

FACULDADE DE ENGENHARIA DA UNIVERSIDADE DO PORTO

Modular Blockchain-Based Data Tampering Detection for Storage Systems

Alexandre Henrique Silva dos Santos



Mestrado em Engenharia Informática e Computação

Supervisor: Prof. Francisco Maia

July 21, 2026

Modular Blockchain-Based Data Tampering Detection for Storage Systems

Alexandre Henrique Silva dos Santos

Mestrado em Engenharia Informática e Computação

Approved in oral examination by the committee:

President: Prof. José Magalhães Cruz

Examiner: Prof. Tânia Esteves

Supervisor: Prof. Francisco Maia

July 21, 2026

Resumo

A manipulação de dados continua a representar um desafio significativo em ambientes onde os sistemas de armazenamento não são totalmente confiáveis. Embora as tecnologias de armazenamento tradicionais ofereçam elevado desempenho e escalabilidade, a sua proteção contra modificações não autorizadas dos dados é limitada. Em contrapartida, sistemas imutáveis descentralizados, como as blockchains, proporcionam fortes garantias de integridade, mas introduzem frequentemente limitações consideráveis em termos de desempenho e escalabilidade. Este compromisso entre desempenho e confiança motiva o desenvolvimento de arquiteturas híbridas capazes de combinar as vantagens de ambas as abordagens.

Esta dissertação investiga a integração de sistemas de armazenamento convencionais com tecnologias de armazenamento imutável externo, com o objetivo de suportar a detecção de manipulação de dados, preservando as vantagens de desempenho das infraestruturas de armazenamento existentes. Foi realizada uma análise abrangente do estado da arte, com foco em sistemas híbridos que integram blockchain com base de dados e em abordagens existentes para a prevenção e detecção de manipulação de dados. Com base nas lacunas de investigação identificadas, foi proposta uma arquitetura na qual a verificação de integridade é introduzida através de uma camada de acesso ao armazenamento no lado do cliente, em vez de modificações no sistema interno de armazenamento.

Para validar a abordagem proposta, foi implementado um protótipo como extensão do Apache OpenDAL. O protótipo armazena âncoras criptográficas de integridade no InterPlanetary File System (IPFS) e disponibiliza, de forma transparente, capacidades de detecção de manipulação em múltiplos sistemas de armazenamento através de uma interface unificada. A solução foi avaliada utilizando os sistemas de armazenamento PostgreSQL, Redis e S3, considerando tanto o desempenho como a eficácia da detecção de manipulação.

Os resultados demonstram que a arquitetura proposta deteta com sucesso diversos cenários de manipulação, sem exigir modificações nos sistemas de armazenamento protegidos nem nas aplicações. Embora os mecanismos de integridade introduzam uma sobrecarga mensurável em termos de latência e taxa de operações, a avaliação demonstra que essa sobrecarga é majoritariamente determinada por custos fixos por operação, tornando o seu impacto relativo menos significativo para objetos de maior dimensão e cargas de trabalho com maior latência. De forma geral, os resultados indicam que a integração de sistemas de armazenamento tradicionais com um sistema de armazenamento imutável externo constitui uma abordagem prática para reforçar a integridade dos dados em aplicações que operam sob pressupostos de confiança limitados.

Palavras-chave: Detecção de manipulação de dados. Sistemas de armazenamento. Bases de dados. Blockchain.

Abstract

Data tampering remains a significant challenge in environments where storage systems cannot be fully trusted. While traditional storage technologies provide high performance and scalability, they offer limited protection against unauthorized data modification. Conversely, decentralized immutable systems such as blockchains provide strong integrity guarantees but often introduce considerable performance and scalability constraints. This trade-off motivates the development of hybrid architectures capable of combining the strengths of both approaches.

This dissertation investigates the integration of conventional storage systems with external immutable storage technologies to support data tamper detection while preserving the performance advantages of existing storage infrastructures. A comprehensive analysis of the state-of-the-art was conducted, focusing on blockchain-database hybrid systems and existing approaches for preventing and detecting data tampering. Based on the identified research gaps, a client-side architecture was proposed in which integrity verification is introduced through a storage access layer rather than through modifications to the underlying storage engines.

To validate the proposed approach, a prototype was implemented as an extension of Apache OpenDAL. The prototype stores cryptographic integrity anchors in the InterPlanetary File System (IPFS) and transparently provides tamper detection capabilities across multiple storage backends through a unified interface. The solution was evaluated using PostgreSQL, Redis and S3 storage backends, considering both performance and tampering detection effectiveness.

The results demonstrate that the proposed architecture successfully detects a range of tampering scenarios while requiring no modifications to the protected storage systems or application code. Although the integrity mechanisms introduce measurable latency and throughput overhead, the evaluation shows that the overhead is primarily driven by fixed per-operation costs, making their relative impact less significant for larger objects and higher-latency workloads. Overall, the findings indicate that integrating traditional storage systems with an external immutable storage provides a practical approach for strengthening data integrity in applications operating under limited trust assumptions.

Keywords: Tampering detection. Storage systems. Databases. Blockchain.

UN Sustainable Development Goals

The United Nations Sustainable Development Goals (SDGs) provide a global framework for achieving a better and more sustainable future for all. It includes 17 goals (<https://sdgs.un.org/goals>) to address the world's most pressing challenges, including poverty, inequality, climate change, environmental degradation, peace, and justice.

This dissertation contributes to these global efforts by focusing on data integrity and security within digital infrastructures. As society increasingly relies on digital records, ensuring the immutability and trustworthiness of data becomes essential for resilient systems and transparent institutions. The proposed hybrid architecture secures existing storage systems against unauthorized data modification. This approach directly aligns with the principles of sustainable technological innovation and institutional accountability.

The specific Sustainable Development Goals mentioned have the following names:

SDG 9 Build resilient infrastructure, promote inclusive and sustainable industrialization and foster innovation

SDG 16 Promote peaceful and inclusive societies for sustainable development, provide access to justice for all and build effective, accountable and inclusive institutions at all levels

SDG	Target	Contribution	Performance Indicators and Metrics
9	9.1	Enhancing the resilience and reliability of data storage infrastructure by integrating robust tamper detection capabilities without disrupting existing systems.	Detection rate of unauthorized data modifications; measured performance overhead relative to baseline storage operations.
	9.4	Upgrading and retrofitting existing IT infrastructures with decentralized immutable technologies, increasing data security while preserving the efficiency of conventional storage engines.	Number of supported traditional storage backends; system throughput and latency evaluated under varied workloads.
16	16.6	Providing strong data integrity guarantees for digital records to support accountability and prevent fraudulent data manipulation in institutional storage.	Percentage of successful cryptographic integrity verifications; number of tampering scenarios successfully mitigated.

Acknowledgements

It is deeply important to me to express my profound appreciation for the people who have contributed so much to this journey and to my personal growth over the years.

I want to thank my friends and family for understanding how strenuous this process has been and helping me get through it mostly intact.

In particular, I want to express my deepest gratitude to my parents for all their unwavering support, not only during this dissertation or these last five years, but throughout my entire life. Thank you for all the opportunities you granted me and for making me most of what I am today. The rest is natural charm.

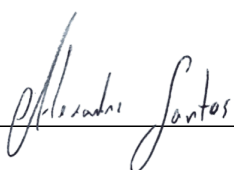
Finally, a big thank you to Professor Francisco Maia for supervising all the work and guiding me through every step of the process.

Alexandre Henrique Silva dos Santos

Declaration of honour

I declare that this dissertation is my own work and has not previously been submitted or assessed at this or any other institution. References to other authors (statements, ideas, thoughts), as well as the use of published works, strictly comply with the rules of attribution and are duly identified in the text and in the bibliographic references, in accordance with the applicable referencing standards. I am aware that the practice of plagiarism or self-plagiarism constitutes an academic offence, as established in the U.Porto Code of Ethics.

Alexandre Henrique Silva dos Santos



“Every disadvantage has its advantage.”

Johan Cruyff

Contents

1	Introduction	1
1.1	Context	1
1.2	Motivation	2
1.3	Problem	3
1.4	Research Questions	4
1.5	Contributions	4
2	Background	6
2.1	Data Storage	6
2.1.1	Data Model	6
2.1.2	Performance and Security	7
2.1.3	Distributed Databases	8
2.2	Blockchain	8
2.2.1	Data Integrity and Traceability	8
2.2.2	Decentralization and Consensus Mechanisms	9
2.2.3	Permissioned and Permissionless Blockchains	10
3	State of The Art	11
3.1	Hybrid Systems	13
3.2	Blockchain-supported Databases	13
3.3	Database-supported Blockchains	17
3.4	Optimized Authenticated Querying for Blockchain Data	18
3.5	Hybrid Distributed File-Storage Blockchain Systems	20
3.6	Discussion	21
4	Proposed Solution	24
4.1	Approach	24
4.2	Design Alternatives	26
4.3	Selected Implementation Path	28
5	Prototype	30
5.1	SDK Interception	31
5.1.1	Apache OpenDAL	31
5.1.2	Interception Strategy	32
5.2	External Immutable Storage Through IPFS	33
5.2.1	InterPlanetary File System	34
5.2.2	Why IPFS?	35
5.2.3	IPFS Integration	36

5.3	Anchor Mapping	37
5.4	Prototype Capabilities	40
5.5	Prototype Operation Flows	42
5.5.1	Write Flow	42
5.5.2	Read Flow	42
5.5.3	Append Write Flow	45
5.5.4	Additional Operation Flows	46
5.6	Trust and Threat Model	46
5.7	Design Trade-offs	47
6	Evaluation	50
6.1	Evaluation Methodology	50
6.1.1	Experimental Environment	51
6.1.2	Evaluation Configurations	51
6.1.3	Evaluated Backends	52
6.1.4	Workload Design	53
6.1.5	Dataset Generation	53
6.1.6	Benchmark Execution	54
6.1.7	Collected Metrics	58
6.2	Evaluation Results	58
6.2.1	MinIO S3	59
6.2.2	PostgreSQL	62
6.2.3	Redis	64
6.3	Tamper Detection Validation	67
6.3.1	Validation Methodology	68
6.3.2	Validation Results	69
7	Discussion	72
7.1	Interpretation of Performance Results	72
7.2	Suitable Application Scenarios	73
7.3	Assessment of Tamper Detection Mechanisms	76
7.4	Limitations	77
7.4.1	Prototype Limitations	77
7.4.2	Evaluation Limitations	78
8	Conclusion and Future Work	80
8.1	Summary of Findings	80
8.2	Main Contributions	81
8.3	Future Work	82
	References	84

List of Figures

4.1	High-level architecture design	24
5.1	Prototype architecture overview.	30
5.2	Write operation flow.	43
5.3	Read operation flow.	44
5.4	Append-based write workflow.	45
6.1	Benchmark execution process.	55
6.2	Relative latency overhead of the prototype compared to the baseline.	59
6.3	MinIO S3 mean latency and throughput per workload for each data size in Baseline vs. Prototype.	61
6.4	PostgreSQL mean latency and throughput per workload for each data size in Baseline vs. Prototype.	63
6.5	Redis mean latency and throughput per workload for each data size in Baseline vs. Prototype.	65
7.1	Baseline latency vs. Prototype relative latency overhead.	74

List of Tables

3.1	Comparison of Hybrid Blockchain-Database Architectures	22
6.1	Chronological execution sequence for a single benchmark run in PostgreSQL and MinIO S3.	56
6.2	Chronological execution sequence for a single benchmark run in Redis.	57
6.3	Relative overhead of the prototype compared to the baseline. Values are ratios.	60
6.4	Summary of the tamper detection validation results.	70

List of Acronyms

ACID	Atomicity, Consistency, Isolation, and Durability
ADS	Authenticated Data Structure
BFT	Byzantine Fault Tolerant
CAP	Consistency, Availability, Partition tolerance
CFT	Crash Fault Tolerant
CID	Content Identifier
CPU	Central Processing Unit
CSP	Cloud Service Provider
DDBS	Distributed Database System
DLT	Distributed Ledger Technology
EIS	External Immutable Storage
EMR	Electronic Medical Record
FPGA	Field Programmable Gate Array
GPU	Graphics Processing Unit
IPFS	InterPlanetary File System
IPNS	InterPlanetary Name System
LSM	Log-Structured Merge
MVCC	Multi-Version Concurrency Control
NFT	Non-Fungible Token
NoSQL	Non-Relational
OLAP	Online Analytical Processing
OLTP	Online Transaction Processing
P2P	peer-to-peer
PBFT	Practical Byzantine Fault Tolerance
PoW	Proof of Work
RBAC	Role-Based Access Control
RDBMS	Relational Database Management System
SDK	Software Development Kit
SQL	Structured Query Language
WAL	Write-Ahead Log

Chapter 1

Introduction

This chapter provides the foundation for the research presented in this dissertation, motivating the study of hybrid architectures that combine traditional storage systems with external immutable storage technologies, while defining the key questions that guide this work.

1.1 Context

Data are a central point of today’s technological progress, as their exponential growth presents one of the most significant challenges for contemporary information systems. Reflecting this trend, data presented in [60] from IDC’s Global Datasphere Forecast 2025 study predicts an expansion of the volume of data worldwide from 173.4 Zettabytes (ZB) in 2024 to exceed 500 ZB by 2029, highlighting the unprecedented pace at which digital information is being generated. To cope with this trend, systems must deliver high performance and scalability, ensuring fast access and management of massive datasets. At the same time, many domains require not only efficiency but also strong guarantees of integrity, availability and anti-tampering protection. Balancing these requirements has proven challenging, as most existing approaches address them unevenly. For instance, strong consistency is non-negotiable in financial transactions systems or real-time inventory management, where every read must reflect the latest write to prevent double-spending or stockouts. In contrast, eventual consistency is adopted in social media feeds or DNS to prioritize high availability.

Traditional database systems have long served as the primary solution for storing and managing digital information. A database organizes data into structured formats, enabling efficient querying and retrieval, while achieving high performance, scalability and availability through decades of optimization. Beyond performance, these systems also provide strong guarantees of consistency, durability and reliability through **ACID** (Atomicity, Consistency, Isolation, and Durability) properties, concurrency control and recovery mechanisms. Additional layers such as access control, auditing and encryption further strengthen their security posture. These systems perform exceptionally well in centralized or trusted environments, such as enterprise or cloud infrastructures, where administrative control is well defined. However, in multi-party or inter-organization

collaboration scenarios, the centralized trust model of traditional databases becomes a limitation, as relying on a central administrator creates a single point of failure where a compromised or malicious central authority puts the entire system's integrity at risk. Furthermore, in decentralized environments where a mutually trusted third-party is unavailable, competing organizations may be reluctant to rely on a database controlled by a single participant, necessitating a shift toward more transparent and distributed trust models.

In recent years, blockchain technologies have emerged as an alternative approach to data management, introducing a decentralized model that eliminates reliance on a single authority. Blockchain systems distribute information across multiple nodes and record transactions through a trust model where participants do not need to trust one another to ensure data integrity, immutability and traceability; however, this often comes at the expense of performance and scalability, making blockchain systems less suitable for high-throughput or data-intensive applications.

1.2 Motivation

The motivation behind this work stems from the growing imbalance between data volume, integrity and usability. As organizations increasingly rely on information systems to store and manage critical assets, the consequences of data tampering, loss or breaches have grown severely, often resulting in significant financial, operational and reputational damage [24]. Data tampering can occur in various scenarios, such as multi-tenant cloud environments where multiple users share the same infrastructure, or through insider misuse and system vulnerabilities that compromise storage or access controls, allowing records to be illegally altered, deleted or injected without detection, potentially affecting the reliability of the information used for critical operations.

Blockchain technologies have proven useful in addressing these challenges, as evidenced by applications in areas such as supply chain management (e.g., IBM-Walmart Blockchain Food Safety Alliance [30]) and financial services (e.g., Bitcoin [43]), where trust and transparency are essential. Nevertheless, traditional database and cloud storage systems remain dominant for their performance and scalability, despite offering weaker guarantees of data integrity and trust.

This trade-off between performance and trust motivates the exploration of hybrid storage architectures that combine traditional storage systems with external integrity-preservation mechanisms. Blockchain technologies are one possible approach, but other immutable distributed systems may provide similar integrity guarantees while offering different trade-offs in terms of complexity, scalability and performance. Therefore, this work investigates how an external immutable storage can be integrated with conventional storage systems to provide tamper detection and stronger integrity guarantees without sacrificing the performance advantages of traditional databases and object stores. A system that merges these approaches could address real-world limitations, ensuring that data remains highly accessible and efficient for large-scale operations without forcing users to rely on the assumed honesty of a central authority, thus enabling new possibilities for secure, scalable and efficient data management.

1.3 Problem

Despite their robustness, most blockchain systems are limited by their data model. They are designed to efficiently record small, structured transactions, but are unsuitable for storing large-scale data or high volumes of information. Each node must maintain a full copy of the ledger, making high-volume data storage prohibitively costly and inefficient. This limitation restricts blockchain adoption in applications that require management of large files, such as multimedia content or enterprise records.

Beyond storage, blockchain systems also face critical performance and scalability limitations. Dinh et al. [14] evaluated Ethereum [9, 15], Parity [47] and Hyperledger [23] using the BLOCKBENCH framework, which benchmarks blockchains against database-style workloads. For key-value (YCSB) workloads, Ethereum and Parity achieved peak throughputs of 284 and 45 transactions per second (tx/s), respectively, while Hyperledger reached 1273 tx/s. For transactional OLTP workloads (Smallbank), throughputs dropped to 255 and 1122 tx/s in Ethereum and Hyperledger and remained similar in Parity (46 tx/s). These figures contrast significantly with a traditional in-memory database system such as H-Store [48], which delivered 142 702 tx/s on YCSB and 21 596 tx/s on Smallbank workloads. These results highlight that, although blockchains provide strong integrity and traceability guarantees, their throughput and scalability are orders of magnitude lower than conventional databases.

An alternative approach is the InterPlanetary File System (IPFS) [8, 27], which can also be viewed as an External Immutable Storage technology and mitigates some of the limitations associated with blockchain systems. IPFS is a decentralized peer-to-peer storage network based on content-addressing, where objects are identified through cryptographic hashes rather than storage locations. As a result, any modification to stored content produces a different identifier, enabling integrity verification and supporting immutability guarantees. Furthermore, by distributing content across multiple nodes, IPFS aims to improve data availability and persistence in untrusted environments. Unlike blockchain systems, however, IPFS does not maintain a global transaction ledger or rely on consensus mechanisms, resulting in different performance and scalability trade-offs. Nevertheless, retrieving content from distributed peers may introduce latency and performance variability, while content availability depends on nodes continuing to host the data, often motivating additional pinning mechanisms or even integrations with blockchain itself to strengthen persistence guarantees.

On the other hand, although traditional database systems provide strong security and consistency guarantees in centralized or well-controlled administrative environments through mechanisms such as Role-Based Access Control (RBAC), auditing and encryption, they fall short in decentralized or multi-entity scenarios. In these settings, they lack the inherent immutability, traceability and decentralized trust mechanisms that systems like blockchain offer and that are increasingly required in many real-world applications.

To tackle the limitations of each when used independently, a hybrid and modular approach that integrates traditional storage infrastructures with an external immutable storage can combine their

complementary strengths. By leveraging an external immutable storage to ensure the security, integrity and availability of critical metadata, while offloading high-performance data operations to traditional storage systems and cloud computing services, this architecture aims to satisfy the requirements of diverse modern applications. Such a solution would be particularly valuable for domains operating in multi-party or otherwise untrusted environments that require strong integrity guarantees while maintaining acceptable levels of performance and scalability (e.g., healthcare, supply chain).

Nevertheless, combining these two paradigms also introduces significant challenges. Incorporating external integrity-preservation mechanisms into database environments may lead to a reduction in system performance, as in blockchain-based approaches, for example, the typical sequential transaction processing and consensus protocols of these systems may increase latency. Likewise, not all blockchain capabilities can be seamlessly adapted to a database context, given their fundamentally different architectural principles. Achieving a system that fully preserves the guarantees of both technologies is therefore unlikely, raising an important question about how much of the strengths of each can be effectively integrated without compromising usability or efficiency.

1.4 Research Questions

To address the problem described in the **Problem (1.3)** section, this work aims to explore how traditional database systems and external immutable storage can be effectively integrated into a hybrid architecture. The following research questions guide this exploration:

- RQ1** How can an external immutable storage and conventional storage systems be combined in a hybrid architecture and which types of data or operations are best suited for each component?
- RQ2** How does the integration of external integrity-preservation mechanisms affect the overall performance and scalability of the system compared to traditional storage systems operating independently?
- RQ3** How effectively can a hybrid system enhance protection against data tampering and ensure the integrity of stored information relative to traditional databases?

1.5 Contributions

To address the research questions presented in the previous section, this dissertation explores the integration of conventional storage systems with external immutable storage technologies through a transparent client-side approach. The work combines a state-of-the-art analysis with the design, implementation and evaluation of a modular solution capable of providing tamper detection across heterogeneous storage backends.

The work first presents a state-of-the-art analysis of hybrid integrity-preservation architectures and existing tamper detection approaches, identifying their main characteristics, limitations and research opportunities. Building on these findings, a modular client-side architecture is proposed to introduce transparent integrity verification through the storage access layer, without requiring modifications to the underlying storage systems.

To demonstrate the feasibility of the proposed approach, the architecture is implemented as an extension of Apache OpenDAL using **IPFS** as the External Immutable Storage for integrity anchoring. The resulting prototype provides transparent tamper detection across multiple storage backends and is capable of detecting a variety of tampering scenarios while introducing a measurable performance overhead whose relative impact decreases for larger objects and higher-latency workloads.

Chapter 2

Background

This chapter establishes the theoretical foundations necessary to understand the hybrid architecture proposed in this work. It offers an overview of the fundamental principles of traditional data storage systems and blockchain technologies, contrasting their operational models and highlighting the inherent strengths and limitations of each paradigm to provide the technical context for the modular integration explored in the following sections.

2.1 Data Storage

At its core, a data storage system acts as a specialized software layer that abstracts physical persistence, offering applications a structured framework to store, retrieve and manage information. Modern computing has seen these systems evolve from simple local file structures into sophisticated, elastic infrastructures. While local storage remains the standard for low-latency single-node access, the transition to cloud environments has shifted the paradigm toward a model in which storage is delivered as a service. This evolution enables massive scalability through object storage and managed services, ensuring that large-scale datasets remain highly available and performant even when distributed across global regions.

2.1.1 Data Model

The architecture of a storage system is largely defined by its underlying data model. Traditionally, Relational Database Management Systems (**RDBMSs**) have dominated the landscape by organizing data into strict two-dimensional tables that favor consistency and complex querying capabilities. However, as noted by Wang, Hsieh, and Li [58], the demands of big data have exposed **RDBMS** bottlenecks in I/O throughput and horizontal scalability. To address these challenges, many systems have moved toward Non-Relational (**NoSQL**) models. Unlike their relational predecessors, **NoSQL** databases often prioritize horizontal scalability and flexible schemas over strict consistency. According to Ge et al. [19], this choice represents a fundamental trade-off between integrity and flexibility since while **SQL** systems enforce data validity through predefined schemas and foreign keys, **NoSQL** allows for rapid development at the risk of storing invalid data.

This architectural shift allows developers to choose a specialized model optimized for the specific requirements of their data structure. For instance, *Key-Value Databases* utilize a hash table structure where each data value is uniquely identified by a specific key, offering outstanding performance for simple operations but lacking support for complex joins (e.g., Redis [54]). Ideally suited for semi-structured information, *Document-Oriented Databases* store data as documents, typically in formats like JSON or XML, allowing each entry to have different attributes (e.g., MongoDB [41]). For analytical workloads, *Columnar Databases* group related data into column families rather than rows, improving efficiency across vast datasets (e.g., Apache HBase [6]). For datasets where the interconnections between entities are the primary focus, such as social networks, *Graph Databases* represent data as nodes and relationships as edges (e.g., Neo4j [44]). Finally, *Time Series Databases* are optimized for handling sequences of data points recorded over time, providing efficient storage for telemetry and monitoring metrics (e.g., InfluxDB[26]).

Although **NoSQL** systems offer the horizontal scaling that **RDBMS** lacks, they frequently compromise on the transactional guarantees required for critical data. This gap led to the development of NewSQL databases, which Zhu et al. [66] briefly discussed as a solution that combines the high scalability of **NoSQL** with the **ACID** characteristics of traditional relational systems.

2.1.2 Performance and Security

The reliability of these systems is governed by the mechanics of a consistent transaction management. A transaction is a logical unit of work that must be processed atomically (it is either completely successful and committed or aborted otherwise). To ensure integrity, traditional databases adhere to the **ACID** properties: Atomicity, Consistency, Isolation and Durability. This process involves complex concurrency control mechanisms, such as locking or Multi-Version Concurrency Control (**MVCC**), to prevent data corruption when multiple users access the system simultaneously, therefore ensuring isolation. A vital part of this process is the database log, commonly known as a Write-Ahead Log (**WAL**). Before any change is permanently applied to the data files, it is first recorded in the log. This ensures that if a system crash happens, the database can rollback the log to recover its state, providing the fundamental guarantee of durability.

Performance within these systems is mainly achieved through optimized indexing and caching techniques. An index is a supplemental data structure that allows the storage engine to locate data without scanning every record on disk. For example, B+ Trees are highly optimized for read-heavy relational workloads, while Log-Structured Merge (**LSM**) Trees are often used in **NoSQL** systems to handle high-speed write operations. These methods are often complemented by caching layers, which store frequently accessed data in high-speed RAM to avoid the latency of disk I/O. By balancing these structures, storage systems can maintain sub-millisecond response times even as datasets grow to large scales.

In controlled enterprise environments, security is managed through a combination of encryption and access controls. Data are typically encrypted both while it is stored and while it is moving across a network. These systems operate under a trusted administrator model, where security policies such as Role-Based Access Control (**RBAC**) ensure that users only see the data they are

authorized to access. Since these environments are centralized, the system relies on a single source of truth for identity and permissions, assuming that the underlying infrastructure and its administrators act responsibly to maintain the integrity of the **ACID** properties.

2.1.3 Distributed Databases

As storage requirements grow, systems transition into distributed architectures to achieve high availability and scalability. A Distributed Database System (**DDBS**) physically stores data across different nodes, each managed by a local management system, while logically appearing as a single centralized system to the user [58]. This is typically accomplished through sharding and replication. Sharding involves partitioning the dataset so that different nodes manage distinct subsets of data, while replication copies information across multiple nodes to ensure the system remains operational if a node fails.

The distribution of data introduces fundamental trade-offs described by the **CAP** Theorem. This principle states that during a network partition, a system must choose between Consistency (all nodes seeing the same data) and Availability (every request receiving a response). Consequently, many distributed systems move away from strong coherence toward eventual consistency, where updates propagate across the network over time. As Ruan et al. [55] note, this is often the most lenient model, making fewer guarantees about the specific order of read and write operations to maintain availability. Despite these benefits, traditional distributed designs often rely on master-slave architectures, which Wang, Hsieh, and Li [58] identify as vulnerable to single points of failure and security issues like Sybil attacks, where malicious nodes can cause the entire cluster to collapse.

2.2 Blockchain

Blockchain technology represents a decentralized and distributed ledger paradigm that enables the secure recording of transactions across multiple nodes without the need for a central authority. At its core, a blockchain is composed of a continuous sequence of blocks, where each block contains a header and a body. The header typically includes metadata such as a timestamp, a nonce used in consensus and a cryptographic hash of the previous block, which creates a chronological and unbreakable link between blocks. The block body contains the actual transaction data, often organized in a Merkle tree structure to facilitate efficient verification.

2.2.1 Data Integrity and Traceability

The architectural design of blockchain provides inherent guarantees regarding data integrity and history. Immutability is achieved through the use of cryptographic hashing, i.e., as all blocks are cryptographically linked, any alteration to the data within a block would change its hash and consequently invalidate all subsequent blocks in the chain. Beyond its immutable and irreversible guarantees, blockchain establishes robust data provenance through its transaction-based storage

process. Every transaction must be digitally signed using public key cryptography, which ensures the authenticity of the data source and provides a strong instrument for non-repudiation and traceability [11]. This structure allows any participant to verify the entire history of a specific data asset from its genesis block to its current state, ensuring that the origin and movement of information remain verifiable in multi-party environments where trust is limited.

