Centralized Ledger System for Document and Process Certification

NUNO ANTÓNIO OLIVEIRA BARTOLOMEU

(Licenciado)

Trabalho de Projeto para obtenção do Grau de Mestre em

Engenharia Informática e de Computadores

Orientadores:

Doutor Nuno Miguel da Costa de Sousa Leite

Eng. João Miguel de Carvalho da Conceição Pereira

Júri:

Presidente: Doutor Nuno Miguel Soares Datia

Vogais:

Doutor José Manuel de Campos Lages Garcia Simão

Doutor Nuno Miguel da Costa de Sousa Leite

Outubro 2025

Centralized Ledger System for Document and Process Certification

NUNO ANTÓNIO OLIVEIRA BARTOLOMEU
(Licenciado)

Trabalho de Projeto para obtenção do Grau de Mestre em
Engenharia Informática e de Computadores

Orientadores:
Doutor Nuno Miguel da Costa de Sousa Leite, ISEL-DEETC
Eng. João Miguel de Carvalho da Conceição Pereira, Cofidis

Júri:

Presidente: Doutor Nuno Miguel Soares Datia, ISEL-DEETC
Vogais:
Doutor José Manuel de Campos Lages Garcia Simão, ISEL-DEETC
Doutor Nuno Miguel da Costa de Sousa Leite, ISEL-DEETC

Outubro 2025

ACKNOWLEDGEMENTS

The development of this project was only possible thanks to the support I received from various people, to whom I am deeply grateful.

Starting with my supervisors, Dr. Nuno Leite and Eng. João Pereira, I thank them both for their guidance and invaluable help, which allowed me to carry out this project with care and appreciation.

I would also like to thank ISEL, as well as all the teachers and classmates who supported me throughout the years, with special thanks to Eng. Pedro Carvalho and Eng. Ricardo Rocha, with whom I worked on all the group projects during the master's program. They are great classmates and friends.

On a more personal note, I want to thank my parents, Ana Teresa Bartolomeu and Nuno Manuel Bartolomeu, and the rest of my family, that are to numerous to mention individually, for their amazing support, motivation and inspiration over the past 23 years.

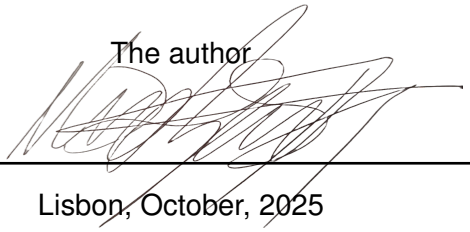
I also want to thank all my friends, especially Eng. Miguel Ferreira, for being there for me through both the good and bad times. I may have lost contact with many people along the way, but I remain thankful for the moments we shared.

Last but not least, I want to thank my girlfriend, Daria Chenkina. With her by my side, I've grown closer to the person I want to become. My grades improved, I fell freer than ever before, and her presence continues to be the light in my days. I owe who I am today to her.

Statement of integrity

I declare that this dissertation is the result of my personal and independent research. Its content is original, and all sources listed in the bibliographic references were consulted and are duly mentioned in the text. I further declare that all scientific and technical references relevant to the development of the work are duly cited and included in the bibliographic references.

The author



Lisbon, October, 2025

Centralized Ledger System for Document and Process Certification

This document was created using the (pdf)LaTeX processor, based in the “iselthesis” template [52], developed at the DEETC of ISEL-IPL.

ABSTRACT

Organizations increasingly require secure document management with integrity guarantees beyond traditional audit logs, particularly in regulated industries where external accountability is critical. While blockchain technologies provide strong tamper-detection, they present significant enterprise adoption challenges including cost volatility, low throughput, and unpredictable operational expenses.

This thesis proposes a Centralized Ledger System (CLS) that provides blockchain-inspired integrity verification through self-hosted architecture without external dependencies. The system implements a three-phase entry lifecycle, signature collection and verification, supporting multi-party transactions with asynchronous workflows. A multi-ledger architecture enables organizational segregation of business domains while maintaining referential integrity.

Key contributions include automated receipt generation for independent verification, selective payload erasure preserving cryptographic validation, entry linking for audit simplification, and integration of security services with two-factor authentication and key management. The modular design enables flexible deployment while maintaining cryptographic guarantees equivalent to blockchain systems.

The solution addresses the gap between traditional audit systems and distributed ledgers by providing cost-predictable, vendor-independent functionality that integrates into existing workflows without specialized blockchain expertise.

Keywords: Blockchain-Inspired Architecture, Cryptographic Receipts, Centralized Ledger, Data Integrity, Digital Signatures, Multi-Ledger System.

RESUMO

As organizações necessitam, cada vez mais, de uma gestão de documentos segura com garantias de integridade que vão além dos registos de auditoria tradicionais, particularmente em indústrias regulamentadas onde a responsabilização externa é crítica. Embora as tecnologias blockchain proporcionem forte deteção de adulteração, estas apresentam desafios significativos para adoção empresarial incluindo volatilidade de custos, baixo débito de dados e despesas operacionais imprevisíveis.

Esta tese propõe um Sistema de Ledger Centralizado (CLS) que fornece verificação de integridade inspirada em blockchain através de uma arquitetura auto-hospedada sem dependências externas. O sistema implementa um ciclo de vida de entrada em três fases, recolha e verificação de assinaturas, suportando transações multi-entidade com fluxos de trabalho assíncronos. Uma arquitetura multi-ledger permite segregação organizacional de domínios de negócio mantendo integridade referencial.

As principais contribuições incluem geração automática de recibos digitais para verificação independente, eliminação seletiva de conteúdo preservando validação criptográfica, ligação de entradas para simplificação de auditorias e integração de serviços de segurança com autenticação dois fatores e gestão de chaves. O design modular permite implementação flexível mantendo garantias criptográficas equivalentes aos sistemas blockchain.

A solução aborda a lacuna entre sistemas de auditoria tradicionais e ledgers distribuídos fornecendo custos previsíveis e independente de fornecedores que se integra em fluxos de trabalho existentes sem conhecimento especializado em blockchain.

Palavras-chave: Arquitetura Inspirada em Blockchain, Assinaturas Digitais, Integridade de Dados, Ledger Centralizado, Recibos Criptográficos, Sistema Multi-Ledger.

CONTENTS

List of Figures

List of Tables

ACRONYMS

2FA	Two Factor Authentication xvii, 2, 4, 41, 42, 46, 47, 49
AOP	Aspect-Oriented Programming 25
API	Application Programming Interface 23, 24, 25, 42, 46
CLS	Centralized Ledger System 1, 4
CSS	Cascading Style Sheets 25
DAO	Data Access Object 24, 26
DSA	Digital Signature Algorithm 9
DTO	Data Transfer Object 24, 26, 44, 45
ECDSA	Elliptic Curve Digital Signature Algorithm 9
EDMS	Electronic Document Management Systems 1
HIPAA	Health Insurance Portability and Accountability Act 43
HTTP	Hypertext Transfer Protocol 24, 46
IPFS	InterPlanetary File System 26, 56
JDBC	Java Database Connectivity 25, 26
JSON	JavaScript Object Notation 25, 55
JWT	JSON Web Token 25
KP	Key Pair 8, 9
MT	Merkle Tree 6, 7, 8, 15
PEM	Privacy Enhanced Mail 46, 47, 56
PK	Public Key xvii, 8, 9

PKC Public Key Cryptography 6
PoS Proof of Stake 18
PoW Proof of Work 13, 18

QLDB Quantum Ledger Database 19

RSA Rivest–Shamir–Adleman 9

SHA Secure Hash Algorithm 6
SIEM Security Information and Event Management 36
SK Secret Key xvii, 8, 9
SOA Service-Oriented Architecture 23, 24
SQL Structured Query Language 19

TxID Transaction ID 11

UTXO Unspent Transaction Output 12, 13

CHAPTER 1

INTRODUCTION

The first chapter provides an overview of the problem's context and significance, outlining the motivation for this research, and presenting the primary objectives of the thesis.

In Section 1.1, the context and motivation for the study are discussed, highlighting the increasing relevance of digital management systems and their inherent challenges. Section 1.2 outlines the main objectives of the thesis, focusing on the design and development of a Centralized Ledger System (CLS) that integrates features inspired by blockchain technology. Lastly, Section 1.3 provides an overview of the document's structure, guiding the reader through the rest of the chapters and their contents.

1.1 Context and Motivation

In recent years, digital management systems have experienced exponential growth as organizations increasingly require secure and organized approaches to handle data and documentation. The global document management software market exemplifies this trend, expected to grow from $7.52 billion in 2024 to $14.82 billion in 2029 [25]. This expansion is driven by compelling returns on investment: organizations adopting Electronic Document Management Systems (EDMS) report reductions of 52% in document related costs and productivity gains of up to 40% [73].

Most organizations today rely on widely adopted document management platforms such as Microsoft SharePoint [43], Google Workspace [26], and Dropbox Business [14] for daily operations. These systems typically maintain audit logs that record user actions, timestamps, and file modifications, with enterprise plans averaging $10–20 per user per month. These solutions have standard logging mechanisms that cannot provide undeniable proof regarding the integrity, non-repudiation, and tamper-detection features required in certain critical situations.

The need for enhanced security, accountability, integrity, auditability, and confidentiality in digital data is particularly present in sectors such as finance, healthcare, legal services, and government contracting. Cofidis [12], a consumer credit company operating in a heavily regulated sector, proposed this research to address their constant need for handling contracts with clients

and partners. The company depends on reliable mechanisms for recording and certifying actions that guarantee legal validity and allow unequivocal association between each user and their performed actions. Among the potential use cases are the digital signature of contracts and the recording of modification actions performed under temporary privilege assignment via 2FA, both demanding high levels of security, traceability, and integrity.

In this context, it becomes essential to have a solution that ensures the aforementioned properties for produced records. The solution reached was the creation of a *digital ledger*, a system capable of maintaining data integrity through precise and immutable record-keeping. This technology can be enhanced with *digital signatures* to provide the critical attribute of non-repudiation. Such a ledger would function as a single source of truth regarding actions performed within the organization's systems.

Several existing ledger solutions could potentially address these requirements, but most face integration challenges as they were not designed to service this exact problem.

The most common type of digital ledgers are *blockchains*, which are public ledgers typically accessible by anyone, making data within easily visible. While this transparency is beneficial for integrity verification, it creates data protection concerns, being unsuitable for any personal data usage. Blockchains are also deeply intertwined with cryptocurrencies as their usual primary focus, making their prices unpredictable, as illustrated in Figure 1.1. Transaction fees can increase by hundreds or thousands of percent in short periods, making long-term budgeting impossible for organizations with operational horizons of 10–20 years.

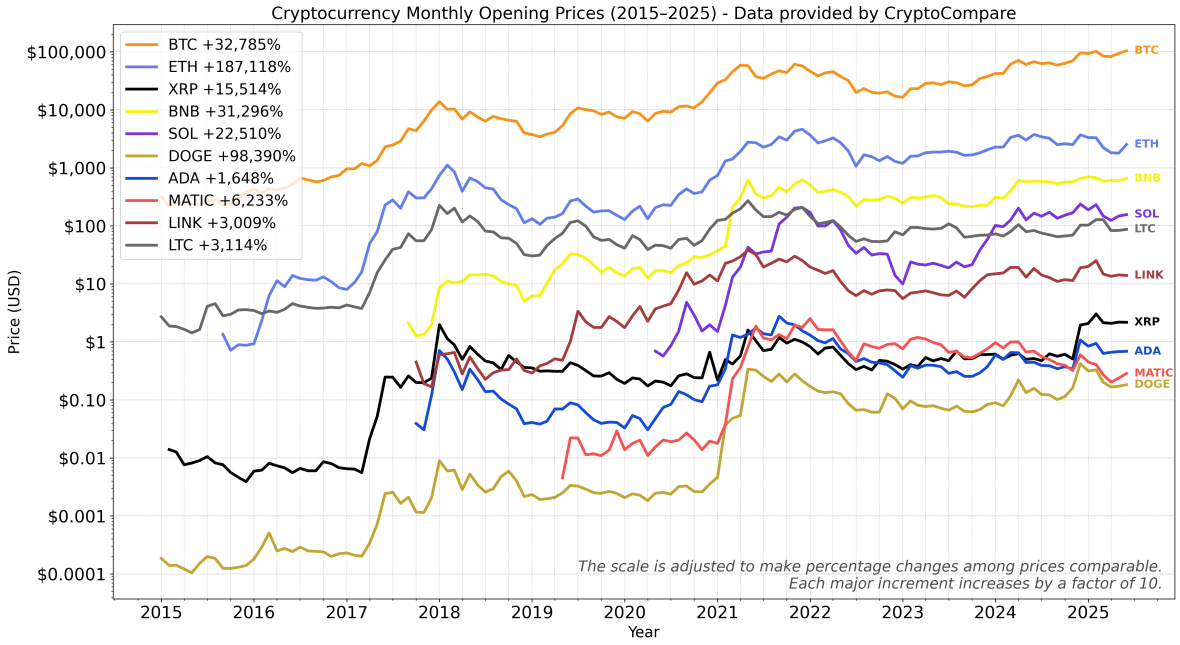


Figure 1.1: *Cryptocurrencies prices evolution 2015–2025*

To work around blockchains, it is possible to use timestamping services that prove a certain file existed in a particular state at a specific point in time. However, this approach alone does not address non-repudiation and would require additional systems to manage that aspect, as well as separate systems to manage the files themselves.

Cloud-platform solutions such as proprietary ledger databases provide comprehensive capabilities but create vendor lock-in concerns and deployment inflexibility. Other centralized ledger implementations, particularly key-value stores, offer better accessibility but lack capabilities for enterprise use: they provide no keyword-based indexing within stored information, cannot map relationships between records for complex audit trails, and lack client-side proof-of-inclusion mechanisms to independently verify data integrity, making clients vulnerable to undetected internal system modifications. Furthermore, no existing solution provides a strong focus on organizational capabilities or the convenient features useful for facilitating integration within companies.

Unlike distributed approaches, a centralized model allows greater administrative control, simplified access management, and efficiency in integration with existing corporate systems, without compromising security and auditability requirements. The proposed research aims to analyze, design, and validate a centralized ledger architecture capable of ensuring traceability and certification of actions within common organizational infrastructures, contributing to increased trust, transparency, and legal compliance in the digital processes.

1.2 Objectives

This thesis aims to design and develop a centralized ledger system that provides secure transaction tracking and data integrity verification through a centralized architecture. By adopting this approach, the system avoids the complexity and costs associated with distributed consensus mechanisms while maintaining the core benefits of ledger-based integrity verification.

The proposed system prioritizes the following design considerations: cost-effective deployment through predictable pricing, organizational control without vendor lock-in, flexible management across multiple ledgers, and adherence to software engineering best practices. The system emphasizes operational simplicity, data security, and auditability while remaining accessible to organizations of varying sizes and technical capabilities.

The primary objectives and functional requirements of the centralized ledger system are:

- Store entry data in a ledger structure with minimal restrictions on users, including flexible payload sizes and content types.
- Guarantee non-repudiation for all registered entries through cryptographic signatures.
- Support data encryption for confidential information requiring secure storage within the ledger.
- Enable selective payload erasure while preserving validation capabilities.
- Facilitate entry linking to connect related operations, simplifying audit processes and information retrieval.
- Provide automated receipt generation with independent verification capabilities for all users.

- Support concurrent operation of multiple ledgers within a single system instance.

To support the core ledger functionality, the system incorporates supporting components organized into different areas. The Security layer includes a Authentication Service for user management, a 2FA Service for enhanced security, and a Key Management Service for cryptographic control. The Workflow layer provides a File Management Service for file upload, download, and deletion operations. These components integrate with a Web Server and Application Server architecture, providing a complete practical implementation.

This thesis contributes to the field by demonstrating a practical approach to secure centralized ledger systems that balances the integrity benefits of ledger-based verification with the operational control and cost-effectiveness required by modern organizations. The modular design and standalone implementation approach make the solution adaptable to various contexts and existing system architectures, providing a scalable foundation for sectors requiring enhanced data integrity, accountability, and auditability.

1.3 Document Organization

The remaining document is organized in the following way:

- Chapter 2: State of the Art - Covers the theoretical foundations underlying ledger systems, including cryptographic concepts, ledger fundamentals, and architectural alternatives. It also examines the current landscape of blockchain applications, cloud-based solutions, timestamp services, and document signature services, whilst identifying the gaps in development for the approaches.
- Chapter 3: Proposed Solution - Details the complete design and implementation of the CLS, beginning with design philosophy and architectural decisions. It covers the technology stack selected, the server-side and client-side implementation, data storage approaches, and containerized deployment strategies.
- Chapter 4: Results and Discussion - Presents the experimental evaluation of the system through comprehensive testing methodologies. It includes cryptographic provider validation, proof-of-concept demonstrations, security analysis with attack vector identification and mitigation strategies, and stress testing with performance analysis under various load conditions.
- Chapter 5: Conclusions - Provides an overview of the research outcomes, demonstrating the feasibility and security of centralized ledger operations, and outlines directions for future work including production testing, algorithm standardization, and system optimization.

CHAPTER 2

STATE OF THE ART

This chapter aims to provide a better understanding of the current landscape on the context behind this thesis. It will start by explaining the theoretical foundations of ledger systems and cryptographic concepts in Section 2.1. In Section 2.2 will be explained the current landscape of ledger systems and their applications. Lastly, in Section 2.3 will be made an overview of the shortcomings of the discussed solutions.

2.1 Theoretical Foundations

To understand the full scope of the thesis, it is necessary to understand the fundamental concepts that were used to develop it. Digital ledgers were only made possible by utilizing a variety of technologies that already existed, such as cryptography, digital signatures, peer-to-peer connections, and more.

2.1.1 Cryptographic Concepts

Ledger systems get their security guarantees from fundamental cryptographic primitives that work in combination to provide comprehensive protection for stored data and transactions. The critical security properties that make ledgers suitable for document certification or financial applications include:

- Integrity: Ensures that data has not been altered or corrupted after storage.
- Authenticity: Verifies the identity of the party who created or submitted a transaction.
- Non-repudiation: Prevents parties from denying their participation in a transaction after the fact.
- Tamper-evidence: Makes any attempt to modify historical records apparent to all participants.

These security properties are achieved through the mathematical foundations of cryptography, particularly Public Key Cryptography (PKC) and cryptographic hash functions. The following subsections examine each primitive in detail, explaining how their mathematical properties translate into practical security guarantees for ledger systems.

2.1.1.1 Hash Functions

Hash functions serve as the fundamental building blocks for data integrity in ledger systems by generating unique "digital fingerprints" for any input data. These mathematical functions transform arbitrary length input into fixed-length output, creating a deterministic mapping that enables efficient verification of data authenticity [37]. As described by Kuchta, "A hash is a sequence of letters and numbers of set length that may be termed the 'digital fingerprint' of a computer file", providing a reliable method for identifying and verifying data integrity in the same way fingerprints identify humans.

The security properties that make hash functions suitable for ledger applications include the following [24]:

- Deterministic: Identical inputs always produce identical outputs, ensuring consistent verification across different systems and time periods
- Pre-image resistance: Given a hash output, it is computationally infeasible to reverse-engineer the original input, ensuring data cannot be easily reconstructed from its hash
- Collision resistance: Finding two different inputs that produce the same hash output is statistically improbable, preventing forgery attacks
- Avalanche effect: A small change in the input produces a significantly different output, making any tampering detectable

The avalanche effect property is particularly crucial for tamper detection in ledger systems. For instance, if a document has the hash `0xa665...`, changing even a single character in the document would result in a completely different hash value, such as `0xb94d...`, immediately revealing the modification. This sensitivity to input changes ensures that any alterations to ledger entries are detectable without the need of checking each character in the file one by one.

The most widely adopted hash function in cryptographic applications is the Secure Hash Algorithm (SHA), particularly SHA-256, which produces a 256-bit hash regardless of input size. This algorithm provides 2^{256} possible outputs, a very large number that makes the probability of hash collisions negligible. For comparison, the estimated number of atoms in the observable universe is approximately 2^{266}, only about one thousand times larger than the number of possible SHA-256 outputs.

2.1.1.2 Merkle Trees

Building upon the hash function principles discussed, Merkle Tree (MT) represents a application of cryptographic hashing for efficient data integrity verification. The MT is a binary tree data

structure where each leaf node contains the hash of a data block, and each non-leaf node contains the hash of its two children [60]. This hierarchical structure culminates in a single root hash that uniquely represents the entire dataset, as seen in Figure 2.1.

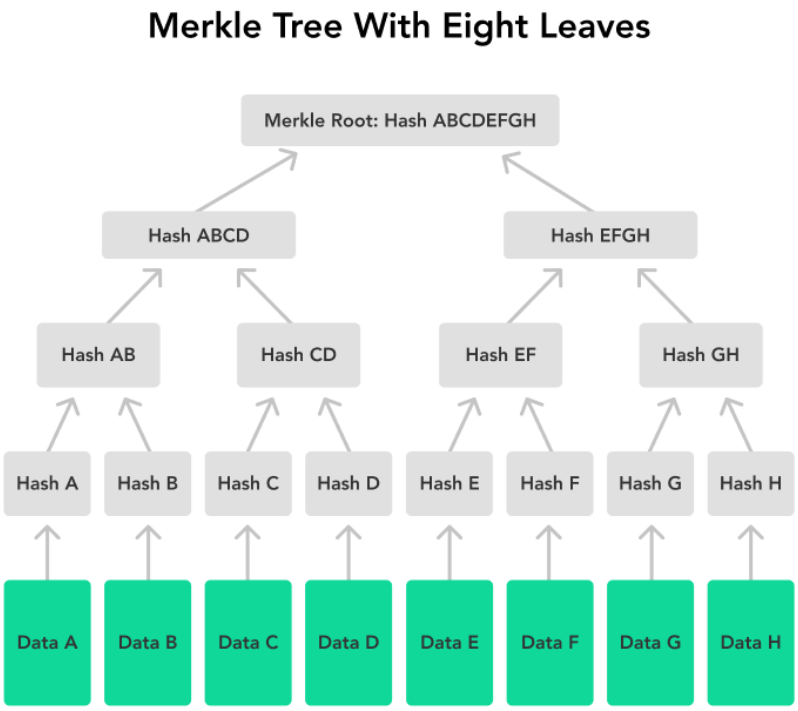


Figure 2.1: *Diagram of a Merkle Tree. Adapted from [36]*

The construction process begins by hashing individual data elements, then systematically combining these hashes in pairs up the tree until a single root remains. This leverages the deterministic and avalanche effect properties of hash functions previously discussed, ensuring that any alteration to the data elements produces a different root hash, making tampering detectable.

A feature of MT is their support for proof of inclusion mechanisms. A *Merkle Proof*, also known as a *Merkle Inclusion Proof* or a *Merkle Path*, is a deterministic proof that a certain data element was present in the dataset used to generate the Merkle Root. Rather than requiring the entire dataset for verification, a proof of inclusion consists in the specific data element and the minimal set of sibling hashes needed to reconstruct the path to the root. This enables efficient verification with logarithmic complexity, making it practical to verify the inclusion of individual elements.

To illustrate using Figure 2.1, suppose we want to verify that *Data C* was included in the dataset that produced the Merkle Root *ABCDEFGH*. The Merkle Proof would consist of the hashes of *D*, *AB*, and *EFGH*, which is the set of hashes needed to reconstruct the path to the root. To actually make the verification is necessary to:

1. Compute the hash of *Data C* -> *Hash C*
2. Combine it with *Hash D* -> *Hash CD*

3. Combine it again with *Hash AB -> Hash ABCD*

4. Finally, combining it and *Hash EFGH -> Hash ABCDEFGH*

If the calculated hash matches the provided root hash than there is proof that *Data C* was included in the dataset that produced *Hash ABCDEFGH*

In ledger systems the MT root is included in the block header, enabling clients to verify transaction inclusion quickly. Figure 2.2 demonstrates the application of MT in such systems.

