SCHOOL OF COMPUTATION, INFORMATION AND TECHNOLOGY — INFORMATICS

TECHNISCHE UNIVERSITÄT MÜNCHEN

Bachelor's Thesis in Information Systems

Tamperproof Logging System for GDPR-compliant Key-Value Stores

Christian Karidas

SCHOOL OF COMPUTATION, INFORMATION AND TECHNOLOGY — INFORMATICS

TECHNISCHE UNIVERSITÄT MÜNCHEN

Bachelor's Thesis in Information Systems

Tamperproof Logging System for GDPR-compliant Key-Value Stores

Manipulationssicheres Protokollierungssystem für DSGVO-konforme Key-Value Stores

Author:	Christian Karidas
Examiner:	Prof. Dr. Pramod Bhatotia
Supervisor:	Dr. Dimitrios Stavrakakis
Submission Date:	16.06.2025

I confirm that this bachelor's thesis is my own work and I have documented all sources and material used.

Munich, 16.06.2025 Christian Karidas

Abstract

This bachelor's thesis addresses the challenge of developing a high-performance, secure, and tamper-evident logging system designed to support data management infrastructures such as key-value stores in ensuring compliance with regulations like the GDPR. Existing logging solutions often fall short of the strict requirements imposed by data protection regulations, particularly when it comes to maintaining verifiable audit trails without degrading system performance. To address this, we propose a new logging architecture based on a lock-free, multi-threaded design optimized for batched writes. Key features include buffered, asynchronous processing of log entries by dedicated writer threads, which handle serialization, compression, and authenticated encryption before appending data atomically to immutable, segmented storage files.

The system's effectiveness is demonstrated by substantial and measurable results obtained through extensive empirical benchmarking. It achieves a log throughput rate of approximately 2.08 GiB per second, coupled with a low write amplification ratio of 0.109, highlighting its storage efficiency. The architecture scales robustly, demonstrating near-linear speedup up to 16 writer threads, and ensures per-batch integrity through authenticated encryption, maintaining tamper-evidence and data confidentiality.

Beyond technical performance, the system offers practical value as a modular, high-efficiency logging layer for GDPR-compliant infrastructures. Designed as a standalone component, it can be easily integrated into existing setups—such as proxies or middleware for key-value stores—allowing organizations to maintain secure and verifiable audit trails. This enables confident responses to regulatory responsibilities and audits, while preserving key performance characteristics like low-latency writes and throughput scalability. The work provides a solid foundation for future improvements in secure data management within high-performance distributed systems.

The source code for this project can be found at: https://github.com/chriskari/bachelor-thesis.

Contents

1 Introduction

The General Data Protection Regulation (GDPR), enacted in May 2018, has fundamentally reshaped how organizations collect, store, and process personal data, imposing stringent requirements on data privacy, security, and user control [18]. Designed to enhance individuals' rights—such as the right to be forgotten, data portability, and explicit consent—GDPR has introduced new challenges for database management systems [18]. In particular, key-value stores, which form the foundation of many modern services [15] with their simplicity and high performance, were originally not conceived for fine-grained data erasure, comprehensive metadata tracking, or rigorous compliance auditing [2, 26]. As a result, integrating GDPR's principles into existing storage engines often incurs significant performance overheads, storage inefficiencies, and increased system complexity [26]. Amid growing regulatory demands and technical challenges, this work proposes a logging system designed from the ground up to meet GDPR's audit, security, and tamper-proofing mandates without substantially sacrificing throughput or scalability.

Considering a real-world scenario in which an online service is faced with a subject access request, the service must identify and report all personal data held about the user, its processing purposes, data sharing practices, and retention policies [7]. Beyond individual cases, external auditors or data protection authorities may also conduct broader compliance audits that could require reconstructing individual processing steps—such as read, write, or deletion events relating to specific users' data—to assess whether the company's data-processing practices across all stored data comply with legal and regulatory standards [8, 9]. Without reliable, tamper-evident records, the service risks regulatory fines or reputational damage [9]. Yet naive logging approaches—appending every operation individually in plaintext—can flood disks and networks, degrade request latencies, and leave sensitive information exposed. Here lies the tension between ensuring accountability and maintaining system performance under heavy write loads. Traditional database logging, focused on crash recovery rather than compliance auditing, falls short of GDPR's requirements for encrypted, integrity-protected, and easily exportable logs.

Academic and industrial research has explored secure logging through various approaches, with different systems prioritizing distinct aspects of the challenge. High-performance logging systems achieve excellent throughput through compile-time optimizations and asynchronous buffering [33, 1, 28], but typically forgo cryptographic security features. Secure logging research, including forward-secure authentication schemes and blockchain-based approaches, provides strong tamper-evidence and integrity guarantees [19, 37, 40], but often at the expense of performance or simplicity. GDPR-compliant systems address regulatory requirements through specialized architectures—pseudonymization with blockchain immutability or centralized log aggregation [41, 12, 30]—but are designed for specific deployment scenarios rather than general-purpose integration. Among the surveyed approaches, none simultaneously delivers the combination of high write throughput, end-to-end encryption, compression, tamper evidence, and simple integration as a standalone library that our system targets.

This gap motivates the problem statement of this thesis: can we build a logging system that, under stringent GDPR constraints, delivers high write throughput, minimizes write amplification, and provides robust security guarantees? Specifically, this work addresses the challenge of designing a system that efficiently handles a high volume of log entries, both in terms of the number of individual records and the total data size written to disk. To maximize efficiency and performance all data written to persistent storage must be compressed to significantly reduce the storage footprint and optimize I/O operations. In compliance with GDPR's mandate for data protection, all log entries must be encrypted before being written to storage, safeguarding sensitive information from unauthorized access [8]. Finally, a robust mechanism must be in place to cryptographically detect any unauthorized modifications or deletions of stored log entries, ensuring the integrity and trustworthiness of audit trails [8].