2.2.2 Decentralization and Consensus Mechanisms

Unlike traditional databases that rely on a central administrator, blockchain systems achieve decentralization by replicating the ledger across all participating nodes. This distributed architecture provides inherent robustness and fault tolerance. By removing the single point of failure characteristic of centralized systems, the network remains resilient against attacks like Denial of Service (DoS) and Distributed Denial of Service (DDoS), as the remaining nodes can continue operating even if specific participants go down or are compromised [11].

To maintain a single, consistent state across the network, these systems employ consensus mechanisms. A primary challenge in these decentralized environments is the double-spending problem, where a user attempts to spend the same digital asset twice. Maiyya et al. [39] explains that Nakamoto consensus, utilized by Bitcoin, addresses this by using a network of miners to achieve serializability. Miners must solve a computationally intensive Proof of Work (PoW) puzzle to add a new block, ensuring that transactions are verified and ordered correctly. If two miners solve the puzzle simultaneously, a fork occurs in the chain. The protocol resolves this through the longest chain rule, where miners join the branch with the most accumulated work, effectively dropping shorter branches.

Beyond Nakamoto consensus, replication protocols are generally categorized by their failure models. As Ruan et al. [55] details, Crash Fault Tolerant (CFT) protocols handle hardware or software failures but assume nodes are honest, requiring $2f + 1$ replicas to tolerate f failures in asynchronous networks. In contrast, Byzantine Fault Tolerant (BFT) protocols handle arbitrary or malicious behavior, requiring $3f + 1$ replicas. A prominent BFT implementation is the Practical Byzantine Fault Tolerance (PBFT) algorithm, which optimizes resilience for asynchronous environments [39]. Unlike the probabilistic finality found in Nakamoto consensus, PBFT ensures agreement through structured communication rounds between a predefined set of participants.

The choice between these models significantly impacts system efficiency. Ge et al. [19] notes that BFT consensus is considerably slower than CFT, with throughput gaps ranging from 5 to 500 times. Consequently, CFT protocols are preferred for high performance scenarios like Online Transaction Processing (OLTP), while BFT is reserved for high security requirements such as digital asset storage. Furthermore, BFT protocols exhibit higher performance variance as the network scales. Since they require a larger number of participating nodes to reach agreement and incur higher network communication costs ($O(N^2)$ compared to $O(N)$ for CFT), they are more sensitive to real world networking conditions [55].

The combination of cryptographic linking and decentralized consensus provides robust anti-tampering guarantees. Because the ledger is distributed, an attacker would need to compromise a

majority of the network (often 51% in permissionless settings) to alter recorded information. This makes the cost of tampering prohibitively high compared to centralized systems where a single compromised account could lead to undetected data manipulation. However, as noted by Maiyya et al. [39], if a group of malicious actors gains control of 51% of the network's mining power, they can manipulate the ledger (*51% Attack*). This level of control allows them to overwrite the existing history of the blockchain and reverse transactions that were previously considered permanent, effectively undoing even those that were deeply buried in the chain.

2.2.3 Permissioned and Permissionless Blockchains

Blockchain systems can be broadly classified based on their access control models. Permissionless (or public) blockchains allow any node or user to join the network, participate in consensus and read the ledger. Examples such as Bitcoin and Ethereum make no identity assumptions, which allows them to achieve a higher degree of security and decentralization, though often at the cost of performance due to the necessity to employ resource intensive mechanisms to protect against malicious actors. For instance, Bitcoin is limited to approximately 7 transactions per second, largely due to the risk of forking and the subsequent need to wait for multiple block confirmations to reach finality [39].

In contrast, permissioned (or private) blockchains require nodes and users to be authorized to join the network. This restricted environment is particularly effective for enterprise settings, like the financial sector, where participants are known and their actions are regulated [55]. Because these systems operate with a known identity assumption, they can utilize more efficient consensus protocols that deliver higher performance and lower latency.

As illustrated by Ruan et al. [55], there is a clear trade-off between security and performance in these designs. Permissionless blockchains prioritize robust security and open participation, while permissioned blockchains lean toward higher throughput and efficiency by restricting participation to a set of known and identified entities.

Chapter 3

State of The Art

While the mechanisms of traditional storage are highly effective in private high-performance environments, they encounter significant limitations in multi-party or trustless scenarios. These systems are perfectly adequate for internal enterprise applications, such as a company inventory management system or a private research database, where data is generated and consumed within a single administrative boundary. In such contexts, the organization maintains full control over the infrastructure, and the primary focus is on operational efficiency rather than external verification. They also remain the most efficient choice when a trusted third party, such as a commercial bank, provides the necessary central oversight to handle high-speed transactions at scale. In these cases, the maturity of established database engines offers a level of performance and throughput that decentralized alternatives cannot yet match.

However, traditional storage falls short in environments where no such central trust exists, such as global supply chain tracking or settlements between competing entities. The primary shortcoming in these contexts is its inherent centralization of the system, since data integrity is only as reliable as the entity that controls it. If an administrator or a cloud provider is compromised or malicious, they can alter records and manipulate the database logs to hide their tracks.

Furthermore, the same techniques used to improve performance, such as caching and eventual consistency, can create brief periods where the overall state of the system is inconsistent. Because these models prioritize availability, some nodes may temporarily hold outdated information while updates propagate across the network. Although these systems excel at managing and presenting the current state of data, they struggle to provide a mathematically verifiable or immutable history of how that state was reached. Without a trusted central authority to guarantee the integrity of the records, traditional systems cannot offer the verifiable immutability needed to prove to an external party that the data has remained untampered since its insertion.

At the same time, while blockchain offers unique security benefits, several core technical limitations remain significant barriers to its adoption in high-performance environments. The most immediate constraints concern performance and scalability. Blockchain systems face significant throughput and latency constraints due to the overhead of reaching consensus and the requirement for every participating node to process every transaction. As network decentralization increases,

the rising communication complexity further hinders the ability of the system to scale effectively. Hong, Sun, and Chen [22] highlights that the typical single-chain ledger structure serves as a major bottleneck, as it forces all transactions to be processed sequentially from a global perspective. This inherent lack of parallelism prevents the technology from meeting the high concurrency requirements of complex environments where transaction volumes are high.

Compounding these processing constraints is the issue of storage overhead. The standard requirement for nodes to maintain a complete history of all transactions leads to a massive and ever-increasing data load. Wei et al. [62] explains that the use of ADSs for data verification incurs significant storage overhead due to frequent serialization, hashing operations and random I/O. Consequently, storing large files directly on the blockchain is highly inefficient, as the total data size grows linearly with every new transaction and never decreases.

Beyond resource consumption, current blockchain implementations suffer from functional deficits regarding their querying and storage models. A major defect in current blockchain systems is the lack of a complete and efficient query engine. Most standard implementations rely on LSM-Tree-based key-value stores, which are designed to prioritize write performance at the expense of read performance [62]. However, many real-world blockchain applications are read-intensive, requiring complex queries that unstructured key-value models cannot easily support. This mismatch between the underlying storage strategy and the application workload results in low query efficiency and limited interface capabilities. Furthermore, there is persistent transaction uncertainty regarding the status of pending data. As explored by Cohen, Rosenthal, and Zohar [12], this occurs because transactions are often interdependent. They may conflict by attempting to use the same input (e.g., double-spend attack) or they may depend on the prior confirmation of a parent transaction. Since these rules prevent all transactions from being accepted at once, the system effectively exists in a state of multiple “possible worlds” regarding its future ledger. This makes it difficult to guarantee which specific combination of transactions will ultimately be finalized and which will be discarded.

These technical limitations dictate where blockchain technology can be safely and effectively applied. As analyzed by Chowdhury et al. [11], blockchains can be highly effective in sectors such as supply chain management, inter-bank transactions and healthcare, where it establishes trust among multiple parties without the need for a central authority. However, the author also notes that its application can be problematic in critical environments, such as healthcare, where the inherent scalability issues of blockchain could delay a doctor’s access to vital information during a crisis, leading to catastrophic consequences. Therefore, the decision to implement a blockchain must be weighed against the specific performance and legislative needs of the target domain.

Furthermore, blockchain may not be the ideal choice for scenarios with strict privacy or resource constraints. Zou et al. [68] points out that the inherent transparency of blockchain can create privacy issues, as sensitive information can be exposed to unauthorized participants or attackers. The author also highlights that blockchain is often ill-suited for environments with low storage and computing capacity, such as the Internet of Things (IoT) or fog computing layers. Similarly, Big Data environments are not particularly suited for blockchain due to the massive

volume of information involved. In these contexts, the traditional requirement for every node to store a full copy of the ledger and participate in resource-intensive consensus becomes a major bottleneck, necessitating the development of lightweight storage and consensus schemes to remain viable. Consequently, blockchain fits poorly in applications requiring high data confidentiality or those involving massive data volumes that exceed the storage and processing capacities of typical network participants.

3.1 Hybrid Systems

Based on the capabilities of data storage systems and blockchain described in the [Background \(2\)](#) and their inherent limitations previously detailed, there is a growing trend in both industry and academia to move beyond single-paradigm systems by integrating both technologies. Hybrid architectures aim for a “best of both worlds” approach, seeking to develop systems that are performant while also ensuring data integrity and traceability.

Evidence for the necessity of integrating both systems is highlighted by the gaps found in the decision tree proposed by Chowdhury et al. [11], which aims to help determine when to use blockchain or database based on trust requirements but fails to address systems requiring both high performance and immutability, as neither paradigm is sufficient on its own. The potential for hybrid systems to bridge this gap is further supported by the empirical findings of Wang, Hsieh, and Li [58], whose benchmarks demonstrate that systems integrating blockchain features with distributed database backends can achieve significantly lower latency and higher throughput in both single record and fuzzy batch queries compared to traditional distributed engines like Hive or HadoopDB.

The following sections explore the current landscape of such integrations by reviewing different implementations and conceptual frameworks found in the literature. An interesting inspiration for this analysis is the blockchain-database spectrum proposed by Zhu et al. [66], which classifies fusion systems into database-oriented blockchains, blockchain-oriented databases and hybrid systems. However, to better distinguish between specific implementation strategies and technical specializations found in the selected literature, the state of the art is categorized into four distinct sections: [Blockchain-supported Databases \(3.2\)](#), [Database-supported Blockchains \(3.3\)](#), [Optimized Authenticated Querying for Blockchain Data \(3.4\)](#) and [Hybrid Distributed File-Storage Blockchain Systems \(3.5\)](#).

3.2 Blockchain-supported Databases

Systems within this category are primarily traditional databases that have been enhanced with blockchain capabilities to introduce integrity and anti-tampering guarantees. Termed Blockchain-supported Databases within this work, this architectural pattern is frequently described in the

literature using similar terminology, such as Out-of-the-database Blockchains [55], Blockchain-oriented Databases [66] or Blockchain-assisted Databases [62]. Implementation strategies generally involve storing a ledger within the database tables themselves, utilizing an external blockchain to anchor cryptographic hashes of the stored records or deploying middleware layers designed to intercept database communications to verify data using a ledger. By integrating these blockchain features, these architectures aim to provide a verifiable data store that preserves the high performance and rich indexing capabilities of conventional storage engines while gaining the auditability and tamper evidence typical of Distributed Ledger Technologies (DLTs).

One of the most common integration strategies involves the use of middleware or external frameworks to secure existing database structures. Rani and Sharma [53] propose a framework for detecting tampering in distributed databases by utilizing blockchain technology to track transaction metadata at each node. By establishing “trust boundaries”, the system requires users to pass access control and validation protocols before interacting with nodes, while the blockchain serves as a verifiable record to ensure the underlying database has not been compromised. Similarly, Lian, Wang, and Xie [34] introduce TDRB, a middleware that intercepts SQL commands to store row-level and column-level hashes on a blockchain. This approach allows for the detection of illegal updates, inserts or deletions by verifying the current state of the database before any operation is executed, using an AES-encrypted cache to maintain performance.

Expanding on the middleware approach to include recovery mechanisms, Muzammal, Qu, and Nasrulin [42] introduce ChainSQL, an open-source middleware that bridges user applications with distributed databases by recording signed transactions on a Ripple-based blockchain. While data is stored in local SQL or NoSQL engines for performance, every update is first validated by the blockchain network and then automatically applied to the local replicas. This architecture provides a multi-disaster recovery mechanism where continuous synchronization between production and backup nodes ensures zero-recovery-time. Additionally, the system supports fine-grained permissioning, allowing data owners to grant auditors read-only access to verify the current database state against the immutable blockchain logs.

In the case of the DEPLEST approach, Chen et al. [10] uses a middleware layer to filter and isolate sensitive user information from centralized storage. The architecture classifies data upon arrival: insensitive records are routed to a traditional distributed database, while sensitive data is encrypted via SHECC (Stochastic Homomorphic Elliptic Curve Cryptography) and stored on a decentralized blockchain that utilizes a novel Proof-of-Communication (PoC) consensus protocol. To maintain performance on resource-constrained client devices, the system implements a partial ledger technique where nodes only store a fraction of the global state proportional to their hardware capacity.

Several studies have tailored these integrity concepts specifically for cloud environments. Liang et al. [35] propose ProvChain, a system that collects data provenance using listeners and hooks to monitor changes in an ownCloud instance. These provenance records are hashed and anchored to the Bitcoin blockchain, allowing a Provenance Auditor (PA) to validate data records through “blockchain receipts” obtained via the Tierion API. This architecture preserves user pri-

vacy through hashing and provides high availability by leveraging the immutability of a public ledger to track the history of cloud records. To further address the integrity of operational logs in cloud environments, Gaetani et al. [17] and Aniello et al. [3] propose and evaluate a two-layer architecture designed to protect redo logs from unauthorized manipulation. The first layer consists of a Cloud Federation where each member maintains a database replica, a local ledger and a miner, while the second layer provides an external Proof of Work (PoW) blockchain used for periodic anchoring of the first layer's state. Transactions are processed via a Mining Rotation and a Three-Phase Commit (3PC) protocol, requiring all miners to sign an operation before it is committed to the local ledgers and executed on the database replicas. At the end of each leader's round, a "witness transaction" containing the current state hash is sent to the Layer 2 blockchain for immutable anchoring. A significant characteristic of this approach is its resistance to collusion due to the total consensus for every operation in Layer 1, which ensures that the system maintains integrity even if only one miner remains honest, although sacrificing overall fault tolerance.

Taking a different approach to cloud security, Awadallah and Samsudin [7] present "BC over cloud-RDB", a hybrid architecture that simulates blockchain properties within relational structures to mitigate internal threats. The system utilizes Homomorphic Encryption and distributes data across at least four independent Cloud Service Providers (CSPs) to achieve consensus via BFT. By comparing cryptographic signatures generated by each CSP, the client can verify database integrity without relying on a single provider. The proposal includes two distinct modules: an "Agile BC-based RDB" for high-performance manipulation detection and a "Secure BC-based RDB" that employs PoW to avoid tampering through significant computational costs.

Moving toward more natively integrated architectures, Antonopoulos et al. [5] present SQL Ledger, where "Ledger Tables" are managed within an Azure SQL Database. These tables can be append-only or updateable, in which case a "History Table" is automatically created to store previous versions of modified or deleted records. The system serializes and hashes database transactions and historic data into a Merkle tree, storing the resulting digests in a "Database Ledger" that mimics blockchain properties by keeping block details and block transactions in relational tables, while anchoring these digests to external immutable storage (e.g., blockchain, trusted blob storage) for formal verification. The system relies on the concept of "Forward Integrity", which assumes that tampering does not occur during an operation, meaning a transaction is committed successfully even before its digest is generated, thus providing integrity for all future verification checkpoints.

Similarly, Yang et al. [64] introduce LedgerDB, a centralized ledger database that replaces decentralized consensus with a Timestamp Authority (TSA) to ensure auditability through a Two-way Peg Protocol. The system follows an execute-commit-index pattern where signed payloads from clients are stored as journals in a storage layer. To facilitate efficient auditing, the platform uses specialized structures including Clue Indices based on skip lists for user-specified labels and a batch Accumulated Merkle Tree (bAMT) for existence proofs. Notably, LedgerDB allows "Purge" and "Occult" operations to remove obsolete records or hide data, intentionally departing from strict blockchain immutability to meet practical database management requirements while

maintaining a verifiable chain of journal blocks.

To address the performance overhead often associated with such local ledgers, Ifrim, Loghin, and Popescu [25] introduce Baldur. This hybrid database offloads cryptographic overhead to Field Programmable Gate Arrays (FPGAs) or Graphics Processing Units (GPUs) to maintain high performance. The system stores data in a key-value engine and records updates in a local ledger, using MVCC to ensure version consistency. To minimize latency, Baldur employs hardware acceleration for Keccak hashing and ECDSA (Elliptic Curve Digital Signature Algorithm) signature verification, utilizing techniques like “Half Point Multiplication” on FPGAs or parallel CUDA threads on GPUs. This hardware-centric approach allows the database to provide blockchain-level integrity guarantees without the typical Central Processing Unit (CPU) bottlenecks found in software-only implementations.

Several architectures focus on collaborative environments and specific domain applications. Peng et al. [49] describe FalconDB, which utilizes a three-layer architecture (Database, ADS and Blockchain) to provide collaborative storage between “Servers”, which store the full database and ledger, and “Clients”, which maintain only block headers. It features a unique mechanism where records are never deleted but instead assigned “valueFrom” (VF) and “valueTo” (VT) attributes representing the block height of its validity. Updates are handled by setting the VT of the current record to the current block height and creating a new record, an approach that ensures traceability and enables “time-travel” queries. While results are returned immediately for performance, Clients can request cryptographic proofs of query honesty from Servers for a computational fee.

Finally, regarding specific use cases, Maheshwarkar, Kumar, and Gupta [38] propose a framework for securing database backups by combining traditional relational engines with IPFS and blockchain. In this architecture, standard transactions are executed on the database and their corresponding logs are committed to a blockchain for auditability. Periodically, the system generates a full SQL export of the database which is stored as chunks in IPFS. The resulting Content Identifier (CID) is then recorded within a specialized “backup” block on the ledger, enabling a verifiable recovery process where the system retrieves the latest valid state from the decentralized file system to restore the database. In the supply chain domain, Preto et al. [52] developed a system for supply chains that combines MongoDB with Hyperledger, signing all data entries with private keys and storing only metadata and hashes on-chain to ensure non-repudiation and data integrity for off-chain documents. For Electronic Medical Records (EMRs), Li and Wang [32] propose a hybrid storage model where full medical records are stored within blockchain blocks, while a corresponding database table maintains metadata and a “mac” field that acts as a logical pointer to the physical location of the full record on the ledger. To facilitate efficient retrieval, the system uses a multi-level indexing strategy involving B+tree and Bitmap indexes within the database layer. Integrity is maintained through a verification process that cross-references database results with the blockchain Merkle tree to ensure the underlying data has not been tampered. Lastly, regarding authorization, Feng et al. [16] propose VDAC, a hybrid scheme in which a Hyperledger Fabric blockchain serves as a decentralized authorization agency to eliminate single points of failure in distributed databases. By integrating CP-ABE (Ciphertext-Policy Attribute-Based Encryption),

the system stores encrypted access parameters and database connection metadata on the ledger, requiring users to possess specific attributes to decrypt authorization tokens.

3.3 Database-supported Blockchains

The architectures in this classification consist of fundamental blockchain systems that incorporate traditional database design principles primarily to mitigate inherent performance bottlenecks and privacy concerns. Referred to in this work as Database-supported Blockchains, these systems are also identified in existing studies as Database-oriented Blockchains [66] or Out-of-the-blockchain Databases [55]. Common implementation strategies include modifying the internal data structure of blocks to facilitate more efficient querying, adding sophisticated indexing layers or replacing standard Merkle trees with optimized structures such as accumulator-based ADSs. Other approaches focus on overcoming the limitations of sequential transaction processing, employing encryption to protect sensitive data or utilizing external database engines to manage large volumes of off-chain information. Furthermore, some systems introduce heterogeneous node roles, mirroring the architecture of Distributed Database Systems (DDBSs) where not every participant is required to maintain a full copy of the ledger. The ultimate goal is to retain the decentralized and trustless nature of the blockchain while integrating database concepts to enhance the overall performance, scalability and privacy of the system.

A primary method of achieving this integration is through the direct embedding of relational structures into the ledger. Zhu et al. [67] introduce SEBDB, which enhances blockchain systems with relational capabilities by storing transactions as tuples in predefined tables according to their type. By incorporating SQL-like interfaces and on-chain indices (such as B+ trees, Bitmap indices and layered indices), the system optimizes block scanning and allows developers to manage on-chain data with traditional database ease. Additionally, it supports “On-Off Joins” that combine immutable on-chain data with private, off-chain local storage. Still within this relational integration, Wang et al. [59] introduce aChain, an analytical blockchain designed to support SQL-based Online Analytical Processing (OLAP) services while maintaining core decentralized properties. The system maps on-chain data into specialized relational tables within a MySQL engine, including dedicated tables for block headers, transactions and user-defined transaction attributes to facilitate fine-grained retrieval. To ensure traceability and state management, aChain maintains both “State” and “History” tables. The architecture follows an Execute-Order-Validate (EOV) workflow where transactions are simulated locally by peers before being ordered into blocks using consensus protocols and validated. Integrity is maintained by allowing users to reconstruct Merkle trees from the transaction table and compare the resulting roots against the values stored in the block header table.

While some systems modify the internal ledger, others utilize external database mirrors to handle high-performance read operations. Şafak, Mendi, and Erol [56] propose a hybrid architecture that utilizes a middleware layer to synchronize a Hyperledger Fabric network with a MongoDB NoSQL database in real-time. Users interact with the MongoDB layer to perform

high-speed queries on mirrored block content while the underlying blockchain maintains the immutable source of truth. The systems are synchronized periodically as a whole to ensure equal states.

Beyond storage and querying, database principles are also applied to the transaction pipeline itself to resolve performance bottlenecks. Sharma et al. [57] introduce Fabric++, an enhancement of Hyperledger Fabric that “databasifies” the transaction pipeline to increase throughput by 12 times. The system moves away from simple arrival-time ordering by constructing a conflict graph of read-write sets and using Tarjan’s and Johnson’s algorithms to reorder transactions, thereby breaking cycles and maximizing block validity. Additionally, Fabric++ replaces the standard “late abort” policy with an early abort strategy that identifies cross-block conflicts with ordered but uncommitted transactions, preventing the waste of resources on doomed transactions.

Implementing a modular approach to scalability, El-Hindi et al. [21] present BlockchainDB, a system that layers traditional distributed database properties over existing blockchain networks. The architecture features a Database Layer containing an Off-chain Verifier, a Transaction Manager and a Shard Manager, which coordinates requests across a Storage Layer composed of multiple independent blockchain shards. To optimize resource utilization, the network distinguishes between “Full peers”, which store a replica of at least one shard, and “Thin peers”, which participate in the network without storing data by connecting to remote peers. Data within shared tables are partitioned into shards, each forming its own blockchain network, thereby avoiding the requirement for every node to maintain a full copy of the entire ledger. The system supports various consistency levels and provides both online and offline verification modes: online verification ensures immediate integrity at the cost of throughput and offline verification batches requests into epochs to be verified asynchronously, allowing for significantly higher performance.

3.4 Optimized Authenticated Querying for Blockchain Data

The specialized architectures reviewed here focus primarily on the performance and verifiability of queries, prioritizing efficient read operations and quick verification over general storage management. While these systems share similarities with Database-supported Blockchains (3.3), they are distinguished by their focus on the authenticated query problem: ensuring that results retrieved from the ledger are both complete and untampered. Implementation typically involves the deployment of dedicated query layers and auxiliary Authenticated Data Structures (ADSs) that sit alongside the blockchain. Although some of these works could theoretically be classified as Database-supported Blockchains, their specific focus on retrieval efficiency warrants a dedicated section. In cases where a system significantly addresses both storage and retrieval, it is analyzed within this section to highlight its specialized query capabilities. These approaches are particularly vital for light clients and third-party applications that require complex search functionalities over blockchain history without the overhead of downloading the entire ledger or the risk of trusting a single centralized provider.

A significant advancement in this field involves re-engineering the blockchain's internal verification structures to support complex search criteria. Xu, Zhang, and Xu [63] propose vChain, a system that introduces a three-party model involving Miners, Service Providers (SP) and Query Users to enable verifiable boolean range queries. Instead of traditional Merkle trees, vChain implements an RSA accumulator-based ADS in each block header, which provides a mathematical summary of the block (*AttDigest*) and allows for dynamic aggregations. This is complemented by specialized indices: a Merkle tree-based intra-block index for clustering similar objects, a skip list-based inter-block index that allows skipping irrelevant block intervals and an Inverted Prefix Tree for processing subscription queries in batches. During retrieval, the Service Provider returns a Verification Object (VO) containing the query results and a Mismatch Proof to confirm that no other records satisfy the criteria. The Query User then performs a mathematical comparison between the received VO and the local *AttDigest* to ensure the integrity and completeness of the result. Although this approach modifies the blockchain structure, its primary objective remains the optimization of query performance and verification speed.

While vChain modifies the ledger structure, other approaches focus on decoupling the query layer through modular middleware that synchronizes with the main chain. Li et al. [33] introduce EtherQL, a modular query layer designed specifically for the Ethereum blockchain. The system functions as a specialized middleware that synchronizes with the network via a Sync Manager, which utilizes a Block Cache to handle chain forks and ensure data stability before committing records to external storage. Once a block is finalized, a Handler Chain extracts specific metadata into block, transaction and account entities, which are then stored in a MongoDB database through a dedicated Persistence Framework. By leveraging the indexing capabilities of a document-oriented database to support efficient range and top-k queries, EtherQL provides developers with a high-performance interface for interacting with blockchain data via native APIs or RESTful services without modifying the underlying ledger protocol.

Similarly, Peng et al. [50] propose VQL, a middleware layer designed to provide efficient query processing while maintaining data authenticity. The system extracts Ethereum blocks and stores them in a MongoDB-based middleware that organizes data into daily "Micro Databases" which are periodically merged into monthly "Key Databases" to optimize storage space. To ensure the integrity of the off-chain data, VQL implements a verification process where the database fingerprint published by the middleware must match the hash calculated from the database contents and a third hash generated by miners who reconstruct the database using their local ledger. Once verified via consensus, the database fingerprint is recorded on the blockchain, providing a proof of authenticity that users can rely on for efficient querying.

Finally, Liu, Li, and Jin [36] propose an auxiliary storage scheme for Hyperledger Fabric that utilizes a MySQL database to enhance query performance. The system organizes data into two primary tables: a Transaction Table for record-keeping and a Hash Table that maintains a chain of digests where each entry represents the hash of current transaction data combined with the preceding record's digest. A distinctive feature of this architecture is the use of "trustworthiness labels" (common, honest or doubtful) assigned to records during regular verification intervals.

When a user executes a query, the system prioritizes the auxiliary database; if a record is missing or flagged as “doubtful”, it falls back to the blockchain for retrieval, subsequently updating the local database with verified results to accelerate future access.

3.5 Hybrid Distributed File-Storage Blockchain Systems

Addressing the constraints of on-chain storage, these systems integrate blockchain with decentralized, content-addressed file systems such as the **IPFS**. The most common pattern involves a hash-on-chain and data-off-chain model, where files or raw data are stored in a distributed file system while only the resulting cryptographic hashes and metadata are recorded on the ledger. This specialization is distinct because it accommodates a wider scope of data formats, including documents, images and videos, at a scale that would be technically or economically unfeasible to store directly on a blockchain. This section explores how these hybrid models manage data availability and links the performance of distributed storage with the immutability and security of the blockchain. Empirical evidence suggests that this integration can yield up to 80% improvement in file upload performance compared to traditional blockchain [29], effectively reducing average system delays by offloading large data objects to a peer-to-peer (**P2P**) file-storage system.

Making use of this hash-on-chain and data-off-chain pattern, Anthal, Choudhary, and Shettiyar [4], Zheng et al. [65] and Kumar and Tripathi [31] leverage the integration of blockchain with InterPlanetary File System (**IPFS**) to overcome storage limitations. Anthal, Choudhary, and Shettiyar [4] utilize Ethereum smart contracts to manage a decentralized file-sharing environment for various formats like documents and images. Zheng et al. [65] focus on the Bitcoin network to reduce ledger size and synchronization time by storing the transactions themselves on **IPFS**, with miners relying on local transaction pools for consensus in order to only fetch records from the distributed file system when missing locally. Similarly, Kumar and Tripathi [31] address the storage pressure caused by the continuous growth of transaction data by replacing bulky records on-chain with fixed-length 46-byte hashes, highlighting a model where miners are responsible for storing transactions on **IPFS** and retrieving the corresponding hash before indexing it on the ledger. These systems eliminate single points of failure and leverage the splitting of large data into chunks for proximity-based retrieval of **IPFS** to ensure high availability and performance, although they both currently lack an incentive or penalty system to guarantee persistent content availability like that found in Filecoin.