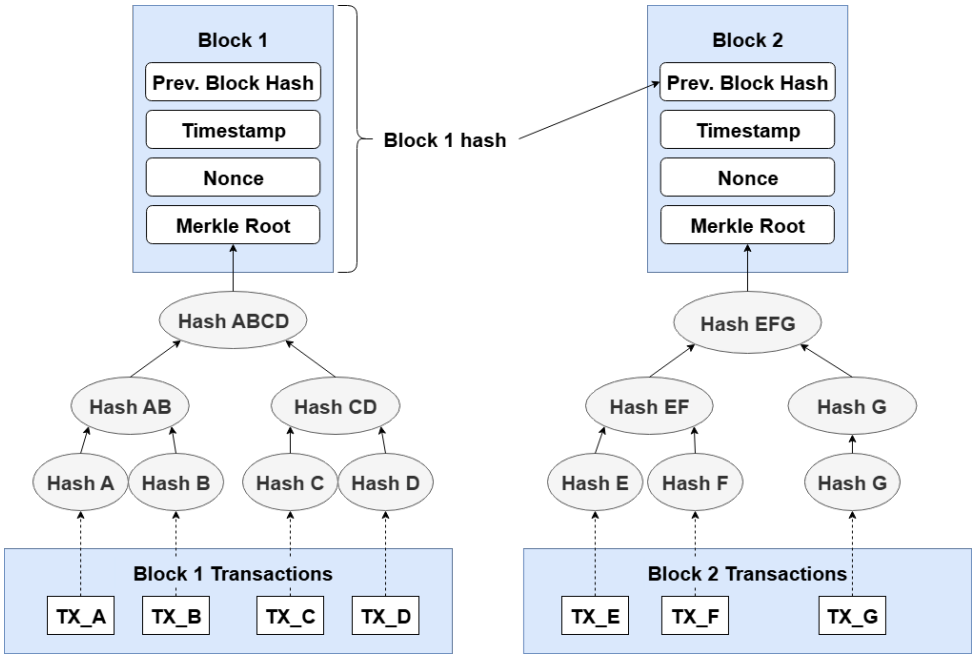


Figure 2.2: *Diagram of the Merkle Tree applied to a Blockchain.*

2.1.1.3 Digital Signatures

Digital signatures provide authentication and non-repudiation for digital data. These cryptographic techniques ensure that data originates from a verified source and cannot be disputed by the signer, making them essential for maintaining trust in distributed ledger systems.

In the physical world, handwritten signatures are used to authenticate the identity of the signer and confirm their approval of the associated content. Digital signatures serve the same purpose but using mathematical schemes that utilize asymmetric cryptography.

Digital signatures have two main components, the Key Pair (KP) containing the SK and the PK and the signature itself. All of these elements are strings of bytes that can be created with cryptography. The keys in the KP are generated by first creating a SK and then using an algorithm to calculate the PK, similar to the hashes, the algorithm used is a one-way function meaning the SK cannot be reverse-engineered from the PK.

The PK is openly accessible and shared, it can be seen as a common identification of an entity in the system. But the SK needs to remain confidential, as its compromise would allow anyone to use the algorithm to calculate the PK and assume the identity of the entity, meaning, they would be able to create signatures in the name of someone else.

Once the signature is created, it can be verified using the original data, the signature, and the PK associated with the SK that generated it. If the verification succeeds, it proves that the holder of the corresponding SK produced the signature and, therefore, cannot deny authorship.

Figure 2.3 illustrates the digital signature process, showing how SKs are used for signing and PKs for verification.

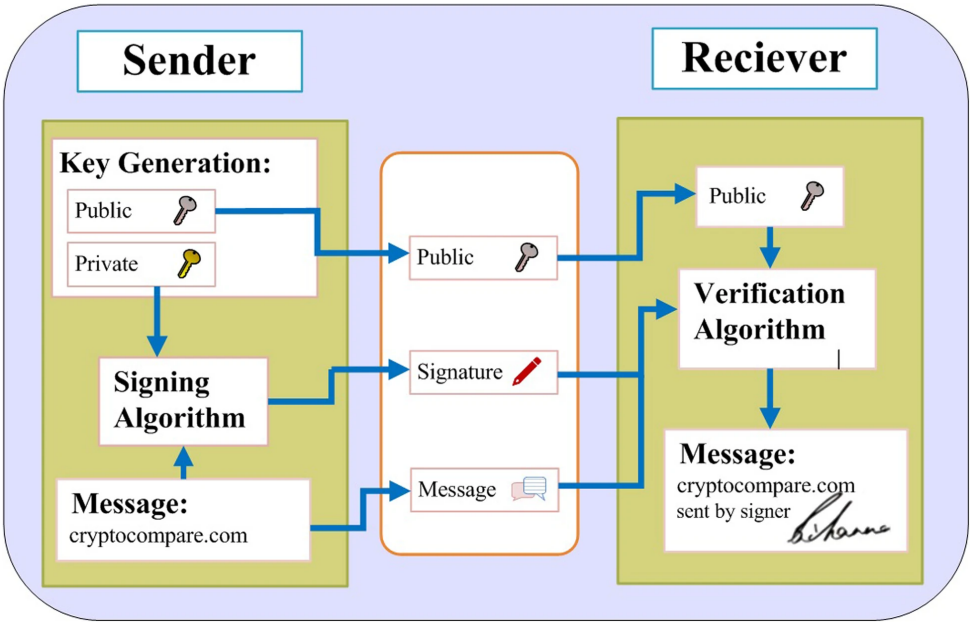


Figure 2.3: *Representation of the digital signature process, showing signature generation with SKs and verification with PKs. Adapted from [17]*

A useful analogy for understanding asymmetric cryptography is thinking of a magical lock and KP: one key locks the mechanism whilst the other unlocks it. Crucially, the same key cannot perform both actions, making both keys necessary for the system's operation.

The most common algorithms used to implement digital signatures are [48]:

- Rivest–Shamir–Adleman (RSA) – Based on the difficulty of factoring large prime numbers.
- Digital Signature Algorithm (DSA) – A U.S. federal standard that operates on discrete logarithms.
- Elliptic Curve Digital Signature Algorithm (ECDSA) – A more efficient variation of DSA that uses elliptic curve cryptography and is widely adopted in modern systems such as blockchains.

2.1.2 Ledger Fundamentals

A ledger is a chronologically ordered data structure that maintains a permanent record of transactions or events, characterized by its append-only nature and immutable historical sequence [47]. It enforces strict temporal ordering, where new entries can only be appended to the end of the sequence, preserving a complete audit trail of all recorded operations. This property

of preserving historical integrity through immutable record keeping makes ledgers suitable for applications requiring trust, transparency, and auditability [20].

While the term *ledger* is less commonly recognized by the general public, most are familiar with *blockchains*. A blockchain is a type of distributed ledger, most often associated with cryptocurrencies, that maintains a synchronized record of all transactions across all peers in its network.

The versatility of ledger systems extends far beyond cryptocurrencies, supporting domains where an immutable record of events is essential. A ledger can be adapted to specific purposes while preserving its core principles of chronological ordering and immutability.

2.1.2.1 Transaction Structure and Purpose

Records, often referred to as *transactions* in the blockchain context, are the fundamental units of information stored within a ledger. They represent discrete events or state changes that define the ledger's purpose and functionality. The structure of these transactions directly determines what the ledger is designed to track and manage. While cryptocurrency ledgers focus on value transfer, the transaction model can be adapted to various cases by modifying the payload and validation rules.

The structure of each transaction serves as the blueprint for the ledger's functionality, defining not only what information is recorded but also how the system validates and processes new entries. This adaptability allows ledgers to serve diverse purposes beyond financial transactions, such as supply chain tracking, identity management, or healthcare records.

For example, consider a specialized ledger for medical prescriptions designed to track patient prescriptions and medication dispensing. Table 2.1 shows how the structure of a record could be tailored to the requirements while maintaining the core properties of a ledger.

Table 2.1: Medical Prescription Ledger Transaction Structure

Field	Purpose	Example Value
Transaction ID	Identify the prescription event	0x7a3b9c2d...
Timestamp	Date and time of event	2024-03-15 14:30:00
Patient ID	Anonymized patient identifier	PAT_789123
Prescriber ID	Licensed physician identifier	DOC_456789
Medication Code	Standardized drug identifier	NDC_12345-678-90
Dosage	Prescribed amount and frequency	500mg, twice daily
Pharmacy ID	Dispensing pharmacy identifier	PHARM_321654
Action Type	Type of transaction	PRESCRIBE/DISPENSE
Digital Signature	Cryptographic authentication	0x9f8e7d6c...
Previous Hash	Link to previous transaction	0x1a2b3c4d...

This medical ledger example demonstrates how transaction structure adapts to domain specific requirements while maintaining essential ledger properties. The inclusion of patient anonymization, prescriber authentication, and medication tracking creates a comprehensive audit trail for

pharmaceutical management without compromising patient privacy.

Regardless of the specific application domain, certain fundamental components remain consistent across ledger implementations:

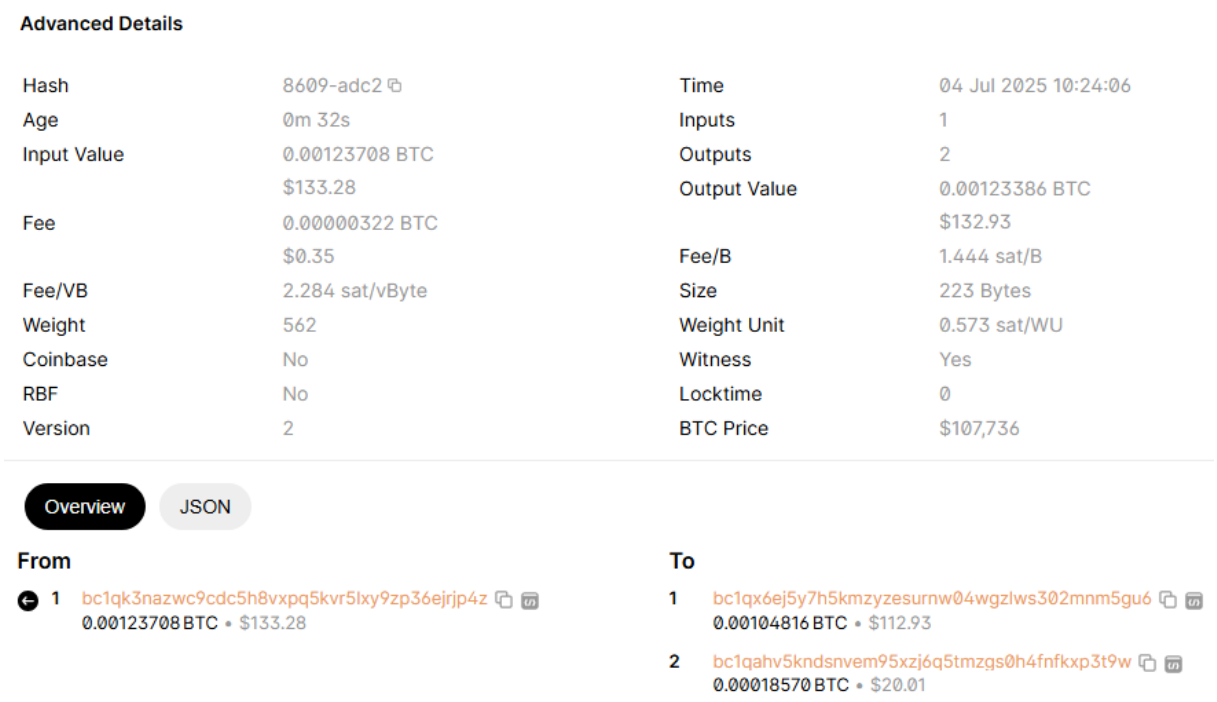
- Transaction ID (TxID): A unique identifier derived from the transaction's content using cryptographic hashing to ensure integrity and prevent duplication.

- Timestamp: The exact time and date the transaction was created, ensuring chronological ordering and enabling temporal queries.

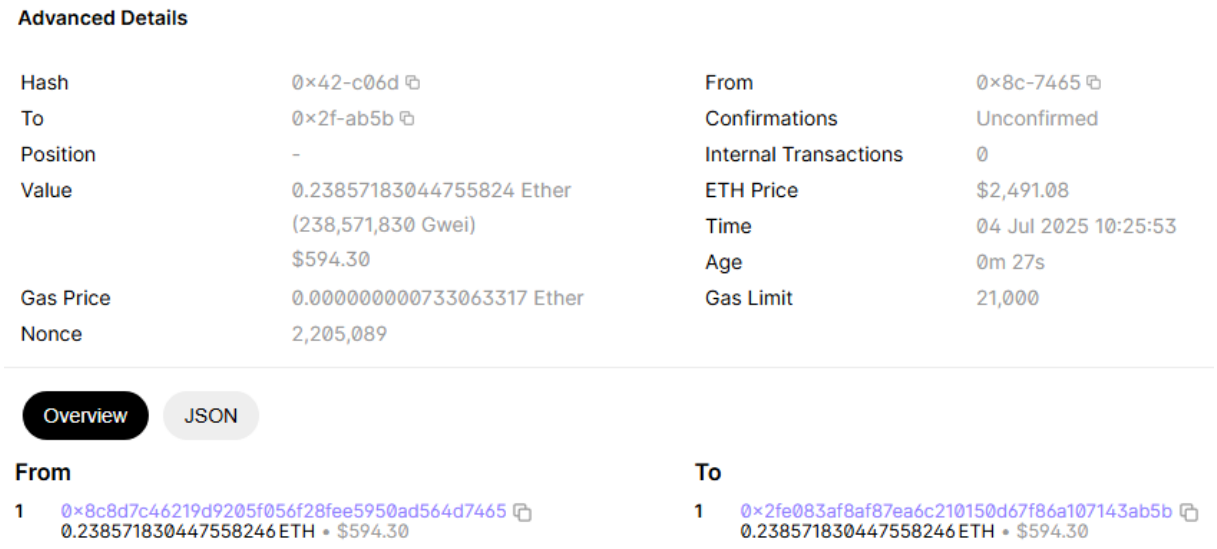
- Authentication Information: Digital signatures or other cryptographic proofs that verify the authenticity and authorization of the transaction originator.

- Payload: The actual data being recorded, which varies significantly based on the ledger's purpose. In the example, the Patient ID, Prescriber ID, and related fields are all part of the payload.

Similarly, domain specific ledgers can implement transaction structures optimized for their particular use cases, such as the medical prescription example above. Figure 2.4 illustrates real world examples of Bitcoin and Ethereum transactions, the two biggest blockchains in the world.



(a) *Bitcoin Transaction Structure from Blockchain.com showing Unspent Transaction Output (UTXO) model with inputs and outputs*



(b) *Ethereum Transaction Structure from Blockchain.com showing account based model with gas fees*

Figure 2.4: *Real world cryptocurrency transactions demonstrating different transaction models*

The Bitcoin transaction in Figure 2.4a exemplifies the UTXO model, where the transaction has one input (0.00123708 BTC) and two outputs (0.00104816 BTC and 0.00018570 BTC), with the difference representing the transaction fee. This structure reflects Bitcoin's design for simple value transfers and privacy through transaction unlinkability.

In contrast, the Ethereum transaction in Figure 2.4b demonstrates the account based model, showing a direct transfer of *0.23857... ETH* from one account to another. The presence of gas price and gas limit fields shows Ethereum's computational model, where transaction fees are

calculated based on computational complexity rather than transaction size alone.

These structural differences highlight how transaction design directly influences the blockchain's capabilities and use cases. Bitcoin's UTXO model prioritizes privacy and simple value transfers, while Ethereum's account based model enables complex smart contract interactions and programmable finance.

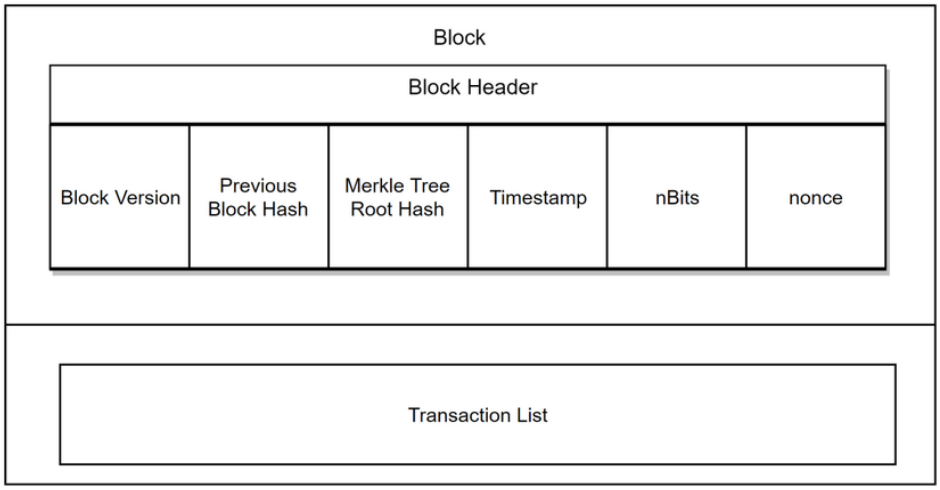
2.1.2.2 Block Structure and Chain Formation

Blocks provide the structural framework for grouping transactions, enabling efficient storage, validation, and distribution of ledger data. Block based ledgers group multiple transactions into discrete units, creating a hierarchical structure that simplifies consensus mechanisms and improves system performance.

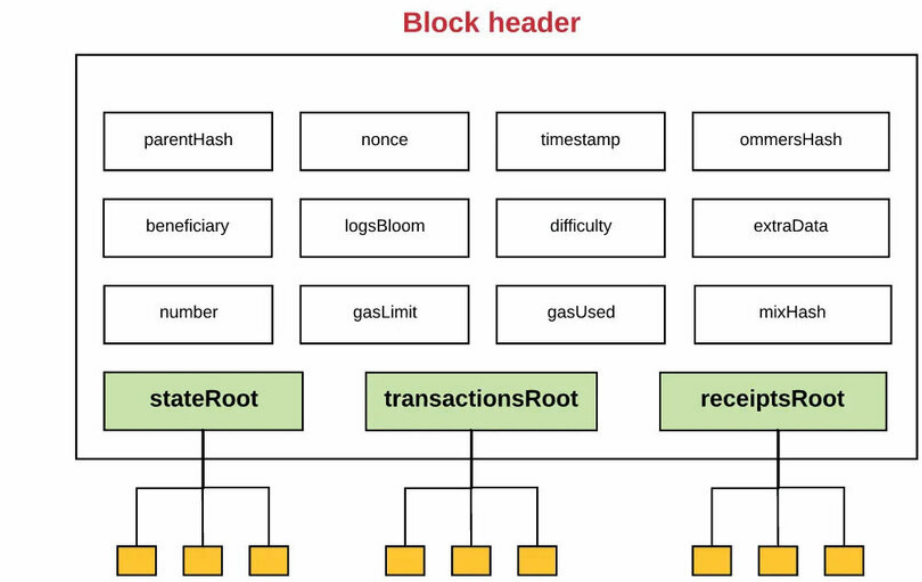
The block structure remains relatively consistent across different ledger implementations, with most sharing common components that are less ledger dependent compared to transaction structures. While there can be variations between different architectures, these changes are typically minor refinements rather than structural overhauls. The most common attributes of blocks are:

- Previous Block Hash: The cryptographic fingerprint of the previous block, creating an immutable chain linkage.
- Timestamp: The creation time of the block, establishing temporal ordering at the block level.
- Merkle Root: The root hash of the Merkle Tree constructed from all transactions in the block, enabling efficient verification, seen previously in Figure 2.2.
- Nonce: A value used in consensus mechanisms, particularly in Proof of Work (PoW) systems.
- Transaction Count: The number of transactions included in the block.
- Block Size: The total size of the block in bytes.

Figure 2.5 demonstrates the block structures implemented in Bitcoin and Ethereum, showing how these fundamental components are organized in practice while highlighting the variations that exist between the architectures.



(a) *Bitcoin block structure showing standard components. Adapted from [56]*



(b) *Ethereum block structure with additional state management fields. Adapted from [57]*

Figure 2.5: *Example of block structures for Bitcoin and Ethereum showing architectural consistency*

The Bitcoin block structure, seen in Figure 2.5a, demonstrates the simplicity of the original blockchain design, focusing on the essential components needed for transaction validation and chain integrity. The structure includes the standard fields creating a straightforward foundation for the cryptocurrency network.

Ethereum's block structure in Figure 2.5b shows additional complexity required for supporting smart contracts and more sophisticated state management. Ethereum maintaining the fundamental block components, Ethereum includes additional fields such as state root, receipts root, and gas limit, reflecting the more complex computational model underlying the Ethereum Virtual Machine.

Despite these differences, both architectures maintain the same principles of block based organization, demonstrating how block structures can accommodate different ledger requirements while preserving the core benefits of batched transaction processing and hierarchical data organization.

2.1.2.3 Architectural Alternatives: Block based vs Blockless Ledger Systems

Block based architectures dominate blockchain implementations due to their ability to maintain a consistent global state across a distributed network. Transactions are grouped into blocks, which serve as natural synchronization points, simplifying consensus and enabling efficient batch processing.

In contrast, blockless systems add transactions directly to a global Merkle Tree without waiting for block formation. This eliminates intentional delays, allowing immediate transaction finalization. This approach can reduce latency and provide predictable logarithmic performance, but it changes the trade-offs in storage and network overhead.

Table 2.2 summarizes the performance characteristics of both approaches across important metrics.

Table 2.2: Performance Comparison of Block based vs Blockless Architectures

Operation	Block based	Blockless
Adding Transaction	$O(1)$ to mempool	$O(\log n)$ to global MT
Generating MT	$O(n)$ per block	$O(\log n)$ per transaction
Validation	$O(\log n)$ within block	$O(\log n)$ in global MT
Storage	$O(n+b)$	$O(3n-1)$
Network Propagation	$O(b)$ per block	$O(t)$ per transaction

As shown in Table 2.2, block based systems excel in high throughput scenarios where batching amortizes costs across multiple transactions. They are particularly effective in distributed environments, where fixed synchronization points simplify consensus coordination. In centralized deployments, block size can be adjusted to match transaction volume, balancing latency with the organizational benefits of batching.

Blockless architectures, such as those demonstrated by ImmuDB [11], use MT based structures for cryptographic integrity without batching. While they provide consistent $O(\log n)$ insertion and verification performance, they typically require higher storage overhead ($O(3n-1)$) compared to block based designs. Despite this, blockless systems are advantageous for applications requiring minimal transaction latency, whereas block based systems remain the preferred choice for scalable, batch oriented processing in both distributed and centralized environments [20].

2.1.3 Types of Ledger Technologies

Ledger technologies can be classified according to their network architecture and governance model, as well as their access control.

2.1.3.1 Centralized vs Decentralized Architectures

From a governance perspective, there are two main architectural models:

- Centralized: All control and validation are performed by a single authority or organization. Clients connect directly to this central server.
- Decentralized: No single authority controls the system. Multiple independent nodes participate in validation and governance.

Both centralized and decentralized ledgers can be *distributed*, meaning their computation and data storage are spread across multiple nodes for performance and fault tolerance. In addition, decentralized systems can be *federated*, where multiple independent authorities (e.g., organizations in a consortium) each operate their own nodes while following shared protocols.

2.1.3.2 The Blockchain Trilemma

Blockchains, being a type of decentralized and distributed ledger, often operate in open environments where any participant can join. A well known design constraint in this space is the *blockchain trilemma* [77], which states that a ledger can at most optimize for two of the following: *security*, *scalability*, and *decentralization*, as seen in Figure 2.6.

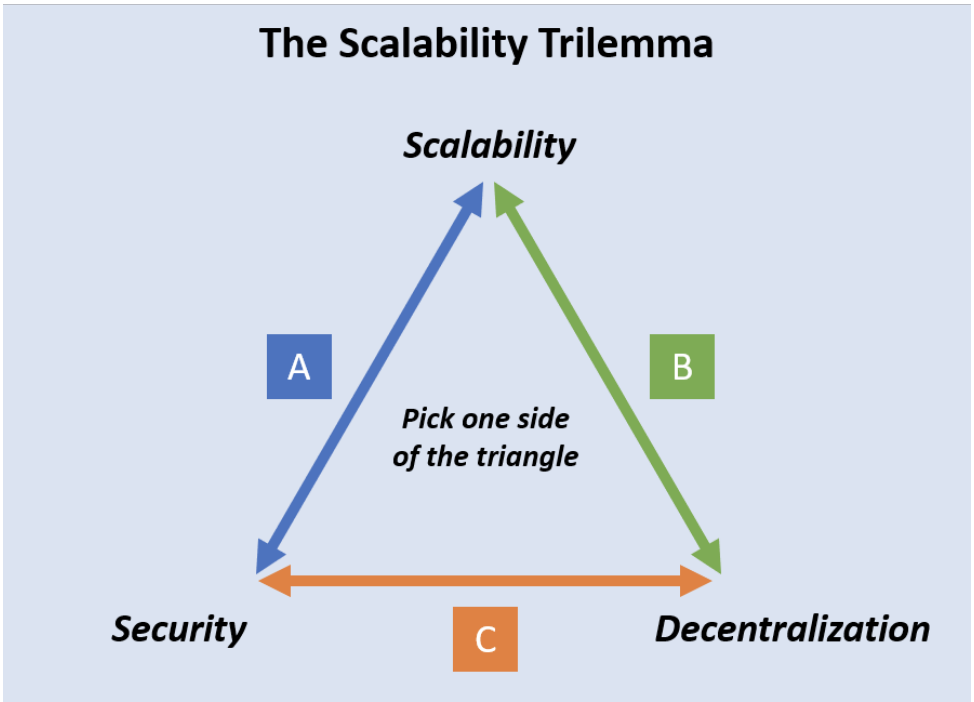


Figure 2.6: *The blockchain trilemma. Adapted from [38]*

Cryptocurrencies tend to maximize the security and decentralization making them unsuitable for high workloads, whilst centralized ledgers focus on security and scalability making them ideal for those scenarios.