To address this, we introduce a lock-free, multi-threaded architecture optimized for batched writes. Client threads enqueue log entries via a minimal-latency Application Programming Interface (API) into a multi-producer, multi-consumer lock-free queue. Dedicated writer threads then drain entries in bulk, grouping each batch by destination file before serializing, compressing and encrypting. This not only ensures confidentiality but also provides per-entry authentication, offering tamper-proof guarantees for each individual batch of log entries. Batches are appended to append-only file segments, with atomic primitives ensuring safe concurrent writes. Segments automatically rotate upon hitting configured size thresholds and are named with embedded timestamps to simplify archival and retrieval.

To evaluate the system's performance and security guarantees, this thesis presents a comprehensive series of benchmarks that examine its behavior across a wide range of configurations and workloads. The evaluation focuses on three central metrics—throughput, client-perceived latency, and write amplification—while exploring the effects of batching, concurrency, buffering, file rotation, compression, and encryption. This analysis provides a detailed understanding of how different design parameters influence system behavior and performance trade-offs.

The broader impact of this work lies in its ability to serve as a modular, high-efficiency logging layer for GDPR-compliant applications. By architecting the system as a standalone component decoupled from the underlying storage engine, it integrates seamlessly into existing infrastructures—acting as a transparent proxy in front of data storage systems. This enables organizations to establish secure, verifiable audit trails without intrusive changes to their infrastructure. In doing so, the system helps bridge the gap between legal compliance and practical deployability, allowing services to respond confidently to audit demands while preserving performance-critical characteristics such as low-latency writes and horizontal scalability.

In summary, this thesis makes three primary contributions. First, we present a lock-free, batched write architecture for secure and tamper-proof logging that achieves high throughput under encryption and compression. Second, we demonstrate a concurrent file rotation and atomic append mechanism that scales across multiple writer threads. Finally, we provide a comprehensive evaluation showing the system's performance and integrity guarantees under both realistic workloads and microbenchmarks. Together, these developments provide a strong starting point for GDPR-compliant data management in high-performance distributed systems.

2 Background

2.1 GDPR and Legal Context

2.1.1 Introduction to GDPR

The General Data Protection Regulation (GDPR), formally known as Regulation (EU) 2016/679, is the cornerstone of data protection legislation within the European Union. Enforced since May 25, 2018, it governs how personal data of individuals within the EU must be handled by both EU-based and non-EU-based organizations [18]. The regulation was introduced to create a uniform set of data privacy laws across Europe, enhance data subjects' control over their personal data, and impose greater accountability on data controllers and processors [38].

Under the GDPR, a data subject is any identified or identifiable natural person whose personal data is being processed [5]. A data controller is the entity—such as an organization or public authority—that determines the purposes and means of processing personal data, and therefore holds primary responsibility for ensuring compliance with GDPR obligations [5]. In contrast, a data processor is a party that processes personal data on behalf of the controller, typically under a contractual obligation [5]. These roles are fundamental to the GDPR's structure of accountability and legal responsibility[5].

At its core, GDPR redefines data protection by placing individuals' rights and their control over personal data at the center [7]. It imposes obligations not only on how data is processed but also on how that processing is documented, monitored, and made transparent to data subjects and regulatory authorities [8].

2.1.2 Key GDPR Principles and Compliance Mechanisms

Among the seven principles outlined in Article 5 of the GDPR, several have direct technical implications for systems that handle personal data. The following principles are particularly relevant for logging and auditing operations:

- Accountability (Art. 5(2)): Organizations must be able to demonstrate compliance with all data protection principles [6]. This implies the need for mechanisms that record and prove how and when personal data was accessed or modified.

- Integrity and Confidentiality (Art. 5(1)(f)): Personal data must be processed in a manner that ensures its security against unauthorized access, alteration, or deletion [6]. Accordingly, audit trails containing personal data must be stored in a way that is tamper-evident, cryptographically secured, and provides authenticity proofs.
- Lawfulness, Fairness, and Transparency (Art. 5(1)(a)): Transparency implies that processing activities on personal data can be audited and reviewed [6]. Logging helps fulfill this principle by creating a traceable record of such activities.

To comply with these principles, especially the accountability requirement, organizations must maintain comprehensive audit trails for all operations involving personal data[12]. These audit trails—often referred to as GDPR logs—should include what data was accessed or modified (i.e., the specific personal data relating to a data subject), by whom (i.e., the identity of the individual or system performing the operation, typically under the data controller's or processor's responsibility), at what time, the type of operation performed (e.g., read, update, delete), and the purpose or context of the access (where applicable) [12].

Logs serve multiple compliance and operational purposes. They enable post-incident investigations in case of data breaches or policy violations, and they provide a verifiable trail of evidence during audits conducted by Data Protection Authorities (DPAs) [17] [33]. In addition, logs support the implementation of the right to access data by enabling traceability of data lifecycle operations, demonstrating what personal data an organization processes and how it has been used [7].

2.1.3 GDPR's Implications for System Design

The legal expectations imposed by the GDPR do not translate directly into concrete system design requirements and are, in fact, rather vague [6]. Thus, one should interpret the law carefully when deriving system design requirements. Generally, though, we can define that logging systems designed to support GDPR compliance in data management infrastructures need to exhibit the following properties:

- Tamper-evidence and accountability: Every log entry must be verifiably authentic. This is typically achieved using cryptographic hashes and chaining mechanisms that detect any unauthorized alterations.
- Data confidentiality and secure storage: Logs may contain sensitive information and must be protected against unauthorized access, typically through robust encryption schemes.

- Auditability and exportability: Since logs are key to demonstrating compliance, the system must support reliable export mechanisms that allow authorized reviewers to inspect complete and verified logs without interrupting normal operations.