To further optimize the efficiency of these hybrid interactions, Alizadeh, Andersson, and Schelén [2] propose a modular approach that introduces an intermediary “Agreement Service” to coordinate data exchange. Unlike simpler models that require multiple direct blockchain interactions, this architecture uses a specialized service to communicate with the ledger via a RESTful API. This allows the system to batch processes and complete blockchain updates in a single transaction, significantly reducing the “referral time” and the high gas costs typically associated with public blockchains like Ethereum.

Gazsi et al. [18] present VAULT, a scalable protocol that combines a permissioned blockchain with IPFS to facilitate secure data collaboration. In this system, encrypted files are stored in IPFS while the ledger records associated CIDs, File Identifiers and timestamps. VAULT distinguishes itself through a rigorous membership and governance model where new nodes must be validated by a Certificate Authority (CA) and existing members before joining the network. Furthermore, the protocol manages collaborative editing by requiring 51% approval from current owners to update a file pointer and utilizes a “key-wrap” mechanism to securely grant data access. To maintain performance, the system employs a quorum-based consensus where nodes are randomly selected for mining tasks based on the preceding block’s hash.

Expanding these hybrid principles to secure logging, Liu et al. [37] propose an audit log system that utilizes Hyperledger Fabric and IPFS to manage large volumes of log data while reducing blockchain storage overhead by 50%. In this architecture, logs are encrypted and stored in IPFS, while the resulting Content Identifiers (CIDs) are recorded on the blockchain as sub-NFTs, allowing non-trusted auditors and a supervisory “Court” entity to verify log integrity by comparing locally calculated hashes against the immutable records on the ledger.

To address the lack of traceability in native IPFS, Nyalety et al. [45] introduce BlockIPFS, which uses Hyperledger Fabric to log metadata and data accesses. For every file stored or accessed, the system records the CID, ownership details and timestamps on the ledger, creating a forensic audit trail that allows authorized users to trace the complete history of content interactions.

3.6 Discussion

The comparative analysis presented in Table 3.1 reveals a significant diversity in how blockchain and database technologies are integrated. Starting from a traditional database system, the predominant trend in *Blockchain-supported Databases* (3.2) is the use of a middleware and/or external blockchain to bridge existing systems. This “out-of-the-database” approach allows organizations to retain high-performance storage engines while anchoring hashes or transactions to an external ledger for tampering detection. Conversely, alternative approaches aim to improve traditional blockchain systems by incorporating database characteristics into the DLT architecture itself (3.3), enhancing query capabilities through specialized authenticated structures or retrieval layers, or scaling storage via external decentralized file-storage systems like IPFS, as detailed in the sections on *Optimized Authenticated Querying for Blockchain Data* (3.4) and *Hybrid Distributed File-Storage Blockchain Systems* (3.5) respectively.

Despite these advancements, several critical gaps remain in the current state of the art. Competing with traditional centralized systems’ performance remains a challenge for several systems [7, 16, 25, 34, 37, 45, 52, 53, 57, 59, 67] as integrating blockchain with databases introduces significant latency (especially for write operations and consensus) and reduces throughput. Additionally, many hybrid systems currently rely on simple Key-Value (KV) stores or limited indexing [21, 32, 33, 42, 49, 52, 57, 63, 67], making complex data retrieval (e.g., joins, nested queries, aggregations) difficult or inefficient compared to standard SQL databases. Scalability is also a gap

Table 3.1: Comparison of Hybrid Blockchain-Database Architectures

Category	System Goal	Integration approach	Storage method	Target Problem	Studies
Blockchain-supported Databases	Add data integrity while maintaining performance	Middleware or External Blockchain	Data off-chain, hash on-chain	Tampering detection	[52, 53], TDRB [34], ProvChain [35]
			Data off-chain, transactions on-chain	Auditability and recovery	[42], FalconDB [49]
				Cryptographic overhead	Baldur [25]
			Data off-chain, transactions internal on-chain, hash external on-chain	Collusion resistance	[3, 17]
			Data off-chain, P2P backup, transactions on-chain	Verifiable backup and recovery	[38]
			Data off-chain, access metadata on-chain	Decentralized authorization	VDAC [16]
			Metadata off-chain, full record on-chain	Medical record indexing	[32]
			Sensitive on-chain, insensitive off-chain	User privacy	DEPLEST [10]
		Blockchain within Database	Ledger blocks in DB	Traceability	SQL Ledger [5], LedgerDB [64]
			Ledger in multi-cloud RDB	Internal threats	[7]
Database-supported Blockchains	Improve scalability, performance and privacy	Indexing and Relational Storage	On-chain indexes and relational tuples	Complex SQL queries	SEBDB [67], aChain [59]
		Middleware or External Database	State on-chain, mirroring off-chain	High-speed reads	[56]
		Transaction Optimization	Transactions on-chain	Transaction throughput	[57]
		Ledger Sharding	Sharded ledger on-chain	Network scalability	BlockchainDB [21]
Optimized Authenticated Querying	Performance and verifiability of read operations	Auxiliary ADS	Digests in block headers	Complex querying	vChain [63]
		External Query Layer	Middleware extraction off-chain	Retrieval speed	[36], EtherQL [33], VQL [50]
Hybrid File-Storage Blockchain Systems	Manage large-scale unstructured data	IPFS Integration	Hash on-chain, data off-chain	Scalability and availability	[2, 4, 31, 65], VAULT [18]
			Logs on-chain, data off-chain	Auditing and traceability	[37], BlockIPFS [45]

in a number of studies [3, 4, 10, 16, 34, 36, 42, 50, 59]: as the number of nodes or the size of the dataset grows, system performance often degrades. This includes the challenge of storing massive historical data on-chain and the communication overhead in large networks.

While blockchain provides integrity, many prototypes [2, 3, 5, 17, 34, 35, 38, 63, 64] lack mature security features like fine-grained access control, privacy for sensitive query parameters or protection against specific attacks or low availability. Moreover, accessibility is an overlooked subject as current solutions are often experimental prototypes that lack user-friendly interfaces, standardized development tools or easy integration with legacy systems [4, 35, 52, 56, 67].

Furthermore, one of the most significant open challenges in the literature is the verification strategy, with current studies presenting a spectrum of trade-offs rather than a unified consensus on the optimal approach [5, 7, 36, 45, 49, 63]. This divergence is particularly evident in the triggering mechanisms for integrity checks, which generally fall into periodical, action-triggered or user-requested categories. Periodical verification [36, 38, 50] minimizes the impact on system throughput but introduces a “window of inconsistency” where tampering may persist undetected until the next cycle. Conversely, action-triggered verification (executed synchronously during read or write operations) [34, 37, 63, 65] offers immediate assurances but can incur severe computational latency and financial overhead, depending on the granularity of the checks. User-requested verification [3, 17, 42, 49, 52, 59] offers a middle ground by delegating cost control to the client; however, this approach compromises system abstraction by exposing internal security maintenance to the user and creates a security risk if audits are performed too infrequently. Beyond the detection phase, a secondary challenge remains in remediation: while many systems successfully identify tampering events, they frequently lack automated self-healing mechanisms to restore valid data states or handle node failures without manual administrative intervention.

Chapter 4

Proposed Solution

Following the current state-of-the-art implementations previously described, in this chapter, a solution approach is proposed.

4.1 Approach

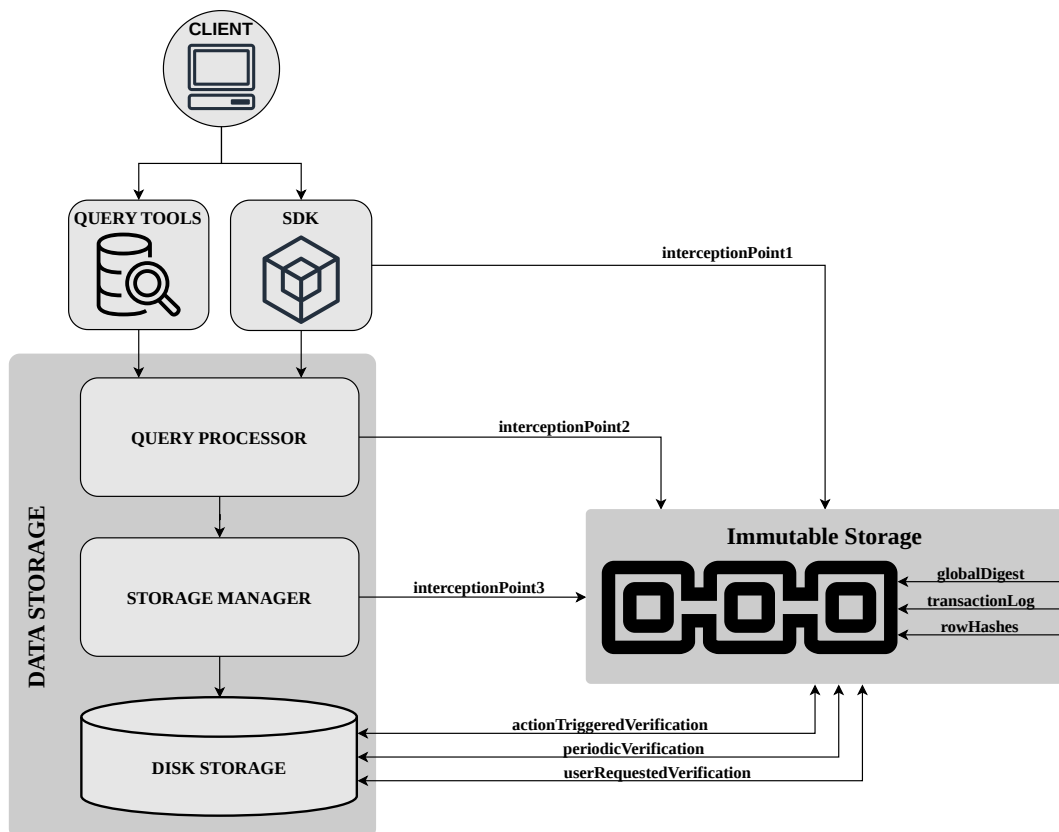


Figure 4.1: High-level architecture design

The proposed solution adopts an architecture based on a database management system augmented with decentralized security properties. This design aligns with the concept of “Blockchain-supported Databases” identified in the State of The Art (3.2), where the primary objective is to extend the mature performance capabilities of standard storage engines with the integrity and anti-tampering guarantees of Distributed Ledger Technology (DLT).

The decision to utilize a traditional database as the foundational layer, rather than building a blockchain-first solution, is driven by the fact that traditional databases remain the predominant standard for data management across most industries. These systems offer highly optimized query engines, Atomicity, Consistency, Isolation, and Durability (ACID) compliance and the ability to handle massive datasets with minimal latency, features that are still being refined in decentralized environments. By starting from a database foundation, the system can leverage these existing strengths while using an External Immutable Storage (EIS) layer specifically to address the lack of built-in cryptographic proof and tampering detection. Therefore, this approach prioritizes a system where data management remains within the database to ensure performance, while an EIS acts as a secure anchor for metadata.

The high-level architecture of this system, detailed in Figure 4.1, is defined by three fundamental design dilemmas:

Interception Point. The first dilemma identifies where the data flow should be intercepted to generate the cryptographic proofs required for the external storage. An option is to intercept at a SDK layer, which happens at the application level to provide end-to-end security. Although this increases the computational burden on the client, it offers the advantage of being database-agnostic, allowing the solution to work with several different databases as long as they are compatible with the SDK. Alternatively, interception can occur internally within the database structure. This approach typically requires making direct changes to a database engine, such as an open-source database, which limits the resulting solution to that specific system. Within the database engine, this can be implemented at the Query Processor level, which captures data as queries are parsed and executed, or at the Storage Manager level, which interacts directly with the Disk Storage. These internal methods involve a much higher level of complexity and development difficulty. In particular, working at the Storage Manager level close to the disk requires deep integration with the database internals, which is technically challenging but ensures that all physical data changes are captured regardless of how the data was accessed. While Query Processor interception provides a higher-level view of logical operations, Storage Manager interception offers the highest performance and transparency.

Anchored Data. The second dilemma involves the nature of the anchored data, addressing what specific information is sent to the EIS. The system could anchor individual row hashes, providing granular traceability for every single record change. While this incurs high storage overhead due to the volume of metadata sent to the external layer, it significantly optimizes the verification process. Since each row has its own hash, the system can perform partial verification by checking

only the specific rows actually being accessed rather than the entire dataset. Another possibility is anchoring transaction logs to preserve the sequence of operations for auditing purposes. This approach generally results in a moderate storage overhead, less intensive than row-level hashing; however, verification is more complex, as it may require reconstructing parts of the log or the state to validate the integrity of the transactions against the anchored record. A third option is the use of global digests, where multiple database operations are aggregated into a single cryptographic summary, such as a Merkle tree root. This approach is the most scalable strictly in terms of storage and communication, as it sends only a single digest to the external layer periodically or for a batch of transactions. The trade-off is that verification becomes very expensive; to verify even a small piece of data, the system might need to recalculate the digest for a large portion of the database to ensure it matches the anchored global state. Ultimately, the choice of anchored data determines the balance between storage efficiency and the agility of the verification mechanism.

Verification Process. The final dilemma defines the trigger mechanism for checking the integrity of the data against the anchors held in the external storage, based on the verification paradigms previously detailed in the State of the Art [Discussion \(3.6\)](#). An action-triggered approach performs verification during read or write operations, making it uniquely suited to explore the advantages of partial verification when paired with row-hash anchoring, as it ensures the data is valid at the exact moment of use. A periodic approach executes integrity checks at set time intervals to mitigate performance overhead; however, this strategy can leave a “window of inconsistency” where tampering remains undetected between scheduled checks. A user-requested approach places the responsibility for integrity and the associated performance impact directly in the hands of the client, who triggers verification only when required. These last two options can be implemented jointly to form a more robust verification framework, providing regular automated monitoring and flexibility for on-demand auditing.

4.2 Design Alternatives

The resolution of the previously described dilemmas involves a set of interdependent choices where a decision in one dimension significantly constrains or guides the valid options available in the others. For instance, choosing an interception point at the application layer naturally limits the ability to aggregate global state, thereby making global digests an impractical anchoring strategy. Similarly, selecting global digests as the anchoring method imposes computational limits that make synchronous action-triggered verification infeasible due to the latency involved in recalculating large cryptographic structures. Based on these inter-dependencies, some distinct design alternatives emerge, each presenting specific trade-offs regarding performance, security and storage costs.

Opting for client-side interception through an [SDK](#), which handles the generation of row hashes before the data ever reaches the storage engine, and utilizing action-triggered verification to verify accessed data at every read against the anchored row hashes, offers a powerful end-to-end

security model where the database itself is treated as an untrusted utility. This alternative provides flexibility and database independence, as the security logic can be applied to various engines without modifying their internal code. By anchoring individual row hashes, the system enables efficient partial verification, as the client only needs to compute and validate the integrity of the specific rows being accessed. While the verification computation is entirely shifted to the client, this model ensures a high level of security by verifying data at the exact moment of use. However, this integrity comes at the cost of massive storage overhead on the external immutable layer due to the granularity of the anchored metadata. Furthermore, this security is enforced strictly at the application level; if a user interacts with the database through other means, such as external query tools, any changes made would be flagged as tampering by the **SDK** since no corresponding anchor was defined. Although this requires strict adherence to the **SDK** for all operations, it also offers significant flexibility as it allows for the intentional “deactivation” of the security layer by simply replacing the secured **SDK** with a standard one in the application. Since no internal changes are made to the storage engine, the same database and data can remain in use without modification even if the requirement for cryptographic verification is removed.

By contrast, an alternative strategy focused on high-throughput environments involves using Storage Manager interception to aggregate multiple database operations into a single global digest, such as a Merkle tree root. This digest is then anchored to the external storage periodically or after a set number of transactions, which is the most efficient method in terms of communication and storage costs. By aggregating numerous internal transactions into a single external write, the database maintains its native performance with minimal interference. However, this scalability does not extend to the verification process, which becomes resource-intensive. To verify even a single record, the system must often recalculate the digest for a large portion of the database state to ensure that it matches the anchored summary. Consequently, this path is only practical when paired with user-requested or periodic verification, as real-time checks would introduce unacceptable latency. Integrating the interception logic internally allows the system to capture data at the lowest level of the database engine, ensuring that the security metadata reflects the actual physical state of the storage. While this approach improves performance and practicality by imbuing the logic directly into the database life-cycle, it significantly increases implementation complexity and limits the solution to a specific database engine. Nevertheless, this internal positioning ensures that security is enforced across all access methods and it avoids restricting options for the other dilemmas.

Moving toward a middle ground, another viable alternative combines internal interception at either the Query Processor or Storage Manager level, with row-hash anchoring and action-triggered verification. Unlike the global digest model, this method allows for immediate, per-row integrity checks during read operations, ensuring that tampering is detected the moment data is accessed. Furthermore, unlike the **SDK** approach, internal integration ensures that security is strictly enforced across all access paths, as even direct administrative interventions or queries from external tools are captured and anchored, eliminating the possibility of bypassing the security layer. However, this robust security retains the significant storage overhead of the client-side model.

Moreover, Query Processor interception offers a simpler logical-level implementation but may miss background state changes, whereas Storage Manager captures the exact data written to disk, offering superior transparency and completeness at the cost of higher development complexity.

Finally, a compromise solution involves anchoring transaction logs or batches to balance the high storage costs of row hashes with the heavy verification penalties of global digests. By anchoring a hash representing a transaction or a small batch of operations, this approach significantly reduces the volume of metadata sent to the external storage compared to row-level strategies. Simultaneously, it allows for a form of partial verification where the system only needs to validate the specific transaction block containing the requested data, rather than reconstructing the entire global state. This makes the verification process lighter than the global digest method, though still more complex than checking a single row hash. Implementing verification in this model requires careful consideration; client-side checks can be difficult due to the need to fetch additional transaction context from the database, potentially increasing network latency. While action-triggered verification is possible, it incurs a higher performance penalty than in row-hash systems due to the need to reconstruct the transaction context. On the other hand, utilizing periodic or user-requested verification in this model might make it more difficult to define the specific boundaries for partial checks, often defaulting to larger audit scopes.

4.3 Selected Implementation Path

For the implementation and evaluation of this dissertation, the client-side interception (**SDK**) with row-hash anchoring and action-triggered verification alternative was selected.

In order to arrive at this decision, it is useful to look at this solution's use in a specific application scenario. An interesting scenario to frame this implementation would be the management of clinical diagnostic results, where a healthcare provider stores sensitive medical data (e.g., blood analysis reports, imaging metadata). Because these records are accessed sporadically but require absolute certainty of integrity upon retrieval, action-triggered verification ensures that a user is alerted to any tampering at the exact moment of consultation. Since medical data for a specific patient is typically queried in isolation, row-hash anchoring allows for efficient, granular verification without the need to process the entire database state or reconstruct complex global structures. Furthermore, the volume of data in such a system is relatively stable compared to high-throughput transactional environments, making the storage overhead of individual row anchors a manageable trade-off. In this highly regulated context, restricting database access to the official application **SDK** is a natural requirement; doctors interact with data exclusively through the medical interface, meaning that any external change would be correctly flagged as an unauthorized deviation from the anchored state.

Beyond specific application requirements, this decision is also driven by a scientific motivation. From a research perspective, existing literature heavily favors internal database engine modifications, leaving a significant gap in the study of client-centric integrity solutions. Exploring

this path allows for a distinct contribution regarding the trade-offs of end-to-end security models where the database is treated as an untrusted utility.

At a high-level, the implementation involves developing an **SDK** for database connections that intercepts read and write operations. For every write, the **SDK** will compute a cryptographic hash of the row and dispatch it to an **EIS** layer. During a read, the **SDK** will simultaneously fetch the record from the database and its corresponding anchor from the immutable layer to perform a comparison. The **SDK** is intended to be a versatile and adaptable tool, designed with the objective of potentially supporting multiple storage engines, from relational systems to **NoSQL** databases. This approach should be useful for several different use cases seeking to improve immediate tampering detection through a solution that is easy to integrate into existing applications.

Chapter 5

Prototype

Based on the design alternative selected in the **Proposed Solution (4)** chapter, the prototype implementation required translating the high-level **SDK** interception, row-level anchoring and action-triggered verification model into a concrete architecture. Figure 5.1 presents the overall architecture of the prototype. The following sections discuss each component and design decision in detail. The complete source code of the prototype is publicly available in the accompanying repository [1].

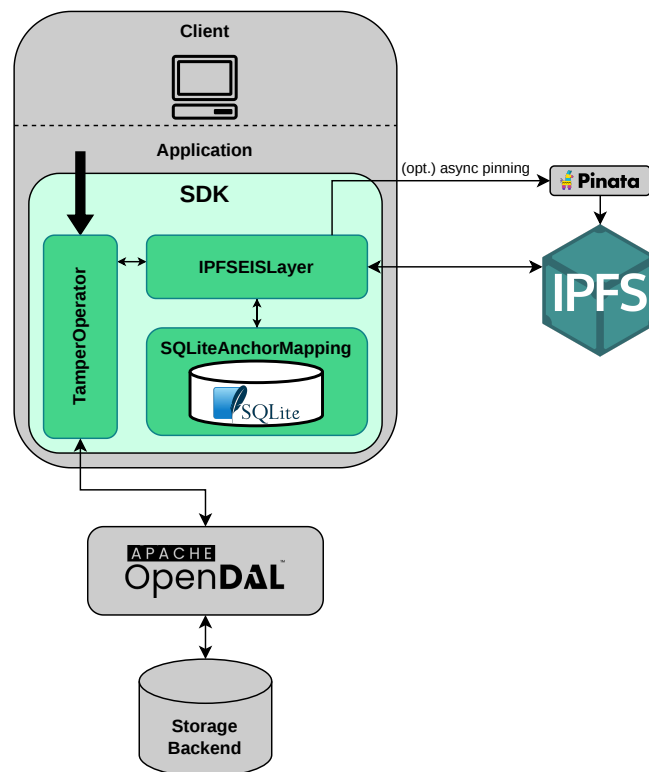


Figure 5.1: Prototype architecture overview.

5.1 SDK Interception

Implementing the proposed client-side tamper detection system required a software layer capable of handling storage interactions while remaining independent from any specific database or storage engine. Rather than developing storage connectivity mechanisms from scratch, there was a preference to build upon an existing open-source abstraction layer already responsible for back-end communication. Re-implementing storage communication logic would duplicate functionality already available in mature storage solutions and shift development effort away from the integration of tampering detection itself. Consequently, the implementation process involved identifying an open-source storage abstraction capable of serving as the interception point for the proposed security layer.

Beyond avoiding unnecessary redevelopment of storage communication logic, another important objective gradually emerged in maintaining backend independence. Since the selected design relies on client-side interception rather than internal database modification, it naturally creates the architectural opportunity to support multiple storage systems through the same security layer. Unlike database-engine interception approaches, which are inherently tied to a specific storage implementation, client-side interception allows the tampering detection logic to remain independent from the underlying backend. The objective, therefore, evolved beyond securing a single database system and toward selecting a storage abstraction capable of transparently supporting heterogeneous storage environments. Achieving this goal required a solution capable of exposing a common operational model across different storage systems while preserving the ability to intercept read and write operations before they reached the backend. Apache OpenDAL [46] emerged as a suitable candidate for this role.

5.1.1 Apache OpenDAL

OpenDAL is an open-source data access layer developed under the Apache ecosystem that provides a unified interface for interacting with heterogeneous storage systems. Rather than exposing backend-specific APIs, OpenDAL abstracts storage interactions through a common operator model supporting operations such as read, write and delete. Although implemented with a Rust core, OpenDAL provides bindings for several major programming languages, allowing its adoption across different software ecosystems.

OpenDAL follows an *Object Storage First* design philosophy, exposing storage systems via a generalized key-to-object interaction model. Under this abstraction, storage operations are expressed through common object-oriented primitives rather than backend-specific mechanisms such as **SQL** queries or proprietary APIs. OpenDAL supports a broad range of storage systems, including relational and **NoSQL** databases such as PostgreSQL and MongoDB, object storage systems such as AWS S3 and Azure Blob, file-based storage such as local filesystems and **IPFS**, and key-value stores such as Redis and RocksDB.

This abstraction enables applications to interact with fundamentally different storage technologies through a largely consistent programming interface while remaining extensible to new storage

backends and language bindings without requiring changes to existing integrations. However, the abstraction is ultimately constrained by the capabilities exposed by each backend. While OpenDAL defines a common operational model, the availability and semantics of specific operations may vary across storage systems. For example, some backends may not support capabilities such as object listing, batch deletion or append operations, causing these features to remain backend-dependent despite the unified interface.

OpenDAL, therefore, satisfied the main architectural requirements identified for the prototype. It provided an open-source storage communication layer, remained independent from storage internals and supported heterogeneous backends through a unified client-side abstraction. As a result, it offered a practical foundation for implementing tamper detection without requiring backend modification or re-implementing storage connectivity logic.

Beyond providing backend abstraction, OpenDAL also exposed a clear interception opportunity. Since storage interactions are mediated through the *Operator* abstraction, additional verification logic could be introduced by extending or wrapping this interaction layer before requests reached the backend. This architectural property made OpenDAL not only a suitable communication layer, but also the practical enforcement point for the proposed tampering detection model.

5.1.2 Interception Strategy

In OpenDAL, storage access is performed through *Operator* instances associated with a particular backend configuration. An operator encapsulates the connection and operational logic required to interact with a storage system and exposes a common interface through which applications perform operations such as reading, writing, deletion, listing, metadata retrieval, copying, renaming and, when supported by the backend, append and batch deletion. Although the availability of individual operations ultimately depends on backend capabilities, the operator abstraction provides a largely consistent interaction model independent of the underlying storage implementation.

The prototype introduces tamper detection as a dedicated **SDK** layer positioned above OpenDAL. Rather than modifying OpenDAL itself or altering backend implementations, the adopted strategy intercepts storage requests before they are delegated to the underlying operator. This preserves OpenDAL as the communication layer while allowing additional integrity logic to be introduced transparently.

The prototype was developed using the Python binding. Direct use of the Rust core was initially considered, as it would provide the closest integration with OpenDAL internals. However, limited prior experience with Rust would pose a challenge to implementation progress. Other language bindings, including C and Java, were also considered. The C binding was regarded as less suitable because the selected client-side interception strategy did not require low-level interaction with storage internals or memory management.

The Python binding was ultimately selected for several practical reasons. Python was already familiar from previous development experience, allowing development effort to remain focused on the tamper detection mechanisms rather than on learning a new language. Its widespread adoption within modern software ecosystems also aligns with the client-side nature of the proposed

solution, where language accessibility may influence practical adoption. Furthermore, OpenDAL documentation indicates that Python is among the most actively used language bindings beyond the Rust core itself, while its ecosystem and tooling also simplified the later benchmarking and evaluation process.

The resulting implementation introduces a dedicated component named *TamperOperator*. Internally, the *TamperOperator* encapsulates a standard OpenDAL *Operator* while extending its behavior with tamper detection mechanisms based on row-level anchoring and action-triggered verification. For storage-modifying operations, the *TamperOperator* generates a cryptographic hash of the affected object data and anchors it in the configured **EIS**. For operations involving access to previously stored data, the component retrieves the corresponding integrity anchor and verifies that the accessed object matches the expected cryptographic state before the requested operation proceeds. Under this design, storage requests issued by the client are first processed by the *TamperOperator* and only then delegated to the underlying OpenDAL operator or validated upon retrieval. From the client perspective, however, interaction remains largely identical to standard OpenDAL usage. Applications continue to operate through an operator-like object configured for a selected backend, while integrity anchoring and verification occur transparently within the intermediate **SDK** layer. This design preserves backend portability and avoids requiring modifications to either storage infrastructure or surrounding application logic while enabling tamper detection to be enforced consistently across supported backends.