2.1.3.3 Permissioned vs Permissionless Networks

Blockchains can be:

- Permissionless (always public): Anyone can join, validate, and submit transactions. They rely on open consensus mechanisms and often use cryptocurrencies for security and participant incentives [16, 47].
- Permissioned: Participation is restricted to authorized entities. These can be:
  - *Private*: controlled by a single organization.
  - *Consortium/Federated*: controlled by a group of organizations.
  - *Hybrid*: combining public and private elements.

Figure 2.7 summarizes the main types.

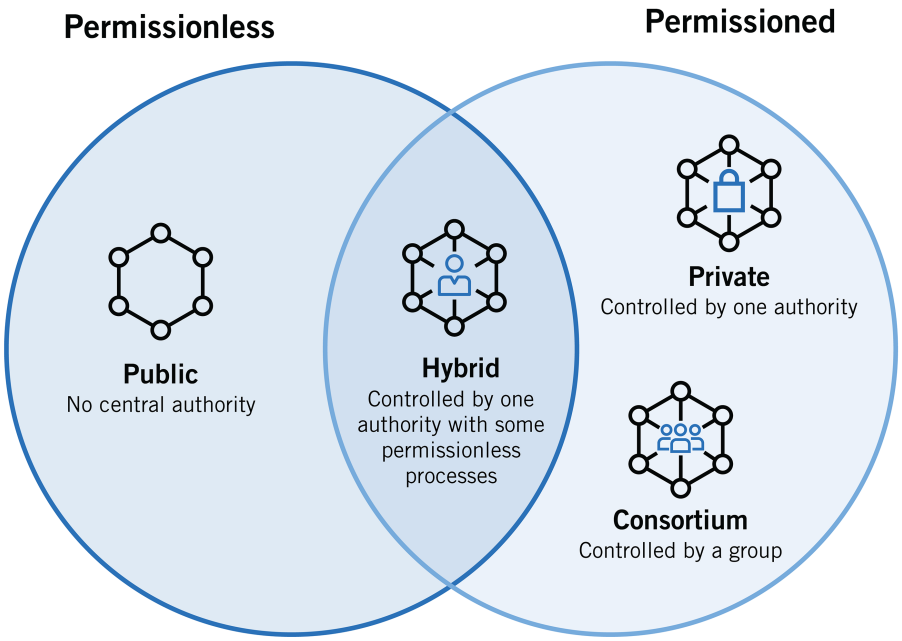


Figure 2.7: *Types of blockchains: Permissionless (always public) and Permissioned (private, consortium, and hybrid). Adapted from [19]*

It is important to note that a *centralized ledger* is inherently *permissioned* due to the presence of a central authority controlling participation. Depending on governance needs, it can take the form of a *private*, *consortium*, or *hybrid* model, as the controlling entity has full discretion over access, validation, and operational policies.

2.1.4 Consensus Mechanisms

Distributed ledgers must ensure that all nodes maintain the same state despite network delays, failures, or malicious actors. Consensus mechanisms achieve this by defining rules for agreeing on the next valid ledger state.

In permissionless networks, the most common approaches are:

- PoW: Requires computational effort to add blocks, making attacks costly (e.g., Bitcoin). Highly secure but energy intensive.
- Proof of Stake (PoS): Validators lock tokens as collateral, risking them if acting maliciously (e.g., Ethereum). More energy efficient but adds economic complexity.

A centralized ledger does not require a distributed consensus, because it has a trusted authority that validates all transactions. This eliminates coordination delays, achieving high throughput and low latency, but concentrates trust and creates a single point of failure.

Both approaches have trade-offs: decentralized consensus increases resilience and trustlessness at the cost of performance and complexity, while centralized validation offers efficiency but requires full trust in a single authority.

2.2 Current Landscape

Ledger systems have a simple core concept, leading to multiple technological approaches for implementing immutable solutions. However, the majority of current research and development focuses on blockchain technologies specifically designed around cryptocurrency applications, emphasizing features such as trustless value transfer and decentralized consensus mechanisms that are irrelevant for centralized enterprise use cases.

The primary objective of the thesis is document certification and it does not require trustless operations, but rather the cryptographic proof that specific events occurred at particular times in a specific order. Consider scenarios where a doctor issues a prescription that a pharmacy cannot later deny, an insurance company processes a claim that cannot be repudiated, or employees register their work attendance creating an immutable record for accountability purposes. These applications assume trust among parties under normal circumstances but require rapid resolution capabilities when disputes arise.

2.2.1 Blockchain Applications

Blockchain technologies, are implementations of ledger systems, but have fundamental limitations for enterprise document certification applications. The financial success of Bitcoin and Ethereum has spawned numerous alternative cryptocurrencies that replicate core blockchain functionalities but remain susceptible to consensus attacks, particularly 51% attacks, when operated with insufficient network participation or economic incentives.

The economic volatility inherent in cryptocurrency based systems creates significant operational challenges for enterprise deployment. Bitcoin transaction fees demonstrate high variability, fluctuating from $0.77 to over $1.50 within a three day period, with costs influenced by network congestion, transaction complexity, and temporal factors [9, 78]. Ethereum exhibits similar volatility patterns, with transaction fees ranging from $0.30 to $0.75 within a single month [79].

This unpredictability contradicts business requirements for cost predictable, long-term operational planning.

For the objectives of this thesis, cryptocurrency dependent solutions are unsuitable due to their inherent volatility, external dependencies, and operational complexity that provides no additional value for centralized use cases where trust relationships already exist between participating parties.

2.2.2 Cloud Based Solutions

Cloud based immutable database services include Amazon Quantum Ledger Database (QLDB) (discontinued in July 2025 and migrated to Amazon Aurora) [4, 5], Azure SQL Database Ledger [42], Oracle Blockchain Tables [49], among others. These solutions provide managed infrastructure and enterprise integration capabilities but introduce vendor dependency and architectural constraints that may not align with organizational requirements.

A critical consideration for enterprises is the necessity of cloud computing for their specific use cases. Research indicates that cloud adoption should be evaluated based on scalability requirements, cost-benefit analysis, and operational complexity rather than assumed as universally beneficial [31]. The predominant focus on Structured Query Language (SQL) based data models in these solutions, while adequate for many applications, limits flexibility for organizations requiring alternative data structures or custom business logic implementations.

Vendor lock-in represents a significant concern with cloud based ledger services, as migration between providers often requires substantial architectural modifications and data transformation processes [30, 34]. Self hosted solutions can address these limitations while maintaining the option for containerized deployment across multiple cloud providers, providing operational flexibility without vendor dependency.

2.2.3 Timestamp Services

Timestamp services such as Stampery [67], ChainPoint [71], and OpenTimestamps [72] provide cryptographic proof of document existence at specific points in time through blockchain anchoring mechanisms. While these services offer valuable timestamping capabilities, they inherit the fundamental limitations of their underlying blockchain dependencies and introduce external service reliance.

Commercial timestamp services often require subscription fees and proprietary integration processes, while free alternatives like OpenTimestamps depend on Bitcoin network stability and community donations for operational sustainability. This creates long term operational risks where service discontinuation could impact historical verification capabilities.

More significantly, timestamp services provide only existence proofs rather than comprehensive business logic integration. Applications frequently require complex relationships between documents and events such as linking medical prescriptions with corresponding medication dispensing records that extend beyond simple temporal verification. These specific requirements necessitate integrated solutions rather than external timestamp service dependencies.

2.2.4 Document Signature Services

Commercial document signature platforms such as DocuSign [13], Adobe Sign [3], and similar services [51] focus primarily on workflow automation and legal signature collection rather than immutable logging or tamper-evident record keeping. These systems reliably manage document approval processes and maintaining audit trails for signature workflows but do not provide cryptographically verifiable immutable storage or comprehensive event logging capabilities.

The subscription based business models of these platforms create ongoing operational costs, while their proprietary formats and APIs introduce vendor dependency similar to cloud based solutions. Additionally, these services are designed for signature workflows rather than general purpose document certification, limiting their applicability to the project.

While document signature services provide valuable insights into user experience design and workflow integration, their fundamental architecture differs significantly from immutable ledger systems, focusing on signature collection and workflow management rather than cryptographic integrity and tamper-evident logging.

2.3 Gaps in Development

The analysis of existing solutions reveals a gap in the availability of fully independent, self-hosted immutable logging systems that operate without reliance on cloud providers, blockchain networks, or external services. Current market offerings require either cryptocurrency network dependencies, cloud service subscriptions, or proprietary platform integration, none of which address the need for complete operational independence and cost predictability.

Specifically absent from the current landscape are open source, self-hosted applications capable of providing cryptographically verifiable immutable logging with multi-ledger architectures that enable logical separation of different business domains while maintaining referential integrity between related records. The capacity for interdependent record relationships requires integrated business logic that existing timestamp services and signature platforms cannot provide.

This thesis addresses these gaps by proposing a centralized, self-hosted immutable ledger management system that provides a controlled document certification without external dependencies, cryptocurrency volatility, or vendor lock-in, while supporting flexible multi-ledger architectures for complex business logic implementation.

CHAPTER 3

PROPOSED SOLUTION

The third chapter provides the proposed solution for the thesis and insight on the design decisions made. Beginning with Section 3.1 with the main design objectives and properties of the solution, followed in Section 3.2 by the project's global architecture overview. Section 3.3 will detail the languages, frameworks and libraries used across the system layers. Section 3.4 covers the server-side implementation including the ledger domain, audit warden, providers, PKI, authentication, and workflow services. Section 3.5 addresses the client application with its models, viewmodels, services, and user interface components. Section 3.6 will cover the data storage approach including both relational database design and file repository management for testing and integration in a practical solution. Finally, Section 3.7 details the containerized deployment architecture and orchestration procedures for both development and production environments.

The implementation of the project is available on GitHub:

https://github.com/NunoBartolomeu/ThesisCode

3.1 Design Philosophy and Priorities

The centralized ledger system was conceived to bridge the gap between traditional audit systems and blockchain-inspired integrity verification. The design philosophy centers on creating a practical solution that organizations can deploy without radical infrastructure changes while maintaining the security guarantees essential for accountability and compliance.

Five core principles guided every architectural decision:

- Compatibility: The system must integrate easily with existing organizational infrastructure, supporting standard authentication protocols, database systems, and deployment patterns without requiring specialized blockchain expertise or infrastructure.
- Self-containment: All components necessary for system operation should be included

within the application, eliminating dependencies on external blockchain networks, third-party validation services, cloud structures or complex distributed infrastructure.

- Consistency: Data integrity and system behavior must remain predictable across all operations, ensuring that ledger entries maintain cryptographic validity regardless of system load, network conditions, or operational complexity.
- Security: The system must provide cryptographic guarantees equivalent to blockchain systems for data integrity and non-repudiation, while maintaining centralized control over access permissions and operational policies.
- Accessibility: Both technical implementation and operational management should remain comprehensible to standard IT professionals, avoiding the specialized knowledge requirements typically associated with distributed ledger technologies.

The solution prioritizes practicality while preserving the cryptographic foundations that make ledger systems valuable for accountability and audit purposes.

3.1.1 Nomenclature

The system deliberately adopts nomenclature that separates itself from blockchains. This approach helps organizations understand the system's capabilities clearly:

- Entries: Instead of *Transactions* the system will use *Entries*, recognizing that not all ledger events represent value transfers or state changes. Entries save any event requiring cryptographic certification, from document approvals to contract confirmations.
- Pages: Substituting the blockchain concept of *Blocks*, the system will use *Pages* drawing inspiration from traditional accounting ledgers that were actual books. Pages provide organization for entries, maintaining cryptographic chains that enable integrity verification.
- Receipts: Absent from public blockchains, *Receipts* are essential for centralized systems. These cryptographic proofs enable users to independently verify their interactions with the ledger, providing protection against disputes with system operators or internal attacks while supporting external audit processes.

3.2 Architecture

The system employs a layered architecture that separates concerns while maintaining clear interfaces between components [21]. This architectural pattern enables organizations to customize specific layers without affecting the integrity of the core ledger functionality. The layered approach consists of three distinct tiers: Presentation Layer, Business Layer, and Persistence Layer [55].

- Presentation Layer: Manages User Interface (UI) components and client-side interactions

- Business Layer: Handles application logic, service orchestration, and business rules
- Persistence Layer: Manages data storage, retrieval, and file system operations

Figure 3.1 illustrates the proposed architectural approach. The system consists of a Web Server forming the Presentation Layer, an Application Programming Interface (API) and Server implementing the Business Layer, and an SQL Server with NoSQL/File Repository constituting the Persistence Layer.

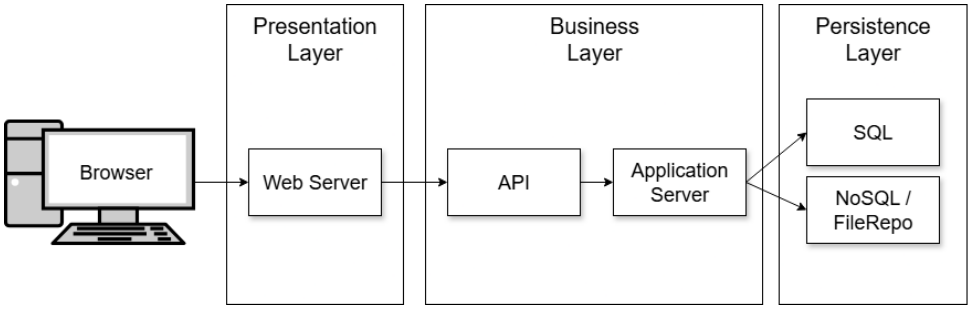


Figure 3.1: *Layered Architecture Overview*

Each layer implements a specific architectural pattern optimized for its responsibilities. The Presentation Layer employs the Model-View-ViewModel (MVVM) architecture [41], represented in Figure 3.2, which provides advantages including separation of concerns, improved testability, and enhanced maintainability [27]. This pattern aligns with the design goals of maintaining clear component boundaries and supporting independent development workflows.

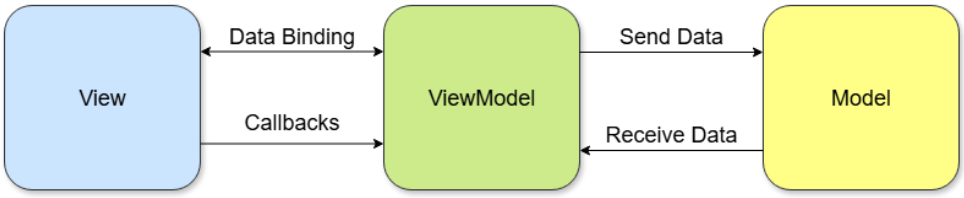


Figure 3.2: *MVVM Architecture Diagram*

The Business Layer follows Service-Oriented Architecture (SOA) principles combined with the Controller-Service-Repository pattern commonly used in Spring applications [65]. The SOA approach provides the following characteristics [15]:

- Service Autonomy: Each service maintains operational independence
- Service Composability: Services can be combined to create composite solutions
- Service Abstraction: Implementation details are hidden from service consumers
- Service Reusability: Services are designed for use across multiple contexts
- Service Standardization: Consistent service contracts and communication protocols

The Controller-Service-Repository pattern implements a clear separation of responsibilities within the Business Layer [66]. In this architectural pattern, represented in Figure 3.3, the Controller functions as the API endpoint, receiving Hypertext Transfer Protocol (HTTP) requests, converting Data Transfer Object (DTO), invoking appropriate services, and returning responses to clients. The Service layer contains the core business logic, operating within SOA principles while coordinating repository calls and managing exception handling. The Repository layer manages data persistence by converting business objects to Data Access Object (DAO) and interfacing with the Persistence Layer.

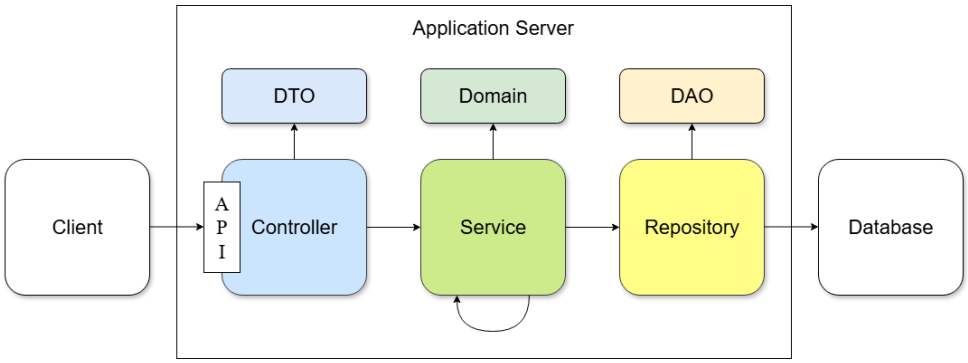


Figure 3.3: *Controller-Service-Repository Pattern Diagram*

These architectural decisions align with the system's design principles by promoting modularity, maintainability, and integration capabilities while preserving the centralized control necessary for audit and compliance requirements.

3.3 Languages, Frameworks and Libraries

A framework is a reusable software platform that provides a foundation and structure for application development, implementing inversion of control where the framework calls application code. A library consists of pre-written code modules that applications call directly to perform specific functions [23]. The distinction lies in control flow: frameworks control the application execution, while applications control library usage [22]. Based on this distinction, the implemented system functions as a library, providing modular components that organizations can integrate and customize within their existing applications according to specific requirements.

3.3.1 Presentation Layer Technologies

The Presentation Layer utilizes TypeScript as the primary programming language [44]. TypeScript extends JavaScript by adding static type definitions, enabling more robust development practices and reducing runtime errors [8]. This type-safe approach aligns with the typed nature of the Business Layer implementation, facilitating consistent data handling across system boundaries.

HTML5 and CSS3 provide the foundational markup and styling capabilities for user interface construction [75, 76]. The layer incorporates React as the primary UI library [40]. React's component-based architecture supports the MVVM pattern implementation by enabling clear

separation between view components and business logic [61]. React's reactive programming model ensures that UI components update automatically in response to state changes, providing responsive user interactions and maintaining interface consistency throughout the application lifecycle [39].

Tailwind CSS serves as the utility-first Cascading Style Sheets (CSS) framework [68], providing a comprehensive set of low-level utility classes that enable rapid user interface development while maintaining visual consistency. Tailwind's approach eliminates the need for separate CSS files by allowing developers to apply styling directly within TypeScript/JSX components, ensuring type safety. The framework includes color palettes, spacing scales, and responsive design utilities that maintain design system coherence across components [69].

Next.js functions as the React framework, offering server-side rendering, static site generation, and optimized performance features [74]. Compared to alternative solutions such as Create React App, Next.js provides automatic code splitting, and improved initial page load performance through pre-rendering strategies [1].

3.3.2 Business Layer Technologies

Kotlin is used as the primary programming language for the Business Layer [33]. Kotlin provides full interoperability with Java, static typing, support for both object-oriented and functional programming paradigms, and enhanced readability compared to traditional Java implementations [32]. The language's integration with Spring Framework components facilitates the implementation of enterprise-grade applications [62].

Spring Framework provides the foundational infrastructure for dependency injection, aspect-oriented programming, and enterprise application development [53]. Aspect-Oriented Programming (AOP) enables the separation of concerns such as logging, security, and transaction management from core business logic, allowing these to be applied declaratively across multiple components [64]. The framework's comprehensive ecosystem supports the implementation of RESTful APIs, security mechanisms, and data access layers through standardized patterns and conventions [63].

Lastly, Maven manages project dependencies, build processes, and artifact generation [6].

Additional libraries support specialized functionality:

- Java Database Connectivity (JDBC): Provides standardized database access mechanisms for SQL operations within the Persistence Layer [50]
- Jackson: JavaScript Object Notation (JSON) processing and object serialization [18]
- JSON Web Token (JWT): Stateless authentication and authorization [35]
- Bouncy Castle: Cryptographic algorithm implementation and PKI support [70]

3.3.3 Persistence Layer Technologies

PostgreSQL serves as the primary relational database management system [54]. PostgreSQL was selected for its ACID compliance, extensibility, and support for complex queries and transactions [45]. The system's requirements for data integrity and consistency align with PostgreSQL's implementation of multi-version concurrency control and transaction isolation mechanisms.

The local file system provides document and artifact storage capabilities for development and testing environments. For production deployments requiring distributed storage, the architecture supports integration with alternative solutions such as the InterPlanetary File System (IPFS) [7], NoSQL document stores such as MongoDB [46], or cloud-based object storage services, enabling scalability without architectural modifications.

This technological stack was selected to balance development efficiency, system performance, and long-term maintainability while supporting the system's core requirements for cryptographic integrity, audit capabilities, and organizational integration.

3.4 Server Implementation

The application server is implemented as a single Kotlin application following modular organization principles. The project structure reflects the Controller-Service-Repository pattern, with each architectural component implemented as a distinct module. The domain model includes DTO and domain models representing business entities, while DAO are omitted in favor of direct JDBC implementation, enabling explicit SQL query construction and execution [50].

Additional utility modules provide specific functionalities including cryptographic providers, such as the *HashProvider* and *SignatureProvider*, and an enhanced color-coded logging component that assigns distinct colors to each component for improved log readability during development and debugging phases.

All modules within the Controller-Service-Repository architecture maintain separation across four primary functional domains: Ledger operations, Authentication services, File management, and Public Key Infrastructure (PKI). This modular separation enables independent development and testing while maintaining clear interface boundaries between distinct areas.

3.4.1 Ledger Domain and Service

The ledger system constitutes the core component of the centralized ledger architecture, comprising four fundamental entities: Entries, Pages, Receipts, and Ledgers. The distinction between domain and service layers reflects the separation of data structures and business logic from operational or management functions. The domain layer contains all structural definitions and lifecycle functions, while the service layer provides multi-ledger management capabilities and external interface abstractions that are used by other services and controllers.

3.4.1.1 Entries

Entries represent the fundamental units of the ledger system. They are designed to accommodate domain-specific requirements while maintaining consistent validation mechanisms. Each entry contains domain-independent attributes necessary for system operation and flexible attributes that adapt to specific use cases.

The entry structure incorporates the following attributes:

- ledgerName: Identifies the specific ledger containing the entry, enabling multi-ledger architecture
- pageNumber: References the page containing the entry, remaining null for entries awaiting page inclusion
- id: Unique identifier for the entry within the ledger context
- timestamp: Creation timestamp using Unix epoch milliseconds for consistency
- content: Domain-specific information requiring cryptographic certification
- senders: List of users creating the entry, with at least one sender required for validation
- recipients: Optional list of users receiving or referenced by the entry
- hash: Cryptographic digest computed at creation for tamper detection and integrity verification
- signatures: Digital signature data for entry senders, supporting asynchronous signing workflows
- keywords: Metadata tags for efficient querying and categorization
- relatedEntries: Links to associated entries for navigation and relationship tracking

The signature structure includes the necessary cryptographic information required for verification:

- signerId: Identifier linking the signature to the corresponding sender
- publicKey: Cryptographic public key for signature verification
- signatureData: Digital signature computed over the entry hash
- signingAlgorithm: Algorithm specification required for proper signature verification

Following are two different examples of applications for the ledger system. Table 3.1 demonstrates entry structures within a medical records scenario, while Table 3.3 illustrates document management applications. Table 3.2 provides detailed signature information for the medical entries, demonstrating the cryptographic components required for verification, while Table 3.4 does the same for the document management entries.

Table 3.1: Medical Records Entry Examples

	MED001	**MED002**
Ledger Name	Medical Records	Medical Records
Page Number	42	42
ID	MED001	MED002
Timestamp	2024-08-12 14:30	2024-08-12 15:45
Content	Patient diagnosis: Type 2 Diabetes, prescribed Metformin 500mg	Medication dispensed: Metformin 500mg, 30 tablets, prescribed by Dr. Smith
Senders	Dr. Smith	Pharmacist Johnson
Recipients	John Doe	John Doe
Hash	ce4cdd...	bbc02a...
Signatures	Dr. Smith signature	Pharmacist Johnson signature
Keywords	diabetes, prescription, Metformin	dispensing, Metformin, pharmacy
Related Entries	MED002	MED001

Table 3.2: Signature Details for Medical Entries

	Dr. Smith (MED001)	**Johnson (MED002)**
Signer ID	Dr. Smith	Pharmacist Johnson
Public Key	30819f...	608d01...
Signature Data	304502...	502d4e...
Algorithm	SHA256withRSA	SHA256withRSA

Table 3.3: Document Management Entry Examples

	DOC001	**DOC002**
Ledger Name	Legal Documents	Financial Reports
Page Number	15	23
ID	DOC001	DOC002
Timestamp	2024-08-05 09:20	2024-08-05 16:00
Content	Contract approval for Project Alpha, version 2.1, including terms for delivery milestones and payment schedule	Financial report Q3 2024 certified by external auditor, showing 15% revenue growth compared to Q2
Senders	CEO Charlie	CFO Dave
Recipients	Project Manager	Director Eric; Director Felix
Hash	cc155e...	f5c5ed...
Signatures	CEO Charlie signature	CFO Dave signature
Keywords	contract, approval, project	financial, audit, Q3, report
Related Entries	null	null

Table 3.4: Signature Details for Document Management Entries

	CEO Charlie (DOC001)	**CFO Dave (DOC002)**
Signer ID	CEO Charlie	CFO Dave
Public Key	d06092...	a86488...
Signature Data	01c8e7...	a9f6c3...
Algorithm	SHA256withRSA	SHA256withRSA

The entry hash computation includes id, timestamp, content, senders, and recipients, while excluding ledgerName to allow ledger name changes, pageNumber due to its mutable nature during the entry lifecycle, and metadata fields such as keywords and relatedEntries that serve only for querying and can be adjusted.