In summary, GDPR mandates not only the protection of personal data but also the ability to demonstrate such protection through audit trails and operational transparency. Logging systems play a critical role in fulfilling these legal requirements, as they provide evidence of data handling activities and help ensure accountability.

2.2 Fundamentals of Logging Systems

2.2.1 Purpose and Role of Logging

A Logging system is a commonly used component in modern software architectures, supporting a wide range of operational and analytical needs. It records events or state changes that occur during runtime, typically in the form of structured or semi-structured messages called log entries. These logs serve as a historical record of system activity and enable various use cases, including:

- Debugging and development: Developers use logs to trace and diagnose issues in code execution.
- Monitoring of operations: Logs offer insights into system behavior and performance, often feeding into observability and alerting platforms.
- Security and forensics: In security-sensitive environments, logs are essential for detecting anomalies, identifying unauthorized access, or investigating potential breaches.
- Legal and compliance auditing: Logs serve as a verifiable audit trail of data processing activities, which is particularly important in regulated environments such as those governed by the GDPR.

The context of this thesis focuses specifically on audit logging in the context of compliance and accountability. In this setting, the trustworthiness and traceability of logs are critical properties that enable organizations to demonstrate responsible data handling and meet legal obligations.

2.2.2 Core Components and Concepts

A typical logging system contains several core elements. At its heart are **Log Entries**, which are individual records of an event or action. Each entry typically includes a timestamp, a message, and metadata such as the log level, the identity of the actor, and the event type. For compliance logging, additional fields might be included, like operation details and identifiers for affected data. These log entries are generally stored in **Log Files or Streams**. These are designed as append-only files to ensure a consistent and chronological order, which supports high write throughput and reduces the risk of data inconsistency or corruption during concurrent writes or system crashes by avoiding in-place updates. Finally, **Serialization Formats** are used to encode the log entries. These can be text-based formats like JSON or CSV, binary formats such as Protobuf [13], or even custom formats optimized for size and parsing efficiency [31].

2.2.3 Design Considerations and Trade-offs

Designing a robust logging system requires carefully balancing several competing factors. **Performance** is a key consideration; high-frequency logging can become a bottleneck if logs are written synchronously to disk. To mitigate this, many systems employ buffering and asynchronous writes, which trade immediate persistence for improved throughput [33, 12].

Another crucial factor is **storage format**. Text-based logs offer the advantage of being human-readable but are often inefficient for storage. Conversely, binary formats reduce size and parsing costs, though they necessitate specific tooling for inspection [31]. In compliance contexts, both space efficiency and long-term readability are important.

Scalability is also vital, especially for systems with high write loads. Such systems benefit from multi-threaded or even distributed logging architectures. In these setups, concurrency control mechanisms, such as thread-safe access to shared log structures, are essential to maintain log integrity and prevent race conditions.

Finally, **crash consistency** is crucial to ensure logs are not lost or corrupted in the event of a system crash. This typically involves mechanisms like write-ahead logging or controlled `fsync` policies. These policies govern when buffered data is flushed to disk—for example, after every write or at fixed intervals—to strike the right balance between performance and durability.

2.2.4 Specialized Logging for Audit and Compliance

Compliance-oriented logging, often referred to as audit logging, brings in a set of stricter requirements that go beyond traditional logging uses [21, 22].

First, there is the need for **Append-Only Guarantees**. This means audit logs must be written sequentially, and it should not be possible to modify them later. This prevents tampering and ensures events are chronologically consistent.

Second, **Traceability and Provenance** are critical. Each log entry needs to be clearly attributable to a specific entity, system component, or process. Accurate timestamps and contextual metadata are absolutely essential for this.

Third, **Verifiability** is a key requirement. To ensure the integrity and authenticity of the entire log history, systems often need to implement mechanisms like hash chaining or digital signatures [11, 36].

Finally, **Archiving and Exportability** are also crucial. Logs should be stored in a long-term, portable format and must be exportable without interrupting ongoing write operations. This supports both operational continuity and the ability to conduct retrospective audits.

2.3 Security and Cryptographic Concepts

Under GDPR, security plays a central role in the design of a logging system that handles sensitive or personal data–particularly in privacy-critical contexts, where logs not only record sensitive operations but may also contain personal data themselves [8]. As such, it is essential to consider potential threats and apply appropriate cryptographic mechanisms to ensure that logs remain confidential, authentic, and tamper-evident throughout their lifecycle.

2.3.1 Threat Model for Logging Systems

Relevant threats can be both external (e.g., unauthorized actors gaining access to log files) and internal (e.g., faulty components within the system). The primary goals of a secure logging system are to ensure the confidentiality, integrity, authenticity, and availability of log data. Typical threats include:

- Log Tampering: Modification or deletion of log entries to hide unauthorized activity.
- Log Injection: Insertion of forged log entries to fabricate events or mislead auditors.
- Unauthorized Access: Reading log data without proper authorization, potentially exposing sensitive personal data.
- Replay Attacks: Re-injection of previously recorded log entries to falsify system behavior.

- Log Truncation or Removal: Deletion of entire log segments to erase evidence.

This threat landscape demands the use of appropriate security measures. While it is difficult to defend against all possible threats—particularly in cases where the entire system is compromised—applying cryptographic techniques can make certain types of attacks significantly more difficult and help detect if logs have been tampered with.

2.3.2 Cryptographic Primitives for Secure Logging

To address the threats outlined above, cryptographic primitives are the fundamental building blocks [24, 20]. These techniques, based on well-established mathematical principles, form the foundation for secure systems and are often employed to safeguard information from malicious activities, ensuring its reliability and privacy.

One essential primitive is symmetric encryption. This method utilizes a single secret key for both encrypting and decrypting data. Its primary role is to ensure confidentiality, meaning that even if unauthorized parties gain access to protected information, its contents remain obscure and meaningless without the correct key. This directly counters threats such as unauthorized access by preventing the exposure of sensitive data to unintended viewers.