5.2 External Immutable Storage Through IPFS

The selected row-level anchoring and action-triggered verification model naturally requires an external storage layer capable of preserving integrity records independently from the protected storage backend. Since every anchored object state generates a corresponding integrity record, the chosen **EIS** must support frequent storage and retrieval operations without introducing prohibitive operational costs or excessive interaction latency.

Beyond practical performance considerations, the **EIS** must also provide immutability and independent verification guarantees. Integrity records cannot be stored within the same trust domain as the protected backend, as a compromised storage system could otherwise alter both the data and its corresponding integrity anchors. The selected solution, therefore, needed to operate as an external, independently verifiable layer while remaining accessible for practical prototype integration.

Under these requirements, the search focused on identifying an immutable storage mechanism capable of supporting high-volume integrity anchoring with low operational complexity and without imposing significant infrastructure or economic overhead. IPFS emerged as a suitable candidate for this role.

5.2.1 InterPlanetary File System

The InterPlanetary File System (**IPFS**) is a decentralized storage system designed around content-addressed data rather than traditional location-based storage. Unlike conventional storage systems, where files are retrieved through paths, filenames or server locations, **IPFS** identifies and retrieves content based on the data itself. This design introduces an immutable storage model where stored objects are uniquely tied to their contents and can be independently verified after retrieval.

The name *InterPlanetary File System* reflects the broader design philosophy behind the system. Rather than depending on centralized servers located at fixed network addresses, **IPFS** was designed as a peer-to-peer storage network where content can be retrieved from any participating node holding the requested data. The term “InterPlanetary” refers to the idea that content retrieval should remain efficient even across highly distributed environments. For example, in a conventional centralized model, requesting data from another planet would require communication with a server located on Earth for every request, introducing substantial delay. Under a decentralized peer-to-peer model such as **IPFS**, once content becomes available on a local peer, subsequent requests can be satisfied locally without repeatedly contacting the original source. Although this scenario remains hypothetical, it illustrates the decentralized and distributed nature of the **IPFS** architecture.

Internally, **IPFS** operates as a distributed storage network composed of interconnected nodes. When content is added to **IPFS**, one or more nodes store or cache the corresponding object and advertise its availability within the network. Retrieval occurs by requesting the associated identifier and locating peers holding the requested content. Rather than contacting a specific server location, **IPFS** searches for available peers and retrieves the corresponding data through peer-to-peer communication. This architecture allows the same content to exist simultaneously across multiple nodes while remaining accessible through a single identifier independent of where the data is physically stored.

The core mechanism enabling this storage model is *content addressing*. When data is stored in **IPFS**, the system computes a cryptographic hash of its contents and uses this value to generate a unique identifier known as a Content Identifier (**CID**). The **CID** acts simultaneously as both the object’s address and its integrity fingerprint. Since the identifier is derived directly from the content, even a minor modification to the stored data produces a completely different **CID**. This content-addressed model forms the basis of **IPFS** immutability. Because the **CID** depends directly on the stored content, data associated with a particular identifier cannot be modified while preserving the same address. Any alteration to the content necessarily produces a different **CID**, meaning that modified versions are no longer accessible through the previous identifier and instead become associated with a new one.

An important practical implication of this model is that **IPFS** does not guarantee permanent storage by default. Objects remain available only while retained by participating nodes. Persistence, therefore, depends on whether content is actively hosted, cached or intentionally retained through mechanisms such as pinning. Nevertheless, even though persistence must be managed

separately, the immutability and verifiability properties of content addressing remain unchanged.

These characteristics make **IPFS** particularly suitable as an **EIS**. Content addressing provides cryptographically verifiable integrity records, immutability prevents silent modification of anchored data, and the decentralized retrieval model enables independent verification outside the protected storage backend. At the same time, **IPFS** can be deployed with relatively low infrastructure requirements and without per-record anchoring costs, making it well-suited to the frequent storage and retrieval operations generated by row-level integrity verification.

5.2.2 Why IPFS?

Selecting an **EIS** required evaluating different technologies capable of storing integrity anchors independently from the protected backend. Since blockchain-based systems were extensively explored throughout the **Background (2)** and **State of The Art (3)** chapters and share many of the security objectives motivating this work, they represented a natural starting point for consideration.

Blockchain infrastructures provide several properties highly relevant to tampering detection. Through append-only ledger structures and distributed validation mechanisms, they offer strong immutability guarantees, decentralized trust and independently verifiable records. Permissionless (or public) blockchains further provide globally replicated state and strong resistance to unauthorized modification, while private or permissioned blockchains allow similar principles to be implemented within more controlled environments. From a security perspective, blockchain systems, therefore, satisfy and often exceed the integrity requirements expected from an **EIS**.

However, suitability depends not only on security guarantees but also on how those guarantees interact with the selected architecture. The adopted prototype relies on row-level anchoring and action-triggered verification, meaning that every protected object version generates a corresponding integrity anchor and that verification occurs whenever data is accessed. Under this model, the number of integrity operations can become substantial.

In public blockchain environments, this interaction model raises two practical concerns. First, anchoring large numbers of records may introduce considerable economic costs due to transaction fees associated with storing hashes on-chain. Second, retrieving and validating anchors during frequent read operations may introduce additional latency associated with blockchain interaction. These characteristics do not make blockchain unsuitable and may be acceptable in scenarios where stronger distributed guarantees justify the additional overhead. However, they become less attractive for a client-side architecture designed around frequent object-level verification.

Private or permissioned blockchains partially mitigate these concerns by reducing or eliminating transaction fees and providing greater control over infrastructure and performance. Nevertheless, such systems still require dedicated blockchain environments and the maintenance of multiple participating nodes to preserve distributed trust assumptions. Although suitable for organizational settings where this infrastructure already exists or can be justified, these operational requirements cannot be assumed for all client environments.

Importantly, these limitations do not imply that blockchain systems are incompatible with the proposed architecture. A blockchain-based implementation remains technically viable even under the selected row-level and action-triggered model, and further experimentation in this direction remains valuable. The decision not to adopt blockchain in this prototype is mainly based on feasibility considerations rather than any technical constraints. Alternative approaches, such as global digest or transaction-log anchoring, would naturally reduce the number of integrity records and could align more favorably with blockchain interaction patterns. Similarly, periodic or user-requested verification could reduce the retrieval frequency and lessen the impact on performance. However, the objective of this prototype was to explore a client-side model centered on independently verifiable object-level integrity with frequent verification, motivating the search for an alternative **EIS** better aligned with these operational characteristics.

Conventional object or blob storage systems represent another possible alternative. Modern cloud storage platforms may provide strong persistence and even immutability guarantees through mechanisms such as object locking or WORM (Write Once, Read Many) policies. However, these guarantees generally remain dependent on the policies and administrative controls of a specific provider or storage environment. While suitable for many use cases, such systems do not inherently provide the decentralized retrieval and independently verifiable storage model offered by distributed approaches.

Under these considerations, **IPFS** emerged as a suitable middle ground for the prototype requirements. Its content-addressed model provides immutable and independently verifiable integrity records, while avoiding the transaction fees, consensus latency and infrastructure requirements commonly associated with blockchain systems. At the same time, it offers a more decentralized and externally verifiable model than conventional provider-controlled object storage. These characteristics made **IPFS** particularly well-suited to supporting the frequent anchoring and retrieval operations generated by the selected row-level anchoring and action-triggered verification strategy.

5.2.3 IPFS Integration

Within the prototype architecture, **IPFS** is integrated as the **EIS** responsible for storing and retrieving integrity anchors independently from the protected storage backend. This functionality is implemented through a dedicated component named *IPFSEISLayer*, derived from the abstract base class *EISLayer*, which defines the common interaction model expected from External Immutable Storage mechanisms. Rather than communicating with **IPFS** directly, the *TamperOperator* delegates anchoring and retrieval requests to this intermediate layer while remaining responsible for request orchestration, integrity verification and hash generation.

Communication with **IPFS** is performed through a dedicated HTTP client connected to an **IPFS** node through its RPC API. Interaction therefore occurs directly with an **IPFS** node capable of exposing the required programmatic interface. Gateway-based access mechanisms, including public, private or service-hosted gateways, are not used for this purpose, as they primarily provide

content retrieval interfaces rather than the full interaction model required for anchoring and management operations. The same communication approach applies whether the connected node is local or remotely hosted.

The *IPFSEISLayer* stores and retrieves integrity anchors represented by cryptographic hashes generated from object contents. Hash generation is performed by the *TamperOperator* using the SHA-256 algorithm, producing a fixed-length digest derived exclusively from the value component under the adopted key-value model. Once received by the *IPFSEISLayer*, the hash is anchored in **IPFS**, locally pinned and associated with the corresponding **CID** returned by the network.

An important practical consideration concerns persistence. As previously discussed, **IPFS** does not guarantee permanent storage by default, meaning that anchored records remain available only while retained by participating nodes. To improve persistence, the prototype locally pins anchored content after submission, instructing the connected node to intentionally retain the corresponding objects and prevent their removal through garbage collection mechanisms.

The prototype also supports optional integration with Pinata [51], a managed **IPFS** pinning service. When enabled, anchored hashes may be pinned through Pinata to improve long-term availability without requiring clients to maintain dedicated persistent nodes. Since external pinning may introduce additional delay, this process is performed asynchronously in the background and therefore does not directly affect the latency of storage operations. Nevertheless, pinning services may introduce economic costs beyond certain usage thresholds, making their adoption a deployment consideration to be evaluated according to persistence requirements and expected anchoring volume.

Although **IPFS** provides immutable and independently verifiable anchor storage, an additional challenge remains. Since backend objects and their integrity anchors exist in separate systems and are identified differently, the prototype requires a mechanism capable of maintaining the association between storage objects and the corresponding **IPFS** identifiers used during verification. The mechanism used to maintain this relationship is discussed in the following section.

5.3 Anchor Mapping

The integration of an External Immutable Storage (**EIS**) layer introduces an additional architectural requirement. Although integrity anchors can be stored independently through **IPFS**, verification still depends on maintaining a reliable association between protected storage objects and their corresponding anchors. Under the adopted row-level anchoring model, where each object version generates its own integrity record, defining how this association should be maintained became a central design decision.

Several alternatives were considered for maintaining this association. One possibility involved using mutable addressing mechanisms such as InterPlanetary Name System (**IPNS**). Unlike standard **IPFS** content addressing, where identifiers are immutable and directly derived from content, **IPNS** provides mutable references capable of pointing to changing **CIDs** over time. Under this approach, a stable logical reference could be maintained while the underlying integrity anchor

evolved alongside object updates. Conceptually, this reduces the need for explicit local association state by delegating reference management to the **IPFS** ecosystem itself. However, this flexibility introduces its own trade-offs. Since **IPNS** references are mutable, their integrity depends on the correctness and security of the update process managing them. Furthermore, experimental evaluation revealed that **IPNS** resolution introduced noticeable latency overhead, making it less suitable for a verification model centered on frequent integrity retrieval.

Another alternative involved deterministic **CID** recomputation from retrieved object contents. Under this model, no explicit mapping would be maintained. Instead, the system would recompute the expected identifier directly from backend data and verify whether the resulting **CID** existed in **IPFS**. This approach is attractive because it removes the need for maintaining local mapping state and produces a fully stateless verification process.

However, deterministic recomputation introduces several practical and security limitations. A first limitation concerns content identity. If anchors were generated exclusively from object contents, distinct objects containing identical data would produce the same hash and therefore the same **CID**. Under such conditions, tampering that replaced an object with content already anchored elsewhere could remain undetected because the recomputed identifier would still correspond to an existing valid anchor.

This issue extends beyond object identity and also affects backend identity. Two objects stored in different backends, storage roots or repositories could still generate identical anchors if they shared the same key-value contents. Preventing this situation would require incorporating backend-specific identity into the anchoring process. However, constructing a universally reliable and backend-independent storage identity proved difficult given the heterogeneous abstractions exposed through OpenDAL.

Historical object states introduce an additional challenge. Since **IPFS** objects are immutable and previously anchored content may remain available indefinitely across the network, reverting an object to an earlier valid state could still produce a valid historical **CID**. Distinguishing legitimate current content from outdated but previously valid versions would therefore require incorporating temporal or version-related information into anchor generation.

Even with such metadata, rollback scenarios remain problematic. Certain storage systems support rollback operations capable of restoring both previous data and associated metadata, potentially reproducing identifiers corresponding to earlier valid anchors. A possible mitigation involved maintaining records of outdated identifiers and explicitly checking for historical matches. However, this approach already introduces explicit state management and effectively converges toward a mapping-based solution.

Deterministic recomputation also complicates protection against anchor injection. Since verification depends on confirming that a recomputed identifier exists in **IPFS**, maliciously anchored content could potentially be treated as valid if it reproduced an accepted historical state or identity combination. Preventing such situations would likely require additional mechanisms such as signed anchors and external key management. Although technically feasible, these additions

introduce further architectural complexity, client setup requirements and trust assumptions regarding key protection and deterministic signing behavior. Furthermore, progressively expanding the information incorporated into the hashing process significantly increases computational overhead.

The adopted solution, therefore, relies on an explicit object-to-CID mapping maintained locally by the prototype. Rather than attempting to discover anchors dynamically, the system preserves a direct association between protected objects and their corresponding integrity records in IPFS. This mapping acts as the authoritative reference used during verification.

Mapping management is implemented through a dedicated abstraction named *AnchorMapping*, defined as an abstract base class responsible for exposing the operations required to maintain anchor associations. The current prototype provides a concrete implementation named *SQLiteAnchorMapping*, responsible for persisting mappings locally using SQLite.

The mapping component is initialized during construction of the *IPFSEISLayer*. Once initialized, the *SQLiteAnchorMapping* component creates or loads the local database required to preserve anchor associations across executions and exposes the operations required to store, retrieve and remove mappings. These interactions remain internal to the *IPFSEISLayer*, which activates mapping operations during anchoring and integrity retrieval without exposing this responsibility to surrounding architectural components.

An important design decision concerns object identification within the mapping itself. Rather than associating CIDs solely with logical object keys, the prototype incorporates backend identity into the mapping key. This avoids ambiguities where identical keys may exist across distinct storage systems or backend contexts. Backend identity is constructed by the *TamperOperator* during initialization using configuration fields considered representative of storage identity, including relevant endpoint, database, repository, container or path information depending on the backend type. The resulting backend context is then introduced into the *IPFSEISLayer*, where it is preserved and combined with logical object keys to form the authoritative identifiers associated with anchored CIDs. Under this design, object hashes remain derived exclusively from object contents through SHA-256, while storage identity is handled separately through the mapping layer.

Persisting mappings through SQLite was primarily motivated by simplicity, portability and lightweight deployment requirements. SQLite requires no dedicated database server, introduces minimal operational overhead and provides sufficient persistence guarantees for prototype execution. At the same time, maintaining mappings locally preserves fast lookup performance and avoids introducing additional network dependencies into the verification process. Nevertheless, the mapping abstraction itself is not inherently tied to SQLite. Alternative implementations may be introduced through the *AnchorMapping* interface if different persistence or deployment requirements are desired.

The adopted mapping design directly addresses several limitations identified in the evaluated alternatives. Because verification retrieves anchors through explicit associations rather than dynamic discovery, replay and rollback attempts are detected whenever outdated content no longer matches the CID currently associated with a given object identifier. Likewise, backend deletions

that would remain undetected under simple **CID** recomputation become observable when a mapping entry exists but the corresponding backend object cannot be retrieved. Anchor injection also becomes substantially more difficult, as independently anchoring tampered content in **IPFS** is insufficient to bypass verification unless the corresponding mapping association is also compromised.

This design introduces an explicit trust assumption. Since the mapping is maintained locally, compromise of the execution environment may affect the integrity guarantees provided by the system. In practice, successful tampering through forged anchors would require simultaneous compromise of the protected backend, the injected or manipulated **IPFS** anchors and the local mapping associations themselves. Nevertheless, although no ideal mechanism was identified that completely eliminates trusted local state, among the alternatives explored, the adopted approach provided the most practical balance between verification performance and protection against replay, rollback, injection and destruction attacks, as well as anchor association ambiguities.

5.4 Prototype Capabilities

The prototype was designed as a client-side extension of OpenDAL that preserves the original storage interaction model while introducing transparent tampering detection. From the client perspective, usage remains largely identical to standard OpenDAL interaction. Applications continue to operate through an operator-like interface, with the *TamperOperator* acting as an intermediate layer responsible for integrity management. As a result, integrating the prototype does not require structural modification of application logic or changes to the underlying storage backend.

Backend configuration follows the same initialization model exposed by OpenDAL. Clients specify the target backend through the corresponding scheme and backend-specific configuration parameters, which are forwarded directly to the underlying OpenDAL *Operator*. In addition to these parameters, the *TamperOperator* introduces optional configuration fields related to integrity management, including the **IPFS** node address, local mapping persistence location and optional Pinata authentication token.

Although these additional parameters enable customization of the integrity infrastructure, they are not strictly required. Default values are provided for all integrity-related configuration fields, meaning that a client may initialize the *TamperOperator* using essentially the same configuration process already used with OpenDAL while obtaining tampering detection capabilities transparently. The only additional operational requirement is access to an **IPFS** node exposing its RPC interface, either locally or remotely.

The architecture also supports extensibility beyond the default implementation. As previously discussed, both *EISLayer* and *AnchorMapping* are defined as abstract base classes. This design allows clients to replace the default *IPFSEISLayer* or *SQLiteAnchorMapping* components with custom implementations adapted to different External Immutable Storages, mapping mechanisms or deployment requirements. Optional Pinata integration is similarly available for environments requiring externally managed **IPFS** persistence.

The prototype implements tamper detection and integrity management for the storage operations directly involved in object access and state modification. These include *read*, *write*, *delete*, *copy*, *rename*, *remove_all* and write operations performed with append semantics enabled. Integrity verification and anchoring behavior are automatically incorporated into these operations through the *TamperOperator* interception layer.

The prototype also introduces a new operation named *read_all*, which is not part of the original OpenDAL interface. This operation retrieves and verifies multiple objects contained within a given path, enabling integrity validation over complete collections rather than individual records. In practice, this functionality supports workflows such as periodic or user-requested large-scale verification of storage contents.

This capability introduces an interesting degree of operational flexibility. Although the architecture was designed around action-triggered verification, clients may still choose to bypass verification during individual reads by interacting directly with the underlying OpenDAL *read* operation and later perform broader integrity validation through *read_all*. This behavior effectively enables periodic or user-requested verification workflows. Nevertheless, such usage represents a workaround rather than the intended interaction model and is not recommended when immediate object-level verification is desired.

Operation availability remains partially dependent on backend capabilities exposed through OpenDAL. Append-based writes are only supported for backends implementing native append functionality. Similarly, *remove_all* and *read_all* depend on backend support for object enumeration through the OpenDAL *list* operation. Consequently, the availability of certain integrity-aware operations may vary across storage systems.

Not all OpenDAL operations require additional tampering logic. Several interfaces unrelated to integrity verification remain available through transparent delegation to the underlying OpenDAL operator. Examples include *list*, *stat*, *exists*, *capability* and *create_dir*. Under this design, the prototype selectively extends only those operations whose semantics involve object modification, retrieval or integrity-sensitive behavior.

Some OpenDAL interfaces were intentionally excluded from the prototype due to semantic incompatibilities with the adopted integrity model. In particular, asynchronous operators (*AsyncOperator*) were not implemented because tampering detection introduces ordering and consistency requirements between storage operations, integrity anchoring and mapping updates. Under asynchronous execution, concurrent operations may complete in different orders from those in which integrity records are generated or persisted, potentially creating temporary inconsistencies between backend state and integrity information. Correctly managing these situations would require additional coordination and concurrency-control mechanisms that fall outside the current prototype scope.

Similarly, stream-oriented interfaces such as *open* were excluded because their semantics do not align cleanly with the object-oriented verification model adopted by the prototype. The current design assumes integrity validation over complete and finalized object representations within the key-value abstraction exposed by OpenDAL. In contrast, stream-oriented interfaces operate

over incremental data flows and may expose partially written states for extended periods. Furthermore, support for such interfaces is not uniformly available across OpenDAL backends, further complicating consistent integration.

5.5 Prototype Operation Flows

The previous sections introduced the architectural components supporting tamper detection and integrity management. This section explains how these components interact during storage operations. The presented workflows first describe the generic interaction model adopted by the SDK and then clarify how this model is instantiated in the current prototype through **IPFS** and SQLite.

At a high level, client applications interact exclusively with the *TamperOperator*, similarly to how they would interact with an OpenDAL *Operator*. The *TamperOperator* orchestrates backend interaction through OpenDAL and delegates integrity management responsibilities to the configured **EIS** layer. Integrity verification and anchor management therefore remain transparent to the client application.

5.5.1 Write Flow

A write operation begins when the client submits data through the *TamperOperator*. The request is first forwarded to the backend storage through the OpenDAL operator associated with the selected backend. At this stage, the operation leaves the SDK and is executed by the storage system itself.

Once OpenDAL confirms successful persistence, control returns to the *TamperOperator*. The operator then computes an SHA-256 hash derived exclusively from the object content under the adopted key-value model. This hash represents the integrity anchor associated with the stored object state.

The generated hash is subsequently forwarded to the configured **EIS** layer. In the current prototype, this corresponds to the *IPFSEISLayer*, which submits the hash to **IPFS** through its dedicated HTTP client connected to the node's RPC API. After anchoring, **IPFS** returns the corresponding **CID**. The *IPFSEISLayer* then locally pins the anchored content and records the object-to-**CID** association through the configured mapping component, implemented in the prototype through *SQLiteAnchorMapping*. If optional Pinata support is enabled, asynchronous remote pinning is also triggered without delaying completion of the write operation.

After anchoring and mapping persistence complete, control returns to the *TamperOperator*, which finalizes the request and returns to the client application.

Figure 5.2 illustrates this interaction sequence. Arrows indicate flow direction, while vertical position determines the order of converging steps.

5.5.2 Read Flow

Read operations follow a verification-oriented workflow designed to validate object integrity before data is returned to the client.

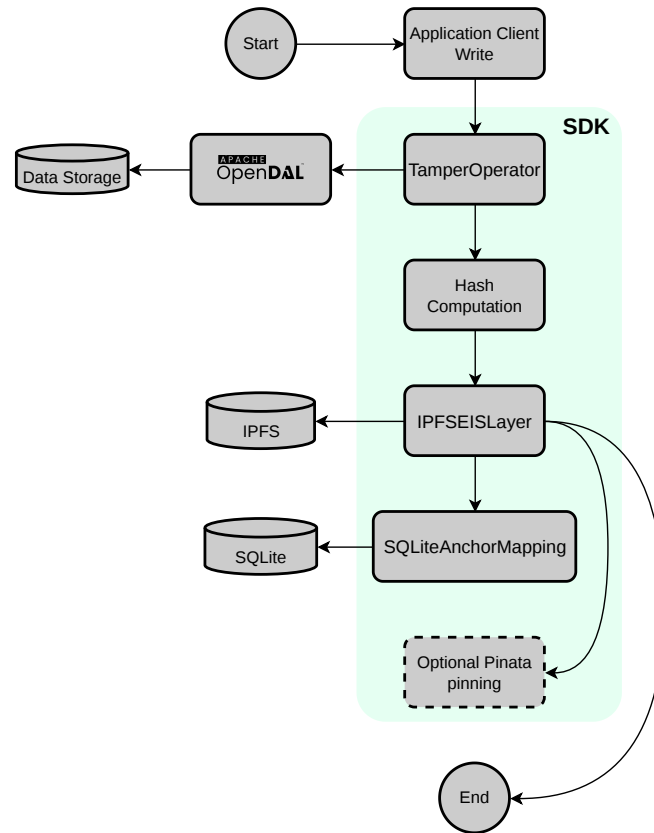


Figure 5.2: Write operation flow.

The process begins when the client requests an object through the *TamperOperator*. The *TamperOperator* first requests the expected integrity anchor from the configured **EIS** layer. In the current prototype, the *IPFSEISLayer* internally consults the mapping component to retrieve the **CID** associated with the requested object and backend context. Using this identifier, the layer retrieves the anchored hash from **IPFS** through the node RPC interface and returns the expected value to the *TamperOperator*.

The *TamperOperator* then retrieves the requested object contents from the backend through OpenDAL. Once the data is obtained, a new SHA-256 hash is computed over the retrieved content and compared against the expected hash recovered from the **EIS**.

If both values match, verification succeeds and the retrieved data is returned to the client application. Otherwise, the operation is treated as a tampering event and the data is not considered valid.

The prototype distinguishes different verification failure conditions through dedicated exception types. A hash mismatch generates a *TamperDetectedException*, indicating that stored contents differ from the anchored integrity record. Missing integrity anchors in the **EIS**, while possessing an existing mapping, generate a *MissingAnchorException*, meaning a possible local anchor dele-

tion or modification may have occurred. Situations where the requested object cannot be found in the backend storage despite the existence of a corresponding mapping, generate a *MissingDataException*, indicating a potential unauthorized deletion or data destruction attack. If an object exists with no mapping to its integrity record, then it may have been added to the storage backend through a method other than the *TamperOperator*, generating an *UntrackedObjectException*. An *OrphanMappingException* occurs when a mapping record exists, but neither the storage data nor the integrity anchor can be found. The *read_all* operation may aggregate multiple verification violations using *BulkTamperException*.

Figure 5.3 presents the verification workflow. Arrows indicate flow direction, while vertical position determines the order of converging steps.

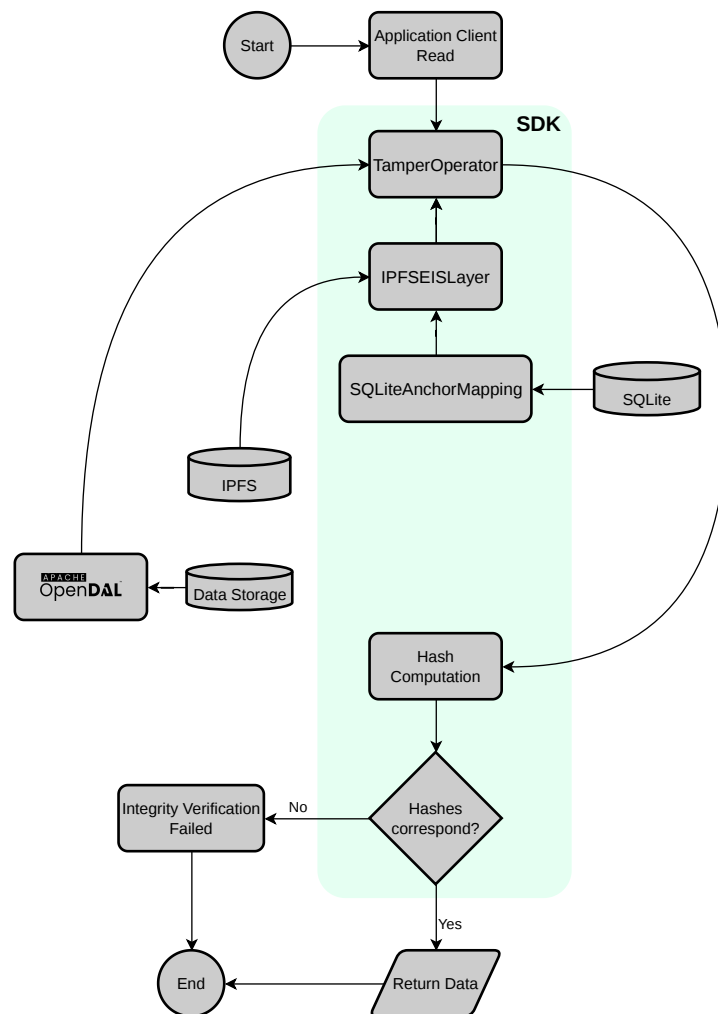


Figure 5.3: Read operation flow.

5.5.3 Append Write Flow

Append operations require additional verification logic because new data is added to an existing object rather than replacing it entirely.

OpenDAL exposes append semantics only for backends providing native support for this capability. Under the adopted tamper detection model, appending to an object whose integrity has not first been validated could allow tampered contents to be silently extended and later anchored as legitimate state.

To prevent this situation, append operations internally combine the previously defined read and write workflows. Before appending occurs, the existing object is first retrieved and verified through the standard read flow. Only if verification succeeds, the append operation is forwarded to the backend through OpenDAL. After successful completion, the updated object state is hashed and anchored again through the standard write workflow, producing a new integrity anchor representing the post-append contents. This process ensures that append operations cannot extend previously compromised object states without detection.