To implement such system is necessary to expose precise hash calculation specifications. Organizations must understand that hash computation requires exact specification of included attributes, their ordering, data types, and concatenation methods. Variations in any of these parameters will result in different hash values, compromising verification capabilities. The system documentation must provide complete hash calculation specifications including attribute serialization formats and concatenation delimiters.

Signature validation was designed so only the computed entry hash needs to be signed, it also supports multiple signing algorithms, even on the same entry, for organizational flexibility. Once validated, signatures become immutable components of the entry record, ensuring non-repudiation for all participants.

Entry deletion handling varies depending on entry status. Entries in the pool phase can be completely deleted without compromising the cryptographic side of the ledger. However, once an entry becomes part of a permanent page, complete deletion would compromise the page's Merkle root integrity. The current implementation addresses this requirement by replacing the entry's content with the hash of the original content and a deletion marker, regardless of the status of the entry. This helps to maintain the page cryptographic integrity while enabling sensitive information removal for privacy compliance.

3.4.1.2 Pages

Pages provide organizational structure for entries, implementing a simplified version of block-based architectures adapted for centralized environments. Unlike blockchain blocks, pages focus on efficient organization and integrity verification, so they don't contain any attributes related to consensus mechanisms.

Page attributes include:

- ledgerName: Identifies the containing ledger
- number: Sequential page identifier within the ledger
- timestamp: Page creation timestamp using Unix epoch milliseconds
- previousHash: Cryptographic link to the preceding page for chain integrity

- merkleRoot: Cryptographic hash summary of all contained entries
- hash: Page integrity digest computed from all page attributes except entries
- entries: Collection of entries within the page

The page hash computation includes all attributes except the entries list, as the Merkle root provides cryptographic coverage of entry data. Merkle tree construction handles odd numbers of entries by duplicating the final node, following the pattern: when nodes exist in pairs, combine left + right; when an odd node remains, combine left + left for hash calculation.

Table 3.5 illustrates page structures containing the previously mentioned entries.

Table 3.5: Page Structure Examples

	Medical Page	**Legal Page**	**Financial Page**
Ledger Name	Medical Records	Legal Documents	Financial Reports
Number	42	15	23
Timestamp	2024-08-12 16:00	2024-08-05 10:00	2024-08-05 18:00
Previous Hash	cd5bbf...	3b8a62...	bdaa7b...
Merkle Root	bb5446...	815de2...	77600b...
Hash	f115a1...	db9ecf...	845677...
Entries	[MED001, MED002, ...]	[DOC001, ...]	[DOC002, ...]

Page creation occurs automatically based on configurable triggers. For this implementation, the accumulation of verified entries reaching a defined number will trigger a new page, but advanced triggering mechanisms could be time based or use the system's load balancing for a more personalized lifecycle.

Page structure has a consistent size regardless of entry count or content volume. For example, using SHA-256 hashing and ledger names up to 16 characters, each page requires approximately 128 bytes for storage (excluding entry data, which is stored separately): 16 bytes for ledgerName, 8 bytes for number, 8 bytes for timestamp, 32 bytes for previousHash, 32 bytes for merkleRoot, and 32 bytes for the page hash. This means pages are lightweight, which is necessary to not bloat the system.

The centralized architecture eliminates network propagation delays, making page creation nearly instantaneous, specially from the user perspective, while providing clear boundaries for data archival and efficient synchronization capabilities.

3.4.1.3 Receipts

Receipts provide cryptographic proof for the entries and are necessary for users to protect themselves against internal attacks to the ledger system. They serve as verifiable objects that demonstrate an entry's inclusion in the ledger structure and provide the necessary information for independent verification by third parties. The receipt can be requested at any moment, even before the entry acquires all signatures, proving the entry was created by the system, this

protects against possible attacks or information losses before the entry has the chance to be included in a page.

The receipt structure incorporates the following attributes:

- entry: Complete entry data for which the receipt is generated
- timestamp: Receipt generation time using Unix epoch milliseconds
- requesterId: Identifier of the user requesting the receipt
- proof: Merkle tree inclusion proof demonstrating entry presence in page structure
- hash: Receipt integrity hash computed from receipt data
- signatureData: Digital signature from the system's private key
- publicKey: System's public key for signature verification
- algorithm: Signature algorithm specification

Receipt generation follows a structured process ensuring both security and accessibility. The system validates that the requester is an authorized participant (either sender or recipient) of the entry before generating the receipt. The inclusion proof consists of the minimal set of hash values required to reconstruct the Merkle tree path from the entry to the page's Merkle root, enabling independent verification of entry inclusion.

The receipt hash computation includes the entry hash, timestamp, requester ID, and concatenated proof values, using pipe (|) delimiters between components. This hash becomes the input for the system's digital signature, providing non-repudiation and authenticity guarantees.

Table 3.6 demonstrates a receipt structure for medical entry MED001.

Table 3.6: Receipt Example for Medical Entry MED001

Attribute	**Value**
Entry ID	MED001
Timestamp	2024-08-12 20:00
Requester ID	John Doe
Inclusion Proof	[bbc02a..., ff3c8d..., bb5446...]
Receipt Hash	a7b8c9...
Signature Data	304602...
Public Key	30819f...
Algorithm	SHA256withRSA

Receipts serve multiple operational purposes including audit trail documentation, regulatory compliance evidence, and third-party verification capabilities. The cryptographic properties enable recipients to prove entry authenticity without requiring direct system access, supporting scenarios where independent verification is necessary for legal or regulatory purposes.

Receipt validation requires verification of multiple parts. Starting with the hash validation of the receipt, to ensure it hasn't been tampered, then the validation of the signature with the system's

public key. Then each signature on the entry should be validated using the same method as the systems signature, and lastly, if the inclusion proof is not empty, the auditor needs to use each hash inside the proof in order then compare it to the Merkle Root of the page. This multi-step validation confirms that the system contains the entry and generated the receipt proving it's existence.

3.4.1.4 Ledgers

Ledgers maintain operational parameters and state management for specific domains within the multi-ledger architecture. Unlike blockchain systems, where the chain represents the primary data structure, the multi-ledger approach requires explicit ledger entities to manage configuration, entry pools, and page sequences.

The system was designed to handle concurrent access from the start, ensuring thread-safe operations across all ledger components. Cross-ledger operations do not affect each other, meaning that activities in one ledger do not block operations in another ledger, enabling true parallel processing of different organizational domains.

Ledger attributes encompass:

- name: Unique identifier for the ledger within the system
- entriesPerPage: Configuration parameter determining page creation triggers
- hashAlgorithm: Cryptographic algorithm specification for entry, page, and Merkle tree computations
- entryPool: Collection of entries awaiting page inclusion, divided into verified and unverified subsets for processing efficiency
- pages: Ordered sequence of completed pages maintaining chronological integrity

Table 3.7 presents examples of different ledger configurations for the various use cases already explored.

Table 3.7: Multiple Ledger Configuration Examples

Name	Medical Records	Legal Documents	Financial Reports
Entries Per Page	50	25	100
Hash Algorithm	SHA-256	SHA-256	SHA-512
Current Page	42	15	23
Entry Pool Size	3	7	15

Concurrency Architecture The entry pool implementation utilizes thread-safe data structures to ensure reliable concurrent access. `ConcurrentHashMap` is employed for both verified and unverified entry collections, providing lock-free read operations and efficient write operations without blocking other threads. The `ConcurrentLinkedQueue` manages the page sequence, enabling atomic additions while maintaining chronological order.

For critical sections requiring atomicity, particularly during page creation, a `ReentrantLock` provides exclusive access to prevent race conditions. The batch lock ensures that only one thread can create a page at a time within a specific ledger, while still allowing concurrent operations in other ledgers. This design maintains data consistency during the complex process of moving entries from the verified pool to permanent pages while computing Merkle trees and updating page sequences.

The separation between verified and unverified entries enables efficient page creation by processing only fully signed entries, while the thread-safe collections allow concurrent signature addition and entry validation without blocking other operations.

This approach enables organizations to maintain separate certification domains with appropriate security and operational parameters while maintaining a shared infrastructure. Each ledger operates independently, allowing domain-specific optimization without compromising system consistency or security guarantees.

3.4.1.5 Ledger Service

The Ledger Service extends the domain functionality by providing management interfaces and operational coordination between system components. The service handles storage delegation to repositories, comprehensive logging of all performed actions, and efficient querying capabilities for entries and related data.

The service implements multi-ledger management through a centralized coordinator that maintains separate ledger instances while providing unified access interfaces. With this is possible to expand the ledger domains without requiring system-wide modifications.

The primary functionalities of the service are:

- Ledger Management:
  - Initial ledger creation with configurable parameters
  - Page creation frequency adjustment based on operational requirements
  - Ledger isolation ensuring organizational boundary enforcement
- Entry Operations:
  - Entry creation with automatic ID generation and timestamp assignment
  - Asynchronous signature collection supporting diverse approval workflows
  - Entry deletion with comprehensive cryptographic audit trails
  - Metadata management including keyword indexing and relationship mapping
- Verification and Receipts:
  - Real-time receipt generation for all significant state changes
  - System-signed verification documents providing cryptographic proof
  - Status tracking throughout the complete entry lifecycle

The service has data retrieval capabilities including individual entry lookups, user-based queries, and keyword-based searches. Additionally, a specialized logging function enables other system services to record their activities within designated audit ledgers, this permits all services to have cryptographic logging.

The service layer abstracts the complexity of multi-ledger coordination while maintaining the cryptographic guarantees and integrity verification capabilities that define the system's core value proposition.

3.4.2 Audit Warden

The ledger domain and service components ensure cryptographic integrity during entry and page creation, but continuous validation requires periodic integrity verification to detect post-creation tampering. Ledger systems provide tamper-evidence rather than tamper-prevention, meaning unauthorized modifications can be detected but not automatically corrected [58]. The Audit Warden addresses this requirement through systematic integrity monitoring and validation reports.

The Warden operates on scheduled intervals to perform health checks across all ledgers within the system. When tampering is detected, it generates incident reports specifying the *when*, *where*, and *what* was compromised. Advanced implementations may incorporate additional log analysis to also determine the *who* and *why* of the tampering attempt.

Regular backup operations become essential in tamper-evident systems, as restoration to consistent states represents the primary recovery mechanism when integrity violations occur. The frequency of backup operations should align with business requirements and acceptable data loss tolerances, balancing operational overhead against recovery capabilities.

3.4.2.1 Validation Architecture

The Warden implements comprehensive validation across multiple cryptographic and structural parts. The validation process encompasses four primary categories of integrity verification:

- Cryptographic Integrity: Hash verification for entries, pages, and Merkle tree structures
- Structural Consistency: Validation of relationships between ledgers, pages, and entries
- Temporal Ordering: Verification of timestamp sequences and chronological constraints
- Digital Signatures: Cryptographic validation of all signature data and participant authorization

The validation process follows a hierarchical approach, beginning with ledger-level consistency checks before proceeding to page-level validation and finally entry-level verification. This structure enables early detection of systematic issues while providing granular identification of specific compromised elements.

3.4.2.2 Validation Results and Reporting

The Warden generates structured integrity reports containing comprehensive validation results for each examined ledger. The reporting framework has a binary classification:

- VALIDATION_SUCCESSFUL: No integrity violations detected across all examined components
- TAMPERING_DETECTED: One or more integrity violations identified with detailed location information

Each integrity report includes the following components:

- ledgerName: Identifier for the examined ledger
- firstPage: Lowest page number in the validation scope
- lastPage: Highest page number in the validation scope
- result: Overall validation outcome classification
- context: Detailed description of findings or detected issues

When tampering is detected, the system provides specific error categorization to facilitate rapid incident response and forensic analysis:

- HASH_DISCREPANCY: Computed hash values do not match stored values
- TIMESTAMP_DISCREPANCY: Temporal ordering violations or invalid timestamp relationships
- SIGNATURE_INVALID: Digital signature verification failures
- MISSING_SIGNATURE: Required signatures absent from entries
- LEDGER_NAME_MISMATCH: Inconsistent ledger identifiers across components
- PAGE_NUMBER_MISMATCH: Invalid page numbering or sequencing
- MERKLE_ROOT_MISMATCH: Merkle tree computation discrepancies
- PREVIOUS_HASH_MISMATCH: Page chaining integrity violations

Table 3.8 demonstrates typical validation error reports with their associated information.

Table 3.8: Validation Error Report Examples

Error Type	Page	Entry ID	Description
HASH_DISCREPANCY	42	MED001	Entry hash mismatch - expected: 'ce4cdd...', actual: 'ce4cde...'
MERKLE_ROOT_MISMATCH	15	null	Merkle root mismatch - expected: '815de2...', actual: '815de3...'
SIGNATURE_INVALID	23	DOC002	Invalid signature from 'CFO Dave'
TIMESTAMP_DISCREPANCY	42	null	Page timestamp not after previous page timestamp

3.4.2.3 Operational Integration

The Warden operates through configurable scheduling mechanisms, enabling organizations to balance validation frequency against system performance requirements.

Integration with existing monitoring infrastructure enables automated alerting when tampering is detected. The structured reporting format facilitates integration with Security Information and Event Management (SIEM) systems, enabling correlation with other security events and automated incident response workflows [59].

For testing and validation purposes, the system includes a companion *tampering service* that introduces controlled integrity violations across all validation categories. This testing capability enables the system to verify the Warden's effectiveness and validate incident response procedures without compromising data integrity.

The tampering service operates through cyclic modification and restoration of ledger components, including:

- Page-level modifications: timestamps, hashes, Merkle roots, and previous hash links
- Entry-level modifications: timestamps, content, and hash values
- Signature-level modifications: signature data and public key information

This testing approach ensures that the Warden can detect all categories of potential tampering while providing businesses with confidence in their integrity monitoring capabilities.

The Warden represents a critical component in the system's security architecture, providing the continuous monitoring necessary to maintain trust in centralized ledger systems. Its integration with backup and recovery procedures enables organizations to maintain data integrity even when faced with tampering attempts.

3.4.3 Providers

The providers are important components for the system, responsible for supplying the cryptographic algorithms required by the ledger. Their design follows the system's core principles

by abstracting the implementations, maintaining consistency through common interfaces, and ensuring that adding new algorithms is simple. This extensibility allows the ledger to evolve without disrupting existing functionalities.

A provider is implemented as an *object* that maintains a registry of available algorithms. An *algorithm* is defined by an *interface*, which specifies the contract that all implementations must follow. Each *implementation* is a `class` that implements the interface and is marked with a dedicated annotation. At runtime, the provider uses reflection to discover all annotated implementations and automatically register them. This removes the need for hardcoded references and guarantees that new algorithms can be introduced with minimal effort. The provider pattern is illustrated in Figure 3.4.

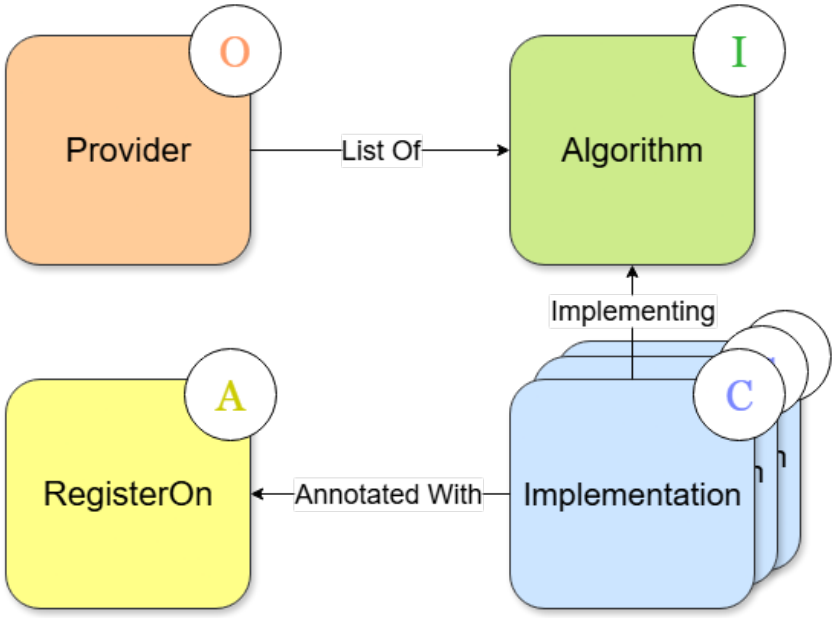


Figure 3.4: *Provider Pattern Diagram*

Every algorithm declares a unique *name*. This identifier is used across all provider functions, such as:

- `isAlgorithmSupported(name)` - Checks if an algorithm exists in the registry.
- `getSupportedAlgorithms()` - Lists all available algorithm names.
- `getDefaultAlgorithm()` - Returns the default algorithm name.

Besides validation, providers also handle conversions that fall under two categories:

- Data conversions: transforming arbitrary data between strings and byte arrays.
- Key and signature conversions: transforming cryptographic object into a hexadecimal string and vice-versa, enabling storage and transmission to be language-independent.

The following subsections describe the specific providers implemented: Hash, Signature, and Crypto. Each follows the same pattern but specializes it for its respective cryptographic purpose.

3.4.3.1 Hash Provider

The Hash Provider manages hashing algorithms used to transform input data into fixed-length digests. Its interface defines a function for hashing strings into byte arrays, and all implementations must provide a unique name. The provider also includes utility functions to represent digests either as hexadecimal strings or as raw byte arrays.

Current implementations include:

- SHA-256
- SHA-512
- SHA3-256

These implementations ensure that the system can support different levels of security and performance requirements while maintaining a consistent interface.

3.4.3.2 Signature Provider

The Signature Provider is responsible for managing asymmetric cryptographic algorithms used in digital signatures. Its interface defines functions for signing data, verifying signatures, generating key pairs, and converting keys between encoded byte arrays, hexadecimal strings, and cryptographic key objects.

The provider exposes different variations of sign and verify functions. These overloads exist solely to offer flexibility in input formats (strings, byte arrays, or hex-encoded values) but internally they all perform the necessary conversions before delegating to the selected algorithm.

In this context, the distinction between conversions is important:

- Data conversions: Transform message content between string and byte array.
- Key and signature conversions: Encode or decode cryptographic keys into hexadecimal strings.

The currently supported implementations are:

- RSA with SHA-256
- Elliptic Curve with SHA256withECDSA, using the `secp256r1` curve

3.4.3.3 Crypto Provider

The Crypto Provider manages symmetric encryption algorithms. Its interface defines functions for encrypting data into an `EncryptedPayload` (containing the encrypted key, initialization vector,

and ciphertext) and for decrypting such payloads back into plaintext. As with the other providers, it also supports conversion utilities for keys, including transformations between key objects, byte arrays, and hexadecimal strings.

The current implementation is AES with a 256-bit key in CBC mode with PKCS5 padding. This ensures confidentiality of data while maintaining compatibility with cryptographic standards.

3.4.4 Public Key Infrastructure

The Public Key Infrastructure (PKI) provides cryptographic identity management and certificate services essential for system security and receipt validation. The PKI implementation focuses on two primary functions: enabling reliable distribution of the system's public key for receipt verification and facilitating secure exchange of user public keys for entry signing operations.

The system employs digital certificates to establish cryptographic trust relationships [28]. Certificates are in a way similar to receipts in their proof mechanism, while receipts demonstrate that the system recognizes a specific entry, certificates prove that a Certificate Authority (CA) recognizes a specific public key for a specific user. This allows for consistency in the system's trust model.

Certificate structures follow a hierarchical chain of trust called *Certificate Chain*, where each certificate is cryptographically signed by its issuing authority. The chain begins with a self-signed certificate, also called *root*, and extends through intermediate certificates until it reaches a end-entity certificate [2]. This hierarchical approach enables scalable trust distribution while maintaining cryptographic verification capabilities.

Figure 3.5 illustrates the Certificate Chain.

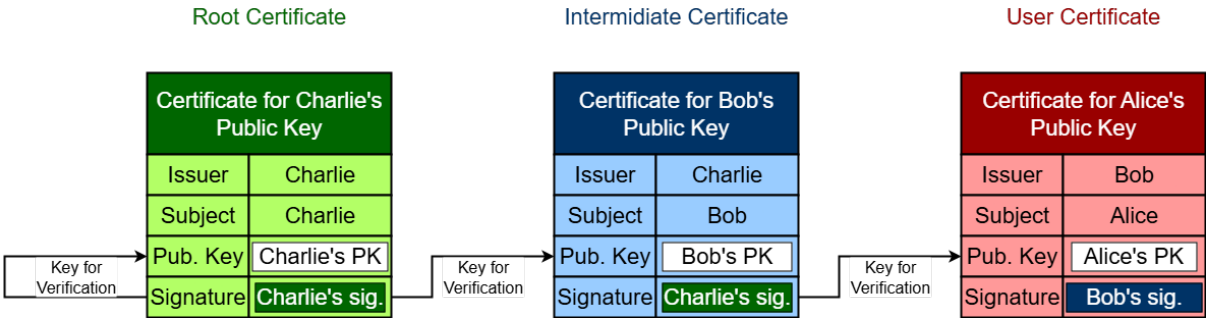


Figure 3.5: *Certificate Chain*

The system implements its own CA for development and testing purposes, maintaining the root certificate and providing certificate generation. However, production deployments should utilize established, widely-trusted CAs to ensure broader interoperability and enhanced security assurance [10]. Commercial CAs provide additional benefits including established trust relationships, comprehensive revocation infrastructure, and regulatory compliance certifications.

The PKI Service manages several important functions:

- System Key Pair Management: Generates and secures the system's cryptographic key pair used for signing receipts and system-generated ledger entries

- Root Certificate Creation: Generates and maintains the self-signed root certificate establishing the system's CA
- User Certificate Operations: Creates, validates, and retries user certificates for ledger entry signing and verification

The implementation utilizes X.509 certificate format [29], providing standardized certificate structure with the following essential components:

- Subject Information: Identity details of the certificate holder
- Public Key: Cryptographic public key for signature verification
- Issuer Information: Certificate authority identification
- Validity Period: Temporal constraints defining certificate lifetime
- Digital Signature: Cryptographic signature from the issuing authority

Certificate distribution requires public accessibility to support independent verification scenarios. The system provides public endpoints for certificate retrieval, enabling external parties to validate receipts and entry signatures. This approach supports audit requirements and regulatory compliance by ensuring verification capabilities remain available even in dispute scenarios.

3.4.5 Authentication

The system implements a basic authentication structure to facilitate integration with existing organizational infrastructure. Many organizations operate complex authentication systems and being able to incorporate the ledger with minimal changes is a design objective of the project, so the only requirement involves providing consistent user identification, *user ids*, across all ledger operations.

3.4.5.1 Authentication Service

The authentication foundation relies on two primary data models supporting user identity and session management:

The user model provides comprehensive identity management:

- id: Unique user identifier serving as the primary key for all ledger operations
- email: Primary identification and communication channel for system notifications
- fullname: Personal identification supporting audit trail clarity
- hashedPassword: Cryptographically protected authentication credential using standard hashing algorithms

The token model manages session persistence and security:

- value: Cryptographic token generated using secure random algorithms for session identification
- expiresAt: Temporal boundary preventing indefinite token usage and limiting unauthorized access windows

The Authentication Service provides some identity management capabilities:

- User Registration: Secure account creation with mandatory email verification preventing unauthorized account establishment
- Credential Validation: Login processing with cryptographic token generation for authenticated session management
- Session Management: Token-based authentication validation ensuring only authorized users access system resources
- User Information Retrieval: Secure user data access supporting other system services requiring identity verification
- Session Termination: Logout functionality with token invalidation preventing session persistence

The service architecture enables organizations to replace authentication backend while maintaining consistent interfaces throughout the system. This abstraction supports integration with existing directory services, single sign-on systems, or specialized authentication providers without requiring modifications to ledger or workflow components.

3.4.5.2 Two Factor Authentication Service

The 2FA Service provides enhanced security for critical operations through additional verification layers. The service operates with ephemeral verification codes that are generated on-demand but not persisted in permanent storage due to their short expiration periods.