Complementing encryption are Message Authentication Codes (MACs). A MAC is a compact cryptographic tag generated from a message's content using a shared secret key. When data is transmitted or stored alongside a valid MAC, a receiver can later verify that the data has not been altered since the MAC was generated. This capability is crucial for ensuring integrity, meaning the data has not been modified or corrupted from its original state. Furthermore, the MAC confirms that the data genuinely originated from a source possessing the correct key, thus providing authenticity by verifying its true origin. Even a minor modification to the data or an attempt to forge its source will cause the MAC verification to fail, making MACs crucial for detecting and preventing data tampering and data injection.

Additionally, cryptographic hash functions play a vital role in ensuring data integrity. A hash function processes any input data to produce a fixed-size, unique digest. Crucially, even tiny changes in the input data will result in a entirely different hash output. When applied in a chaining pattern, where each piece of data includes the hash of the preceding one, hash functions create a verifiable sequence. Such a chaining mechanism is foundational for building tamper-evident systems, allowing to detect unauthorized changes, such as deletion of entire data segments (truncation or removal), insertion of new entries, or reordering of records.

These cryptographic techniques are complementary. Modern authenticated encryption schemes, such as Advanced Encryption Standard - Galois/Counter Mode (AES-GCM), go beyond basic confidentiality, inherently offering mechanisms to verify both the integrity and authenticity of the encrypted data within a single process. While encryption and MACs protect the confidentiality, integrity and authenticity of individual entries, hash chaining mechanisms add structural integrity across the log as a whole, ensuring that the sequence and completeness of records can also be verified.

2.3.3 Security and Performance Trade-offs

Despite their strengths, cryptographic techniques come with inherent performance costs. In high-throughput systems, applying encryption and authentication to individual data elements can quickly become a computational bottleneck. As a result, secure systems often need to strike a careful balance between security guarantees and system performance.

One common strategy to mitigate this overhead is batch processing. Instead of encrypting each item individually, elements can be collected in memory and processed in groups. This reduces the repeated cost of cryptographic operations and allows more efficient use of parallel processing capabilities, such as GPUs and hardware-accelerated encryption [23]. However, batching introduces a window of vulnerability: if the system crashes before a batch is flushed to persistent storage, that data may be lost. This trade-off must be managed carefully, depending on the threat model and operational constraints.

Storage efficiency is another important consideration. Encrypted data is often less compressible than plaintext, and additional metadata (e.g., nonces, tags, or hashes) can increase the overall size of record batches. Systems that must store large volumes of data over long periods may benefit from applying compression before encryption or using compact binary serialization formats to minimize overhead.

Finally, cryptographic operations must be implemented and managed with great care to avoid introducing new vulnerabilities. Key management is especially critical: if encryption keys are poorly stored or exposed, the confidentiality and authenticity guarantees of the system collapse entirely. While this thesis focuses on system-level design rather than key management practices, it is important to recognize the central role secure key handling plays in any cryptographically-secure infrastructure.

2.4 Key-Value Stores and GDPR Compliance

Key-value stores (KVSs) play a central role in many modern data architectures. They are a form of database system that manages data as collections of key–value pairs, where each key is a unique identifier mapped to a value, typically stored in a raw or serialized format without internal structure. Unlike relational databases, KVSs do not impose a fixed schema or support complex relationships between records, which gives them great flexibility and makes them easy to scale horizontally.

Designed with speed and efficiency in mind, most KVSs rely on lightweight data structures—such as in-memory hash tables or log-structured merge trees—to achieve high throughput and low latency [16]. As a result, they are widely used in performance-critical workloads like caching layers, session management, and real-time analytics pipelines. In these contexts, applications expect extremely fast access to mutable data.

But this performance-oriented design creates complications when personal data is involved, especially under privacy regulations like the GDPR. Traditional KVSs typically operate without maintaining historical context, internal logs exist purely to support crash recovery, not to preserve an audit trail of user data activity.

Two key challenges emerge in this context:

1. **Non-transparent Operations:** By design, KVSs expose only the current value associated with a given key. PUT and DELETE operations replace or remove data without preserving earlier versions, and GET operations typically leave no trace at all. This lack of built-in versioning or provenance makes it difficult to demonstrate how a piece of personal data has been handled over time—a serious issue for systems subject to audit or data subject access requests.
2. **Strict Performance Requirements**: The low-latency characteristics of KVSs are often essential to the applications that depend on them. Introducing compliance-related overhead such as logging as a side effect of every key operation must therefore be done without impacting the store's performance guarantees.

To address this lack of built-in audit trails in KVSs, organizations must introduce an external layer—often a proxy or middleware—that intercepts every single client call. In this proxy, two functions occur in parallel: policy enforcement (e.g., making sure the requested operation complies with GDPR rules) and audit logging (recording the operation in a tamper-evident store). By decoupling logging from the KVS itself and embedding it in the proxy layer, organizations can achieve compliance without sacrificing performance—while keeping both application code and the underlying KVS engine unchanged.

3 Overview

3.1 System Overview

This chapter presents the architectural design and operational flow of the logging system developed in this thesis. Built specifically for low-latency, high-throughput environments where GDPR compliance is imperative, the system acts as a tamper-evident, cryptographically secure log recorder for operations on personal data. Rather than extending existing database logging infrastructure, this system offers a modular, standalone solution which is designed to function independently of the underlying storage layer, integrating seamlessly into middleware such as GDPR-compliant proxies for key-value stores.

It receives structured log entries representing operations on personal data from external components and ensures they are securely encoded and appended to an immutable audit trail. The act of logging is separated from immediate persistence through the use of buffering and asynchronous batch processing, allowing the system to achieve high throughput and low client-side latency while maintaining data integrity and confidentiality. However, this approach introduces latency in log persistence and subsequently increases the risk of data loss in crash scenarios—a trade-off that must be taken into consideration. If immediate persistence is required, the system would need to be tuned by users to reduce or eliminate buffering latency at the cost of performance.