Figure 5.4 illustrates this interaction sequence. The predefined process symbols labeled *Write* and *Read* correspond to the workflows described in the sections [Write Flow \(5.5.1\)](#) and [Read Flow \(5.5.2\)](#) respectively.

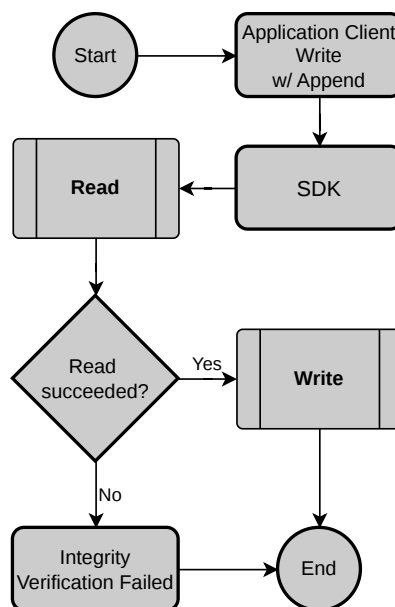


Figure 5.4: Append-based write workflow.

5.5.4 Additional Operation Flows

Other supported operations follow variations of the same integrity management principles. Deletion removes both the backend object and its associated mapping entry. The object is first removed through OpenDAL and, after successful completion, the corresponding anchor association maintained by the **EIS** layer is deleted.

Copy and rename operations first validate the source object through the standard read workflow before applying the corresponding OpenDAL operation. This verification step prevents already tampered objects from being propagated or relocated without detection. Once the backend operation succeeds, a new mapping is created for the destination object using the verified integrity anchor. In rename operations, the original source mapping is subsequently removed.

The *remove_all* operation introduces additional coordination requirements. Since OpenDAL removes entire paths recursively, the prototype first lists all objects contained in the target path. After the backend *remove_all* operation completes successfully, the mappings associated with all previously listed objects are removed from the local mapping store.

The *read_all* operation performs integrity verification across all objects contained within a given path. Internally, the operation enumerates objects through OpenDAL and applies the standard read workflow to each entry individually. Verification results are accumulated throughout execution. If one or more violations are detected, a *BulkTamperException* is raised containing both the detected exceptions and the successfully validated results. Otherwise, the complete set of retrieved objects is returned normally.

5.6 Trust and Threat Model

Tamper detection relies on explicit trust assumptions defining which components are expected to behave correctly and which may behave maliciously or become compromised. The proposed prototype does not eliminate trust requirements but rather establishes where trust is placed and under which conditions integrity guarantees remain valid.

The prototype assumes a trusted client execution environment where the tamper detection logic executes. This includes the *TamperOperator*, the configured *EISLayer*, the mapping implementation and all local state maintained by these components. In particular, the correctness of the locally maintained mapping is assumed to be preserved throughout operation.

The storage backend itself is considered untrusted. Under this model, backend systems may return modified, deleted, replayed or otherwise inconsistent data. Tampering affecting stored data, regardless of how it is introduced, therefore remains possible and is explicitly included within the considered threat model.

The **IPFS** network is similarly not assumed to be inherently secure or trusted against malicious participation. Anchored content may be replicated, cached or hosted by arbitrary nodes and may coexist with unrelated or maliciously introduced data. However, the content-addressed model of **IPFS** provides an important guarantee. Since **CIDs** are derived directly from content, retrieval

using a correct **CID** will inevitably return the expected content whenever that content remains available. Consequently, integrity does not depend on trusting the nodes storing the data but rather on knowing the correct identifier associated with the expected anchor.

A critical implication of this model concerns the locally maintained mapping between backend objects and their corresponding integrity anchors. The mapping acts as the authoritative reference used during verification to determine which anchor is expected for a given object state. If an attacker can modify this mapping, the system may retrieve incorrect anchors and potentially validate tampered data or fail to detect integrity violations. For this reason, protection of the local execution environment and its associated state constitutes the primary trust assumption of the architecture.

Under these assumptions, the prototype provides detection of several relevant tampering scenarios. Unauthorized data modification is detected whenever retrieved contents produce hashes differing from the expected anchored values. Data destruction attacks are identified when an object expected according to the mapping is absent from backend storage. Replay and rollback attacks are mitigated because verification relies on the current object-to-anchor association maintained by the mapping rather than merely checking whether some historical anchor exists in **IPFS**. Similarly, anchor injection attacks are constrained because independently anchoring malicious content in **IPFS** does not affect verification unless the attacker can also compromise the trusted mapping associating objects with their expected anchors.

The proposed model does not prevent tampering from occurring at the storage level. Instead, it detects integrity violations at data access. Likewise, the system does not defend against compromise of the trusted execution environment itself. If an attacker simultaneously controls the backend, manipulates integrity anchors and compromises the trusted local state, including the mapping, the integrity guarantees provided by the prototype collapse and tampering may become undetectable. Such scenarios fall outside the intended threat model.

Consequently, the role of the prototype is centered on reliable tampering detection rather than tampering prevention. By identifying integrity violations in untrusted storage environments, the system can support higher-level protection mechanisms such as recovery mechanisms, backup strategies, audit processes and incident response strategies once tampering is detected.

5.7 Design Trade-offs

The proposed prototype reflects a set of deliberate design decisions balancing integrity guarantees, deployment practicality and architectural flexibility. The adopted design prioritizes practical tamper detection across heterogeneous storage environments while remaining transparent to client applications.

A central design decision concerns operating above backend-specific implementations through OpenDAL. By introducing tamper detection at the abstraction layer, the prototype can support multiple storage systems without requiring modifications to individual backends or application logic. From the client perspective, interaction remains largely unchanged and integrity verification becomes transparently incorporated into supported operations. However, this generality introduces

its own limitations. Since the prototype operates through the interfaces exposed by OpenDAL, support for certain capabilities remains constrained by backend features and abstraction boundaries, reducing opportunities for backend-specific optimization. Additionally, OpenDAL follows an object-storage-first abstraction model centered around key-value interactions. Although this promotes portability across heterogeneous backends, it also limits access to backend-specific functionality that may be highly relevant in practical environments, particularly richer interfaces such as native SQL querying and relational data operations.

The adopted row-level anchoring and action-triggered verification model similarly reflects a deliberate trade-off. Verifying integrity whenever data is accessed provides strong object-level detection guarantees and enables immediate identification of integrity violations during normal usage. This model avoids relying exclusively on periodic verification schedules or manual user-requested verification and ensures that each object remains independently verifiable whenever accessed. However, these guarantees introduce operational costs. Every write generates a new integrity anchor and every read requires verification, increasing computational work and interaction with the external integrity storage system.

Cryptographic hashing contributes additional overhead proportional to the size of the processed data. Although SHA-256 computation remains relatively efficient, integrity verification still incurs processing costs not present in conventional storage access. Similarly, anchoring and retrieving integrity records through an **EIS** introduces latency beyond that of backend interaction alone. The use of **IPFS** mitigates some of these costs by avoiding blockchain consensus and transaction fees, but network communication and anchor management remain unavoidable parts of the verification process.

The choice of **IPFS** itself reflects another architectural compromise. Compared to blockchain approaches, **IPFS** provides immutable and independently verifiable anchor storage with lower infrastructure requirements and without per-anchor economic costs. These characteristics align naturally with the high frequency of anchoring and retrieval produced by the selected architecture. Nevertheless, **IPFS** does not guarantee permanent persistence by default. Maintaining long-term anchor availability, therefore, depends on local pinning, external pinning services or continued content replication within the network. Optional services such as Pinata may improve persistence but may also introduce additional economic considerations.

Maintaining an explicit local mapping between objects and integrity anchors introduces an additional trade-off. The adopted mapping provides efficient lookup performance and avoids the latency and ambiguity associated with mutable addressing or deterministic anchor discovery. It also strengthens protection against replay, rollback and anchor injection scenarios by preserving an authoritative association between objects and their expected anchors. However, this design introduces a trusted local state that must be protected. The correctness of the mapping directly influences the validity of integrity verification, making compromise of the local execution environment a critical limitation of the architecture.

The use of SQLite for mapping persistence reflects similar practical considerations. SQLite provides lightweight deployment, portability and minimal operational overhead while avoiding

dependence on dedicated database infrastructure. At the same time, the prototype does not require SQLite specifically, since alternative mapping implementations may be introduced through the *AnchorMapping* abstraction when different persistence or scalability requirements exist.

Overall, the adopted architecture favors transparent tampering detection, backend independence and practical deployment while explicitly acknowledging the costs required to achieve these properties. The resulting design introduces additional verification overhead and preserves limited trusted local state, but in exchange provides strong and independently verifiable integrity guarantees across heterogeneous storage systems. Therefore, the prototype demonstrates that practical tamper detection can be integrated into generalized storage abstractions without fundamentally altering how client applications interact with storage.

Chapter 6

Evaluation

The previous chapters presented the proposed architecture and described the implementation decisions adopted to integrate tamper detection into the OpenDAL abstraction layer. While the resulting prototype provides independent integrity verification through external anchoring, its practical applicability depends not only on its ability to reliably detect tampering attempts but also on the costs introduced by these additional mechanisms.

To assess these aspects, an experimental evaluation was conducted focusing on two complementary objectives. First, a benchmarking-based evaluation measures the performance impact introduced by integrity anchoring, verification and mapping management when compared with standard OpenDAL operations. Since the proposed architecture is intended to operate across heterogeneous storage systems, the evaluation also examines how these costs vary across different backend types and workload characteristics. Second, the evaluation validates the effectiveness of the proposed integrity mechanisms by examining whether different tampering scenarios can be successfully detected.

Beyond quantifying overhead, the evaluation aims to determine whether the adopted architectural decisions provide a practical balance between tamper detection capabilities and operational performance. Particular attention is given to the scalability of the approach as the size of protected data increases and to the behavior of the prototype under different storage environments.

The remainder of this chapter first describes the performance evaluation methodology, including the experimental environment, benchmark design, tested configurations and measurement procedures. The corresponding results are then presented and analyzed to assess the performance impact of the proposed architecture. Finally, the chapter introduces the methodology adopted to validate tamper detection capabilities and presents the results obtained from the considered tampering scenarios.

6.1 Evaluation Methodology

This section describes the methodology adopted to evaluate the proposed prototype. The objective of the evaluation is not to compare the performance characteristics of different storage systems,

since such differences reflect the behavior of the underlying backends themselves. Instead, the evaluation focuses on quantifying the overhead introduced by the tamper detection mechanisms and assessing how this overhead behaves across different storage technologies, workload patterns and object sizes.

To achieve this objective, the prototype was benchmarked against a baseline implementation using standard OpenDAL operations without integrity verification. Both configurations were executed under identical conditions, allowing the performance impact of the complete tamper detection workflow to be measured. The evaluation, therefore, captures the combined effects of integrity anchoring, verification and mapping management required by the proposed architecture.

The evaluation considers three main dimensions: storage backend, workload type and object size. By varying these factors independently, it becomes possible to analyze how the overhead introduced by tamper detection evolves under different storage environments and data workloads.

6.1.1 Experimental Environment

The evaluation was conducted on a Linux-based system equipped with an 8-core AMD EPYC Milan processor operating at approximately 2.2 GHz, 32 GB of memory and 225 GB of available storage. The system was accessed remotely through SSH and all benchmark executions were performed on the same machine to ensure consistent testing conditions throughout the evaluation.

The prototype and benchmark infrastructure were built in Python using CPython 3.12.13. Dependency management and environment isolation were handled through UV 0.11.8. Using an isolated Python environment ensured that all benchmark executions relied on the same package versions and runtime configuration, improving reproducibility and reducing the risk of external software dependencies influencing the results. Apache OpenDAL 0.46.0 was used as the storage abstraction layer for both the baseline and prototype configurations.

This environment was selected because it provided sufficient computational resources to execute large-scale benchmark workloads while remaining representative of a practical deployment platform. At the same time, the available storage and memory resources imposed realistic operational constraints that influenced some benchmark execution decisions, particularly regarding backend cleanup procedures discussed later in this section.

6.1.2 Evaluation Configurations

The evaluation compares two configurations executed under identical conditions. The first configuration corresponds to the baseline provided by Apache OpenDAL, where storage operations are performed through the standard *Operator* without any integrity verification mechanism. The second configuration corresponds to the proposed prototype, where the same storage operations are performed through the *TamperOperator*, incorporating integrity anchoring and verification through the IPFS-based EIS implementation and the SQLite-based mapping layer.

For each benchmark scenario, both configurations used the same storage backend, backend configuration and workload definition. This ensured that any observed differences reflected the

additional processing performed by the prototype rather than variations in backend deployment or benchmark execution conditions.

The comparison was designed to quantify the practical costs associated with introducing transparent tamper detection into OpenDAL-based storage operations. In particular, the evaluation captures the combined impact of integrity anchor creation, anchor retrieval, cryptographic hashing, mapping management and verification procedures performed by the prototype during normal storage access.

Although the prototype supports optional remote pinning through Pinata, this functionality was disabled throughout the evaluation. The Pinata integration operates asynchronously and, therefore, is not expected to directly affect the latency of storage operations measured in the benchmarks. It was excluded because it is not required for evaluating the performance overhead of the tamper detection mechanisms and because extensive benchmark executions could introduce unnecessary economic costs.

6.1.3 Evaluated Backends

The evaluation was conducted using three storage backends supported by OpenDAL: PostgreSQL [20], Redis [54] and S3-compatible object storage through MinIO [40]. These backends were selected because they represent distinct storage paradigms commonly encountered in practical deployments while remaining accessible through a common OpenDAL interface.

All storage backends and supporting services were deployed through Docker Compose using Docker version 29.4.1. Containerized deployment simplified environment management and contributed to experiment reproducibility by ensuring consistent service configurations across benchmark executions. This deployment also included the IPFS Kubo [28] node used by the prototype for integrity anchoring, deployed through the *ipfs/kubo.41.0* Docker image.

PostgreSQL was included as a representative relational database system. Although relational databases provide capabilities beyond simple object storage, OpenDAL exposes PostgreSQL as a key-value abstraction, allowing it to be evaluated under the same access model used by the remaining backends. The evaluation used the *postgres:16* Docker image.

Redis was selected as a representative in-memory key-value store. Its architecture prioritizes low-latency data access and differs substantially from both relational databases and object storage systems. Evaluating Redis helps assess how the prototype behaves in environments where storage operations are typically very fast and the relative impact of integrity verification may therefore become more visible. The evaluation used the *redis:7.2* Docker image.

Object storage was represented through MinIO, deployed using the *minio/minio.2024-01-16T16-07-38Z* Docker image. MinIO provides an S3-compatible interface while allowing the entire evaluation environment to be executed locally and under controlled conditions. Ideally, the evaluation would use a managed cloud S3 service, as this more closely reflects typical object storage deployments. However, the large number of benchmark operations required by the evaluation would make extensive use of cloud services costly and potentially subject to service limits. Therefore, a local MinIO deployment provided a practical and reproducible alternative. Nevertheless,

this choice represents a limitation of the evaluation, since it does not capture the network latency and variability that would be present when interacting with a remotely hosted S3 service. Consequently, the measured results should be interpreted as representative of a local S3-compatible deployment rather than a managed cloud object storage service.

6.1.4 Workload Design

The evaluation considered three workload categories: write-only workloads, read-only workloads and mixed write-read workloads. These workloads were selected to evaluate the overhead introduced by the prototype under different storage access patterns and to separately observe the costs associated with integrity anchoring and integrity verification.

The write-only workload consists exclusively of write operations. This workload primarily captures the overhead associated with hash computation, integrity anchor creation, anchor storage and mapping management performed after data is successfully persisted in the backend.

The read-only workload consists exclusively of read operations performed over previously stored objects. This workload focuses on the costs introduced by integrity verification, including anchor retrieval from the **EIS**, hash computation and comparison with the expected integrity value before data is returned to the client.

The mixed workload combines write and read operations within the same benchmark iteration. Each object is first written and then immediately retrieved. This workload approximates common usage scenarios in which recently stored data is accessed shortly afterwards and allows the combined effects of anchoring and verification to be evaluated within a single execution flow.

Together, these three workloads provide visibility into the main operational paths of the prototype. Evaluating them independently makes it possible to distinguish the impact of write-related and read-related integrity mechanisms while also assessing their combined behavior under a more representative access pattern.

6.1.5 Dataset Generation

The benchmark datasets consisted of pseudo-randomly generated binary objects with four different sizes: 4 KB, 64 KB, 1 MB and 16 MB. These sizes were inspired by the benchmark workloads available in the OpenDAL repository [13] and were selected to represent both small objects and larger storage operations. Evaluating multiple object sizes allows the impact of hashing and verification operations to be observed across different data granularities.

Dataset generation was designed to ensure reproducibility while avoiding runtime data generation overhead during benchmark execution. Two independent resource pools were prepared before benchmark execution: a *key pool* and a *data pool*.

The *key pool* contains the object identifiers used during benchmark operations. For each object size category, a unique key was generated for every benchmark round. This ensured that write operations created new objects instead of repeatedly overwriting previously stored ones, while also providing a deterministic set of identifiers for subsequent read operations. These keys were

then used to create objects during write-based workloads and to identify objects accessed during read operations.

The *data pool* contains the object contents used throughout the evaluation. For each object size category, a set of $N_{\text{FILES}} = 500$ binary objects was generated using pseudo-random bytes produced from a deterministic random number generator initialized with seed value 42. This deterministic generation process ensures that identical datasets can be recreated across independent benchmark executions. During write operations, object contents were selected randomly from the corresponding data pool and associated with the key being written.

The same pools were reused across both evaluation configurations and all benchmark workloads. This ensured that the baseline OpenDAL implementation and the proposed prototype operated on identical inputs, improving the consistency and comparability of the results. Furthermore, objects created during the write workload were subsequently used by the read workload in order to allow for the reuse of already written resources.

However, Redis constituted a special case due to its in-memory storage model and the memory constraints of the evaluation environment. Persisting all objects generated by the write workload across benchmark executions would have resulted in memory consumption exceeding available resources. Consequently, the Redis read workload uses a dedicated pre-benchmark dataset initialization step. In this case, the same data pool is reused, but the key pool is reduced to $N_{\text{FILES}} = 500$ identifiers per object size. These objects are written before benchmark execution begins and subsequently accessed during read operations, providing a stable dataset while maintaining memory usage within the available system resources.

6.1.6 Benchmark Execution

Benchmark execution was implemented using *pytest* version 9.0.3 together with *pytest-benchmark* version 5.2.3. The *pedantic* execution mode provided by *pytest-benchmark* was used throughout the evaluation in order to control the number of rounds and warm-up rounds executed.

For each workload, object size, backend and evaluation configuration, benchmark operations were executed for 10,000 rounds, with an additional 10 warm-up rounds before each benchmark evaluation. Each benchmark configuration was repeated across five independent runs. These repetitions were performed to reduce the influence of temporary fluctuations in system performance and improve the stability of the reported results. The complete benchmark process is shown in Figure 6.1.

The benchmark process generated a substantial amount of data. The total storage required for a complete benchmark run can be estimated as shown in Equation 6.1. The calculation considers one object of each benchmark size being created per round, across all benchmark rounds and warm-up rounds, for both the baseline and prototype configurations, and for both write-based workloads.

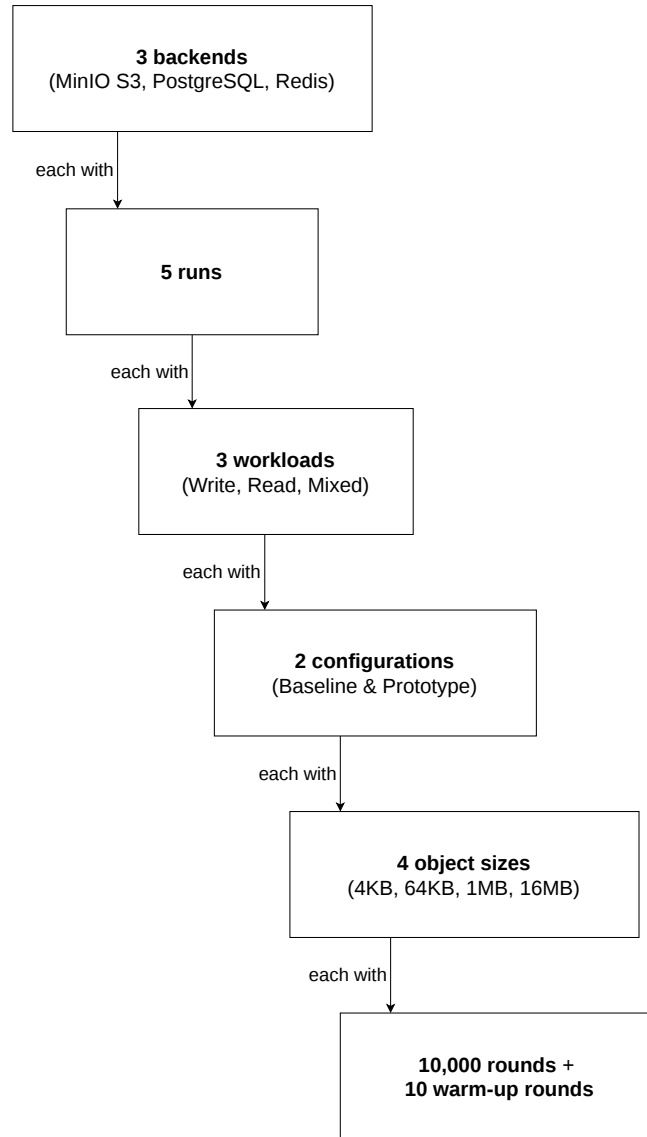


Figure 6.1: Benchmark execution process.

$$\begin{aligned}
 4 \text{ KB} + 64 \text{ KB} + 1 \text{ MB} + 16 \text{ MB} &= 0.017068 \text{ GB} && [\text{Total storage per round}] \\
 0.017068 \text{ GB} \times (10,000 + 10) &= 170.85068 \text{ GB} && [\text{Total storage for all rounds}] \\
 170.85068 \text{ GB} \times 2 &= 341.70136 \text{ GB} && [\text{Baseline \& Prototype}] \\
 341.70136 \text{ GB} \times 2 &= 683.40272 \text{ GB} && [\text{Write \& Mixed workloads}]
 \end{aligned} \tag{6.1}$$

The evaluation environment provided approximately 225 GB of available storage and 32 GB of

memory. As shown in Equation 6.1, executing both write-based workloads for both configurations without cleanup would require approximately 683.4 GB of storage, more than three times the available storage capacity. Even a single write-based workload executed for both configurations exceeds the available storage budget. Consequently, cleanup procedures were incorporated into the benchmark methodology to prevent storage exhaustion and, in the case of Redis, memory exhaustion.

For all backends, cleanup was performed between independent benchmark runs by removing all benchmark-generated objects and associated integrity information. Without this cleanup, benchmark data would accumulate across the five repetitions, rapidly exceeding the available storage capacity. Cleanup therefore ensured that each repetition started from a consistent initial state while remaining within the storage constraints of the evaluation environment.

Additional cleanup was performed before write-based workloads, namely the write-only and mixed workloads, for both the baseline OpenDAL configuration and the proposed prototype. Executing both configurations of a write-based workload requires approximately 341.7 GB of storage, which exceeds the available 225 GB. As a result, the data generated by the baseline configuration had to be removed before executing the prototype configuration. Similarly, data generated during the write-only workload had to be removed before executing the mixed workload. These cleanup steps ensured that sufficient storage capacity remained available throughout the benchmark process.

For PostgreSQL and MinIO S3, the execution sequence within a benchmark run was as shown in Table 6.1.

Step	Workload Phase	Executed Action
1	Write Workload	Cleanup backend state
2		Execute baseline write workload
3		Cleanup backend state
4		Execute prototype write workload
5	Read Workload	Execute baseline read workload
6		Execute prototype read workload
7	Mixed Workload	Cleanup backend state
8		Execute baseline mixed workload
9		Cleanup backend state
10		Execute prototype mixed workload

Table 6.1: Chronological execution sequence for a single benchmark run in PostgreSQL and MinIO S3.

Redis required additional considerations due to its in-memory architecture. The evaluation environment provided 32 GB of memory, while a write workload for a single configuration generates approximately 170.9 GB of object data, as shown in Equation 6.1. Even considering only a single configuration, the amount of generated data exceeds the available memory by more than a factor of five. Redis was therefore configured with a maximum memory limit and an LRU-based eviction policy to prevent benchmark execution from failing due to memory exhaustion.

However, this eviction behavior makes it impossible to reliably reuse the objects generated during previous benchmark phases, since it cannot be guaranteed which objects remain stored after eviction. Consequently, the Redis read workload cannot operate on the data produced by the write workload. Instead, as described in the [Dataset Generation \(6.1.5\)](#) section, a dedicated dataset initialization step is performed before the read workload to create a stable dataset for benchmarking.

The dedicated Redis read dataset contains 500 objects for each benchmark size. The memory required for this dataset can be estimated as seen in Equation 6.2.

$$500 \times (4 \text{ KB} + 64 \text{ KB} + 1 \text{ MB} + 16 \text{ MB}) = 8.534 \text{ GB} \quad (6.2)$$

Although this dataset fits within the available 32 GB of memory, it cannot be safely prepared while retaining the objects generated during the write workload, which already exceeds the available memory capacity. Consequently, Redis required an additional cleanup step before the read workload, followed by dataset initialization.

The execution sequence for Redis can be seen in Table 6.2.

Step	Workload Phase	Executed Action
1	Write Workload	Cleanup backend state
2		Execute baseline write workload
3		Cleanup backend state
4		Execute prototype write workload
5	Read Workload	Cleanup backend state
6		Prepare read benchmark dataset
7		Execute baseline read workload
8		Execute prototype read workload
9	Mixed Workload	Cleanup backend state
10		Execute baseline mixed workload
11		Cleanup backend state
12		Execute prototype mixed workload

Table 6.2: Chronological execution sequence for a single benchmark run in Redis.

It is important to note that all cleanup procedures described were performed outside the benchmarked code path and were therefore not included in the reported performance measurements. Only the operations executed within the *pytest-benchmark* pedantic evaluation were timed. Consequently, the reported results reflect the performance of the storage operations themselves and are not affected by the overhead of dataset preparation or cleanup activities.

Furthermore, the adopted cleanup strategy reflects the characteristics of the evaluation environment used in this work and should not be interpreted as a mandatory requirement of the proposed prototype. Different execution environments may provide substantially larger storage or memory resources, allowing alternative cleanup policies or longer benchmark processes without intermediate cleanup operations. Consequently, the described procedures should be understood as

part of the experimental methodology employed to obtain reproducible results under the available hardware constraints.

6.1.7 Collected Metrics

The evaluation focuses on measuring the performance impact introduced by the proposed tamper detection mechanisms. Since the objective is to quantify the additional cost of integrity verification rather than compare storage systems themselves, the collected metrics emphasize the differences between the baseline OpenDAL implementation and the proposed prototype.

The primary metric collected was operation latency. For each benchmark configuration, latency corresponds to the execution time required to complete a benchmark operation under the evaluated workload. This metric captures the cumulative effect of all activities performed during the operation, including storage access, cryptographic hashing, integrity anchoring, integrity verification and mapping management whenever applicable.

Throughput was also derived from the benchmark results and expressed as the number of operations completed per unit of time. While latency measures the cost of an individual operation, throughput provides a complementary view of the system's capacity to process storage requests under continuous execution.

To facilitate comparison between the baseline and prototype configurations, relative overhead was additionally calculated for both latency and throughput. Latency overhead was computed by comparing the average operation latency of the prototype against the baseline implementation and is reported as a multiplicative factor, calculated as:

$$\frac{\text{Latency}_{\text{prototype}}}{\text{Latency}_{\text{baseline}}}$$

Throughput overhead was computed by comparing the throughput achieved by the prototype against the baseline implementation. Since integrity verification introduces additional processing costs that typically reduce throughput, throughput overhead is reported as a multiplicative factor, calculated as:

$$\frac{\text{Throughput}_{\text{baseline}}}{\text{Throughput}_{\text{prototype}}}$$

Together, these metrics provide a comprehensive view of the operational impact of the proposed approach across different storage backends, workload patterns and object sizes.