The 2FA Service implements the following security mechanisms:

- Code Generation: Six-digit verification codes generated for specific email addresses using cryptographically secure random number generation
- Code Validation: Secure verification with timeout handling preventing replay attacks and brute-force attempts
- Email Integration: Automated code delivery through organizational email systems ensuring delivery through trusted communication channels
- Audit Logging: Comprehensive logging of all authentication attempts supporting security monitoring and incident investigation

The service operates as an internal component without direct API exposure, preventing external attack vectors while enabling other services to request enhanced authentication when required. This architecture supports selective application of 2FA based on operation criticality or organizational security policies.

The current implementation focuses on email-based verification, but the modular design supports extension to SMS delivery, authenticator applications, or hardware token integration. Organizations can customize verification methods based on their security requirements and existing infrastructure.

2FA integration remains optional throughout the system, allowing organizations to apply enhanced security selectively based on operation sensitivity or user role requirements.

3.4.6 Workflow Services

Workflow Services contain domain-specific business logic and application functionality, representing the most flexible component within the system architecture. This modular approach enables the ledger and security infrastructure to remain consistent across implementations while supporting diverse organizational requirements through specialized workflow components.

The workflow layer serves as the primary integration point between organizational business processes and the underlying ledger infrastructure. Organizations can implement custom workflows tailored to their specific operational requirements while leveraging shared security and audit capabilities.

For enhanced consistency and maintainability, workflows should utilize structured content classes when interacting with the ledger service at scale. Rather than using simple string content, organizations can implement domain objects that provide consistent serialization and deserialization. For example, a contract management workflow might implement a Contract class with standardized fields for parties, terms, dates, and conditions, ensuring consistent representation across all contract related ledger entries.

3.4.6.1 File Management Service

The File Management Service demonstrates a practical workflow implementation, providing secure document management with comprehensive audit trail integration. This service illustrates how domain-specific functionality can leverage core ledger infrastructure while maintaining operational independence.

The service operates through a FileMetadata model supporting complete document lifecycle management:

- id: Unique file identifier for system-wide file tracking
- originalFileName: User-defined file names preserving original document identification
- actualFileName: System-generated unique names preventing filename conflicts and enhancing storage security

- filePath: Physical storage location within the system's file repository
- fileSize: Complete file size information supporting storage management and transfer optimization
- contentType: Specification enabling proper file handling and security validation
- uploadedAt: Creation timestamp for audit trails and document lifecycle tracking
- uploaderId: Primary user responsible for file upload and ownership
- senders: Additional users with sending privileges for collaborative file management
- receivers: Users granted access to file content supporting controlled information sharing
- ledgerEntries: References to associated ledger entries providing immutable audit trails
- wasDeleted: Deletion status flag supporting soft deletion and audit trail preservation

The service implements participant-based access control through clearly defined roles: uploaders maintain primary ownership and deletion rights, while senders and recipients can only download the files.

Core File Operations provide comprehensive document management:

- Upload: Upload with automatic metadata generation, participant validation, and filename collision detection
- Download: Permission-based file retrieval with access logging
- Deletion: File removal available only to uploaders, with cryptographic audit trails
- File Listing: Permission-filtered directory services displaying only files accessible to requesting users

The service creates comprehensive audit trails through multiple ledger integration points. File operations are logged in a dedicated files ledger, while user operations generate entries in individual user ledgers. This dual logging approach supports both the system's audit requirements and user's activity tracking.

Organizations can implement specialized workflows tailored to their operational requirements:

- Medical Records Management: Patient data handling with Health Insurance Portability and Accountability Act (HIPAA) compliance, clinical workflow integration, and healthcare provider collaboration
- Email Archive Management: Communication retention with legal hold capabilities, keyword indexing, and regulatory compliance support
- Financial Transaction Processing: Accounting integration with multi-currency support, regulatory reporting, and audit trail requirements

- Legal Document Management: Contract lifecycle management with version control, approval workflows, and court admissibility features

Each workflow implementation can utilize shared ledger and security infrastructure while providing domain-specific business logic. This approach ensures consistency in security guarantees and audit capabilities across diverse organizational use cases while enabling specialized functionality appropriate to specific business requirements.

The modular workflow design supports incremental deployment, allowing organizations to implement workflows progressively based on operational priorities while maintaining system coherence and security standards throughout the expansion process.

3.5 Client Implementation

The client application created for this thesis was designed to test the server endpoints and system functionality, whilst being able to visualize its data. The architecture follows production patterns that organizations can reference when developing or integrating the system with their own interfaces. When in production, companies should maintain similar public accessibility features to enable independent ledger verification, supporting the transparency requirements necessary for audit and compliance scenarios.

The client implementation follows the MVVM architectural pattern [41], making a clear separation of concerns between data representation, user interface components, and business logic. This separation allows for maintainable code organization while supporting the reactive UI of modern web applications.

The client architecture divides functionality across four primary domains mirroring the server organization: Authentication, File Management, Ledger Operations, and Public Key Infrastructure. This structure simplifies development while maintaining the coverage for all system capabilities.

3.5.1 Models and Data Transfer Objects

The data models correspond directly to the server-side entities, ensuring consistency in data representation across the application. DTOs facilitate the transformation between server responses and client data structures, handling format conversions and providing type safety, which was a big reason to use TypeScript.

The data model layer has several categories of data structures:

- Authentication Models - Manage user identity and session information, including user profiles, authentication tokens, and session state, providing the identity context required for all authenticated operations.
- File Management Models - Represent document metadata and file system interactions. The FileMetadata model extends the server data with specific properties such as file type classifications and download URLs.

- Ledger Models - Represent ledger components including ledgers, pages, entries, signatures, and participants. These models support both summary views for efficient browsing and detailed representations for analysis, enabling users to navigate the ledger structures intuitively.
- PKI Models - Handle certificate data and cryptographic key information. Certificate models include both raw certificate data and parsed representations with human-readable atributes such as: validity periods, subject information, and algorithm specifications.

The DTO transformations are done in the *Fetcher Services* and convert server responses into client models while handling data validation, format normalization, and error scenarios. This abstraction enables the client to have less dependency on the server when it comes to preventable exceptions.

3.5.2 ViewModels

ViewModels function as the coordination center for the application. It connects UI components and backend services, managing application state, user interactions, and data synchronization. Each ViewModel encapsulates the business logic specific to its domain while providing reactive state management that automatically update UI when data changes.

The ViewModel architecture employs React hooks to provide stateful logic that components can consume through declarative interfaces. This approach enables component reusability while centralizing complex state management logic in isolated units.

For this implementation, the ViewModels have a lesser role considering its applications are simple. They heavily utilize the logic already present in the server and only do small validations and checks to prevent faulty requests from being sent. Most of their importance is to gather the information from the application server and make it visible for the user.

3.5.2.1 Ledger ViewModel

The Ledger ViewModel provides ledger navigation and analysis, enabling users to examine individual entries, and understand cryptographic relationships. The ViewModel supports standard visualization and forensic analysis of ledger contents, being able to validate hashes and signatures.

3.5.2.2 PKI ViewModel

The PKI ViewModel manages cryptographic key operations, certificate lifecycle management, and verification workflows. The ViewModel coordinates key generation, certificate requests, and validation operations while presenting complex cryptographic information in accessible formats.

Key pair management includes generation, secure storage, and validation operations. All certificates and keys are saved with the *local storage services* to maintain persistence while providing clear information about key status and validity. But the system also allows for a direct download

of the key pair to a Privacy Enhanced Mail (PEM) format to allow for recovery of the key pair in case the local storage is erased.

Certificate operations encompass creation requests, validation workflows, and certificate chain verification, with the ViewModel presenting certificate information including validity periods, issuer details, and technical specifications in formats appropriate for both technical and business users.

System certificate management enables users to access and verify the system's cryptographic credentials, supporting independent verification of receipts and system generated signatures. The ViewModel ensures certificate information remains publicly accessible while maintaining security boundaries for private key operations.

3.5.2.3 Authentication ViewModel

The Authentication ViewModel manages user identity, session state, and authentication workflows throughout the application lifecycle. It provides comprehensive authentication including registration, login, 2FA, and session management while maintaining security best practices.

The authentication flow incorporates password hashing on the client side using the same cryptographic providers found in the server, ensuring passwords remain protected during transmission. Session management includes automatic token validation and renewal for a simplified user experience.

3.5.2.4 Files ViewModel

The Files ViewModel handles document management operations including upload, download, deletion, and metadata retrieval. For file upload is possible to use drag-and-drop or a regular file system search, while download and deletion are also one button operations.

The ViewModel coordinates with the authentication system to ensure only authorized users can perform file operations while providing clear feedback about access permissions.

3.5.3 Fetcher Services

The Fetcher Services provide communication between the client and server endpoints, handling HTTP requests, response parsing, authentication integration, and error management. This implementation uses a wrapper class for the fetch API, defining request types, token management, a base URL, and a single request function that returns an API Response. The wrapper made coordination between the Fetcher Services and server Controllers straightforward and consistent.

Each controller has a separate service:

- Authentication Service: Manages registration, login, 2FA verification, token validation, and logout operations
- Files Service: Handles uploads, downloads, metadata retrieval, and file management through multipart form data and browser APIs

- Ledger Service: Provides ledger data access including listings, page retrieval, entry examination, and metadata management with hierarchical data retrieval

- PKI Service: Manages certificate operations including creation requests, validation workflows, user certificate retrieval, and system certificate access

3.5.4 Local Storage Services

Local Storage Services provide client-side data persistence using browser storage, which can be deleted by users or browser cleaning operations. The authentication storage saves user session objects including profiles, tokens, and session state for automatic login persistence.

The PKI storage manages cryptographic key pairs and certificates for improved performance and offline functionality. While keys and certificates are stored locally, it is highly recommended to download everything to separate PEM files for backup and recovery purposes, as mentioned before. This ensures key pair recovery is possible if local storage is cleared.

3.5.5 Views and Components

The View layer implements the UI components that present system functionality to users through intuitive, responsive interfaces. The component architecture uses React's declarative programming model with hooks for state management, enabling reusable interface components.

Component design is inspired by other applications and tries to be intuitive and functional. The interface supports both casual usage and detailed analysis workflows to mimic a possible company UI with the separation of the basic utilities, like file management, and the complex ones, like the ledger components and PKI.

The application follows a consistent navigation pattern where users begin at the home page and proceed through authentication before accessing the main functional areas. Common UI components are reused throughout the application, including navigation bars, form elements, action buttons, and detail cards, ensuring visual consistency and reducing development complexity.

3.5.5.1 Authentication Components

Authentication components provide user registration, login, 2FA, and the navigation bar. The components use forms validation and real-time feedback while maintaining security best practices for credential handling, like blocking the password to protect against shoulder surfing.

When registering a new account, users fill the form shown in Figure 3.6, which includes fields for full name, email address, password, and password confirmation. The registration process enforces password strength requirements and performs email validation before account creation.

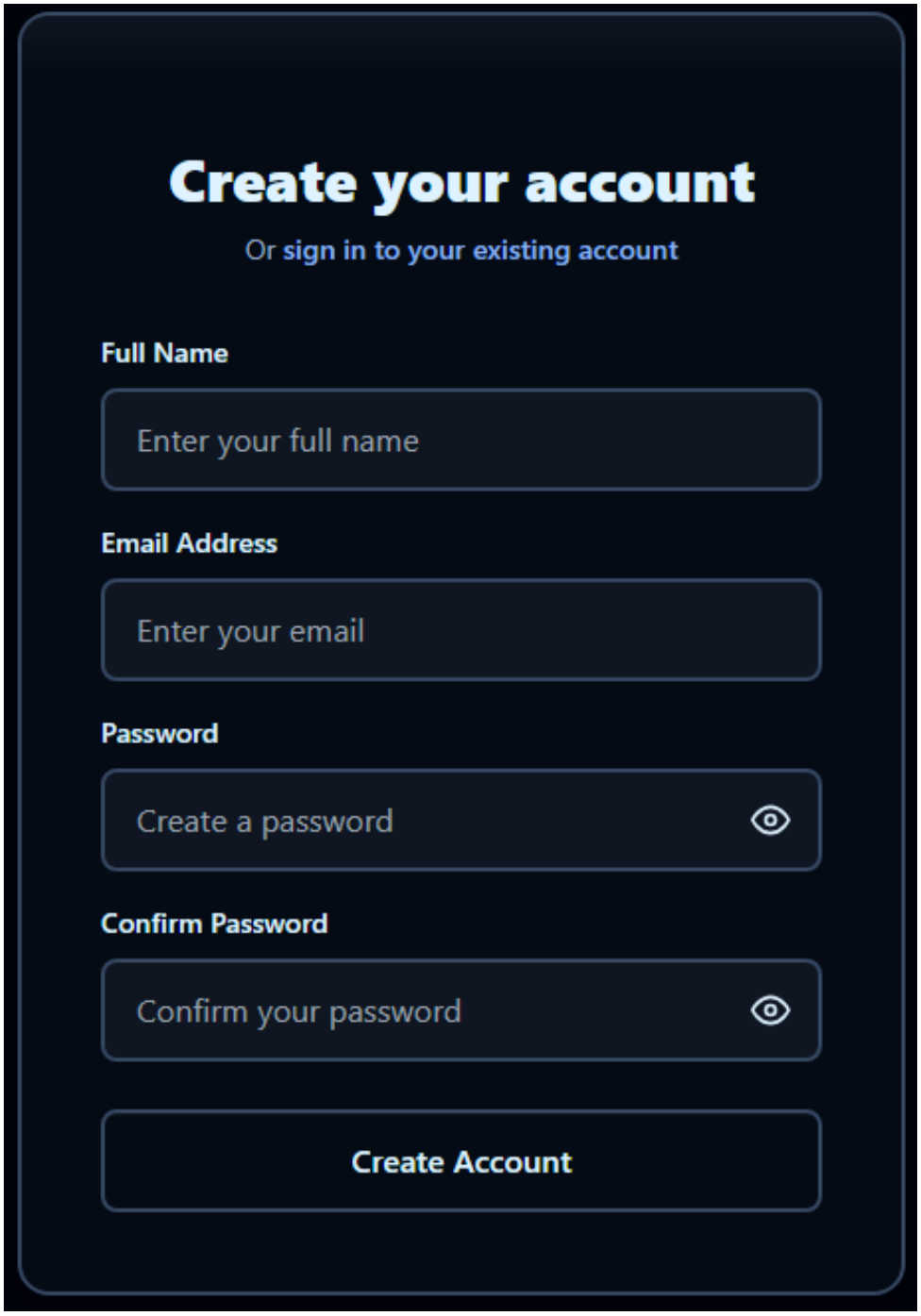


Figure 3.6: *User Registration Form*

Login utilizes the interface presented in Figure 3.7, only containing the email and password fields.

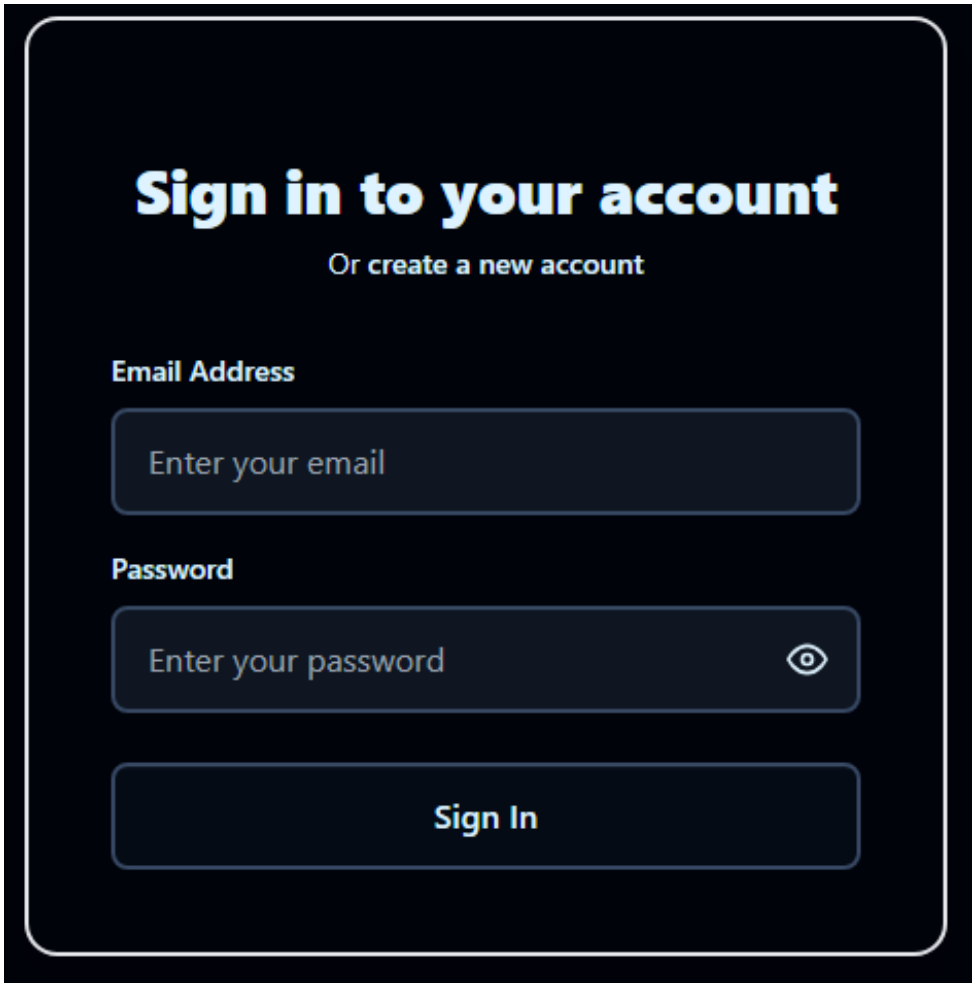


Figure 3.7: *User Login Form*

Upon successful authentication, either by registering or login, users receive a verification code via email, as demonstrated in Figure 3.8, which must be entered to complete the process.

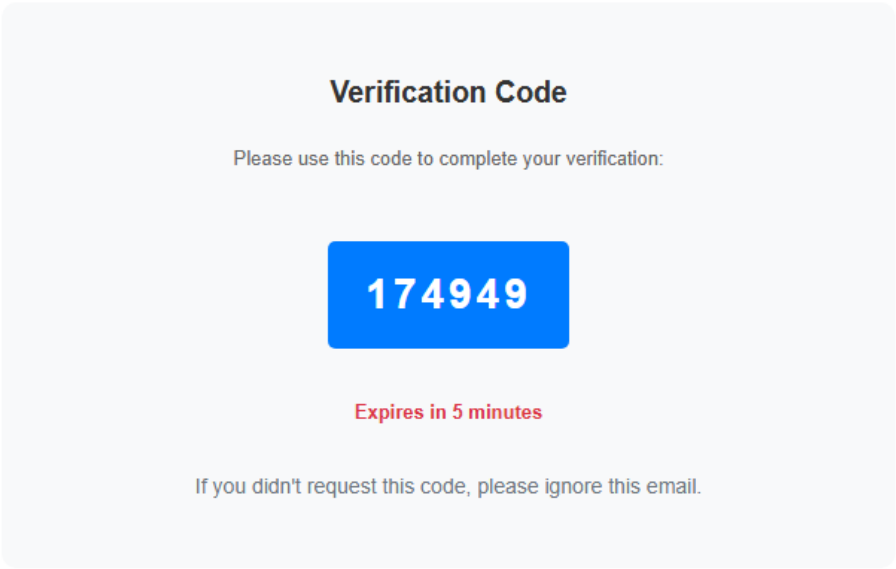
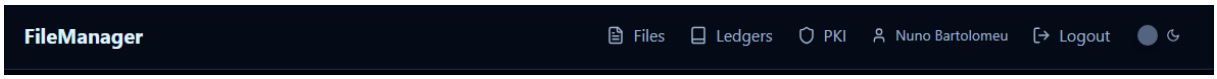


Figure 3.8: *2FA Email Verification*

The navigation bar differs based on authentication status. Unauthenticated users see the navigation bar shown in Figure 3.9a, providing access to home, login, and registration pages. Once authenticated, users access the expanded navigation shown in Figure 3.9b, which includes file management, ledger operations, PKI functions, user profile, and logout capabilities.

(a) *Navigation Bar for Unauthenticated Users*



(b) *Navigation Bar for Authenticated Users*

Figure 3.9: *Navigation Bars for the client UI*

3.5.5.2 File Management Components

File management components enable document upload, browsing, metadata viewing, and file operations through intuitive drag-and-drop system and file listings.

The main file management interface, shown in Figure 3.10, provides drag-and-drop upload functionality and the file listing. Users can upload files by dragging them into the designated area or clicking to browse their computer's file system.

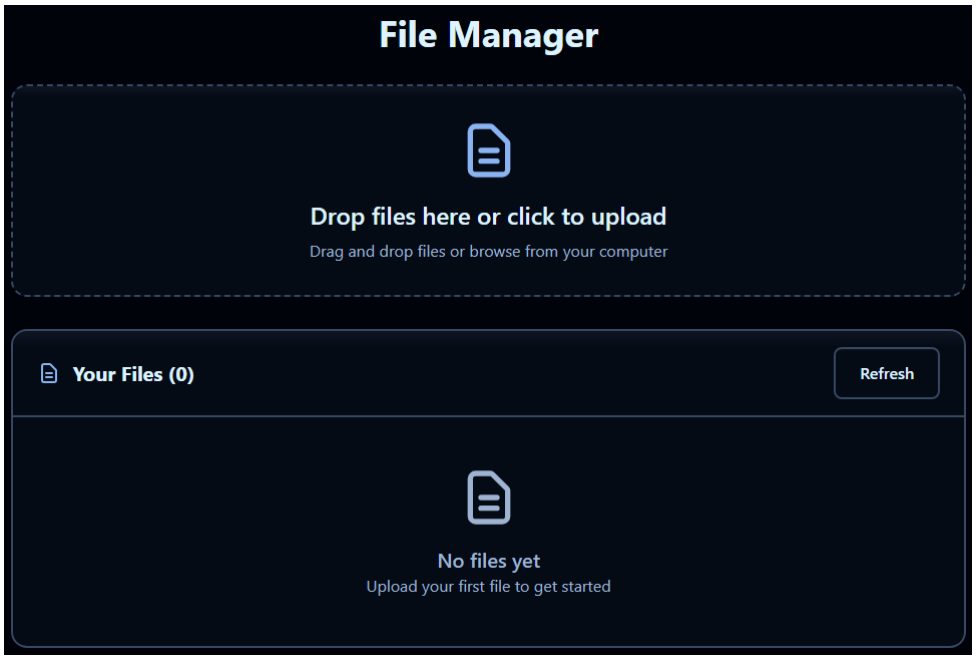


Figure 3.10: *File Management Interface*

Individual file details are presented through comprehensive metadata displays, as demonstrated in Figure 3.11. The interface provides file information including size, type, and content type, ownership details showing uploader identification, participant management for collaborative file sharing, and direct access to associated ledger entries for audit trail.

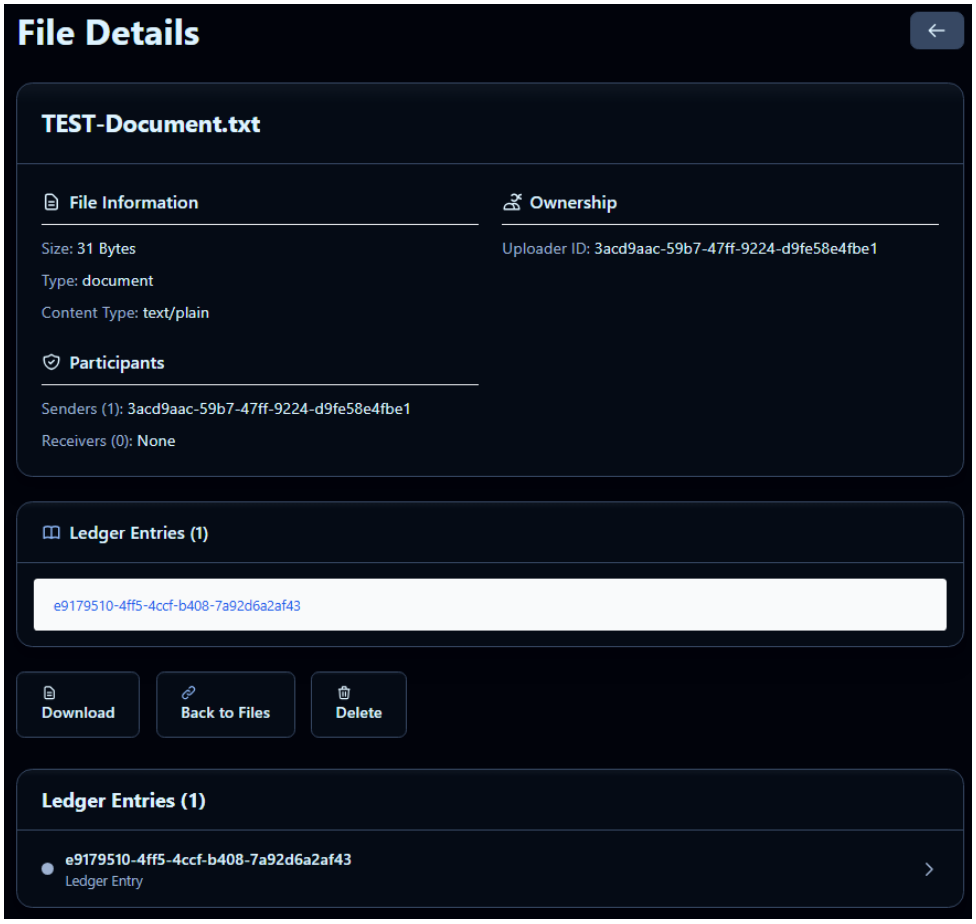


Figure 3.11: *File Details View*

File operations include download, navigation back to the file listing, and deletion (available only to file uploaders), with clear action buttons for access control management.