Figure 3.1 illustrates the overall architecture of the proposed logging system, highlighting its key components and their interactions on a high-level. The architecture prioritizes real-world constraints observed in regulated environments: heavy write loads, strict tamper-evidence, and the eventual need for verifiable log export. Each design choice—from lock-free queues to append-only storage segments—is aimed at enabling efficient, secure, and scalable operation in compliance-focused systems.

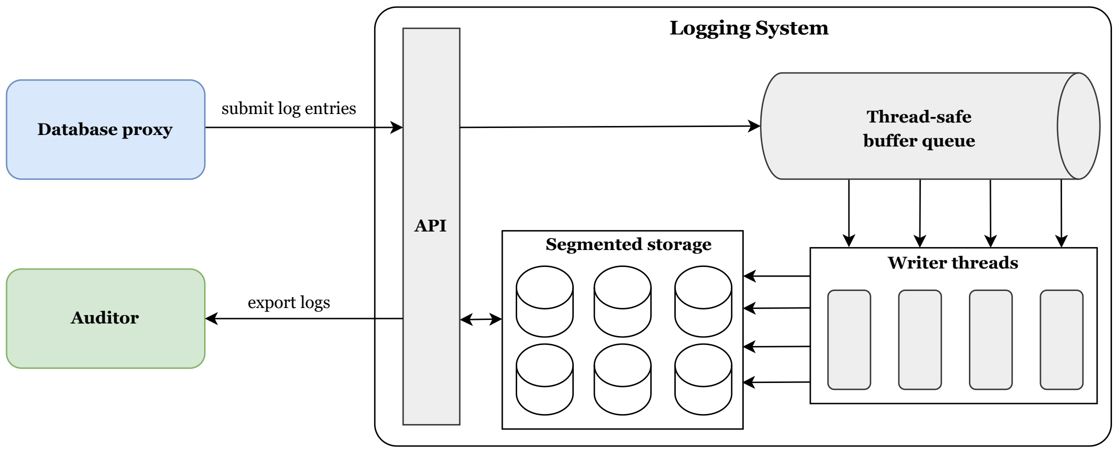

Figure 3.1: High-level system diagram

The sections that follow detail the goals driving the design, the specific components that realize those goals, and the exact flow of data from incoming log entry to securely persisted record.

3.2 Design Goals and Principles

The logging system was developed with a set of clearly defined goals, each motivated by the technical and regulatory demands of GDPR-compliant infrastructure. These goals shaped every aspect of the system design, from API design to the layout of persistent storage. While it has to be noted that not all features are fully realized within the scope of this thesis (e.g., export functionality, complete tamper-proofing), the system is structured to integrate them with little architectural change to achieve production readiness, with the necessary steps for their implementation also documented in this thesis.

Decoupling of Concerns

The logging system is fundamentally designed to operate as a standalone component, distinct from the underlying data storage mechanisms, such as key-value stores, and the client applications that generate log entries. This architectural separation is critical, as it allows the logging system to integrate seamlessly into existing infrastructures, acting as a transparent proxy for databases without requiring intrusive changes to their core functionalities.

High Write Throughput

To accommodate the volume of operations generated by high-performance data management engines, the system prioritizes write throughput above all else. This is achieved through a lock-free, multi-threaded architecture where log entries are collected asynchronously via a low-latency API and processed in batches. By deferring computationally intensive operations such as compression and encryption to dedicated worker threads, the system ensures that logging overhead does not become a bottleneck in the critical path of data operations.

Low Latency

Minimizing the time it takes to enqueue a log entry is essential to preserving application responsiveness. The system ensures low latency by decoupling log submission from log persistence: client threads simply enqueue entries into a concurrent buffer and return immediately, without waiting for disk or cryptographic processing. This design allows the logging system to be used in latency-sensitive paths without introducing noticeable delay.

Concurrent Scalability

To handle high logging throughput demands, the system supports multiple producer threads enqueuing log entries concurrently, alongside multiple consumer (writer) threads that process these entries in parallel. The internal buffer leverages a lock-free, multi-producer/multi-consumer queue, which reduces the need for traditional locking mechanisms and helps minimize thread contention under load. When writing to disk, threads can perform parallel appends to the same file by atomically reserving byte ranges using a shared offset. This approach avoids write serialization and allows safe concurrent writes without coordination via locks, preserving high throughput even under heavy concurrency.

Encryption

To comply with GDPR requirements for data confidentiality, all log entries are encrypted before being written to disk. The system uses `AES` in `Galois/Counter Mode` (AES-GCM), a widely adopted authenticated encryption scheme that provides both confidentiality and integrity in a single operation. Encryption is applied at the batch level, meaning each group of serialized log entries is compressed and then encrypted as a single unit.

This batching strategy improves performance by reducing cryptographic overhead and enables efficient use of system resources during high-throughput workloads. AES-GCM additionally produces an authentication tag for each batch, which is later used to verify that the encrypted data has not been altered.

Tamper Evidence

Integrity and authenticity are fundamental to auditability. As already mentioned, the system leverages the authentication tags generated by AES-GCM during batch encryption to detect unauthorized modifications. Rather than attaching a separate cryptographic hash to each individual log entry, the integrity of an entire batch is protected as a whole. Any tampering with the encrypted data—whether a single byte or the entire batch—will cause the authentication check to fail during decryption, ensuring that corrupted or maliciously altered logs cannot be read without detection.

While this mechanism provides tamper evidence at the batch level, it does not yet guarantee integrity across an entire log segment. A production-ready extension would, for example, incorporate a strictly increasing counter included in each batch and used in the generation of the AES-GCM authentication tag—forming a verifiable chain across batches. The system has been designed with this extensibility in mind. Additionally, any real-world deployment would require a mechanism to detect the removal of entire log files, which is beyond the scope of this prototype but could for instance be achieved with the use of Confidential computing techniques that can preserve a secure and tamper-resistant list of expected log files.