6.2 Evaluation Results

This section presents the benchmark results obtained for the evaluated storage backends. The objective is to quantify the performance impact introduced by the proposed tamper detection mechanisms and to understand how this impact varies across different storage technologies, workload patterns and object sizes.

Figure 6.2 and Table 6.3 summarize the relative overhead observed across all evaluated configurations. The detailed results for each backend are presented separately in the following sub-sections.

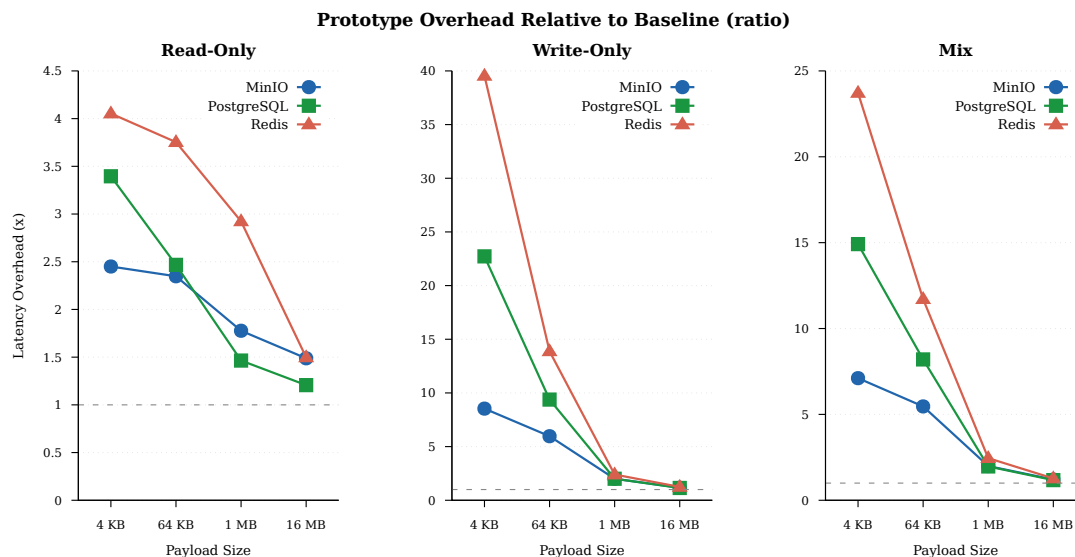


Figure 6.2: Relative latency overhead of the prototype compared to the baseline.

6.2.1 MinIO S3

The results obtained for MinIO S3 are presented in Figure 6.3. The latency and throughput measurements show a consistent scaling behavior across all workload types. As object size increases, latency grows and throughput decreases for both the baseline and prototype implementations, reflecting the higher cost of transferring larger objects. At the same time, the relative overhead introduced by the prototype decreases as object size increases. This can be observed in the overhead ratios reported in Table 6.3, where the prototype approaches the performance of the baseline implementation for larger objects across all workload categories.

Comparing the workload types, the read workload consistently achieves the lowest latency and highest throughput in both the baseline and prototype implementations. Write operations are the most expensive, resulting in higher latency and lower throughput in the write-based workloads. Mixed workloads produce the highest latency and lowest throughput because they combine both read and write operations, accumulating the costs of both access patterns. This behavior is visible across all object sizes and remains consistent throughout the evaluation.

Compared to the other evaluated backends, MinIO exhibits higher baseline latencies and lower throughput for 4 KB and 64 KB objects. For these smaller object sizes, operation latency can be up to five times higher than in PostgreSQL and Redis. As a result, the fixed costs associated with hashing, integrity anchoring and verification represent a smaller fraction of the total execution time, making their relative impact less pronounced. This behavior is reflected in the overhead

Table 6.3: Relative overhead of the prototype compared to the baseline. Values are ratios.

Backend	Workload	Size	Latency Overhead ($\times$)	Throughput Overhead ($\times$)
MinIO	Read	4 KB	2.45	2.45
		64 KB	2.35	2.35
		1 MB	1.78	1.78
		16 MB	1.49	1.49
	Write	4 KB	8.54	8.54
		64 KB	5.97	5.97
		1 MB	2.00	2.00
		16 MB	1.17	1.17
	Mixed	4 KB	7.11	7.11
		64 KB	5.46	5.46
		1 MB	1.99	1.99
		16 MB	1.21	1.21
PostgreSQL	Read	4 KB	3.40	3.40
		64 KB	2.47	2.47
		1 MB	1.46	1.46
		16 MB	1.21	1.21
	Write	4 KB	22.72	22.73
		64 KB	9.38	9.39
		1 MB	2.00	2.00
		16 MB	1.15	1.15
	Mixed	4 KB	14.92	15.01
		64 KB	8.20	8.20
		1 MB	1.97	1.97
		16 MB	1.17	1.17
Redis	Read	4 KB	4.05	4.05
		64 KB	3.75	3.76
		1 MB	2.92	2.92
		16 MB	1.50	1.54
	Write	4 KB	39.54	39.54
		64 KB	13.88	13.88
		1 MB	2.40	2.40
		16 MB	1.24	1.24
	Mixed	4 KB	23.72	23.72
		64 KB	11.71	11.71
		1 MB	2.45	2.45
		16 MB	1.27	1.27

results, where MinIO consistently achieves the lowest overhead ratios among all backends for small objects across read, write and mixed workloads.

Among the evaluated backends, MinIO exhibits the most stable overhead behavior across object sizes and workload types. The difference between the overhead observed for small and large objects is considerably smaller than that observed for PostgreSQL and Redis, resulting in a flatter overhead curve across all workloads, as seen in Figure 6.2.

For read operations, the latency overhead decreases from $2.45\times$ for 4 KB objects to $1.49\times$ for 16 MB objects, a reduction of $0.96\times$. Write operations show a larger reduction, from $8.54\times$ to $1.17\times$, corresponding to a decrease of $7.37\times$. Mixed workloads have a reduction of $5.90\times$, decreasing from $7.11\times$ to $1.21\times$. Although the overhead consistently becomes smaller as object size increases, these reductions are considerably smaller than those observed in PostgreSQL and Redis. For comparison, PostgreSQL exhibits reductions of $2.19\times$, $21.57\times$ and $13.75\times$ for read, write and mixed workloads, respectively, while Redis shows reductions of $2.55\times$, $38.30\times$ and

MinIO (S3-Compatible Storage) — Latency and Throughput

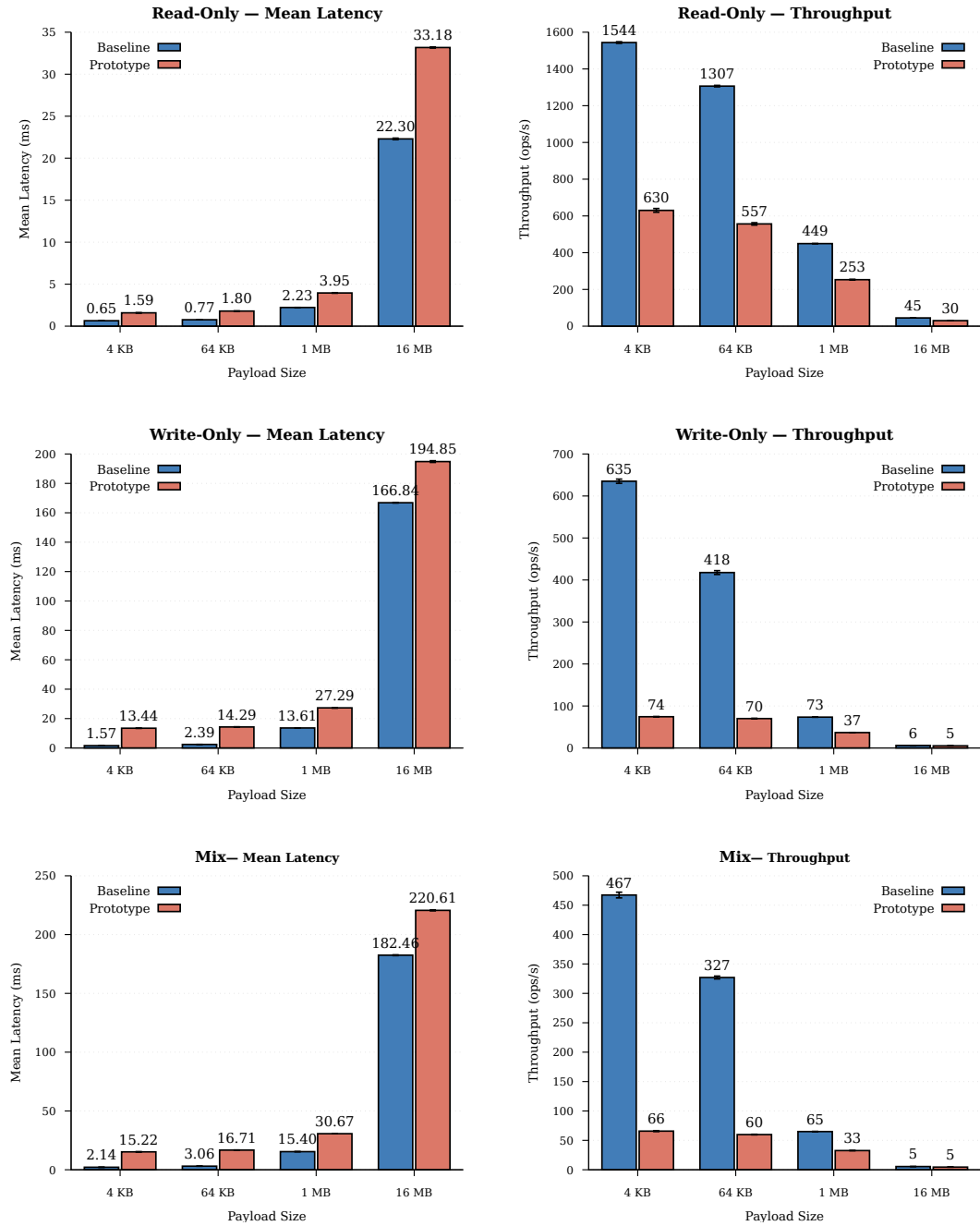


Figure 6.3: MinIO S3 mean latency and throughput per workload for each data size in Baseline vs. Prototype.

$22.45\times$. This indicates that the relative impact of the tamper detection mechanisms remains more uniform across different data sizes.

The comparison between workloads also reveals a change in the relative impact of the tamper detection mechanisms as object size increases. For 4 KB objects, read operations incur the lowest overhead, at $2.45\times$, compared to $8.54\times$ for writes and $7.11\times$ for mixed workloads. However, the

overhead of write and mixed workloads decreases more rapidly as object size grows. At 16 MB, write and mixed workloads reach $1.17\times$ and $1.21\times$, respectively, while read operations remain at $1.49\times$, becoming the highest overhead among the three workload types. This suggests that the additional costs associated with write operations are more effectively amortized by large object transfers, whereas the verification steps performed during reads continue to represent a larger fraction of the total execution time.

The relative overhead introduced by the prototype is nevertheless the lowest among all evaluated backends for small objects. For 4 KB write operations, the overhead reaches $8.54\times$ in MinIO, compared to $22.72\times$ in PostgreSQL and $39.54\times$ in Redis. The same pattern is observed for read and mixed workloads. As object size increases, the overhead values of all three backends gradually converge, reaching similar levels for 1 MB and 16 MB objects.

One possible explanation is that the baseline cost of MinIO operations is already higher for small objects due to the characteristics of S3-compatible object storage. As a result, the additional costs associated with hashing, integrity anchoring and verification represent a smaller fraction of the total execution time than in PostgreSQL and Redis, where baseline operations complete much faster. Consequently, the relative impact of the tamper detection mechanisms becomes less pronounced.

Overall, MinIO demonstrates the most consistent scaling behavior among the evaluated backends. It achieves the lowest overhead ratios for small objects and maintains relatively stable overhead across increasing object sizes, converging to approximately $1.2\text{--}1.5\times$ for large objects across all workload categories. The results also show that read workloads remain the most efficient, while write and mixed workloads exhibit similar performance characteristics, reinforcing that write-related processing is the primary contributor to the overhead introduced by the tamper detection mechanisms.

6.2.2 PostgreSQL

The PostgreSQL results are presented in Figure 6.4. Similar to MinIO, latency increases and throughput decreases as object size grows for both the baseline and prototype implementations. The same workload ordering is also observed across all object sizes: read workloads achieve the lowest latency and highest throughput, followed by write workloads, while mixed workloads exhibit the highest latency and lowest throughput due to combining both write and read operations within the same benchmark round.

For small objects, PostgreSQL achieves lower latencies and higher throughput than MinIO. For example, the baseline latency of 4 KB read operations is 0.32 ms in PostgreSQL, compared to 0.65 ms in MinIO. However, this behavior reverses as object size increases. At 16 MB, PostgreSQL becomes the slowest backend among the evaluated systems, with baseline latencies reaching 54.78 ms for reads, 209.27 ms for writes and 243.69 ms for mixed workloads. In comparison, the corresponding MinIO latencies are 22.30 ms, 166.84 ms and 182.46 ms. This indicates that PostgreSQL scales less efficiently for large binary objects, where data transfer and storage costs become increasingly dominant.

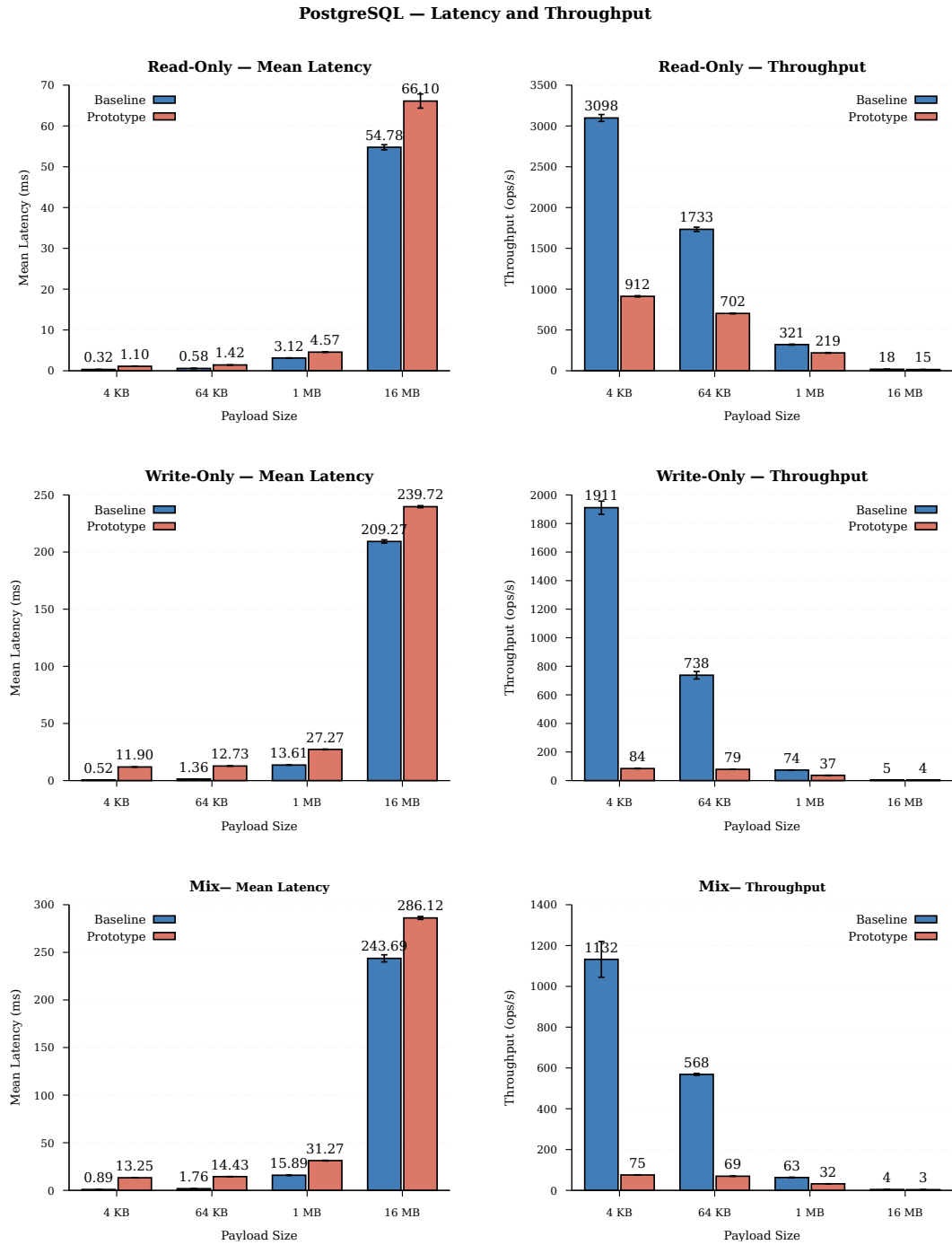


Figure 6.4: PostgreSQL mean latency and throughput per workload for each data size in Baseline vs. Prototype.

The overhead introduced by the prototype follows the same general trend observed in MinIO: overhead decreases as object size increases. However, the reduction is considerably more pronounced. For read workloads, latency overhead decreases from $3.40\times$ for 4 KB objects to $1.21\times$ for 16 MB objects, a reduction of $2.19\times$. Write workloads exhibit an even larger decrease, from $22.72\times$ to $1.15\times$, corresponding to a reduction of $21.57\times$. Mixed workloads decrease from

14.92 $\times$ to 1.17 $\times$, a reduction of 13.75 $\times$. These reductions are substantially larger than those observed in MinIO, where the corresponding decreases were 0.96 $\times$, 7.37 $\times$ and 5.90 $\times$.

The relationship between workload type and overhead is also similar to that observed in MinIO. For 4 KB, 64 KB and 1 MB objects, read workloads consistently exhibit the lowest overhead. At 4 KB, read operations incur an overhead of 3.40 $\times$, compared to 14.92 $\times$ for mixed workloads and 22.72 $\times$ for write workloads. However, as object size increases, the overhead associated with write-intensive workloads decreases more rapidly. For 16 MB objects, write workloads become the most efficient, with an overhead of only 1.15 $\times$, followed by mixed workloads at 1.17 $\times$, while read workloads remain at 1.21 $\times$. The same behavior was observed in MinIO and suggests that the additional costs introduced during writes are more effectively amortized by large data transfers than the verification operations performed during reads.

Compared with the other evaluated backends, PostgreSQL occupies an intermediate position. For small object sizes, its overhead is higher than MinIO but significantly lower than Redis. For example, 4 KB write operations incur an overhead of 22.72 $\times$ in PostgreSQL, compared to 8.54 $\times$ in MinIO and 39.54 $\times$ in Redis. As object size increases, however, PostgreSQL consistently achieves the lowest overhead among all evaluated backends. At 1 MB, PostgreSQL records the lowest overhead for read (1.46 $\times$), write (2.00 $\times$) and mixed (1.97 $\times$) workloads. The same pattern is observed at 16 MB, where PostgreSQL achieves overheads of 1.21 $\times$, 1.15 $\times$ and 1.17 $\times$ for read, write and mixed workloads, respectively. Although the differences become small at these sizes and all backends converge towards similar values, PostgreSQL consistently maintains the lowest relative overhead.

One possible explanation for the substantial overhead reduction is the rapid increase in baseline storage costs as object size grows. For example, PostgreSQL write latency increases from 0.52 ms for 4 KB objects to 209.27 ms for 16 MB objects, representing an increase of approximately 400 times. During the same interval, prototype latency increases from 11.90 ms to 239.72 ms, an increase of approximately 20 times. As the cost of the underlying storage operation becomes dominant for large objects, the fixed costs associated with hashing, anchor management and integrity verification represent a progressively smaller fraction of the total execution time, causing the relative overhead to decrease substantially.

Overall, the PostgreSQL evaluation reinforces the general pattern observed throughout the study: the impact of the tamper detection mechanisms becomes progressively less significant as object size increases. Although PostgreSQL exhibits higher overhead than MinIO for small objects, it achieves the lowest overhead for larger objects, with all workloads remaining between 1.15–1.21 $\times$ at 16 MB.

6.2.3 Redis

The Redis results are presented in Figure 6.5. As observed for MinIO and PostgreSQL, latency increases and throughput decreases as object size grows for both the baseline and prototype implementations. The same workload ordering is also maintained across all object sizes. Read workloads consistently achieve the lowest latency and highest throughput, followed by write workloads,

while mixed workloads exhibit the highest latency and lowest throughput because they combine both write and read operations within the same benchmark round.

Redis — Latency and Throughput

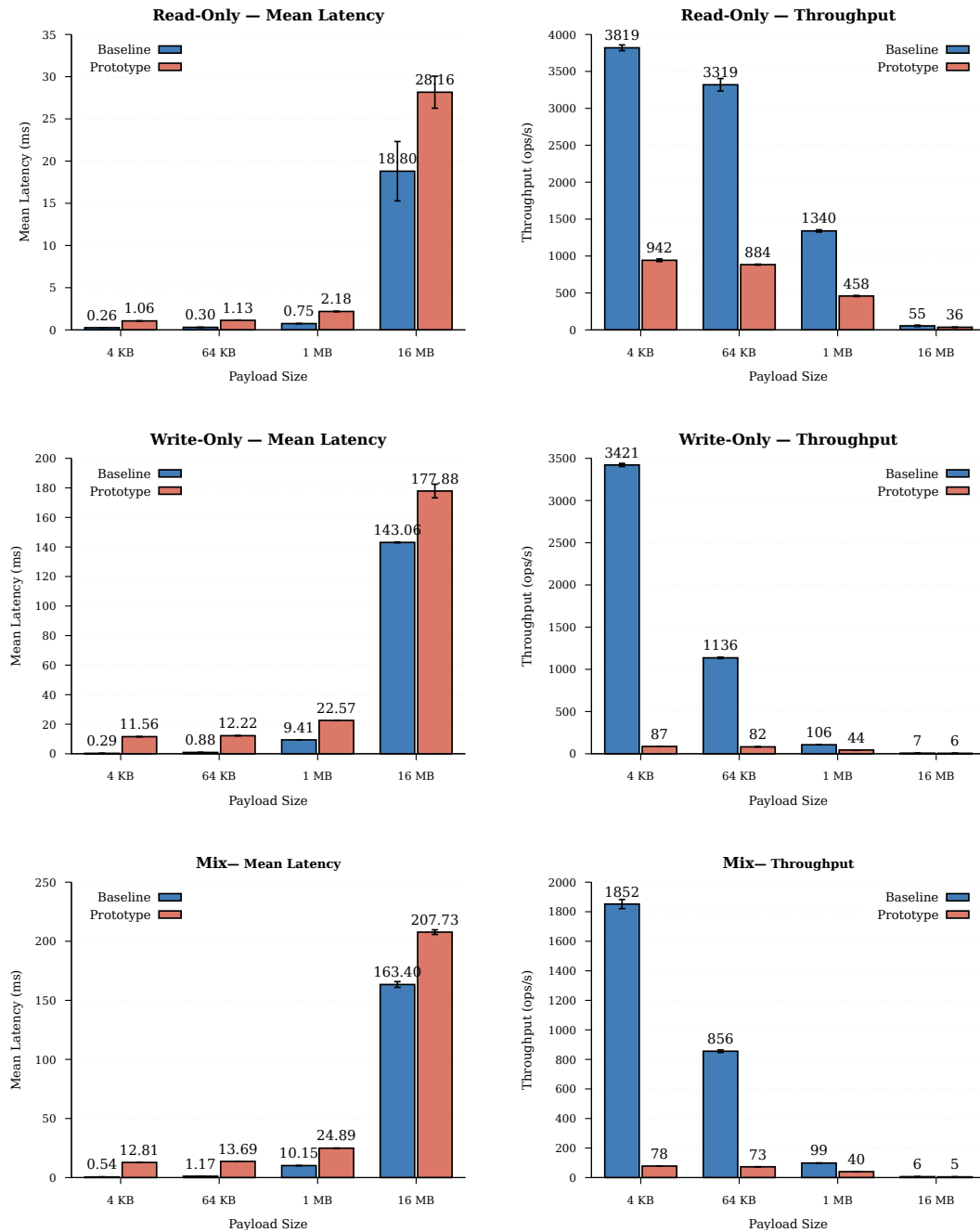


Figure 6.5: Redis mean latency and throughput per workload for each data size in Baseline vs. Prototype.

Among all evaluated backends, Redis provides the best baseline performance. For every workload and object size, Redis achieves the lowest latencies and highest throughput values. For example, the baseline latency of 4 KB read operations is only 0.26 ms, compared to 0.32 ms in

PostgreSQL and 0.65 ms in MinIO. Similarly, Redis achieves 3819 operations per second for 4 KB reads, exceeding both PostgreSQL (3098 ops/s) and MinIO (1544 ops/s). This advantage remains visible across all workloads and object sizes.

The scaling behavior of Redis, however, differs between workloads. For write and mixed workloads, latency begins to increase substantially at 1 MB. Baseline write latency rises from 0.88 ms at 64 KB to 9.41 ms at 1 MB, while mixed workload latency increases from 1.17 ms to 10.15 ms over the same interval. In contrast, the read workload scales much more gradually up to 1 MB. Baseline read latency increases from only 0.26 ms at 4 KB to 0.75 ms at 1 MB, remaining below 1 ms despite an increase of 256 times in object size. The substantial increase only occurs at 16 MB, where latency rises to 18.80 ms. A possible explanation is the in-memory nature of Redis, which allows data to be accessed much faster than disk-based systems. For read operations, this enables Redis to maintain very low latencies across small and medium-sized objects, with the impact of transferring larger amounts of data becoming more noticeable only at the largest evaluated object size. Despite this increase, Redis remains the fastest backend for read operations at all evaluated object sizes.

The overhead introduced by the prototype decreases as object size increases, but Redis exhibits the least stable overhead behavior among the evaluated backends. The difference between the overhead observed for small and large objects is considerably larger than in MinIO and PostgreSQL. For read workloads, latency overhead decreases from $4.05\times$ at 4 KB to $1.50\times$ at 16 MB, a reduction of $2.55\times$. Write workloads decrease from $39.54\times$ to $1.24\times$, corresponding to a reduction of $38.30\times$. Mixed workloads decrease from $23.72\times$ to $1.27\times$, a reduction of $22.45\times$. These are the largest overhead reductions observed among all evaluated backends.

The relationship between workload type and overhead follows a pattern similar to the previous backends, although with a notable difference. For 4 KB and 64 KB objects, read workloads exhibit the lowest overhead, followed by mixed workloads and then write workloads. At 4 KB, the overhead reaches $4.05\times$ for reads, $23.72\times$ for mixed workloads and $39.54\times$ for writes. However, from 1 MB onward, the ordering changes. At 1 MB, write workloads exhibit the lowest overhead at $2.40\times$, followed by mixed workloads at $2.45\times$, while read workloads remain at $2.92\times$. The same ordering is maintained at 16 MB. Unlike MinIO and PostgreSQL, where this inversion only becomes visible at 16 MB, Redis exhibits it already at 1 MB.

The shape of the overhead curves also differs between workload types. For write and mixed workloads, the overhead drops sharply between 64 KB and 1 MB, closely matching the behavior observed in PostgreSQL. For example, write overhead decreases from $13.88\times$ to $2.40\times$ between these two object sizes. In contrast, the read workload exhibits a more gradual reduction, decreasing from $3.75\times$ to $2.92\times$. The largest reduction in read overhead occurs later, between 1 MB and 16 MB, where it drops from $2.92\times$ to $1.50\times$. This behavior is visible in Figure 6.2, where the read overhead curve remains elevated at 1 MB instead of converging with the other backends as quickly.

Compared to MinIO and PostgreSQL, Redis consistently exhibits the highest overhead across all workloads and object sizes. For example, 4 KB write operations incur an overhead of $39.54\times$

in Redis, compared to $22.72\times$ in PostgreSQL and $8.54\times$ in MinIO. Even at 1 MB, Redis write overhead remains higher at $2.40\times$, compared to approximately $2.00\times$ in both PostgreSQL and MinIO. Although the differences become much smaller at 16 MB, Redis still maintains the highest overhead values for all workloads.

A possible explanation for this behavior is the extremely low latency of the Redis baseline implementation. Because standard Redis operations complete very quickly, the additional costs introduced by hashing, anchor management and integrity verification represent a much larger fraction of the total execution time. This effect is particularly visible in the read workload, where baseline latencies remain below 1 ms up to 1 MB objects. As a result, the prototype overhead remains comparatively high even when object sizes increase.