3.5.5.3 Ledger Analysis Components

Ledger analysis components display information about the ledger system showing, detailed entries, pages and cryptographic verification.

The ledger overview interface, illustrated in Figure 3.12, presents detailed ledger information including configuration parameters, statistical summaries and organized listings of unverified and verified entries and completed pages.

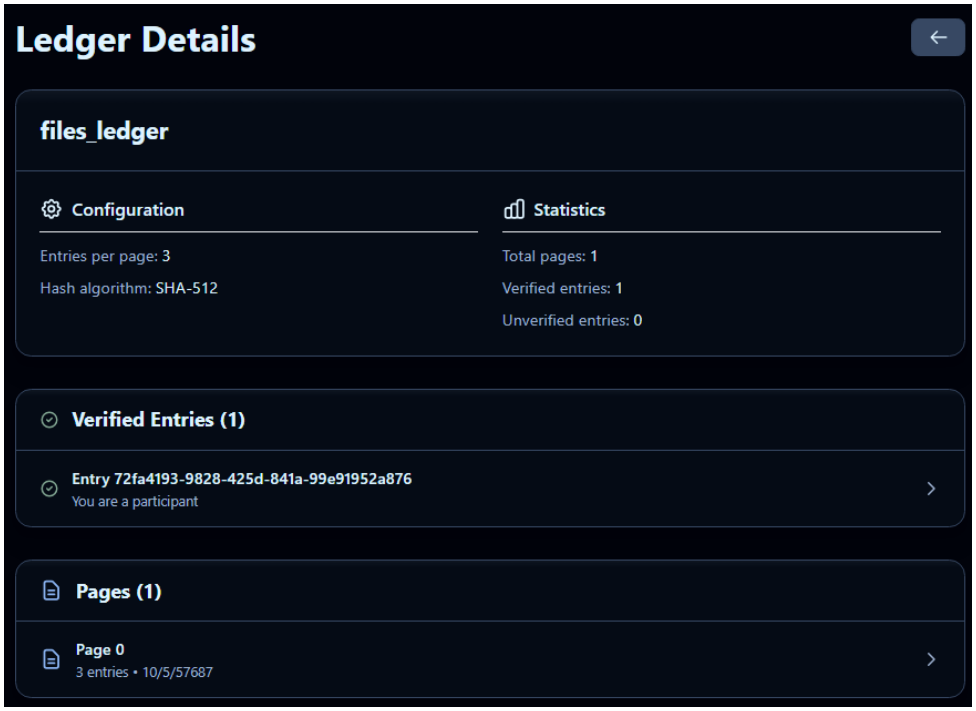


Figure 3.12: *Ledger Details and Overview*

Page analysis is provided through a detailed view as shown in Figure 3.13, displaying page information including number, entry count, and timestamp, cryptographic hashes showing both page hash and Merkle root for integrity verification. It also contains a list with all entries in the page with participant indicators and direct navigation to individual entries.

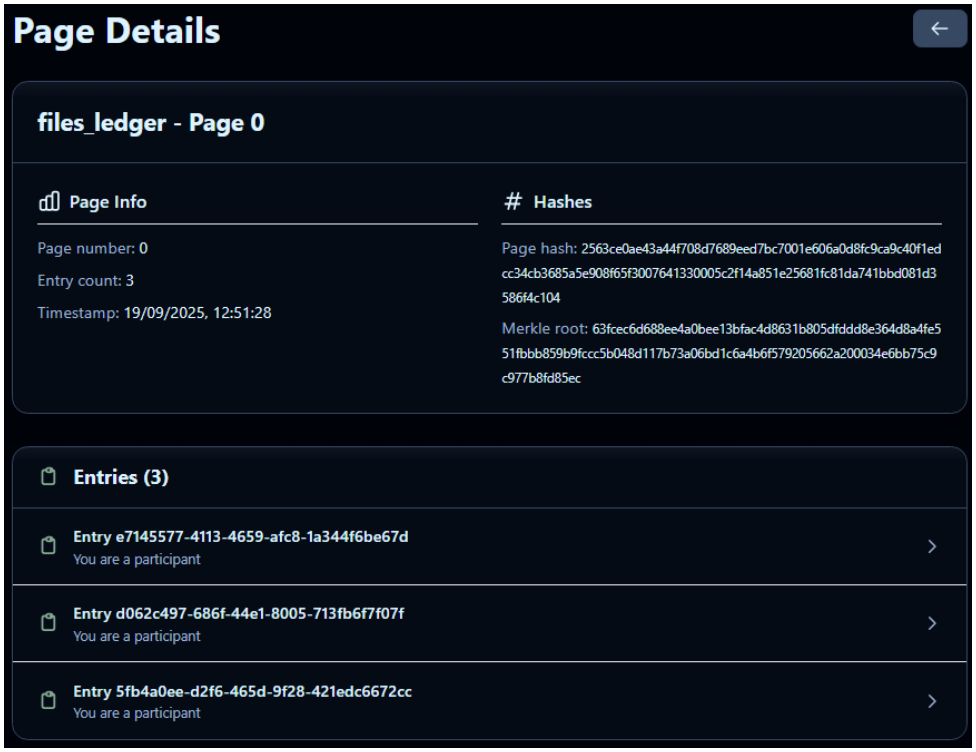


Figure 3.13: *Page Details and Entry Listings*

The entry interfaces, demonstrated in Figure 3.14, presents detailed information like the previous pages. It includes basic details such as ledger name, page number, and timestamps, security information displaying hash values and signature counts. It also contains the actual entry data and the participant listing, being both senders and recipients related to the entry. Lastly it has keyword management supporting entry categorization and searchability.

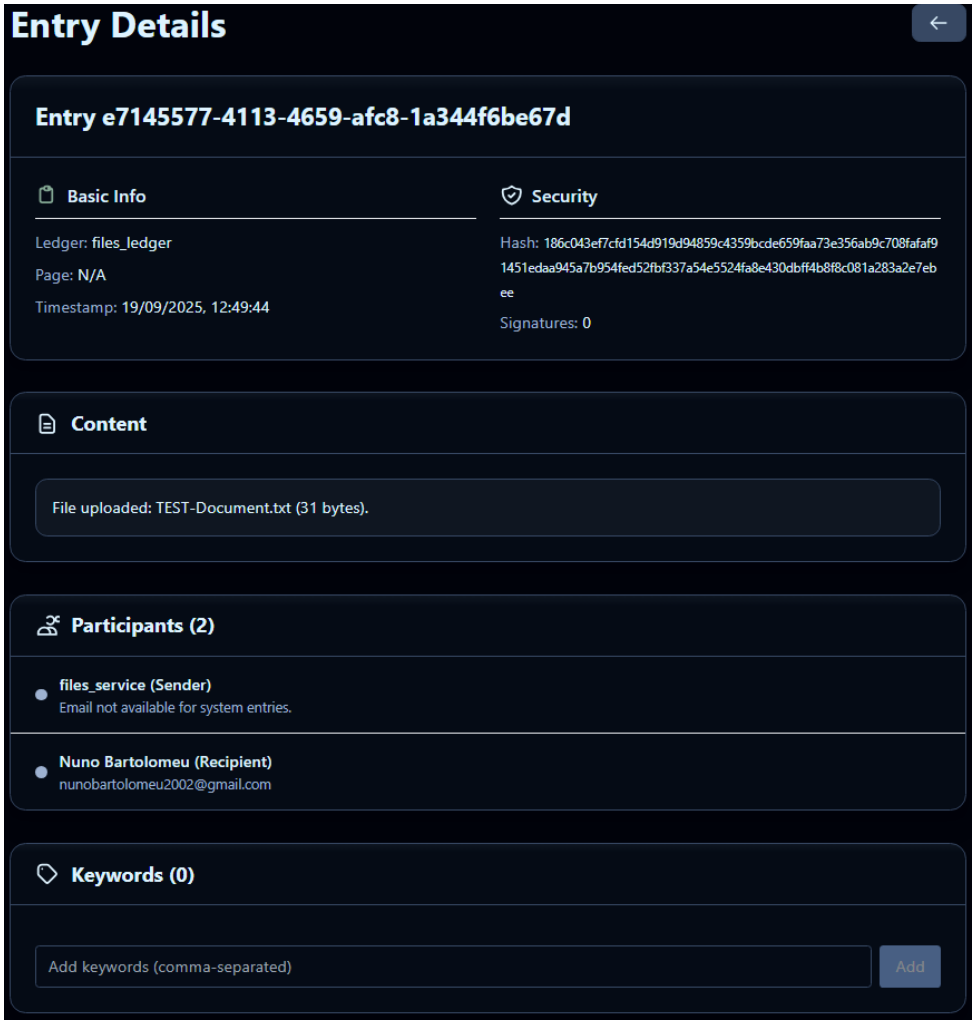


Figure 3.14: *Entry Details and Cryptographic Information*

3.5.5.4 PKI Management Components

PKI management components provide key and certificate visualization for users requiring cryptographic operations. The components present complex cryptographic information in user friendly formats while maintaining access to technical details.

Key management interfaces enable key pair generation, certificate requests, and validation operations with clear status information and progress tracking. Certificate viewing components present certificate details including validity, issuer information, and technical specifications in accessible formats for all users.

Figure 3.15 presents the implemented PKI management page.

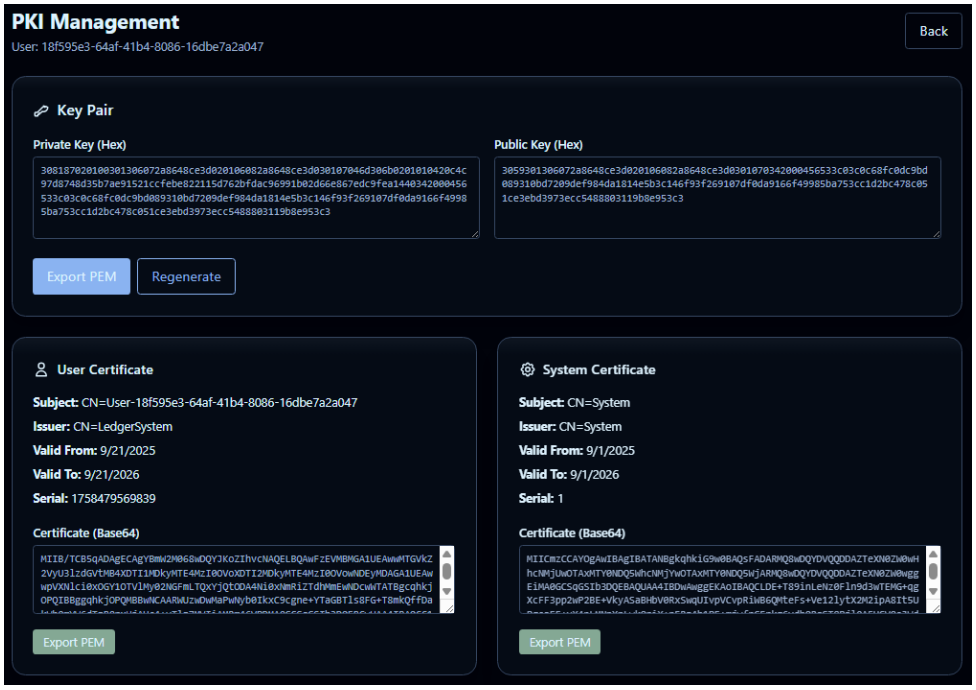


Figure 3.15: *PKI Management Interface*

3.6 Data Storage

The data storage implementation represents the persistence layer of the system architecture, utilizing a hybrid approach that combines relational database management with file system storage. The design aims for simplicity while maintaining flexibility for diverse business requirements.

The SQL database is used for structured ledger data and user management, and the file system serving as a pseudo-NoSQL repository is used for document storage and cryptographic artifacts. This approach demonstrates practical implementation patterns while avoiding unnecessary complexity in the storage architecture.

3.6.1 Relational Database

The SQL database schema, illustrated in Figure 3.16, implements the core ledger components alongside supporting tables for user management and file metadata. The schema design prioritizes clarity and direct representation of the domain model over complex optimization strategies.

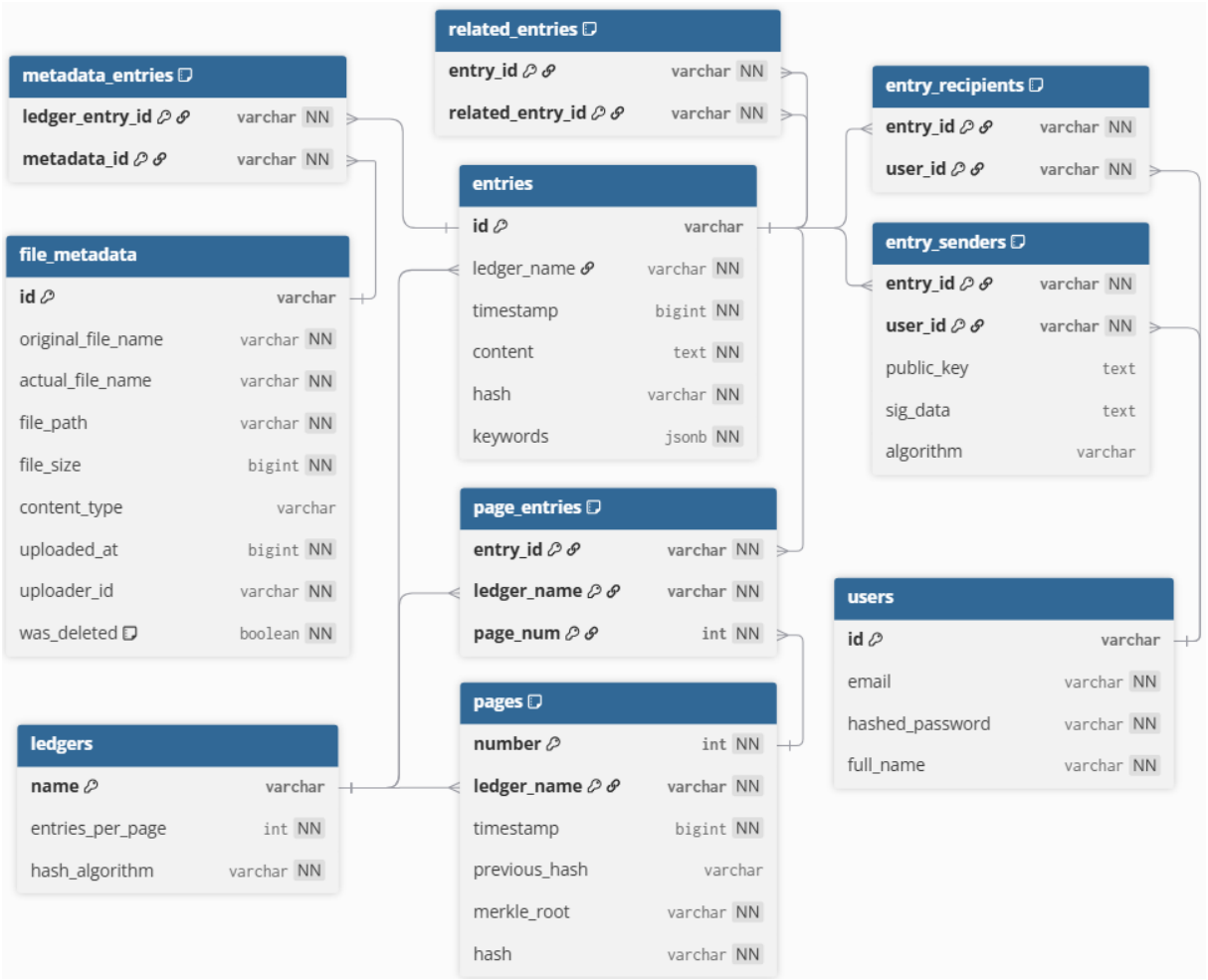


Figure 3.16: *SQL Database Schema*

The database structure includes the following primary tables:

- Core Ledger Tables - The *ledgers*, *pages*, and *entries* tables implement the fundamental ledger hierarchy, while *page_entries* provides the many-to-many relationship between pages and their contained entries. Supporting tables include *entry_senders*, *entry_recipients*, *related_entries*, and *metadata_entries* for the normalization of the schema.

- User Management - The *users* table maintains authentication credentials and user profile information necessary for system access and audit trails.

- File Metadata - The *file_metadata* table tracks uploaded documents with references to associated ledger entries, enabling complete audit trail integration for file operations.

The schema employs normalized table structures to maintain data consistency and eliminate redundancy. However, normalization is not strictly required for system operation. Organizations may implement denormalized schemas, alternative data formats such as JSON documents, or entirely different storage mechanisms based on their specific performance and scalability requirements.

The relational approach serves as a reference implementation demonstrating how ledger concepts can be represented in traditional database systems. The flexibility of the system architecture enables organizations to substitute alternative persistence mechanisms including NoSQL databases, distributed storage systems, or specialized ledger databases without requiring modifications to the business logic layer.

3.6.2 File Repository

The file system component provides document storage and cryptographic artifact management through a straightforward directory structure. This approach eliminates dependencies on external storage services while maintaining compatibility with distributed file systems for production deployments.

File storage utilizes two primary directory structures. The */uploads* directory contains user-uploaded documents organized by user identifier, creating isolated storage spaces that prevent filename conflicts and enhance access control management. The PKI directory maintains cryptographic certificates and key materials in PEM format, supporting both the system and user certificates.

The file system implementation supports migration to distributed storage solutions including cloud object storage, IPFS, or network-attached storage systems. Organizations requiring enhanced scalability, redundancy, or geographical distribution can integrate alternative storage backends without modifying the application logic.

3.7 System Deployment

The system deployment utilizes containerized architecture through Docker, enabling consistent deployment across different environments while maintaining isolation between system components. This approach simplifies installation procedures, manages dependencies automatically, and provides scalable infrastructure suitable for both development and production environments.

The deployment architecture consists of three primary containers orchestrated through Docker Compose: a PostgreSQL database container, a Kotlin-based server application container, and a Next.js client application container. This multi-container approach enables independent scaling, maintenance, and updates for each system component.

3.7.1 Database Container

The PostgreSQL database container provides persistent data storage for ledger information, user management, and file metadata. The container utilizes the official PostgreSQL 16 image, ensuring stability and compatibility with the system requirements.

Database configuration includes:

- Image: PostgreSQL 16 official Docker image

- Container Name: ledger_db
- Port Mapping: 5432:5432 (host:container)
- Environment Variables: Database name, username, and password configuration
- Volume Mounting: Persistent storage through named volumes ensuring data persistence across container restarts

The database container initializes automatically upon first deployment, creating the necessary database schema and establishing the connection endpoints required by the server application.

3.7.2 Server Container

The server container hosts the Kotlin-based application server implementing the ledger domain, security services, and workflow components. The container builds from the application source code using a multi-stage Docker build process for optimization.

Server container specifications:

- Base Image: Eclipse Temurin 21 JDK for Java runtime environment
- Build Process: Multi-stage Maven compilation with dependency caching
- Container Name: ledger_server
- Port Mapping: 8080:8080 (host:container)
- Dependencies: PostgreSQL database container
- Volume Mounts: Persistent storage for PKI certificates, user files, and system artifacts

The multi-stage build process optimizes container size by separating build dependencies from runtime requirements. The first stage utilizes Maven for dependency resolution and application compilation, while the second stage creates a lean runtime container containing only the compiled application and necessary runtime environment.

Volume mounting provides persistent storage for:

- system_pki: System-generated certificates and cryptographic keys
- users_pki: User certificates and public key information
- uploads: File repository for document management

3.7.3 Client Container

The client container serves the Next.js-based web application providing the user interface for system interaction. Similar to the server container, the client utilizes multi-stage building for optimization.

Client container configuration:

- Base Image: Node.js 20 Alpine for lightweight runtime
- Build Process: Multi-stage npm compilation with dependency caching
- Container Name: ledger_client
- Port Mapping: 3000:3000 (host:container)
- Dependencies: Server application container

The Alpine-based Node.js image provides a minimal runtime environment, reducing container size while maintaining full compatibility with the Next.js application requirements.

3.7.4 Docker Compose Orchestration

Docker Compose manages container orchestration, networking, and volume management through declarative configuration. The composition defines service dependencies, ensuring containers start in the correct sequence: database first, followed by the server, and finally the client application.

Key orchestration features include:

- Service Dependencies: Automated container startup sequencing
- Network Configuration: Internal container networking with external port exposure
- Volume Management: Named volumes for persistent data storage
- Environment Variable Management: Centralized configuration for database connections and application settings

The orchestration configuration enables single-command deployment through `docker-compose up`, automatically building containers, initializing the database, and establishing service connectivity.

3.7.5 Deployment Process

System deployment follows a streamlined process suitable for both development and production environments:

1. Repository Preparation: Ensure Docker and Docker Compose installation on the target system
2. Configuration Review: Verify environment variables and volume mount paths match system requirements
3. Container Build and Start: Execute `docker-compose up -build` to build containers and initialize services

4. Database Initialization: Allow PostgreSQL container to complete initial setup and schema creation
5. Application Verification: Access the client application through `localhost:3000` to confirm successful deployment

The containerized approach eliminates host system dependencies, ensuring consistent deployment across different operating systems and infrastructure configurations. This deployment strategy supports both local development environments and production server deployments with minimal configuration changes.

3.7.6 Production Considerations

While the provided Docker configuration serves development and testing purposes effectively, production deployments should incorporate additional security and performance optimizations:

- Security: Implementation of proper secrets management, network security policies, and access control mechanisms
- Scalability: Container orchestration through Kubernetes or Docker Swarm for multi-node deployments
- Monitoring: Integration with logging and monitoring solutions for operational visibility
- Backup: Automated backup procedures for database volumes and file repositories
- SSL/TLS: Implementation of encrypted communication channels through reverse proxy configuration

The modular container architecture facilitates these enhancements without requiring modifications to the core application components, supporting smooth transition from development to production environments.

CHAPTER 4

RESULTS AND DISCUSSION

This chapter presents the experimental evaluation of the centralized ledger system, including testing methodologies, performance analysis, and security validation. The evaluation encompasses cryptographic provider testing, proof-of-concept demonstrations, security attack analysis, and comprehensive stress testing to validate system capabilities and identify performance characteristics.

Section 4.1 describes the testing environment and hardware specifications. Section 4.2 presents the evaluation of cryptographic providers. Section 4.3 demonstrates core ledger functionality through unit testing and validation. Section 4.4 analyzes potential attack vectors and mitigation strategies. Section 4.5 provides comprehensive performance evaluation under various load conditions.

4.1 Testing Environment

All testing procedures were conducted on a standardized hardware configuration to ensure consistent and reproducible results. The testing environment specifications provide the baseline for understanding performance measurements and system limitations observed during the evaluation process.

4.1.1 Hardware Specifications

The testing system utilizes the following hardware configuration:

- Processor: Intel Core i7-8550U CPU @ 1.80GHz
- Memory: 12GB DDR4 RAM
- Storage: Solid State Drive (SSD) with sufficient capacity for test data generation
- Graphics: 2GB dedicated graphics memory
- Operating System: Windows 11 Home

4.1.2 Software Environment

The testing environment includes:

- Java Runtime: Eclipse Temurin 21 JDK
- Database: PostgreSQL 16 running in Docker container
- Application Server: Spring Boot 3.x with Kotlin
- Testing Framework: JUnit 5 for unit and integration testing
- Containerization: Docker Desktop for Windows

This software stack mirrors the production deployment environment, ensuring testing results accurately reflect real-world performance characteristics.

4.2 Cryptographic Provider Testing

The cryptographic providers are the foundation of system security. Testing ensures correct implementation, protection against misuse, and reliable behavior. Each provider was validated for algorithm correctness, key management, and resistance to common mistakes (e.g., weak hashes, incorrect signature verification, or broken encryption). The values for the hash, keys and signatures are only partially shown due to their size that can be hundreds or thousands of characters.

4.2.1 Hash Provider Testing

The Hash Provider implements three standard algorithms: SHA-256, SHA-512, and SHA3-256. The goal of testing is to confirm:

- Algorithms are correctly registered and available.
- Deterministic outputs: same input produces the same output.
- Correctness against known test vectors.
- Proper hex conversion for storage and retrieval.

Because the *hash* is a one-way function it's not possible to get to the input with the output, unlike the other providers. To properly test the implementations a online tool was used to acquire the expected value before the test to compare it to the algorithms result.