Compression

Given the potentially large volume of log data, the system employs compression to reduce the size of data written to disk. By compressing log batches before encryption, storage efficiency is significantly improved without sacrificing security. This also reduces I/O pressure, making it feasible to retain long audit histories even in storage-constrained environments.

Immutable, Append-Only Storage

Especially important for GDPR compliance is the strict adherence to an append-only storage model. This means that once a log entry is written, it cannot be modified or deleted in place. This design choice is fundamental for ensuring tamper-resistance and maintaining chronological consistency of events, which are vital for auditability under

GDPR's accountability principle. The append-only nature facilitates verifiable audit trails, enabling post-incident investigations and providing reliable evidence for Data Protection Authorities (DPAs) during audits. This approach inherently supports the principle of transparency, as processing activities on personal data can be audited and reviewed through an unalterable record.

Exportability

Although not implemented within the scope of this thesis, the system is designed to support secure and efficient log export. Since all writes are append-only and files are rotated upon reaching a size threshold, closed segments can be safely exported without interfering with ongoing operations. Export functionality would involve decrypting, decompressing, and verifying the integrity of log segments to produce a complete, verifiable audit trail—essential for responding to data access requests or regulatory audits.

3.3 Threat Mitigation

As outlined in the background chapter in Section 2.3.1, our logging system faces several key threats, such as log tampering, log injection, unauthorized access, replay attacks, and log truncation or removal. The architectural design and cryptographic mechanisms implemented in this system directly address these security concerns through multiple complementary approaches. We adopt a threat model in which external log-producing components are assumed to be trusted, whereas the persistent storage layer is considered untrusted. We assume that all incoming append requests to the logging system are valid, originating from legitimate sources, and should therefore be processed and written to persistent storage. Should these external components generating the log entries be compromised, our system is not capable of detecting or mitigating such scenarios. Instead, we assume that attackers may gain access to the physical storage layer where log files are persisted, and our security safeguards are specifically designed to protect against and detect attacks at this level.

Unauthorized access to log data is prevented through the system's use of AES-GCM authenticated encryption. Each batch of log entries is encrypted using a secret key, ensuring that even if attackers gain access to the stored files, the sensitive data remains confidential without access to the correct decryption key. Log tampering is partially mitigated through the authentication tags generated by AES-GCM for each batch of entries. Any modification to the encrypted data within a batch will cause the authentication check to fail during decryption, making such tampering immediately detectable. Additionally, under the assumption that the secret key used for authenticated encryption remains secure and is not compromised, the system reliably protects against log injection attacks that attempt to insert forged batches of entries, as attackers lacking the correct secret key are unable to produce a valid authentication tag, causing the verification process to fail.

However, the current implementation has notable limitations in its threat mitigation capabilities. The system remains vulnerable to log truncation or removal attacks, as there is no chaining mechanism between batches in place that would detect the deletion of entire batch segments. Similarly, replay attacks cannot be detected, as an attacker could simply rewrite previously processed valid batches to disk without the need to preserve a continuous verification chain. To address these crucial threats, the system architecture leaves room for future improvements like cryptographic chaining between batches, but such integrity safeguards aren't yet part of the current prototype (see Chapter 8 for further discussion).

3.4 System Components

In this section, we examine the individual components that comprise the logging system, detailing their roles, interactions, and key implementation choices.

3.4.1 Logging Manager

The `Logging Manager` serves as the top-level coordinator. On initialization, it reads configuration parameters—such as batch size, file-rotation thresholds, and worker counts—and instantiates the buffer queue, writer workers, and file managers accordingly. During shutdown, it signals all writer workers to drain remaining entries, ensures that in-flight batches are flushed to disk, and cleanly closes any open segment files. By centralizing lifecycle management in this module, the system guarantees that resources are allocated and released in a controlled and predictable fashion.

3.4.2 Logger

The `Logger` presents a minimal, low-latency interface for clients to submit audit data. Upon invocation, it immediately enqueues the result into the lock-free buffer queue—returning control to the caller without waiting for any disk or cryptographic operations. Additionally, the API defines an export endpoint to support future retrieval and audit workflows, though this functionality is not implemented in the current prototype.

3.4.3 Concurrent Buffer Queue

At the heart of the ingestion pipeline is a multi-producer/multi-consumer, lock-free queue. This structure enables many client threads to submit entries simultaneously, while writer workers pull batches without blocking producers. The queue's internal ring buffers and atomic pointers help ensure that push and pop operations incur minimal cache contention by providing each producer thread with its own internal queue structure, reducing shared memory access and coordination overhead between threads and making it well-suited for high-throughput logging scenarios.

3.4.4 Writer Workers

A configurable pool of writer workers continuously processes the queue. Each worker follows the workflow outlined in section 3.5: dequeue up to N entries, group them by base filename, serialize and compress each group, then perform authenticated encryption. Upon completion, the worker atomically reserves file offsets and appends the encrypted data to the corresponding segment. Writer workers also monitor segment sizes and trigger rotation when thresholds are reached. By batching work and isolating I/O and CPU-intensive tasks to these dedicated workers, the system shields client code from performance variability.

3.4.5 Segmented Storage

Logs are persisted in append-only segment files organized by base filename and rotation timestamp. Each active segment maintains an atomic counter representing the current write offset; workers increment this counter to reserve their write region and then write directly at that position. When a segment exceeds its configured maximum size, it is atomically closed (no further writes allowed) and a new segment file is opened. This pattern enables safe, concurrent writes to the same logical log stream without locks, and ensures that closed segments remain immutable—facilitating future export and verification.