It should also be noted that the Redis evaluation required the use of an LRU eviction policy, as described in the [Benchmark Execution \(6.1.6\)](#) section, to prevent memory exhaustion during the benchmark process. Without eviction, the continuous insertion of objects would eventually exceed the available memory and prevent the completion of the experiments. Although this configuration was necessary, eviction activity may have introduced additional processing costs, particularly in write and mixed workloads where new objects are continuously inserted. As a result, some portion of the measured latency and throughput degradation may be attributable to memory management operations performed by Redis in addition to the costs introduced by the prototype. Nonetheless, the objective of this evaluation is to compare the baseline and prototype configurations under the same conditions. Since both configurations were subject to the same eviction policy and memory constraints, the relative differences observed between them remain representative of the overhead introduced by the tamper detection mechanisms.

Overall, Redis demonstrates the best absolute performance among the evaluated backends but also the highest relative overhead. The results show that the impact of the tamper detection mechanisms is amplified when the underlying storage system is extremely fast, making the additional integrity-related operations more visible in relative terms. Nevertheless, as object size increases, the overhead converges towards values close to those observed in the remaining backends, reaching between $1.24\times$ and $1.50\times$ for 16 MB objects.

6.3 Tamper Detection Validation

Beyond measuring the performance overhead introduced by the prototype, it is also necessary to validate that the proposed tamper detection mechanisms operate correctly under different attack scenarios. The objective of this validation is not to measure performance, but rather to verify that unauthorized modifications to objects or the unavailability of integrity records are consistently detected and that legitimate objects remain accessible without false positives.

The validation process focused on representative scenarios that may occur in practice, including object modification, object deletion, integrity anchor unavailability, recovery from local anchor loss and rollback attempts. Together, these scenarios evaluate the prototype's ability to detect both direct attacks against stored data and attacks targeting the anchor storage itself.

6.3.1 Validation Methodology

Tamper detection validation was implemented using the *pytest* testing framework. Each test case was designed to simulate a specific attack or operational scenario and verify whether the prototype produced the expected outcome. Successful validation required legitimate objects to be accepted and tampered objects to be rejected.

All validation scenarios were executed using the local file-system backend. Since the integrity verification logic is implemented independently of the underlying storage backend, the expected tamper detection behavior remains the same regardless of the storage technology used. The file-system backend was therefore selected for practicality and ease of direct object manipulation.

Valid Object. This scenario evaluates the normal operation of the prototype and serves as a baseline for the remaining validation tests. An object is written through the prototype and subsequently read through the prototype. The verification process is expected to succeed, confirming that valid objects are accepted and that the integrity mechanisms do not generate false positives.

Object Without Anchor. This scenario evaluates objects that were not created through the prototype and therefore do not have associated integrity anchors. An object is written using a standard OpenDAL operator instead of the prototype, bypassing the integrity mechanisms. When the object is later accessed through the prototype, verification is expected to fail because no corresponding integrity anchor exists.

Object Content Tampering. This scenario evaluates the detection of unauthorized modifications to stored object contents. After an object is written through the prototype, its contents are modified using a standard OpenDAL operator, bypassing the prototype. A subsequent read operation should detect the mismatch between the stored content and its integrity anchor, causing verification to fail.

Multiple Tampered Objects. This scenario evaluates batch verification when multiple objects are processed simultaneously. Multiple objects are written through the prototype, after which some are modified using standard OpenDAL operators. Executing a *read_all* operation should correctly identify the tampered objects and report the integrity violation.

Deleted Object. This scenario evaluates the detection of unauthorized object removal from the storage backend. An object written through the prototype is deleted using a standard OpenDAL operator without updating its integrity information. Any attempt to access the object through the prototype should detect the missing data and result in a verification failure.

Anchor Deletion. This scenario evaluates the impact of removing local integrity anchors. After writing an object through the prototype, the corresponding local anchor record is deleted. Verification should fail because the integrity information required to validate the object is no longer available.

Pinned-Anchor Recovery. This scenario evaluates the recovery of integrity anchors from **IPFS** when local anchor information is unavailable. An object is written through the prototype with Pinata-based pinning enabled, after which the local anchor is removed. During verification, the prototype should retrieve the original anchor from IPFS and successfully validate the object, demonstrating resilience against local anchor loss.

Anchor Injection Attack. This scenario evaluates whether integrity records can be forged outside the normal prototype workflow. An object is created (modifying a previously valid object would have the same effect) using a standard OpenDAL operator and, therefore, bypasses the prototype. A matching anchor is then manually generated and stored in **IPFS**. Verification is expected to fail, demonstrating that attackers cannot create valid integrity records outside the normal prototype workflow.

Rollback Attack. This scenario evaluates the detection of unauthorized restoration of previous object states. An object is first written through the prototype and later updated to a newer version. The original version is then restored directly in the storage backend using a standard OpenDAL operator, bypassing the prototype. When the object is subsequently accessed through the prototype, verification should fail because the restored version no longer corresponds to the most recent integrity information. This scenario verifies that tampering is detected even when a valid anchor for the restored state already exists.

6.3.2 Validation Results

Table 6.4 summarizes the results of the tamper detection validation scenarios. All tests produced the expected outcome, demonstrating that the prototype correctly distinguishes between legitimate and tampered objects under the evaluated conditions.

The **Valid Object** scenario established a baseline for the remaining validation tests. An object written and read through the prototype, following the normal workflow and without any tampering, was successfully verified. This demonstrates that legitimate operations are accepted by the integrity mechanisms and that verification succeeds when the object remains unchanged.

The **Object Without Anchor** scenario failed verification because the object had been created outside the prototype and, therefore, was never associated with an integrity anchor. During verification, the prototype detected that no anchor existed for the object and raised a *UntrackedObjectException*. This confirms that externally created objects remain untracked and cannot be treated as valid protected objects.

Table 6.4: Summary of the tamper detection validation results.

Scenario	Expected Result	Success
Valid Object	Verification succeeds	Yes
Object Without Anchor	Verification fails	Yes
Object Content Tampering	Verification fails	Yes
Multiple Tampered Objects	Tampered objects detected	Yes
Deleted Object	Verification fails	Yes
Anchor Deletion	Verification fails	Yes
Pinned-Anchor Recovery	Verification succeeds	Yes
Anchor Injection Attack	Verification fails	Yes
Rollback Attack	Verification fails	Yes

The **Object Content Tampering** scenario was detected through a mismatch between the hash computed from the retrieved object and the integrity anchor associated with it. When the mismatch was identified, the prototype raised a *TamperDetectedException*. The same verification mechanism was exercised in the **Multiple Tampered Objects** scenario, where the *read_all* operation recomputed and compared hashes for each object. All detected integrity violations were accumulated and reported through a *BulkTamperException*, allowing the operation to identify every tampered object processed in the batch.

The **Deleted Object** scenario was also detected successfully. Although the object could not be retrieved from storage, a corresponding mapping to a valid integrity anchor was still present. This inconsistency indicates that the object was illegally removed after being registered by the prototype and is therefore treated as tampering rather than a simple *NotFound* condition. As a result, the verification process raised a *MissingDataException*, preventing deleted objects from being silently accepted as valid.

The **Anchor Deletion** scenario resulted in a verification failure. In this case, the local mapping pointed to an unavailable **CID**, preventing the prototype from locating the corresponding anchor. As a result, the integrity information required for verification could not be retrieved and a *MissingAnchorException* was raised. These results show that attackers cannot bypass verification by removing local integrity records.

The **Pinned-Anchor Recovery** scenario produced the expected successful outcome. After the local anchor was removed, the prototype retrieved the corresponding anchor from another **IPFS** peer that was holding it as a result of the Pinata pinning and completed the verification process successfully. The recovered anchor matched the computed object hash, allowing verification to succeed without raising an exception. This demonstrates the ability of the architecture to tolerate local anchor loss or corruption when external anchor pinning is available.

The **Anchor Injection Attack** scenario also failed as expected. Although a matching anchor was manually generated and stored in **IPFS**, the prototype had no mapping between the object and the injected anchor **CID**. As a result, the injected anchor had no effect and the object was treated as untracked, causing an *UntrackedObjectException* to be raised. The same principle applies if an attacker tampers with an existing tracked object and generates a valid anchor for the modified

state. The prototype would still use the mapping associated with the previous valid anchor, causing a mismatch between the tampered object's hash and the stored integrity information, which would result in a *TamperDetectedException*. These results show that attackers cannot create apparently valid objects by independently generating integrity information.

Finally, the **Rollback Attack** scenario was successfully detected. Restoring a previous version of an object caused a mismatch with the most recent integrity information maintained by the prototype. Although a previous state was illegally restored, the mapping still maintained a reference to the latest valid anchor **CID**. During verification, the hash of the restored object therefore differed from the anchor referenced by the mapping, causing the prototype to raise a *TamperDetectedException*. This confirms that the system protects not only against direct content modification but also against unauthorized restoration of older object states.

Overall, the validation results demonstrate that the proposed integrity architecture correctly detects all evaluated tampering scenarios while preserving normal access to legitimate objects. These results provide evidence that the prototype can enforce integrity guarantees across a range of common attack and failure situations.

Chapter 7

Discussion

The previous [Evaluation \(6\)](#) chapter presented the performance evaluation and tamper detection validation results of the proposed prototype. This chapter discusses the meaning of those results and their implications for the practical adoption of the system. The goal is to interpret the observed behavior, explain the main performance trends and assess how the introduced integrity guarantees affect storage operations under different conditions.

7.1 Interpretation of Performance Results

The evaluation results revealed several consistent trends across all evaluated storage backends. As expected, latency increased and throughput decreased as object size increased. Larger objects require more data to be transferred, processed and stored, naturally increasing operation times. A similar behavior was observed across workloads, where read operations consistently achieved lower latencies and higher throughput than write operations. Mixed workloads generally exhibited the lowest performance because they combine both read and write activities within the same execution.

Beyond these expected performance characteristics, the results also show the cost of introducing tamper detection mechanisms. In every evaluated scenario, the prototype increased latency and reduced throughput when compared to the baseline implementations. This behavior was expected since each operation requires additional work, including integrity verification, hash computation and anchor management.

However, the relative impact of these mechanisms was not constant across all scenarios. One of the most important observations is that the overhead introduced by the prototype decreased as object size increased. Small objects consistently exhibited the highest latency and throughput overheads, while larger objects showed substantially lower overhead values. Although the prototype performs additional integrity-related operations regardless of object size, the cost of these operations becomes less significant as the amount of transferred data increases. Consequently, the integrity verification cost is progressively amortized by the larger storage operation.

The same pattern can be observed from another perspective. Scenarios with lower baseline latencies generally exhibited higher relative overheads, whereas scenarios with higher baseline latencies exhibited lower overheads. In other words, the faster the underlying storage operation, the more visible the impact of the tamper detection mechanisms becomes. Conversely, when the storage operation is already expensive, the additional verification cost represents only a small fraction of the total execution time. This relationship can be visualized by examining the relation between the baseline latency of each scenario and the relative latency overhead introduced by the prototype. Figure 7.1 shows that as baseline latency increases, the observed overhead tends to decrease.

This observation also explains why overhead values differed between storage backends. Redis consistently achieved the lowest baseline latencies and highest throughput values, but it also exhibited the highest overheads across most scenarios. PostgreSQL showed the opposite behavior, particularly for larger object sizes, where its higher baseline latencies resulted in some of the lowest observed overheads. MinIO exhibited a different pattern. Its baseline latencies were comparatively higher for smaller object sizes, resulting in lower overheads in those scenarios. Because of that, as object size increased, the difference between small-object and large-object overheads was less pronounced than in Redis and PostgreSQL, leading to a more stable overhead profile across the evaluated object sizes.

Taken together, these results suggest that the integrity mechanisms behave primarily as a fixed per-operation cost. Regardless of the storage backend, every operation must perform a similar set of integrity-related tasks. The cost of these tasks is largely independent of the storage backend and depends mainly on the object size due to the hash computation process. As a result, for a given object size, the additional integrity-related work remains similar across backends, making its impact more noticeable when storage operations are fast and less noticeable when storage latency is higher. This interpretation explains both the reduction of overhead with increasing object sizes and the differences observed between the evaluated storage backends.

Overall, the results demonstrate that integrity guarantees introduce a measurable performance cost. However, they also show that this cost becomes progressively less significant as the underlying storage operation becomes more expensive. This behavior was consistently observed across the evaluated object sizes, workloads and storage backends, indicating that the relative impact of the tamper detection mechanisms on overall performance is strongly influenced by the baseline cost of the storage operation itself, even though the cost of these mechanisms is largely independent of the underlying storage backend.

7.2 Suitable Application Scenarios

The suitability of the prototype depends first on the conditions that motivated this work. As discussed earlier, the prototype is intended for application scenarios where the storage backend cannot be fully trusted, either because it is managed by a third-party, shared among multiple actors or otherwise exposed to unauthorized modifications. It is also intended for situations where integrity

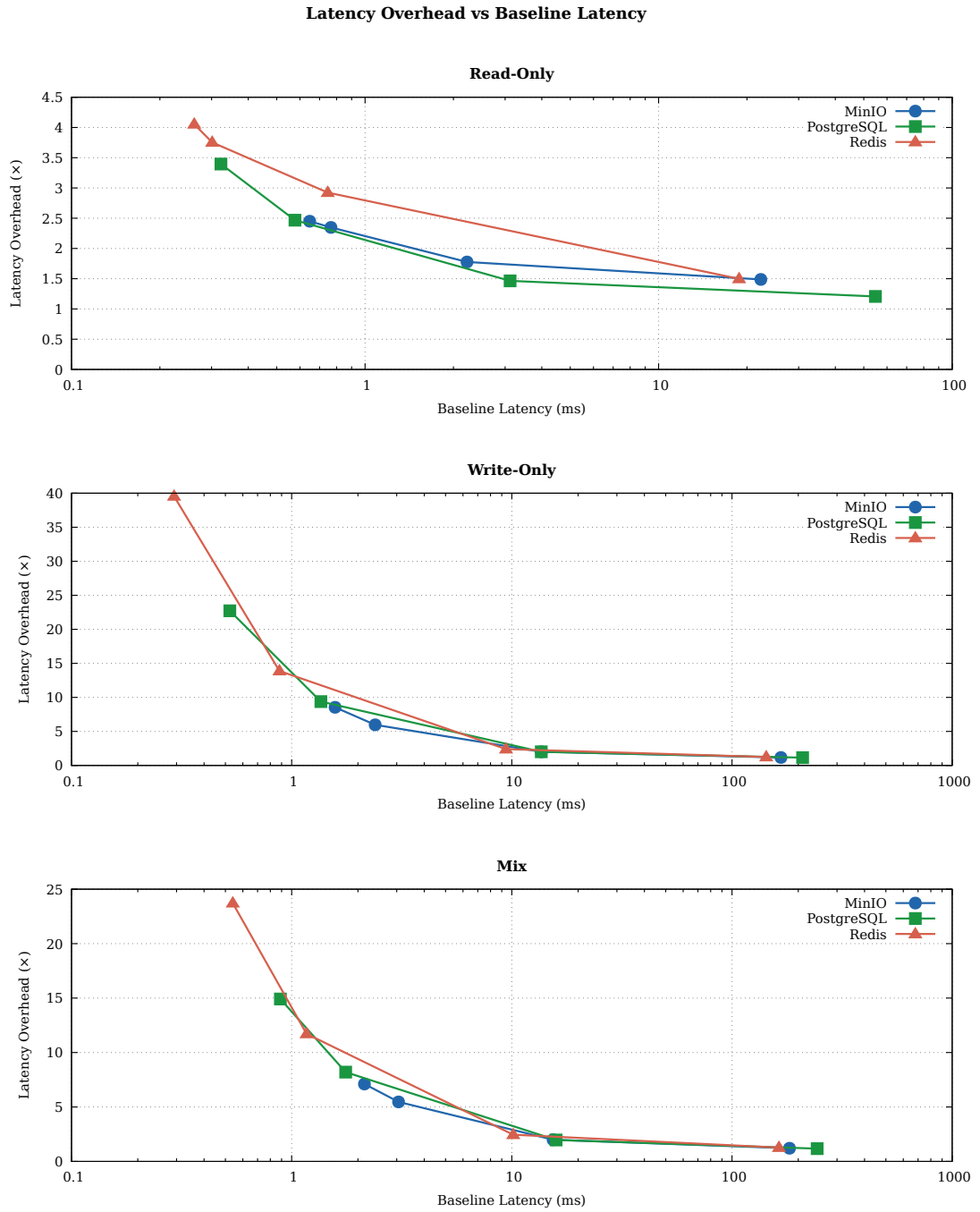


Figure 7.1: Baseline latency vs. Prototype relative latency overhead.

violations must be detected immediately when data is accessed, rather than through periodic verification processes.

The suitability of the prototype also depends on its trust assumptions. The prototype assumes that the execution environment of the application using the **SDK** remains trusted. In particular, it does not protect against attackers who are able to compromise the client environment and directly manipulate internal components such as the local key-to-reference mapping. As with many

integrity verification systems, the guarantees provided by the prototype are therefore bounded by the security of the trusted computing environment in which it operates.

The evaluation results indicate that the proposed prototype is most suitable for applications where integrity guarantees are prioritized over achieving the highest possible performance. Although the introduced tamper detection mechanisms successfully provide immediate verification of data integrity, they also increase latency and reduce throughput in every storage operation. Therefore, adopting the prototype involves accepting a performance cost in exchange for stronger integrity guarantees.

The results also show that the relative impact of this cost depends on the characteristics of the workload. While latency naturally increases and throughput naturally decreases as object size grows, the overhead introduced by the prototype becomes less significant relative to the total operation cost. This does not mean that larger objects achieve better performance. On the contrary, larger objects are inherently slower to process. However, the additional cost associated with integrity verification represents a progressively smaller proportion of the total operation time as object size increases. As a result, the prototype is generally more suitable for applications that manage larger objects than for applications dominated by very small storage operations.

Another important observation is that the prototype operates as an application-level solution. Integrity guarantees are only enforced when storage operations are performed through the provided **SDK**. Consequently, the proposed approach is most appropriate for applications where all access to the underlying storage backend can be controlled through a single software layer. In such environments, any modification performed outside the application workflow can be treated as unauthorized and reliably detected by the integrity verification mechanisms.

The application scenario introduced in the **Selected Implementation Path (4.3)** remains consistent with these findings. Clinical diagnostic records typically require strong integrity guarantees, since unauthorized modifications may have serious consequences for patients and healthcare providers. At the same time, these systems are generally accessed through dedicated software applications rather than direct database or storage access, making **SDK**-enforced integrity verification a realistic assumption. Furthermore, medical reports, diagnostic images, laboratory results and other clinical documents are often accessed individually and at a relatively moderate frequency, reducing the practical impact of the additional verification overhead observed during the evaluation. Many of these records can also be relatively large, making the overhead introduced by the tamper detection mechanisms less noticeable relative to the overall cost of the storage operation.

Another suitable use case for the prototype is as a supporting component for other data protection mechanisms. For example, applications can use the **SDK** to support backup strategies or recovery procedures that depend on identifying integrity violations in stored data. By providing immediate and reliable detection of tampering when data is accessed, the prototype can supply the information required for these higher-level mechanisms to take corrective actions. While the prototype does not prevent tampering or automatically restore compromised data, it can play an important role within broader protection and recovery architectures.

In contrast, the prototype is less suitable for applications where maximizing performance is the primary objective. Examples include caching systems, high-frequency transactional workloads, real-time analytics platforms and other applications that execute a very large number of storage operations and depend on extremely low latency. In these scenarios, even though tamper detection remains effective, the additional processing introduced by integrity verification may represent an unacceptable performance cost.

Overall, the evaluation suggests that the proposed approach is best suited for application-controlled environments where strong integrity guarantees are required, immediate tamper detection is valuable and a moderate performance overhead is considered an acceptable trade-off for increased trust in stored data.

7.3 Assessment of Tamper Detection Mechanisms

The tamper detection validation demonstrated that the proposed integrity mechanisms were able to identify all evaluated integrity violations while accepting legitimate objects. Across all tested scenarios, the prototype behaved as expected, successfully distinguishing between valid operations performed through the **SDK** and unauthorized modifications introduced outside the trusted workflow.

The results also highlight that integrity anchors alone are not sufficient to guarantee protection. Several validation scenarios showed that an attacker may be able to modify stored objects or even create matching anchors for tampered data. The anchor injection attack is a clear example of this situation. Although a valid anchor was manually created and stored in **IPFS**, verification still failed because the object had not been registered through the prototype. This demonstrates the importance of the local key-to-**CID** mapping, which acts as an additional integrity control layer linking application objects to their expected anchors.

The key-to-**CID** mapping was also fundamental for detecting rollback attacks. Even though a previously valid version of the object and its corresponding anchor already existed, restoring that version directly in the storage backend resulted in a verification failure because the current mapping pointed to a newer anchor. Without this mechanism, older valid states could potentially be reintroduced without detection.

Another important observation is the resilience provided by anchor recovery through **IPFS**. The validation results showed that local anchor deletion does not necessarily prevent successful verification when the corresponding anchor remains available through Pinata pinning. In such cases, the prototype can retrieve the anchor from **IPFS** peers and continue the verification process normally. This reduces dependence on the local node and improves robustness against accidental loss of integrity information.

More generally, the results indicate that distributing integrity anchors outside the primary **IPFS** node increases resistance against attacks targeting a single node. While this work relies on Pinata-based pinning for anchor availability, the same principle could be extended by employing additional **IPFS** nodes to replicate and retrieve anchors through the **IPFS** network itself. Such

approaches could further improve resilience by reducing reliance on a single anchor storage location.

Overall, the validation results provide evidence that the combination of integrity anchors, key-to-CID mappings and optional external pinning services is capable of detecting a wide range of tampering attempts, including unauthorized modifications, deletions, local anchor unavailability, injected integrity records and rollback attacks.

7.4 Limitations

7.4.1 Prototype Limitations

Although the prototype successfully provides immediate tamper detection across multiple storage backends, several limitations remain.

A first limitation is the storage abstraction model adopted by the prototype. While OpenDAL allows access to relational backends such as PostgreSQL, it exposes them through a unified object-oriented storage interface rather than their native relational model. As a result, backend-specific features are not exposed to applications. In particular, relational database capabilities such as SQL queries, joins, aggregations and transactions are not directly supported through the SDK. Applications must therefore operate on complete objects rather than relying on backend-specific query mechanisms.

Another limitation is the trust model assumed by the architecture. The prototype relies on the integrity of the execution environment where the SDK operates. In particular, the local key-to-CID mapping is considered trusted and protected from unauthorized modification. If an attacker gains sufficient privileges to compromise the client environment and manipulate this mapping directly, the integrity guarantees provided by the prototype may no longer hold. Therefore, the security guarantees of the prototype are only valid as long as the client environment remains trusted and protected from compromise.

The evaluation results also showed that integrity verification introduces additional processing costs on every storage operation. Because the architecture follows an anchor-per-object strategy combined with action-triggered verification, every write operation requires anchor creation and every read operation requires integrity validation. Although this overhead becomes proportionally smaller for larger objects, it remains present regardless of backend or workload type. The impact is particularly noticeable for very fast operations, where the baseline storage latency is low and the verification cost represents a larger fraction of the total execution time.

The architecture also depends on the availability of integrity anchors. Although locally unavailable anchors can be recovered through the IPFS network from peers that hold the content due to external pinning being enabled, this recovery mechanism still relies on the availability of the corresponding pinned content and accessible IPFS peers. Consequently, the prototype remains dependent on these mechanisms for anchor retrieval. If an anchor cannot be accessed locally or

by finding it in the **IPFS** network, integrity verification cannot be performed for the corresponding object and it may be assumed invalid.

Finally, the current implementation maintains the key-to-**CID** mapping locally within each client environment. While this approach is sufficient for single-user use cases, it becomes a limitation in multi-party scenarios where multiple users or applications interact with the same storage backend. Since each client only maintains its own mapping state, tamper detection remains accurate only for the changes that client has registered through the **SDK**. As a result, the mapping is not shared across participants and cannot serve as a common integrity reference for all users. Given the modular design of the prototype, this limitation could be addressed through alternative mapping implementations, such as a cloud-based service or a replicated mapping mechanism shared among trusted participants. Such approaches would provide a consistent view of object-to-anchor associations across clients and improve suitability for collaborative environments.

7.4.2 Evaluation Limitations

The evaluation presented in this work has several limitations that should be considered when interpreting the results.

The evaluation was conducted using three storage backends: MinIO S3, PostgreSQL and Redis, all deployed in a local environment. While these backends represent different storage paradigms and provide a broad view of the prototype's behavior, they do not cover the full range of storage systems supported by OpenDAL. Ideally, the evaluation would also include remote cloud storage services, particularly for the S3-compatible backend. Initial attempts were made to use public S3 providers, but the available free storage limits were insufficient for the volume of data required by the benchmarking process. As a result, MinIO was deployed locally to provide a controlled and reproducible S3-compatible environment. However, this setup does not capture network latency, bandwidth constraints and other factors typically present in real cloud deployments. Future work could extend the evaluation to additional backend types and remote storage environments.

The evaluation was also constrained by the available storage capacity. Consequently, the largest object size considered was 16 MB. Although the results revealed clear trends across the evaluated sizes, it remains uncertain whether the same behavior would continue for significantly larger objects. Extending the experiments to larger datasets would provide a more complete characterization of the prototype's scalability.

For Redis, the memory limits of the evaluation environment required the use of the LRU eviction policy during the evaluation. Although benchmark data was cleaned between workloads, configurations, backends and repetitions, eviction activity may still have influenced some measurements, particularly for larger object sizes where memory consumption was higher. Therefore, the Redis results should be interpreted with this constraint in mind.

Finally, the evaluation was performed using synthetic benchmark workloads specifically designed to isolate and measure the costs introduced by the integrity mechanisms. While this approach provides controlled and reproducible results, it does not fully capture the behavior of real-world applications. A more comprehensive assessment would involve integrating the proposed prototype into an existing application that already relies on OpenDAL and evaluating its impact under realistic usage patterns and operational workloads.

Chapter 8

Conclusion and Future Work

This chapter concludes the dissertation by summarizing the main outcomes of the work, highlighting its contributions from both research and practical perspectives, and identifying opportunities for future research and development.

8.1 Summary of Findings

This dissertation addressed the problem of ensuring data integrity in storage systems operating in environments where trust cannot be fully assumed, such as multi-party or collaborative settings. In such contexts, conventional storage systems provide strong performance and scalability, but do not offer independent guarantees against tampering. On the other hand, decentralized immutable systems, such as blockchains, provide strong integrity and traceability guarantees, but introduce significant performance and scalability overheads. Alternative immutable storage technologies, such as **IPFS**, follow a different design approach based on content-addressing, offering similar integrity properties while exposing different trade-offs in terms of availability and retrieval performance. These observations motivated the exploration of hybrid architectures that combine conventional storage systems with external integrity-preservation mechanisms.

The state-of-the-art analysis showed that such hybrid systems can be implemented in multiple ways. Existing approaches include introducing blockchain concepts into database systems, integrating database capabilities into blockchain technology or introducing intermediary layers between the two paradigms. These solutions pursue different objectives, including tamper detection, auditability, traceability, verifiable backup and recovery, and improved blockchain querying capabilities and scalability. However, the literature also consistently highlights challenges related to performance overhead, scalability limitations, verification strategies and integration complexity.

Building on these findings, the proposed approach addresses **RQ1** by combining conventional storage systems with an External Immutable Storage (**EIS**) through a client-side architecture centered on tamper detection. Instead of modifying storage engines directly, the system intercepts operations at the storage access layer, allowing integrity mechanisms to be applied transparently from the perspective of the application. Conventional storage systems are used for high-performance

data management, while the **EIS** is responsible for maintaining integrity anchors independently from the backend. Each object version generates a corresponding anchor and verification is performed individually during data access, ensuring that tampering is detected at the moment data is retrieved.

The resulting prototype was implemented as an extension of OpenDAL, enabling tamper detection mechanisms to be applied consistently across multiple storage backends through a unified interface. The immutability and independently verifiable nature of **IPFS** content-addressing, combined with its decentralized storage model and more favorable operational costs and scalability characteristics compared to blockchain technologies under the proposed architecture, motivated its selection as the **EIS**. Integrity anchors are stored in **IPFS**, while a trusted local mapping associates storage objects with their corresponding anchors. The architecture remains modular, allowing alternative immutable storage systems or mapping mechanisms to be incorporated in future implementations without requiring changes to application-level interactions.