All algorithms were tested against the input `"Hello, World!"`. Results are shown in Table 4.1.

Table 4.1: Hash Provider Algorithm Validation Results

Algorithm	Output Size	Resulting Hash
SHA-256	32 bytes	`dffd6021bb2bd5b0af676290809ec3a53191dd...`
SHA-512	64 bytes	`374d794a95cdcfd8b35993185fef9ba368f160...`
SHA3-256	32 bytes	`1af17a664e3fa8e419b8ba05c2a173169df761...`

All outputs matched the expected vectors, confirming correctness.

4.2.2 Signature Provider Testing

The Signature Provider was tested for RSA and ECDSA. The focus was:

- Key pair generation produces valid related keys.
- Signatures created with a private key verify with the matching public key.
- Verification fails with non-matching keys.

The verification process for the signature algorithms is direct and follows the following steps:

1. Generate a key pair
2. Sign `"Hello World!"` using the private key
3. Verify the signature using the public key

The following shows a full RSA test case:

```
Algorithm: RSA
Private key: 308204be020100300d06092a864886f70d010101050004...177cc
Public key:  30820122300d06092a864886f70d01010105000382010f...010001
Signature:   4e0700cd9e943cde0675d111f862c5e06a59c4e869ecb4...4b648
Verification: true
Different key verification: false
```

ECDSA test case:

```
Algorithm: ECDSA
Private key: 3041020100301306072a8648ce3d020106082a8648ce3d...15546
Public key:  3059301306072a8648ce3d020106082a8648ce3d030107...11326
Signature:   3046022100896d224122b346095290b7d8c80b6e28d94c...b52fa
Verification: true
Different key verification: false
```

4.2.3 Crypto Provider Testing

The Crypto Provider implements AES-256 in CBC mode with PKCS5 padding. Testing confirmed:

- Encryption produces ciphertext with required randomness (different ciphertexts for same input with different keys).
- Decryption recovers the original plaintext exactly.

- Keys are generated securely and match the expected size (256 bits).

Validation results are shown in Table 4.2.

Table 4.2: Crypto Provider Algorithm Validation Results

Algorithm	Key Size	Encryption	Decryption
AES-256-CBC	256 bits	Success	Success

4.3 Proof of Concept Testing

Proof of concept testing validates the correctness and reliability of the core ledger components before advancing to more complex deployment scenarios. All tests were implemented as automated unit tests, ensuring reproducibility and isolation of individual components. The focus was on verifying that the fundamental building blocks functioned as designed under both normal and edge-case conditions.

4.3.1 Entry Testing

Entry tests concentrated on verifying the automatic generation of essential metadata:

- ID Generation: Ensured globally unique identifiers were consistently produced.
- Timestamp Creation: Confirmed timestamps were accurate and correctly integrated into hash calculations.
- Hash Integrity: Verified that entry content, metadata, and references were included in hash computation and that any modification was detectable.

Signature validation received particular attention. Unit tests confirmed that:

- Only valid signatures from declared senders were accepted.
- Invalid or tampered signatures caused deterministic rejection.
- Multiple cryptographic algorithms (e.g., RSA, ECDSA) were supported and correctly verified.

4.3.2 Page Testing

Page-level unit tests ensured that:

- Each page correctly included the hash of the previous page, maintaining chain integrity.
- Merkle tree roots accurately reflected the underlying entries, providing tamper evidence.
- Page hashes were correctly computed over all internal structures.

Concurrency scenarios were also unit tested by simulating parallel entry additions, validating that page assembly remained deterministic and consistent under multi-threaded conditions.

4.3.3 Receipt Testing

Receipt unit tests confirmed the validity of the Merkle proof mechanism:

- Entries could be independently verified against a page root.
- System-generated signatures on receipts could not be forged or modified without detection.
- Verification processes remained robust across different hash and signature algorithms.

These tests ensured that receipts provide reliable, independently verifiable evidence of ledger state.

4.3.4 Ledger Testing

The ledger, as the system abstraction layer, was validated through comprehensive unit tests combining entry, page, and receipt operations. Tests confirmed that:

- Concurrency handling allowed multiple entries to be processed without data corruption or race conditions.
- Receipt generation functioned consistently across varying ledger states.
- The system could securely and irreversibly erase the content of an entry while preserving its metadata and historical footprint.

These tests demonstrated that the ledger maintained correctness under both isolated and integrated operations, ensuring trustworthiness of higher-level functionality.

4.4 Security Analysis and Attack Mitigation

The centralized architecture introduces specific security considerations requiring systematic analysis and mitigation strategies. This section examines potential attack vectors, their implications, and the defensive mechanisms implemented within the system.

4.4.1 Attack Vector Analysis

The centralized ledger system faces several categories of potential attacks, each requiring specific mitigation approaches:

4.4.1.1 Data Integrity Attacks

Data integrity attacks attempt to modify ledger contents without detection, compromising the fundamental trust guarantees of the system.

- Hash Manipulation Attacks: Attackers with database access might attempt to modify entry content while updating corresponding hash values to maintain apparent consistency. The system mitigates this through:
  - Merkle tree construction that propagates hash changes through page structures
  - Receipt generation that creates external evidence of original hash values
  - Audit Warden continuous monitoring that detects hash discrepancies
  - Digital signatures on receipts that prevent forging of hash evidence
- Timestamp Manipulation: Modifying timestamps could enable retroactive ordering of entries or creation of false chronological sequences. Mitigation strategies include:
  - Cryptographic inclusion of timestamps in hash calculations
  - Page-level timestamp validation ensuring chronological consistency
  - External receipt timestamps providing independent temporal evidence
  - Audit trail logging with external timestamp correlation
- Signature Forgery: Attempts to create false digital signatures or replace legitimate signatures with forged alternatives. The system addresses this through:
  - Public key infrastructure validation requiring certificate chains
  - Receipt inclusion of signature data for external verification
  - Cryptographic signature validation at multiple system layers
  - Immutable signature storage preventing post-creation modifications

4.4.1.2 Access Control Attacks

Access control attacks target authentication and authorization mechanisms to gain unauthorized system access or privilege escalation.

Credential Compromise: Stolen or compromised user credentials enable unauthorized system access. Defensive measures include:

- Two-factor authentication for critical operations
- Session timeout and token expiration mechanisms
- Comprehensive audit logging of all authentication attempts
- Password complexity requirements and secure storage

4.4.1.3 Internal Threat Attacks

Internal threats represent particularly serious risks due to privileged system access and organizational knowledge.

Administrator Privilege Abuse: System administrators with database access might attempt unauthorized ledger modifications. Mitigation approaches include:

- Receipt generation providing external evidence of system state
- Audit Warden operating independently of normal administrative access
- Cryptographic signatures preventing silent data modifications
- External backup systems limiting the scope of internal tampering

4.4.2 Mitigation Effectiveness

The implemented security measures provide defense-in-depth protection against identified attack vectors. The combination of cryptographic integrity, external verification capabilities, and continuous monitoring creates multiple barriers that attackers must overcome simultaneously.

Critical security properties maintained by the system include:

- Tamper Evidence: All unauthorized modifications can be detected through cryptographic validation
- Non-Repudiation: Digital signatures prevent denial of participation in ledger entries
- External Verification: Receipts enable independent validation without system access
- Audit Trail Integrity: Comprehensive logging with cryptographic protection
- Time-Bounded Risk: Regular backups and monitoring limit the duration of undetected attacks

The centralized architecture inherently creates single points of failure that distributed systems avoid. However, the implemented security measures provide practical protection appropriate for organizational deployments where the benefits of centralized control outweigh the theoretical advantages of distributed consensus.

4.5 Stress Testing and Performance Analysis

Comprehensive stress testing evaluates system performance under various load conditions, identifying capacity limits, resource utilization patterns, and scalability characteristics. The testing methodology examines key performance metrics across different system configurations using controlled experiments with multiple iterations to ensure statistical reliability.

4.5.1 Testing Methodology

The stress testing employed systematic variation of key parameters while maintaining controlled conditions to understand performance relationships and identify optimal configurations for different organizational requirements.

4.5.1.1 Experimental Design

Each configuration was tested five times to reduce entropy and ensure statistical reliability. The configurations were organized by varying specific attributes while maintaining default values for non-tested parameters to ensure fair comparisons. Default baseline configuration used:

- Entry Count: 10,000 entries
- Signatures Per Entry: 3 signatures
- Entries Per Page: 100 entries
- Content Size: 2,048 bytes
- Number of Threads: 12 threads
- Hash Algorithm: SHA-256
- Signature Algorithm: ECDSA

4.5.1.2 Testing Environment

For each test iteration, 10 new users were generated with fresh cryptographic key pairs, and a new ledger instance was created. Timing measurements focused exclusively on internal ledger operations, excluding external factors such as network delays, exception handling, or system failures that would affect real-world applications.

The measured metrics include:

- Total Time: Complete test execution duration (ms)
- Average Entry Creation: Time per entry creation operation (ms)
- Average Signature Processing: Time per signature verification and integration (ms)
- Average Page Creation: Time per page creation including Merkle tree computation (ms)

4.5.2 Performance Test Results

A specific test was created for each configurable attribute, maintaining the default values apart from the tested attribute.

4.5.2.1 Hash Algorithm Test

The first test examined the impact of different cryptographic hash algorithms on system performance, comparing SHA-256, SHA-512, and SHA3-256.

Test Configuration: 10,000 entries, 3 signatures each, 100 entries per page, 2,048 bytes content, 12 threads, ECDSA signatures

Figure 4.1 illustrates the performance characteristics across different hash algorithms. SHA-256 provides the best overall performance with consistently lower processing times across all

operations. The chart reveals that SHA-512 has a dramatic impact on page creation performance, nearly doubling the processing time compared to SHA-256 and SHA3-256. This is due to the increased computational overhead of the 512-bit hash function during Merkle tree construction, where multiple hash operations are performed sequentially. Entry creation and signature processing times remain relatively similar across algorithms, indicating that the primary performance difference lies in batch operations like page creation.

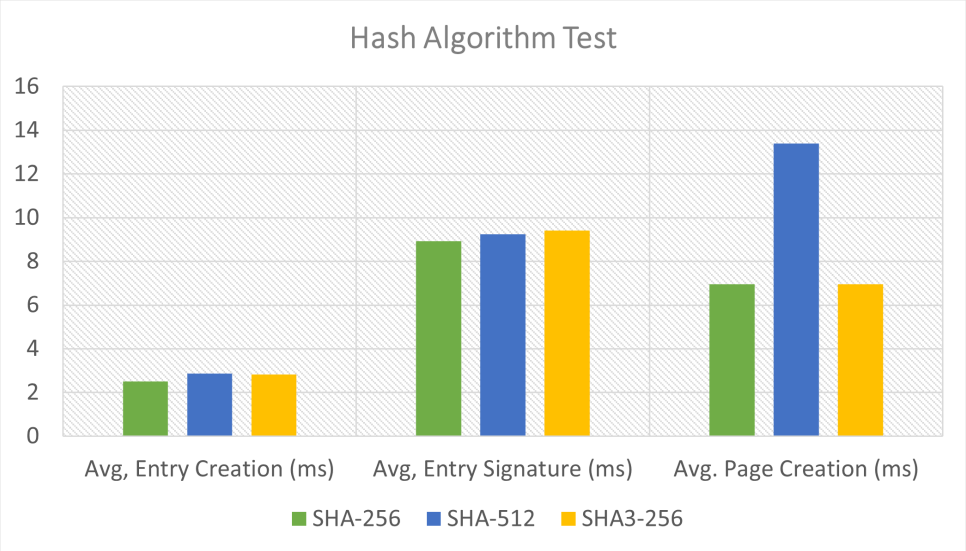


Figure 4.1: *Hash Algorithm Performance Comparison showing average processing times for entry creation, signature processing, and page creation across different hash algorithms*

4.5.2.2 Signature Algorithm Test

This test compared RSA and ECDSA signature algorithms to evaluate their computational overhead.

Test Configuration: 10,000 entries, 3 signatures each, 100 entries per page, 2,048 bytes content, 12 threads, SHA-256 hash

Figure 4.2 clearly demonstrates RSA's performance advantage across all operations. RSA shows dramatically faster entry creation times (0.30ms vs 2.93ms for ECDSA) and moderately better signature processing and page creation performance. The chart reveals that RSA provides approximately 24% better total execution time. The substantial difference in entry creation suggests that RSA key operations are more efficient in the Java/Kotlin environment, though ECDSA maintains the advantage of smaller key sizes for storage and transmission requirements.

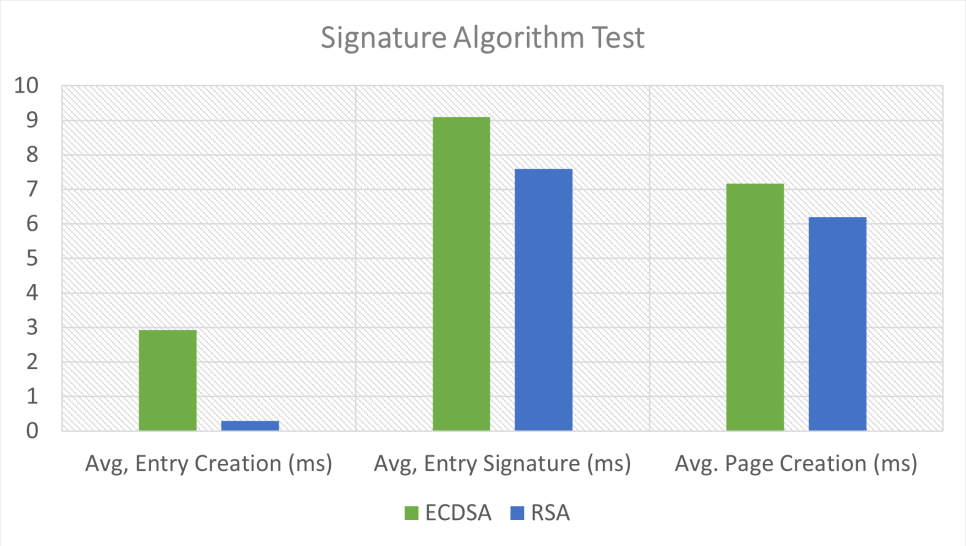


Figure 4.2: *Signature Algorithm Performance Comparison showing processing time differences between RSA and ECDSA for entry creation, signature processing, and page creation*

4.5.2.3 Thread Concurrency Test

This test evaluated the impact of concurrent thread processing on system throughput.

Test Configuration: 10,000 entries, 3 signatures each, 100 entries per page, 2,048 bytes content, SHA-256 hash, ECDSA signatures

Figure 4.3 illustrates the complex relationship between thread count and performance. The chart shows that individual operation times increase with thread count due to concurrency overhead, while total execution time decreases up to 12 threads. The optimal performance occurs at 12 threads, matching the processor's logical core count (4 cores × 2 hyperthreads + system overhead). Beyond 12 threads, context switching overhead begins to impact performance, with 16 threads showing slightly worse total execution time than 12 threads.

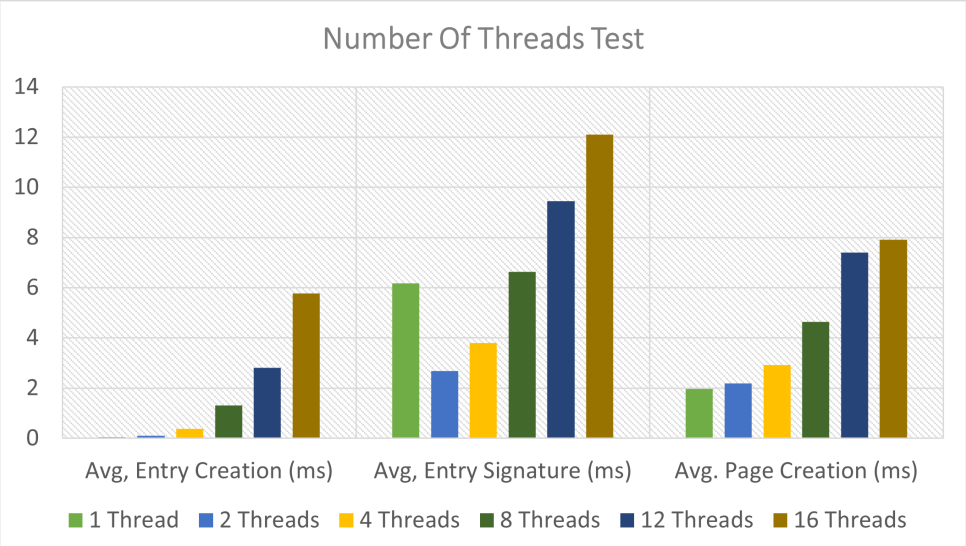


Figure 4.3: *Thread Concurrency Performance Analysis showing the relationship between thread count and processing times across different operations*

4.5.2.4 Content Size Analysis

This test examined how entry payload size affects processing performance.

Test Configuration: 10,000 entries, 3 signatures each, 100 entries per page, various content sizes, 12 threads, SHA-256 hash, ECDSA signatures

Content size shows minimal impact on processing performance across the tested range. The relatively flat performance profile suggests that hash computation overhead is not the limiting factor for content sizes up to 16KB. This indicates the system can efficiently handle typical document metadata and small file contents without performance degradation. A further isolated test was performed with 100kB, and the results showed a similar performance profile. Figure 4.4 illustrates the relationship between content size and processing time, confirming that payload size has minimal impact on system throughput up to 16kB.

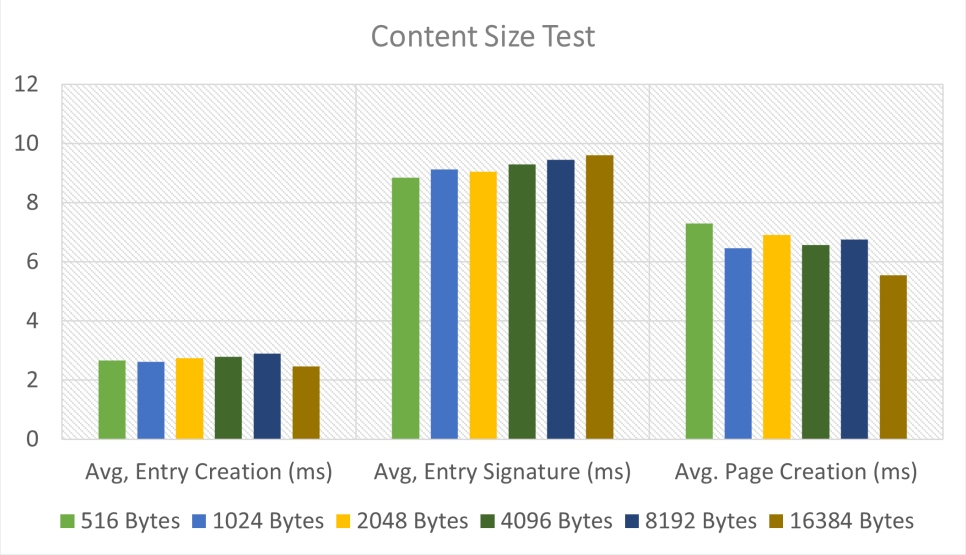


Figure 4.4: *Content Size Performance Analysis showing the effect of varying payload size on entry creation, signature processing, and page creation times*

4.5.2.5 Entries Per Page Configuration Analysis

This test examined how page size affects system performance and resource utilization.

Test Configuration: 10,000 entries, 3 signatures each, various page sizes, 2,048 bytes content, 12 threads, SHA-256 hash, ECDSA signatures

Smaller page sizes (10-50 entries) provide faster individual page creation but require more frequent page operations. Large page sizes (500-1000 entries) show a higher page creation time due to Merkle tree computational complexity. The optimal configuration depends on user concurrency, since page creation temporarily blocks thread execution. Figure 4.5 highlights how different page sizes impact processing efficiency, showing the trade-off between frequent page operations and Merkle tree complexity.

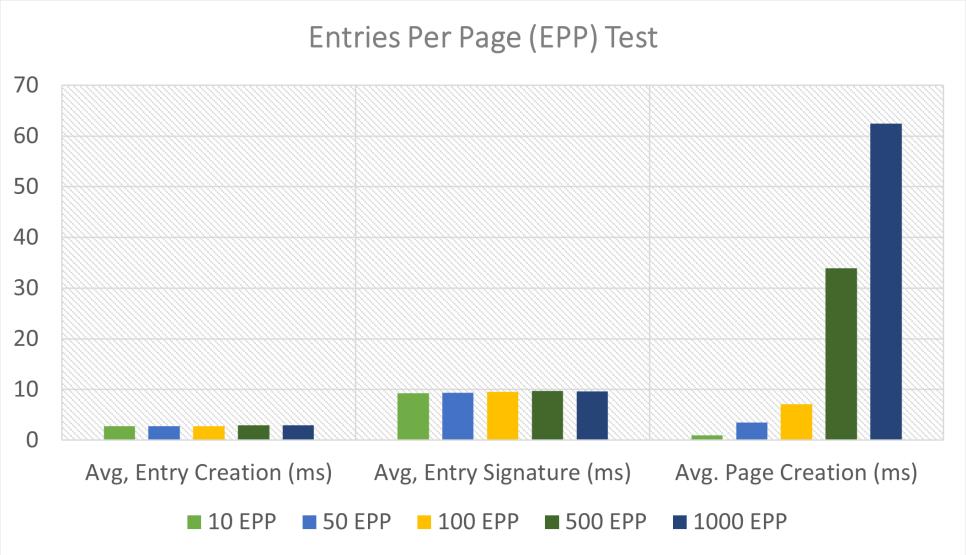


Figure 4.5: *Entries Per Page Performance Analysis showing how different page sizes affect page creation time and overall throughput*

4.5.2.6 Signature Count Analysis

This test evaluated the impact of multiple signature requirements on processing performance.

Test Configuration: 10,000 entries, various signature counts, 100 entries per page, 2,048 bytes content, 12 threads, SHA-256 hash, ECDSA signatures

The results show stable performance for signature addition and page creation, with the primary impact observed in entry creation times. As Figure 4.6 demonstrates that increasing the number of required signatures per entry causes higher creation times due to extended verification delays and concurrency overhead.

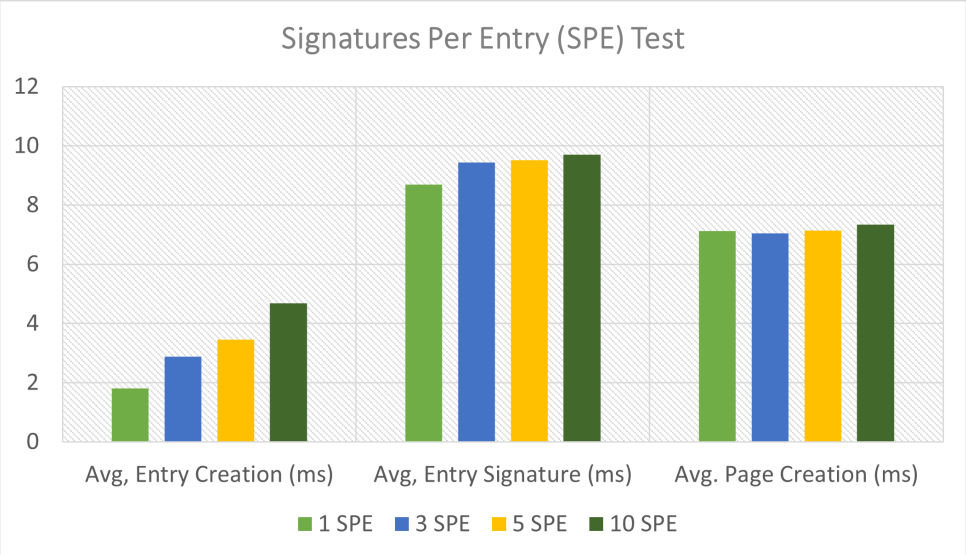


Figure 4.6: *Signature Count Performance Analysis showing how increasing the number of required signatures per entry affects entry creation, signature processing, and page creation times*

4.5.2.7 Entry Count Analysis

This test evaluated system performance across different total entry counts to understand scalability characteristics.

The scale analysis demonstrates excellent linear scalability. Processing times per operation remain consistent across all tested entry counts, as shown in Figure 4.7. This indicates that the system sustains performance even under increasing workloads, with slight improvements at larger scales likely due to JVM optimizations. The slight improvement in per-operation times at larger scales suggests efficient resource utilization and possible JVM optimization effects.