3.5 System Workflow

The system workflow can be divided into eight key stages, illustrating how log entries travel from the moment an application submits them until they are durably stored on disk. This flow maximizes throughput by decoupling producer work from I/O and cryptographic tasks, while preserving the append-only, tamper-evident guarantees. Figure 3.2 revisits the high-level system architecture shown earlier, now annotated with numbers to illustrate the key stages of the log processing workflow described in this section.

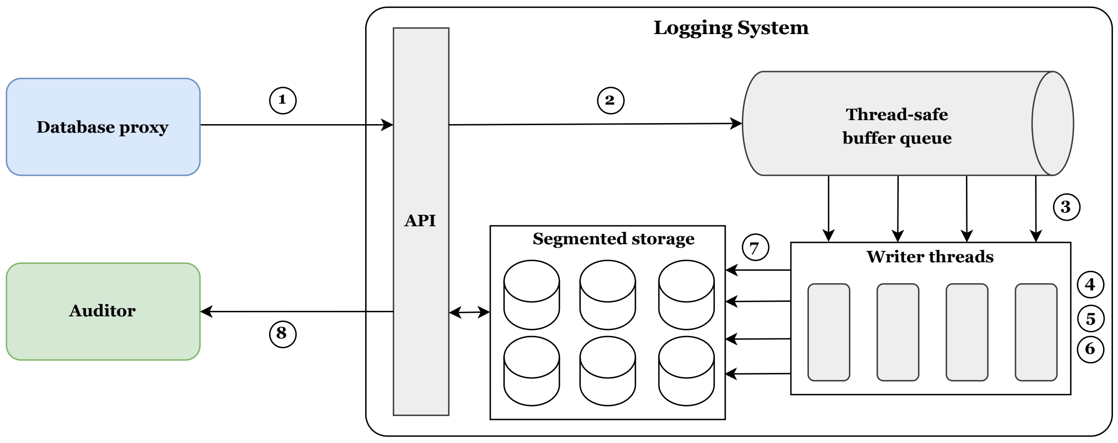

Figure 3.2: High-level system workflow diagram

1. **Entry Submission**: Client code invokes the `append()` API–either for a single entry or a batch–and immediately returns control to the caller. Internally, each entry is timestamped, tagged with metadata (e.g., operation type, key identifier, client ID), and wrapped into an in-memory record object.
2. **Enqueueing**: Records are placed into a multi-producer/multi-consumer, lock-free queue. This concurrent buffer allows many threads to submit entries in parallel with minimal coordination overhead.
3. **Batch Dequeueing**: Dedicated writer threads periodically poll the queue. Each thread drains up to a configurable batch size, collecting a group of records to process together.
4. **Grouping by Destination**: Within a batch, entries are partitioned by their destination filename. This ensures that each writer thread can efficiently assemble contiguous data destined for the same segment file.
5. **Serialization & Compression**: For each destination group, the thread serializes

records into a compact binary blob, then applies streaming lossless compression. By compressing before encryption, the system preserves confidentiality without sacrificing compression ratios, while ensuring logs can be entirely restored for export and audit processes.

6. **Authenticated Encryption**: The compressed blob is encrypted with AES-GCM using a per-batch nonce and a secret key. AES-GCM produces both ciphertext and an authentication tag, binding integrity to confidentiality.
7. **Append & Rotate**: To write to disk, the thread atomically reserves a byte range in the target file by incrementing a shared offset. If this new offset would exceed a configured file-size limit after the append, the current segment is marked "closed" and a new file is opened—named with the same base filename plus a timestamp suffix. The thread then writes the ciphertext and tag at the reserved position.
8. **Export and Verification (Future Work)**: In later iterations, closed segments can be exported for auditing. During export, the system will decrypt, decompress, and verify each batch's authentication tag to ensure end-to-end data integrity without disrupting active writes.

4 Design

In this chapter, we describe the detailed design of our high-performance, secure, and tamper-evident logging system. Building on the high-level overview presented in Chapter 3, we focus here on each core component's internal responsibilities, data structures, and interactions. We assume the reader is already familiar with core concepts mentioned in the previous chapters. Our goal is to explain how each component fits into the overall pipeline, illustrate its internal workflow via pseudocode, and clarify how data moves from the moment a client calls *append(...)* through to a finalized, encrypted, compressed batch on disk.

4.1 Detailed Component Design

4.1.1 Log Entry Structure

A `LogEntry` represents the fundamental unit of information in the logging system, encapsulating all relevant metadata about operations performed on personal data. Each `LogEntry` corresponds to a single data interaction and is designed to ensure traceability and auditability in compliance with the GDPR. Conceptually, a `LogEntry` captures the **type of operation** performed (e.g., CREATE, READ, UPDATE, DELETE), the **location of the data** affected (such as a key in a key-value store), the identities of the **data controller**, **data processor**, and **data subject** involved, a **timestamp** indicating when the operation occurred, and finally a **custom payload** used to store additional context or operation-specific data—such as the **purpose** of the data processing. This structured approach ensures that each log entry maintains a complete and consistent audit trail. The design is extensible to support future changes in data compliance requirements or logging semantics. To support efficient storage and transmission within the logging pipeline, the `LogEntry` is designed to be serializable, including in batches, which enables performant handling by downstream components such as writer threads.