The evaluation addressed **RQ3** by assessing how effectively the proposed hybrid architecture can detect data tampering. The results demonstrated that the approach successfully identifies integrity violations across a variety of attack scenarios, including unauthorized modifications, object deletion, anchor unavailability and rollbacks. These findings show that combining conventional storage with an External Immutable Storage can effectively enhance data integrity guarantees compared to relying solely on traditional storage systems.

The evaluation further addressed **RQ2** by analyzing the performance impact of integrating external integrity-preservation mechanisms. The results showed that the integrity mechanisms introduce measurable overhead, increasing latency and reducing throughput compared to baseline storage operations. This overhead behaves mostly as a fixed per-operation cost, making its relative impact more noticeable for fast operations and smaller objects, while becoming proportionally less significant as operation costs increase. Despite these costs, the underlying storage systems retain their scalability characteristics, highlighting the trade-off between stronger integrity guarantees and performance.

Overall, the findings demonstrate that an External Immutable Storage can be effectively integrated with traditional storage systems through a modular client-side architecture. The proposed approach enables tamper detection across multiple storage backends without requiring modifications to the underlying storage engines. Although the integrity mechanisms introduce measurable performance overhead, the evaluation shows that the resulting trade-off remains acceptable for applications where data integrity is a primary concern.

8.2 Main Contributions

This dissertation contributes to the study of hybrid integrity-preservation architectures by investigating a client-centric alternative to the dominant approaches found in the literature. While most existing solutions focus on integrating blockchain technologies directly into database engines or

storage infrastructures, this work expands the research landscape by exploring integrity verification through the storage access layer.

A first contribution is the state-of-the-art analysis of hybrid blockchain-database systems and related integrity-preservation approaches. By consolidating and comparing existing solutions, the dissertation provides a structured overview of the field, identifies common architectural patterns and trade-offs, and highlights research gaps that remain insufficiently explored. This analysis helped position the proposed approach within the broader research landscape and highlighted the limited exploration of client-side integrity verification mechanisms.

These findings led to a further contribution in the proposal of a transparent application-layer architecture for tamper detection across heterogeneous storage systems, implemented through an extension of Apache OpenDAL. This contribution demonstrates that integrity verification can be decoupled from the underlying storage technology and provided through a common abstraction layer, allowing the same tamper detection mechanisms to be applied consistently across different storage backends.

Another contribution is the exploration of **IPFS** as an External Immutable Storage for integrity preservation. This broadens the range of technologies considered within hybrid integrity architectures and provides evidence that decentralized content-addressable storage can serve as a viable alternative to blockchain-based integrity anchoring.

The dissertation also contributes empirical evidence regarding the practical implications of application-layer integrity verification. Through a systematic evaluation, it provides quantitative insights into the performance trade-offs associated with the proposed approach and demonstrates its effectiveness in detecting unauthorized data modifications.

Beyond its academic contributions, the work also has practical relevance. The proposed approach can be integrated into existing OpenDAL-based applications with minimal changes to application logic, making it a viable option for environments that require independent integrity verification of stored data while relying on storage systems that cannot be fully trusted. Its modular design further provides a foundation for future adaptations to different **EISs**, mapping strategies and application requirements, increasing its potential applicability in real-world environments.

Overall, this dissertation contributes both a new architectural perspective and supporting empirical evidence for transparent tamper detection in heterogeneous storage environments, advancing the study of integrity-preservation mechanisms and demonstrating how they can be applied consistently across different storage technologies without requiring modifications to the underlying storage systems.

8.3 Future Work

Several opportunities exist to extend and improve the proposed architecture. A first direction concerns the key-to-**CID** mapping mechanism. The current implementation relies on a trusted local mapping maintained by each client. Future work could explore alternative approaches that reduce or eliminate this trust assumption, such as storing mapping information within **IPFS** itself

or another immutable storage system. Approaches that improve support for multiple clients are also worth investigating. Possible solutions include cloud-hosted mappings or mechanisms for mapping replication across multiple trusted clients. Additionally, in-memory mapping implementations could be explored as a way to improve performance. Studying these alternatives could address some of the limitations identified in this work.

Another promising direction is the evaluation of additional External Immutable Storage technologies. Although **IPFS** was selected as the most suitable solution for the proposed architecture, blockchain-based implementations remain valuable from a research perspective. Using blockchain for anchoring mechanisms would allow for a direct comparison between different immutable storage technologies within the same architectural model and contribute further evidence regarding their respective trade-offs.

The evaluation itself could also be extended. Future studies should consider larger datasets, larger object sizes and additional storage backends. In particular, evaluating the prototype with cloud-based storage systems would provide a more realistic assessment of its behavior under network latency. Furthermore, integrating the prototype into existing applications and evaluating it under realistic workloads would provide insights that cannot be fully captured through synthetic benchmarks alone.

Beyond these improvements, two broader research challenges remain open. The first concerns support for relational and **SQL**-based interaction models. The proposed solution relies on an object-oriented abstraction layer that provides a common interface across different storage backends. Although this abstraction enables backend independence, it does not naturally support relational concepts or direct **SQL** interactions. Future work could investigate mechanisms for intercepting and validating relational operations, either by processing **SQL** operations directly or through integration with Object-Relational Mappings (ORMs), which are widely used in modern web frameworks and could provide a practical interception point while preserving a higher-level programming interface.

The second challenge concerns remediation after tampering detection. Once tampering is identified, deciding how to recover from the incident remains application-dependent. Different environments may require different responses, ranging from application-level integrity violation notifications (such as the exceptions currently raised by the prototype) followed by manual corrective actions, to dashboard-based alerting mechanisms such as those provided by file integrity monitoring systems like Wazuh [61], or even automated restoration from backups. Future work could explore flexible remediation mechanisms that allow clients to select and configure recovery strategies according to their specific operational requirements.

References

- [1] AlexTrofalealeo. *SecureDAL*. June 2026. URL: <https://github.com/AlexTrofalealeo/SecureDAL> (visited on 06/15/2026).
- [2] Morteza Alizadeh, Karl Andersson, and Olov Schelén. “Efficient Decentralized Data Storage Based on Public Blockchain and IPFS”. In: *2020 IEEE Asia-Pacific Conference on Computer Science and Data Engineering (CSDE)*. Dec. 2020, pp. 1–8. DOI: [10.1109/CSDE50874.2020.9411599](https://doi.org/10.1109/CSDE50874.2020.9411599). URL: <https://ieeexplore.ieee.org/abstract/document/9411599>.
- [3] Leonardo Aniello et al. “A Prototype Evaluation of a Tamper-Resistant High Performance Blockchain-Based Transaction Log for a Distributed Database”. In: *2017 13th European Dependable Computing Conference (EDCC)*. Sept. 2017, pp. 151–154. DOI: [10.1109/EDCC.2017.31](https://doi.org/10.1109/EDCC.2017.31). URL: <https://ieeexplore.ieee.org/abstract/document/8123568>.
- [4] Jyotsna Anthal, Shakir Choudhary, and Ravikumar Shettiyar. “Decentralizing File Sharing: The Potential of Blockchain and IPFS”. In: *2023 International Conference on Advancement in Computation & Computer Technologies (InCACCT)*. May 2023, pp. 773–777. DOI: [10.1109/InCACCT57535.2023.10141817](https://doi.org/10.1109/InCACCT57535.2023.10141817). URL: <https://ieeexplore.ieee.org/document/10141817>.
- [5] Panagiotis Antonopoulos et al. “SQL Ledger: Cryptographically Verifiable Data in Azure SQL Database”. In: *Proceedings of the 2021 International Conference on Management of Data*. SIGMOD ’21. New York, NY, USA: Association for Computing Machinery, June 2021, pp. 2437–2449. ISBN: 978-1-4503-8343-1. DOI: [10.1145/3448016.3457558](https://doi.org/10.1145/3448016.3457558). URL: <https://dl.acm.org/doi/10.1145/3448016.3457558>.
- [6] *Apache HBase*. URL: <https://hbase.apache.org/> (visited on 06/15/2026).
- [7] Ruba Awadallah and Azman Samsudin. “Using Blockchain in Cloud Computing to Enhance Relational Database Security”. In: *IEEE Access* 9 (2021), pp. 137353–137366. ISSN: 2169-3536. DOI: [10.1109/ACCESS.2021.3117733](https://doi.org/10.1109/ACCESS.2021.3117733). URL: <https://ieeexplore.ieee.org/document/9558818>.
- [8] Juan Benet. “Ipfs-content addressed, versioned, p2p file system”. In: *arXiv preprint arXiv:1407.3561* (2014). URL: <https://arxiv.org/abs/1407.3561>.
- [9] Vitalik Buterin et al. “A next-generation smart contract and decentralized application platform”. In: *white paper* 3.37 (2014), pp. 2–1. URL: https://ethereum.org/content/whitepaper/whitepaper-pdf/Ethereum_Whitepaper_-_Buterin_2014.pdf.

- [10] Yun Chen et al. “DEPLEST: A blockchain-based privacy-preserving distributed database toward user behaviors in social networks”. In: *Information Sciences* 501 (Oct. 2019), pp. 100–117. ISSN: 0020-0255. DOI: [10.1016/j.ins.2019.05.092](https://doi.org/10.1016/j.ins.2019.05.092). URL: <https://www.sciencedirect.com/science/article/pii/S0020025519305250>.
- [11] Mohammad Javed Morshed Chowdhury et al. “Blockchain Versus Database: A Critical Analysis”. In: *2018 17th IEEE International Conference On Trust, Security And Privacy In Computing And Communications/ 12th IEEE International Conference On Big Data Science And Engineering (TrustCom/BigDataSE)*. ISSN: 2324-9013. Aug. 2018, pp. 1348–1353. DOI: [10.1109/TrustCom/BigDataSE.2018.00186](https://doi.org/10.1109/TrustCom/BigDataSE.2018.00186). URL: <https://ieeexplore.ieee.org/abstract/document/8456055>.
- [12] Sara Cohen, Adam Rosenthal, and Aviv Zohar. “Reasoning about the Future in Blockchain Databases”. In: *2020 IEEE 36th International Conference on Data Engineering (ICDE)*. ISSN: 2375-026X. Apr. 2020, pp. 1930–1933. DOI: [10.1109/ICDE48307.2020.00206](https://doi.org/10.1109/ICDE48307.2020.00206). URL: <https://ieeexplore.ieee.org/document/9101185>.
- [13] OpenDAL Contributors. *OpenDAL*. June 2026. URL: <https://github.com/apache/opendal> (visited on 06/15/2026).
- [14] Tien Tuan Anh Dinh et al. “Blockbench: A framework for analyzing private blockchains”. In: *Proceedings of the 2017 ACM international conference on management of data*. 2017, pp. 1085–1100. URL: <https://dl.acm.org/doi/abs/10.1145/3035918.3064033>.
- [15] *Ethereum - The complete guide from ethereum.org*. en. URL: <https://ethereum.org/> (visited on 06/15/2026).
- [16] Xiaowen Feng et al. “Verifiable Decentralized Access Control for Distributed Databases”. In: *2020 International Conference on Cyber-Enabled Distributed Computing and Knowledge Discovery (CyberC)*. Oct. 2020, pp. 248–253. DOI: [10.1109/CyberC49757.2020.00046](https://doi.org/10.1109/CyberC49757.2020.00046). URL: <https://ieeexplore.ieee.org/document/9329408>.
- [17] Edoardo Gaetani et al. “Blockchain-based database to ensure data integrity in cloud computing environments”. en. In: Num Pages: 10. Jan. 2017. URL: <https://eprints.soton.ac.uk/411996/>.
- [18] Justin S. Gazsi et al. “VAULT: A Scalable Blockchain-Based Protocol for Secure Data Access and Collaboration”. In: *2021 IEEE International Conference on Blockchain (Blockchain)*. Dec. 2021, pp. 376–381. DOI: [10.1109/Blockchain53845.2021.00059](https://doi.org/10.1109/Blockchain53845.2021.00059). URL: <https://ieeexplore.ieee.org/document/9680475>.
- [19] Zerui Ge et al. “Hybrid blockchain database systems: design and performance”. en. In: *Proceedings of the VLDB Endowment* 15.5 (Jan. 2022), pp. 1092–1104. ISSN: 2150-8097. DOI: [10.14778/3510397.3510406](https://doi.org/10.14778/3510397.3510406). URL: <https://dl.acm.org/doi/10.14778/3510397.3510406>.
- [20] PostgreSQL Global Development Group. *PostgreSQL*. en. June 2026. URL: <https://www.postgresql.org/> (visited on 06/15/2026).
- [21] Muhammad El-Hindi et al. “BlockchainDB: a shared database on blockchains”. en. In: *Proceedings of the VLDB Endowment* 12.11 (July 2019), pp. 1597–1609. ISSN: 2150-8097. DOI: [10.14778/3342263.3342636](https://doi.org/10.14778/3342263.3342636). URL: <https://dl.acm.org/doi/10.14778/3342263.3342636>.

- [22] Alan Hong, Caiping Sun, and Meixuan Chen. “A Survey of Distributed Database Systems based on Blockchain”. In: *2020 3rd International Conference on Smart BlockChain (Smart-Block)*. Oct. 2020, pp. 191–196. DOI: 10.1109/SmartBlock52591.2020.00042. URL: <https://ieeexplore.ieee.org/document/9415662>.
- [23] *hyperledger/fabric*. June 2026. URL: <https://github.com/hyperledger/fabric> (visited on 06/15/2026).
- [24] IBM. *Cost of a Data Breach Report 2025*. en. Cost of a Data Breach. 2025. URL: <https://www.ibm.com/reports/data-breach>.
- [25] Rares Ifrim, Dumitrel Loghin, and Decebal Popescu. “Baldur: A Hybrid Blockchain Database with FPGA or GPU Acceleration”. In: *Proceedings of the 1st Workshop on Verifiable Database Systems. VDBS '23*. New York, NY, USA: Association for Computing Machinery, June 2023, pp. 19–27. ISBN: 979-8-4007-0775-9. DOI: 10.1145/3595647.3595650. URL: <https://dl.acm.org/doi/10.1145/3595647.3595650>.
- [26] *InfluxData*. URL: <https://www.influxdata.com/> (visited on 06/15/2026).
- [27] *IPFS: Building blocks for a better web*. en-US. URL: <https://ipfs.tech> (visited on 06/15/2026).
- [28] *ipfs/kubo*. June 2026. URL: <https://github.com/ipfs/kubo> (visited on 06/15/2026).
- [29] Meenu Jain, Manisha Jailia, and Manisha Agarwal. “Performance Analysis of Blockchain Technology using Inter Planetary File System (IPFS) database in comparison with Block chain database”. In: *2023 International Conference on Computer Communication and Informatics (ICCCI)*. ISSN: 2473-7577. Jan. 2023, pp. 1–4. DOI: 10.1109/ICCCI56745.2023.10128177. URL: <https://ieeexplore.ieee.org/document/10128177>.
- [30] Reshma Kamath. “Food traceability on blockchain: Walmart’s pork and mango pilots with IBM”. In: *The Journal of the British Blockchain Association 1.1* (June 2018). URL: <https://jbba.scholasticahq.com/article/3712-food-traceability-on-blockchain-walmart-s-pork-and-mango-pilots-with-ibm>.
- [31] Randhir Kumar and Rakesh Tripathi. “Implementation of Distributed File Storage and Access Framework using IPFS and Blockchain”. In: *2019 Fifth International Conference on Image Information Processing (ICIIP)*. ISSN: 2640-074X. Nov. 2019, pp. 246–251. DOI: 10.1109/ICIIP47207.2019.8985677. URL: <https://ieeexplore.ieee.org/abstract/document/8985677>.
- [32] Jingwen Li and Jian Wang. “Index Design of Electronic Medical Record Database Using Blockchain”. In: *2020 5th International Conference on Mechanical, Control and Computer Engineering (ICMCCE)*. Dec. 2020, pp. 2003–2008. DOI: 10.1109/ICMCCE51767.2020.00438. URL: <https://ieeexplore.ieee.org/document/9421611>.
- [33] Yang Li et al. “EtherQL: A Query Layer for Blockchain System”. en. In: *Database Systems for Advanced Applications*. Ed. by Selçuk Candan et al. Cham: Springer International Publishing, 2017, pp. 556–567. ISBN: 978-3-319-55699-4. DOI: 10.1007/978-3-319-55699-4_34.
- [34] Jie Lian, Siqian Wang, and Yanmiao Xie. “TDRB: An Efficient Tamper-Proof Detection Middleware for Relational Database Based on Blockchain Technology”. In: *IEEE Access* 9 (2021), pp. 66707–66722. ISSN: 2169-3536. DOI: 10.1109/ACCESS.2021.3076235. URL: <https://ieeexplore.ieee.org/document/9417201>.

- [35] Xueping Liang et al. “ProvChain: A Blockchain-Based Data Provenance Architecture in Cloud Environment with Enhanced Privacy and Availability”. In: *2017 17th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGRID)*. May 2017, pp. 468–477. DOI: [10.1109/CCGRID.2017.8](https://doi.org/10.1109/CCGRID.2017.8). URL: <https://ieeexplore.ieee.org/document/7973733>.
- [36] Chongjun Liu, Yongzhen Li, and Zhexue Jin. “Blockchain Data Query Scheme Based on Auxiliary Storage”. In: *2023 5th International Academic Exchange Conference on Science and Technology Innovation (IAECST)*. Dec. 2023, pp. 367–370. DOI: [10.1109/IAECST60924.2023.10503158](https://doi.org/10.1109/IAECST60924.2023.10503158). URL: <https://ieeexplore.ieee.org/document/10503158>.
- [37] Zhonghao Liu et al. “A Secure and Reliable Blockchain-based Audit Log System”. In: *ICC 2024 - IEEE International Conference on Communications*. ISSN: 1938-1883. June 2024, pp. 2010–2015. DOI: [10.1109/ICC51166.2024.10623012](https://doi.org/10.1109/ICC51166.2024.10623012). URL: <https://ieeexplore.ieee.org/document/10623012/references>.
- [38] Aashay Maheshwarkar, Animesh Kumar, and Manas Gupta. “Securing Database Backups Using Blockchain and Peer-to-Peer Network”. In: *2021 2nd International Conference on Communication, Computing and Industry 4.0 (C2I4)*. Dec. 2021, pp. 1–6. DOI: [10.1109/C2I454156.2021.9689443](https://doi.org/10.1109/C2I454156.2021.9689443). URL: <https://ieeexplore.ieee.org/document/9689443>.
- [39] Sujaya Maiyya et al. “Database and distributed computing fundamentals for scalable, fault-tolerant, and consistent maintenance of blockchains”. In: *Proc. VLDB Endow.* 11.12 (Aug. 2018), pp. 2098–2101. ISSN: 2150-8097. DOI: [10.14778/3229863.3229877](https://doi.org/10.14778/3229863.3229877). URL: <https://dl.acm.org/doi/10.14778/3229863.3229877>.
- [40] *minio*. June 2026. URL: <https://github.com/minio/minio> (visited on 06/15/2026).
- [41] *MongoDB: The World’s Leading Modern Data Platform*. en-us. URL: <https://www.mongodb.com/> (visited on 06/15/2026).
- [42] Muhammad Muzammal, Qiang Qu, and Bulat Nasrulin. “Renovating blockchain with distributed databases: An open source system”. In: *Future Generation Computer Systems* 90 (Jan. 2019), pp. 105–117. ISSN: 0167-739X. DOI: [10.1016/j.future.2018.07.042](https://doi.org/10.1016/j.future.2018.07.042). URL: <https://www.sciencedirect.com/science/article/pii/S0167739X18308732>.
- [43] Satoshi Nakamoto. “Bitcoin: A peer-to-peer electronic cash system”. In: *Available at SSRN 3440802* (2008). URL: <https://bitcoin.org/bitcoin.pdf>.
- [44] *Neo4j Graph Intelligence Platform*. en. June 2026. URL: <https://neo4j.com/> (visited on 06/15/2026).
- [45] Emmanuel Nyaletey et al. “BlockIPFS - Blockchain-Enabled Interplanetary File System for Forensic and Trusted Data Traceability”. In: *2019 IEEE International Conference on Blockchain (Blockchain)*. July 2019, pp. 18–25. DOI: [10.1109/Blockchain.2019.00012](https://doi.org/10.1109/Blockchain.2019.00012). URL: <https://ieeexplore.ieee.org/abstract/document/8946164>.
- [46] *One Layer, All Storage | Apache OpenDAL™*. en. URL: <https://opendal.apache.org/> (visited on 06/15/2026).
- [47] *Parity*. en. URL: <https://www.parity.io> (visited on 06/15/2026).
- [48] Andy Pavlo. *h-store*. June 2026. URL: <https://github.com/apavlo/h-store> (visited on 06/15/2026).

- [49] Yanqing Peng et al. “FalconDB: Blockchain-based Collaborative Database”. In: *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data*. SIGMOD ’20. New York, NY, USA: Association for Computing Machinery, May 2020, pp. 637–652. ISBN: 978-1-4503-6735-6. DOI: [10.1145/3318464.3380594](https://doi.org/10.1145/3318464.3380594). URL: <https://dl.acm.org/doi/10.1145/3318464.3380594>.
- [50] Zhe Peng et al. “VQL: Providing Query Efficiency and Data Authenticity in Blockchain Systems”. In: *2019 IEEE 35th International Conference on Data Engineering Workshops (ICDEW)*. ISSN: 2473-3490. Apr. 2019, pp. 1–6. DOI: [10.1109/ICDEW.2019.00-44](https://doi.org/10.1109/ICDEW.2019.00-44). URL: <https://ieeexplore.ieee.org/document/8750921>.
- [51] Pinata | IPFS. en. URL: <https://pinata.cloud/ipfs> (visited on 05/29/2026).
- [52] Lourenço Preto et al. “Dependable Food Traceability Data Through Blockchain and Database Integration”. In: *2025 20th European Dependable Computing Conference Companion Proceedings (EDCC-C)*. Apr. 2025, pp. 163–167. DOI: [10.1109/EDCC-C66476.2025.00050](https://doi.org/10.1109/EDCC-C66476.2025.00050). URL: <https://ieeexplore.ieee.org/document/11144795>.
- [53] Kritika Rani and Chinmay Sharma. “Tampering Detection of Distributed Databases using Blockchain Technology”. In: *2019 Twelfth International Conference on Contemporary Computing (IC3)*. ISSN: 2572-6129. Aug. 2019, pp. 1–4. DOI: [10.1109/IC3.2019.8844938](https://doi.org/10.1109/IC3.2019.8844938). URL: <https://ieeexplore.ieee.org/document/8844938>.
- [54] Redis. *Redis - Real-time data for agents & apps*. en-US. URL: <https://redis.io/> (visited on 06/15/2026).
- [55] Pingcheng Ruan et al. “Blockchains vs. Distributed Databases: Dichotomy and Fusion”. In: *Proceedings of the 2021 International Conference on Management of Data*. SIGMOD ’21. New York, NY, USA: Association for Computing Machinery, June 2021, pp. 1504–1517. ISBN: 978-1-4503-8343-1. DOI: [10.1145/3448016.3452789](https://doi.org/10.1145/3448016.3452789). URL: <https://dl.acm.org/doi/10.1145/3448016.3452789>.
- [56] Emre Şafak, Arif Furkan Mendi, and Tolga Erol. “Hybrid Database Design Combination of Blockchain And Central Database”. In: *2019 3rd International Symposium on Multidisciplinary Studies and Innovative Technologies (ISMSIT)*. Oct. 2019, pp. 1–5. DOI: [10.1109/ISMSIT.2019.8932763](https://doi.org/10.1109/ISMSIT.2019.8932763). URL: <https://ieeexplore.ieee.org/document/8932763>.
- [57] Ankur Sharma et al. “Blurring the Lines between Blockchains and Database Systems: the Case of Hyperledger Fabric”. In: *Proceedings of the 2019 International Conference on Management of Data*. SIGMOD ’19. New York, NY, USA: Association for Computing Machinery, June 2019, pp. 105–122. ISBN: 978-1-4503-5643-5. DOI: [10.1145/3299869.3319883](https://doi.org/10.1145/3299869.3319883). URL: <https://dl.acm.org/doi/10.1145/3299869.3319883>.
- [58] Yajun Wang, Chao-Hsien Hsieh, and Changfeng Li. “Research and Analysis on the Distributed Database of Blockchain and Non- Blockchain”. In: *2020 IEEE 5th International Conference on Cloud Computing and Big Data Analytics (ICCCBDA)*. Apr. 2020, pp. 307–313. DOI: [10.1109/ICCCBDA49378.2020.9095589](https://doi.org/10.1109/ICCCBDA49378.2020.9095589). URL: <https://ieeexplore.ieee.org/document/9095589>.
- [59] Yuchao Wang et al. “aChain: A SQL-Empowered Analytical Blockchain as a Database”. In: *IEEE Transactions on Computers* 72.11 (Nov. 2023), pp. 3099–3112. ISSN: 1557-9956. DOI: [10.1109/TC.2023.3287036](https://doi.org/10.1109/TC.2023.3287036). URL: <https://ieeexplore.ieee.org/abstract/document/10154133>.

- [60] Brad Warbiany. *White Paper: The Long-Term Case for HDD Storage*. en. White Paper. Western Digital, Oct. 2025, p. 3. URL: https://documents.westerndigital.com/content/dam/doc-library/en_us/assets/public/western-digital/collateral/white-paper/white-paper-the-long-term-case-for-hdd-storage.pdf (visited on 01/17/2026).
- [61] *Wazuh - Open Source XDR. Open Source SIEM*. en-US. URL: <https://wazuh.com/> (visited on 06/14/2026).
- [62] Qian Wei et al. “A Survey of Blockchain Data Management Systems”. In: *ACM Trans. Embed. Comput. Syst.* 21.3 (May 2022), 25:1–25:28. ISSN: 1539-9087. DOI: 10.1145/3502741. URL: <https://dl.acm.org/doi/10.1145/3502741>.
- [63] Cheng Xu, Ce Zhang, and Jianliang Xu. “vChain: Enabling Verifiable Boolean Range Queries over Blockchain Databases”. In: *Proceedings of the 2019 International Conference on Management of Data*. SIGMOD ’19. New York, NY, USA: Association for Computing Machinery, June 2019, pp. 141–158. ISBN: 978-1-4503-5643-5. DOI: 10.1145/3299869.3300083. URL: <https://dl.acm.org/doi/10.1145/3299869.3300083>.
- [64] Xinying Yang et al. “LedgerDB: a centralized ledger database for universal audit and verification”. In: *Proc. VLDB Endow.* 13.12 (Aug. 2020), pp. 3138–3151. ISSN: 2150-8097. DOI: 10.14778/3415478.3415540. URL: <https://dl.acm.org/doi/10.14778/3415478.3415540>.
- [65] Qihong Zheng et al. “An Innovative IPFS-Based Storage Model for Blockchain”. In: *2018 IEEE/WIC/ACM International Conference on Web Intelligence (WI)*. Dec. 2018, pp. 704–708. DOI: 10.1109/WI.2018.000–8. URL: <https://ieeexplore.ieee.org/abstract/document/8609675>.
- [66] Changhao Zhu et al. “A Survey on the Integration of Blockchains and Databases”. en. In: *Data Science and Engineering* 8.2 (June 2023), pp. 196–219. ISSN: 2364-1541. DOI: 10.1007/s41019-023-00212-z. URL: <https://doi.org/10.1007/s41019-023-00212-z>.
- [67] Yanchao Zhu et al. “SEBDB: Semantics Empowered BlockChain DataBase”. In: *2019 IEEE 35th International Conference on Data Engineering (ICDE)*. ISSN: 2375-026X. Apr. 2019, pp. 1820–1831. DOI: 10.1109/ICDE.2019.00198. URL: <https://ieeexplore.ieee.org/document/8731416>.
- [68] Jinglin Zou et al. “Integrated Blockchain and Cloud Computing Systems: A Systematic Survey, Solutions, and Challenges”. In: *ACM Comput. Surv.* 54.8 (Oct. 2021), 160:1–160:36. ISSN: 0360-0300. DOI: 10.1145/3456628. URL: <https://dl.acm.org/doi/10.1145/3456628>.