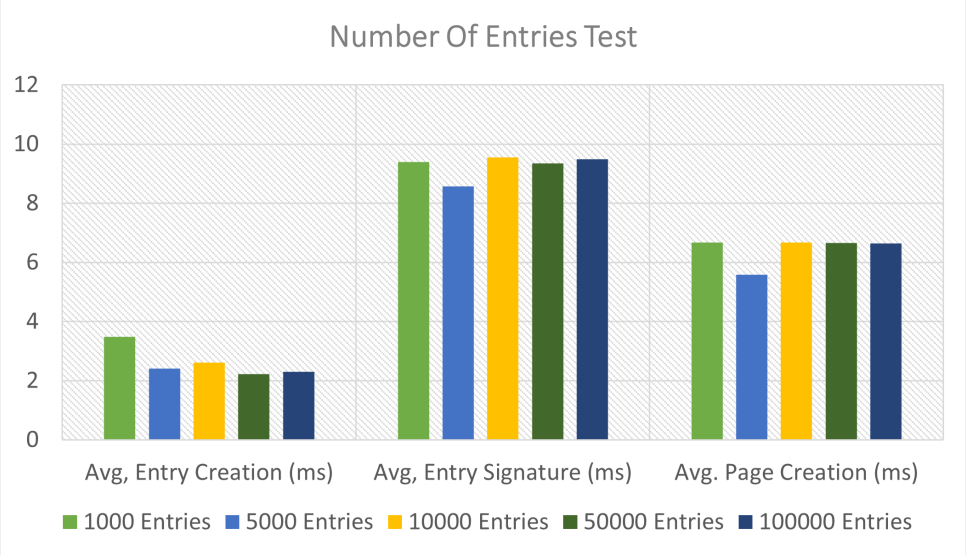


Figure 4.7: *Entry Count Scalability Analysis showing the effect of increasing total entries on processing performance*

4.5.3 Performance Analysis Summary

4.5.3.1 Optimal Configuration Recommendations

Based on the comprehensive testing results, the following configurations provide optimal performance for different scenarios:

High-Throughput Scenario:

- RSA signatures for fastest processing
- SHA-256 hashing for balanced security and performance
- 50-100 entries per page for optimal page creation efficiency
- Thread count matching available CPU cores
- Maximum of 3 senders per entry

High-Security Scenario:

- ECDSA signatures for smaller key sizes and forward security

- SHA-512 or SHA3-256 for enhanced cryptographic strength
- Small page sizes (> 50 entries) acceptable if security justifies performance cost

4.5.3.2 Scalability Recommendations

Vertical Scaling: The system would benefit most from:

1. CPU Enhancement: Higher core count processors for signature processing parallelization
2. Memory Expansion: 32-64GB RAM for larger in-memory entry pools

Horizontal Scaling: The modular architecture supports:

1. Database Read Replicas: For improved query performance without affecting write operations
2. Application Server Load Balancing: Multiple instances handling different ledger domains

The signature processing component would benefit most from vertical scaling due to its CPU-intensive cryptographic operations, while the database layer would benefit from both vertical and horizontal scaling strategies depending on the specific bottlenecks encountered in production deployments.

CHAPTER 5

CONCLUSIONS

This chapter provides an overview (Section 5.1) of this study and contemplates its conclusions. Possible improvements and alternatives resulting from self-assessment are presented in Section 5.2

5.1 Overview

This thesis demonstrates the practical viability of implementing a Centralized Ledger System (CLS) that provides blockchain-inspired integrity verification without the operational complexities and economic volatilities of distributed blockchain networks. Through comprehensive design, implementation, and evaluation, the research establishes that centralized ledger systems can bridge the gap between traditional audit logging and distributed ledger technologies.

The implemented system validates several core principles:

- Cryptographic Integrity Preservation: The system achieves cryptographic guarantees equivalent to blockchain systems while maintaining centralized control. Hash-based integrity verification, digital signature validation, and Merkle tree construction provide tamper-evidence capabilities that detect unauthorized modifications across all ledger components.
- Multi-Ledger Architecture Viability: The ability to segregate business domains while maintaining referential integrity enables enterprises to implement domain-specific security policies without compromising system-wide consistency. Testing confirms that concurrent operations across multiple ledgers operate independently without interference.
- Receipt-Based External Verification: The cryptographic receipt mechanism addresses the trust gap inherent in centralized systems by providing users with independently verifiable evidence of their interactions. Receipt validation demonstrates reliable tamper detection and authentication verification across all tested scenarios.
- Linear Scalability Characteristics: Performance evaluation reveals consistent per-operation processing times across varying workload conditions, from 1,000 to 100,000 entries. The

system maintains stable performance with slight improvements at larger scales, indicating suitability for enterprise deployments requiring predictable performance.

Security analysis confirms that while centralized architectures create single points of failure, the implemented defensive mechanisms provide practical protection for organizational deployments. The combination of cryptographic integrity verification, external receipt validation, and continuous audit monitoring creates multiple barriers against identified attack vectors.

The selective content erasure capability addresses privacy and regulatory compliance requirements while preserving cryptographic validation capabilities. This functionality enables organizations to respond to data protection regulations without compromising ledger structure integrity, a feature typically difficult to implement in immutable systems.

The research validates that centralized ledger systems offer a practical alternative to distributed blockchain networks for organizations requiring enhanced audit capabilities, cost predictability, and operational control while maintaining cryptographic security guarantees.

5.2 Future Work

The research opens several directions for continued development and optimization of centralized ledger systems:

5.2.1 Production Validation and Standardization

- Conduct comprehensive production testing with real organizational workloads over extended operational periods to validate long-term integrity preservation and system stability

- Develop formal cryptographic standards for algorithm naming conventions, parameter specifications, and interoperability requirements to facilitate system integration and cross-organizational verification capabilities

5.2.2 Architectural Enhancements

- Integrate distributed storage solutions, NoSQL databases, or specialized document management systems to handle larger document volumes and provide enhanced redundancy

- Extract core ledger components into independent microservices to enable organizations to integrate ledger functionality without comprehensive system replacement

- Implement lazy loading mechanisms, page-based data retrieval, and intelligent caching strategies to handle larger ledger volumes without proportional memory consumption increases

5.2.3 Advanced Capabilities

- Implement hybrid blockchain integration through periodic anchoring of ledger state to public networks, combining centralized operational control with enhanced external security guarantees
- Create sophisticated audit trail analysis, automated compliance reporting, and business intelligence integration to transform the system from passive ledger to active compliance tool
- Standardize RESTful API specifications and provide client libraries for multiple programming languages to reduce integration complexity

The modular architecture developed in this thesis provides a foundation for these enhancements, enabling incremental improvement without fundamental architectural modifications. The combination of proven security mechanisms, demonstrated scalability characteristics, and flexible deployment options positions the system for continued development and broader organizational adoption.

BIBLIOGRAPHY

[1] D. Abramov. *Create React App vs Next.js*. Accessed: 29/09/2025. 2023. URL: https://blog.logrocket.com/next-js-vs-create-react-app/ (cit. on p. 25).

[2] C. Adams and S. Lloyd. *Understanding PKI: Concepts, Standards, and Deployment Considerations*. 2nd. Accessed: 29/09/2025. Addison-Wesley Professional, 2003. ISBN: 978-0672323911 (cit. on p. 39).

[3] Adobe Inc. *Adobe Sign Electronic Signature Solutions*. Accessed: 29/09/2025. 2025. URL: https://www.adobe.com/sign.html (cit. on p. 20).

[4] Amazon Web Services. *Amazon Aurora PostgreSQL with History Tables*. Accessed: 29/09/2025. 2024. URL: https://aws.amazon.com/rds/aurora/postgresql-features/ (cit. on p. 19).

[5] Amazon Web Services. *Amazon QLDB End of Support Announcement*. Accessed: 29/09/2025. 2024. URL: https://aws.amazon.com/pt/qldb/ (cit. on p. 19).

[6] Apache Software Foundation. *Apache Maven Project*. Accessed: 29/09/2025. 2024. URL: https://maven.apache.org/ (cit. on p. 25).

[7] J. Benet. *IPFS: Content Addressed, Versioned, P2P File System (DRAFT 3)*. arXiv:1407.3561; Accessed: 29/09/2025. 2014. arXiv: 1407.3561 [cs.NI]. URL: https://arxiv.org/abs/1407.3561 (cit. on p. 26).

[8] G. Bierman, M. Abadi, and M. Torgersen. "Understanding TypeScript". In: *ECOOP 2014 – Object-Oriented Programming*. Accessed: 29/09/2025. 2014, pp. 257–281. DOI: 10.1007/978-3-662-44202-9_11 (cit. on p. 24).

[9] BitInfoCharts. *Bitcoin Transaction Fees*. Accessed: 29/09/2025. 2025.URL: https://bitinfocharts.com/comparison/bitcoin-transactionfees.html (cit. on p. 18).

[10] CA/Browser Forum. *Baseline Requirements for the Issuance and Management of Publicly-Trusted Certificates*. Tech. rep. Accessed: 29/09/2025. CA/Browser Forum, 2024. URL: https://cabforum.org/baseline-requirements-documents/ (cit. on p. 39).

[11] CodeNotary Inc. *ImmuDB: The Open-Source Immutable Database*. Accessed: 29/09/2025. 2023. URL: https://immudb.io/ (cit. on p. 15).

[12] Cofidis. *Cofidis Portugal: Crédito Pessoal e Soluções de Financiamento*. Accessed: 21/10/2025. 2024. URL: https://www.cofidis.pt/ (cit. on p. 1).

[13] DocuSign Inc. *DocuSign Electronic Signature Platform*. Accessed: 29/09/2025. 2024. URL: https://www.docusign.com/ (cit. on p. 20).

[14] Dropbox Inc. *Dropbox Business: Secure File Sharing and Collaboration Platform*. Accessed: 29/09/2025. 2024. URL: https://www.dropbox.com/business (cit. on p. 1).

[15] T. Erl. *SOA Design Patterns*. Accessed: 29/09/2025. Prentice Hall (The Prentice Hall Service-Oriented Computing Series from Thomas Erl), 2009. ISBN: 0136135161. DOI: 10.5555/1538586. URL: https://www.thomaserl.com/book/soa-design-patterns/overview/index.html (cit. on p. 23).

[16] Ethereum. *Ethereum: A Global, Open-Source Platform for Decentralized Applications*. Accessed: 29/09/2025. 2014. URL: https://www.ethereum.org (cit. on p. 17).

[17] W. Fang, W. Chen, W. Zhang, J. Pei, W. Gao, and G. Wang. "Digital signature scheme for information non-repudiation in blockchain: a state of the art review". In: *EURASIP Journal on Wireless Communications and Networking* 2020 (2020). Accessed: 29/09/2025. DOI: 10.1186/s13638-020-01665-w (cit. on p. 9).

[18] FasterXML. *Jackson JSON Processor - documentation and project pages*. Jackson project (GitHub) – API docs also at https://javadoc.io and https://fasterxml.github.io/jackson-databind/. Accessed: 29/09/2025. 2024. URL: https://github.com/FasterXML/jackson-docs (cit. on p. 25).

[19] Foley and Lardner. *Types of Blockchain: Public, Private, or Something in Between*. Accessed: 29/09/2025. 2021. URL: https://www.jdsupra.com/legalnews/types-of-blockchain-public-private-or-5282575/ (cit. on p. 17).

[20] L. Foschini, A. Gavagna, G. Martuscelli, and R. Montanari. "Hyperledger Fabric Blockchain: Chaincode Performance Analysis". In: *ICC 2020 - 2020 IEEE International Conference on Communications (ICC)*. Accessed: 29/09/2025. 2020, pp. 1–6. DOI: 10.1109/ICC40277.2020.9149080 (cit. on pp. 10, 15).

[21] M. Fowler. *Patterns of Enterprise Application Architecture*. Accessed: 29/09/2025. Addison-Wesley Professional, 2002. ISBN: 978-0321127426. URL: https://martinfowler.com/books/eaa.html (cit. on p. 22).

[22] M. Fowler. *Inversion of Control Containers and the Dependency Injection pattern*. Accessed: 29/09/2025. 2004. URL: https://martinfowler.com/articles/injection.html (cit. on p. 24).

[23] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Accessed: 29/09/2025. Addison-Wesley Professional, 1994. ISBN: 978-0201633610. URL: https://www.pearson.com/en-us/subject-catalog/p/design-patterns-elements-of-reusable-object-oriented-software/P200000009480/9780321700698 (cit. on p. 24).

[24] GeeksforGeeks. *Hash Functions and Types of Hash functions*. Accessed: 29/09/2025. 2025. URL: https://www.geeksforgeeks.org/hash-functions-and-list-types-of-hash-functions/ (cit. on p. 6).

[25] P. Gharib. *Document Management Statistics for Efficiency*. Accessed: 29/09/2025. 2024. URL: https://profiletree.com/document-management-statistics-for-efficiency/ (cit. on p. 1).

[26] Google LLC. *Google Workspace: Business Collaboration and Productivity Tools*. Accessed: 29/09/2025. 2025. URL: https://workspace.google.com/ (cit. on p. 1).

[27] J. Gossman. "Introduction to Model/View/ViewModel pattern for building WPF apps". In: *Microsoft Developer Network* (2005). Accessed: 29/09/2025. URL: https://docs.microsoft.com/en-us/archive/blogs/johngossman/introduction-to-modelviewviewmodel-pattern-for-building-wpf-apps (cit. on p. 23).

[28] R. Housley, W. Polk, W. Ford, and D. Solo. *Internet X.509 Public Key Infrastructure Certificate and Certificate Revocation List (CRL) Profile*. RFC 3280. Accessed: 29/09/2025. 2002. DOI: 10.17487/RFC3280. URL: https://tools.ietf.org/html/rfc3280 (cit. on p. 39).

[29] International Telecommunication Union. *X.509: Information technology - Open Systems Interconnection - The Directory: Public-key and attribute certificate frameworks*. Accessed: 29/09/2025. 2019. URL: https://www.itu.int/rec/T-REC-X.509/en (cit. on p. 40).

[30] P. Jamshidi, A. Ahmad, and C. Pahl. "Cloud Migration Research: A Systematic Review". In: *IEEE Transactions on Cloud Computing* 1.2 (2013). Accessed: 29/09/2025, pp. 142–157. DOI: 10.1109/TCC.2013.10 (cit. on p. 19).

[31] M. Janssen and A. Joha. "Challenges for adopting cloud-based software as a service (SaaS) in the public sector". In: *Proceedings of the European Conference on Information Systems (ECIS)*. Accessed: 29/09/2025. Association of Information Systems (AIS), 2011, pp. 1–10. URL: https://aisel.aisnet.org/ecis2011/80/ (cit. on p. 19).

[32] D. Jemerov and S. Isakova. *Kotlin in Action*. Accessed: 29/09/2025. Manning Publications, 2017. ISBN: 978-1617293290. URL: https://www.manning.com/books/kotlin-in-action (cit. on p. 25).

[33] JetBrains. *Kotlin Language Reference*. Accessed: 29/09/2025. 2024. URL: https://kotlinlang.org/docs/home.html (cit. on p. 25).

[34] S. Jeyaseelan. "Vendor Lock-In Issues in Cloud Computing and How to Neutralize Them". Accessed: 29/09/2025. PhD thesis. Capella University, 2025. URL: https://search.proquest.com/openview/d36c3a47a259e8fe30e2107afbabb1a4/1.pdf?cbl=18750&diss=y&pq-origsite=gscholar (cit. on p. 19).

[35] M. Jones, J. Bradley, and N. Sakimura. *JSON Web Token (JWT)*. RFC 7519. Accessed: 29/09/2025. 2015. DOI: 10.17487/RFC7519. URL: https://tools.ietf.org/html/rfc7519 (cit. on p. 25).

[36] A. Kamsky. *How Bitcoin Uses Merkle Trees To Ensure Transaction Integrity*. Accessed: 29/09/2025. 2024. URL: https://www.ccn.com/education/crypto/bitcoin-merkle-trees-ensure-transaction-integrity/ (cit. on p. 7).

[37] R. Kuchta. *The hash - a computer file's digital fingerprint*. Accessed: 29/09/2025. 2023. URL: https://newtech.law/en/articles/the-hash-a-computer-files-digital-fingerprint (cit. on p. 6).

[38] I. Mandiri and D. McIntyre. *Decentralism*. Accessed: 29/09/2025. 2022. URL: https://ethereumclassic.org/why-classic/decentralism (cit. on p. 16).

[39] Meta Platforms Inc. *React Concepts: Describing the UI*. Accessed: 29/09/2025. 2024. URL: https://react.dev/learn/describing-the-ui (cit. on p. 25).

[40] Meta Platforms Inc. *React Documentation*. Accessed: 29/09/2025. 2024. URL: https://react.dev/ (cit. on p. 24).

[41] Microsoft Corporation. *Model-View-ViewModel Pattern*. Accessed: 29/09/2025. 2023. URL: https://learn.microsoft.com/en-us/dotnet/architecture/maui/mvvm (cit. on pp. 23, 44).

[42] Microsoft Corporation. *Azure SQL Database Ledger*. Accessed: 29/09/2025. 2024. URL: https://learn.microsoft.com/en-us/azure/azure-sql/database/ledger-overview (cit. on p. 19).

[43] Microsoft Corporation. *Microsoft SharePoint: Collaboration and Content Management Platform*. Accessed: 29/09/2025. 2024. URL: https://www.microsoft.com/en-us/microsoft-365/sharepoint/collaboration (cit. on p. 1).

[44] Microsoft Corporation. *The TypeScript Handbook*. Accessed: 29/09/2025. 2024. URL: https://www.typescriptlang.org/docs/ (cit. on p. 24).

[45] B. Momjian. *PostgreSQL: Introduction and Concepts*. Accessed: 29/09/2025. Addison-Wesley / Pearson Education, 2001. ISBN: 978-0201703313. URL: https://momjian.us/main/presentations/book.html (cit. on p. 26).

[46] MongoDB Inc. *MongoDB Documentation*. Accessed: 29/09/2025. 2024. URL: https://docs.mongodb.com/ (cit. on p. 26).

[47] S. Nakamoto. *Bitcoin: A Peer-to-Peer Electronic Cash System*. SSRN; also available at https://bitcoin.org/bitcoin.pdf. Accessed: 29/09/2025. 2008. DOI: 10.2139/ssrn.3440802. URL: https://bitcoin.org/bitcoin.pdf (cit. on pp. 9, 17).

[48] O. Nem. *Digital Signature Algorithms: The Science Behind Secure Online Signatures*. Accessed: 29/09/2025. 2024. URL: https://tribulant.com/blog/business/digital-signature-algorithms-the-science-behind-secure-online-signatures/ (cit. on p. 9).

[49] Oracle Corporation. *Blockchain or Immutable Tables (Oracle Database blog post)*. Oracle Blogs (CoreTech / Blockchain posts). Accessed: 29/09/2025. 2024. URL: https://blogs.oracle.com/coretec/post/blockchain-or-immutable-tables (cit. on p. 19).

[50] Oracle Corporation. *JDBC API Documentation*. Accessed: 29/09/2025. 2024. URL: https://www.oracle.com/database/technologies/appdev/jdbc.html (cit. on pp. 25, 26).

[51] PandaDoc Inc. *PandaDoc Document Workflow Platform*. Accessed: 29/09/2025. 2024. URL: https://www.pandadoc.com/ (cit. on p. 20).

[52] M. Pato. *The ISELthesis LaTeX Template's Manual*. Instituto Superior de Engenharia de Lisboa (ISEL-IPL). 2024. URL: https://github.com/matpato/iselthesis (cit. on p. viii).

[53] Pivotal Software Inc. *Spring Framework Documentation*. Accessed: 29/09/2025. 2024. URL: https://spring.io/projects/spring-framework (cit. on p. 25).

[54] PostgreSQL Global Development Group. *PostgreSQL Documentation*. Accessed: 29/09/2025. 2024. URL: https://www.postgresql.org/docs/ (cit. on p. 26).

[55] R. S. Pressman and B. R. Maxim. *Software Engineering: A Practitioner's Approach*. 9th. Accessed: 29/09/2025. McGraw-Hill Education, 2019. ISBN: 978-1260548006 (cit. on p. 22).

[56] K. S. Puranam, M. Gaddam, V. K, S. Panda, and G. S. M. Reddy. "Anatomy and Lifecycle of a Bitcoin Transaction". In: *SSRN Electronic Journal* (2019). Accessed: 29/09/2025. DOI: 10.2139/ssrn.3355106 (cit. on p. 14).

[57] T. Rafaj, L. Mastilak, K. Kost'al, and I. Kotuliak. "DeFi Gaming Platform Using the Layer 2 Benefits". In: *2023 33rd Conference of Open Innovations Association (FRUCT)*. Accessed: 29/09/2025. 2023, pp. 236–242. DOI: 10.23919/FRUCT58615.2023.10143054 (cit. on p. 14).

[58] B. Schneier and J. Kelsey. "Secure Audit Logs to Support Computer Forensics". In: *ACM Transactions on Information and System Security*. Vol. 2. 2. Accessed: 29/09/2025. 1999, pp. 159–176. DOI: 10.1145/317087.317089 (cit. on p. 34).

[59] A. Silberschatz, P. B. Galvin, and G. Gagne. "Operating System Concepts – Security and Protection (chapter/resource reference)". In: *Operating System Concepts (book)* (2018). Accessed: 29/09/2025 (cit. on p. 36).

[60] Simplilearn. *Merkle Tree in Blockchain: What is it and How does it work*. Accessed: 29/09/2025. 2023. URL: https://www.simplilearn.com/tutorials/blockchain-tutorial/merkle-tree-in-blockchain (cit. on p. 7).

[61] J. Smith and M. Johnson. "Implementing MVVM Pattern in React Applications: A Systematic Approach". In: *International Journal of Software Engineering* 14.2 (2023). Accessed: 29/09/2025, pp. 87–102. DOI: 10.1016/j.ijse.2023.02.014 (cit. on p. 25).

[62] Spring Team. *Kotlin support in Spring Framework*. Accessed: 29/09/2025. 2024. URL: https://docs.spring.io/spring-framework/reference/languages/kotlin.html (cit. on p. 25).

[63] Spring Team. *Spring Boot Reference Guide*. Accessed: 29/09/2025. 2024. URL: https://spring.io/projects/spring-boot (cit. on p. 25).

[64] Spring Team. *Spring Framework AOP Reference*. Accessed: 29/09/2025. 2024. URL: https://docs.spring.io/spring-framework/reference/core/aop.html (cit. on p. 25).

[65] Spring Team. *Spring Framework Reference Documentation*. Accessed: 29/09/2025. 2024. URL: https://docs.spring.io/spring-framework/reference/ (cit. on p. 23).

[66] Spring Team. *Spring MVC Architecture and Components*. Accessed: 29/09/2025. 2024. URL: https://docs.spring.io/spring-framework/reference/web/webmvc.html (cit. on p. 24).

[67] Stampery Inc. *Stampery: Blockchain-based Data Certification*. Accessed: 29/09/2025. 2024. URL: https://stampery.com/ (cit. on p. 19).

[68] Tailwind Labs Inc. *Tailwind CSS Documentation*. Accessed: 29/09/2025. 2024. URL: https://tailwindcss.com/docs (cit. on p. 25).

[69] Tailwind Labs Inc. *Utility-First Fundamentals*. Accessed: 29/09/2025. 2024. URL: https://tailwindcss.com/docs/utility-first (cit. on p. 25).

[70] The Legion of the Bouncy Castle. *Bouncy Castle Cryptography APIs*. Accessed: 29/09/2025. 2024. URL: https://www.bouncycastle.org/ (cit. on p. 25).

[71] Tierion Inc. *Chainpoint: A Global Network for Anchoring Data to the Blockchain*. Accessed: 29/09/2025. 2024. URL: https://chainpoint.org/ (cit. on p. 19).

[72] P. Todd. *OpenTimestamps: A Timestamping Proof Standard*. Accessed: 29/09/2025. 2024. URL: https://opentimestamps.org/ (cit. on p. 19).

[73] M. Venter. *100 Document Management Statistics For The Digital Era*. Accessed: 29/09/2025. 2023. URL: https://www.pdfreaderpro.com/blog/document-management-statistics/ (cit. on p. 1).

[74] Vercel Inc. *Next.js Documentation*. Accessed: 29/09/2025. 2024. URL: https://nextjs.org/docs (cit. on p. 25).

[75] W3C. *CSS3 Specification*. Accessed: 29/09/2025. 2024. URL: https://www.w3.org/Style/CSS/current-work (cit. on p. 24).

[76] W3C. *HTML5 Specification*. Accessed: 29/09/2025. 2024. URL: https://www.w3.org/TR/html52/ (cit. on p. 24).

[77] J. Werth, M. H. Berenjestanaki, H. Barzegar, N. E. Ioini, and C. Pahl. "A Review of Blockchain Platforms Based on the Scalability, Security and Decentralization Trilemma". In: *IEEE Access* (2023). Accessed: 29/09/2025, pp. 146–155. DOI: 10.5220/0011837200003467 (cit. on p. 16).

[78] YCharts. *Bitcoin Average Transaction Fee*. Accessed: 29/09/2025. 2024. URL: https://ycharts.com/indicators/bitcoin_average_transaction_fee (cit. on p. 18).

[79] YCharts. *Ethereum Average Transaction Fee*. Accessed: 29/09/2025. 2024. URL: https://ycharts.com/indicators/ethereum_average_transaction_fee (cit. on p. 18).