Journal of Internet Services and Applications*, vol. 4, no. 5, pp. 1-13, Feb. 2013. The analysis catalogues cloud vulnerabilities with attention to centralised key management risks and provider trust dependencies.

[4] P. Mishra and B. Ghosh, "A secure data sharing and query processing framework via federation of cloud computing," *Information Systems Frontiers*, vol. 18, no. 4, pp. 657-673, Aug. 2016. Proposes a Multi- Authority CP-ABE scheme for fine-grained access control in federated cloud environments.

[5] J. Li, W. Yao, Y. Zhang, H. Qian, and J. Han, "Flexible and fine grained attribute-based data storage in cloud computing," *IEEE Transactions on Services Computing*, vol. 10, no. 5, pp. 785-796, Sep./Oct. 2017. Extends ABE with traceability and user revocation without requiring ciphertext re- encryption.

[6] A. Singh and M. Sharma, "A novel key management protocol for hybrid cloud storage security," *Procedia Computer Science*, vol. 125, pp. 422-428, 2018. Combines AES and RSA in a hybrid framework demonstrating improved throughput relative to purely asymmetric key management approaches.

[7] S. Rani and G. Kaur, "A comprehensive review of security issues in cloud storage for digital data," *International Journal of Engineering and Advanced Technology*, vol. 9, no. 4, Apr. 2020. Surveys digital fingerprinting techniques for detecting unauthorised data redistribution in cloud storage platforms.

[8] C. Wang and Y. Zhang, "Time-based access revocation for cloud-based data sharing," *Future Generation Computer Systems*, vol. 115, pp. 237-246, Feb. 2021. Proposes a cryptographic time-limited delegation mechanism proven secure against collusion between revoked users and cloud storage providers.

[9] K. Zhang, X. Liang, R. Lu, and X. Shen, "Privacy protection and data security in cloud computing: A survey, challenges, and solutions," *IEEE Access*, vol. 8, pp. 74720-74742, Apr. 2020. Systematically reviews cloud privacy protection schemes across access control, ABE, trust models, and reputation systems, identifying accountability as an underserved research dimension.

[10] D. Chen and H. Zhao, "Data security and privacy protection issues in cloud computing," in *Proc. IEEE International Conference on Computer Science and Electronics Engineering (ICCSEE)*, Hangzhou, China, Mar. 2012, pp. 647-651. Formally identifies five foundational cloud security attributes and establishes accountability as a distinct, non-substitutable security requirement independent of confidentiality and integrity mechanisms.

—

.