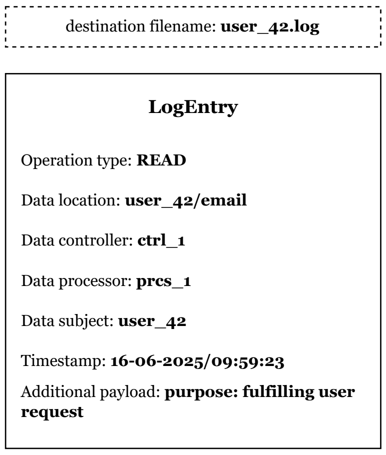
destination filename: **user_42.log**
LogEntry
Operation type: **READ**
Data location: **user_42/email**
Data controller: **ctrl_1**
Data processor: **prcs_1**
Data subject: **user_42**
Timestamp: **16-06-2025/09:59:23**
Additional payload: **purpose: fulfilling user request**

Figure 4.1: Simplified structure of a log entry

As an example, consider a READ operation on the key `user_42/email` intercepted by a GDPR-compliant key-value store proxy. This operation is performed by a data processor `prcs_1` on behalf of data controller `ctrl_1` and concerns the data subject `user_42`. The purpose of the access—such as fulfilling a user request—is recorded in the payload field, and the resulting `LogEntry` is assigned a designated destination file name `user_42.log` to maintain a dedicated audit trail for that individual. If, within a very short time frame, another operation on data belonging to the same data subject occurs, the proxy captures it in a second `LogEntry` and groups the two entries into a batch. These entries, being related to the same data subject and contextually linked in time, can then be submitted together as a batch to the logging system. Otherwise, captured log entries remain independent records.

4.1.2 Logging Manager

The `Logging Manager` is the central orchestrator of the logging system. It is responsible for **initializing** all internal subsystems, managing **configuration**, and controlling the overall **runtime behavior**. Its design ensures that the system starts in a well-defined state, operates reliably under load, and shuts down gracefully without data loss.

During initialization, the `Logging Manager` receives a structured configuration object that defines the overall behavior of the logging pipeline. This configuration includes parameters such as timeouts for enqueue operations, internal buffer sizes, and the number of concurrent writer threads. It also allows tuning of storage-related aspects, such as the maximum size of individual log files, the location where logs are written, and limits on the number of files that can be open at the same time. Additionally, optional features like encryption and compression can be enabled or disabled through the configuration, with adjustable compression levels to balance performance and storage efficiency. These settings are intended to give users fine-grained control over system behavior making it flexible to use in various scenarios with differing requirements.

Once configured, the `Logging Manager` sets up the core components of the system, including the buffer queue and the storage backend. It ensures that required directories exist on the filesystem and that all components are initialized consistently. The system is designed to operate safely in a multithreaded environment, with internal safeguards to prevent race conditions or inconsistent states. Specifically, it uses internal flags to track whether the system is currently running and whether it is still accepting new log entries, which helps coordinate startup and shutdown phases reliably.

In relation to these phases, the Logging Manager provides explicit methods to control its runtime lifecycle: A start method that activates the writer threads according to the configured parameters and transitions the system into an active state, ready to receive and process log entries; and a stop method that initiates a controlled shutdown by preventing further entries from being accepted, waiting for the buffer queue to be drained, allowing writer threads to complete any remaining processing, and flushing all buffered data to disk–all in a clean and predictable manner.

Additionally, the `Logging Manager` exposes the logging system's API endpoints to producers; however, these calls are simply delegated to the underlying `Logger` component. For this reason, further details will be discussed in the following subsection dedicated to that component.

4.1.3 Logger

The `Logger` serves as the primary interface through which external producers interact with the internal logging system. Although technically accessed via the `Logging Manager`, the responsibility for handling **log submission** and **producer registration** is encapsulated entirely within this component. Its main objective is to provide a lightweight and efficient mechanism for submitting log entries to the buffer queue, while shielding producers from the complexities of the internal system behavior.

Before use, the `Logger` must be initialized with the necessary components and parameters—a reference to the buffer queue and a configured timeout value that governs how long a producer may wait when attempting to enqueue data. This initialization process is managed centrally by the `Logging Manager`, which ensures that all dependencies are properly configured before any producer interaction takes place.

The Logger exposes a minimal set of endpoints that allow producers to register, submit individual or batched log entries, and, in future versions, export stored logs. Before submitting entries, each producer must register with the system to obtain a unique producer token. This token must be maintained throughout the producer's lifecycle and passed along with subsequent append requests. Its primary purpose is to optimize the performance of the internal concurrent buffer queue under load, a topic discussed in more detail in the next subsection. Submission methods are designed to be thread-safe and to respect system constraints such as queue capacity and timeout thresholds. When attempting to enqueue log entries into the system's internal buffer, each entry is submitted together with its associated destination filename. Under high workloads, append calls will block until sufficient space becomes available in the queue, or fail if this does not occur within the configured timeout. Batched submissions are supported to improve throughput and reduce contention in high-concurrency scenarios.

Internally, the Logger enforces strict checks to ensure it is properly initialized before any operation is performed. These safeguards help prevent undefined behavior and make the component robust against incorrect usage. Overall, the design maintains a clear separation between log submission and log persistence, allowing producers to be released immediately after their entries are enqueued. This ensures that the system remains responsive even when downstream processing–including serialization, compression, and encryption of logs–or storage operations are temporarily constrained.

Although the current version of the API includes a placeholder for log export functionality, this feature is not yet supported. It is intended to evolve into a more comprehensive interface that supports filtered retrieval of persisted data for auditing or analysis. When implemented, it will delegate tasks such as decompression and decryption to dedicated components.

The following pseudocode-based algorithms describe the producer submission workflow within the logging system, from registration through log entry submission, assuming the system has already been properly initialized and started:
Algorithm: Producer Log Submission

```
producerToken = createProducerToken()

appendLogEntries(vector<LogEntry>, destinationFilename, producerToken):
    queueItems = []
    for entry in vector<LogEntry>:
        queueItems.append(QueueItem(entry, destinationFilename))
    enqueueBatchBlocking(queueItems, producerToken, configuration.appendTimeout)
```

4.1.4 Concurrent Buffer Queue

The `Buffer Queue` is a high-performance, lock-free queue implementation that forms the core of the logging system's buffering layer. It acts as the communication bridge between the `Logger` and the writer threads, providing thread-safe concurrent enqueueing and dequeueing of log entries under high load. Figure 4.2 illustrates the internal components of the buffer queue, highlighting its architectural breakdown and interaction between producers and consumers.

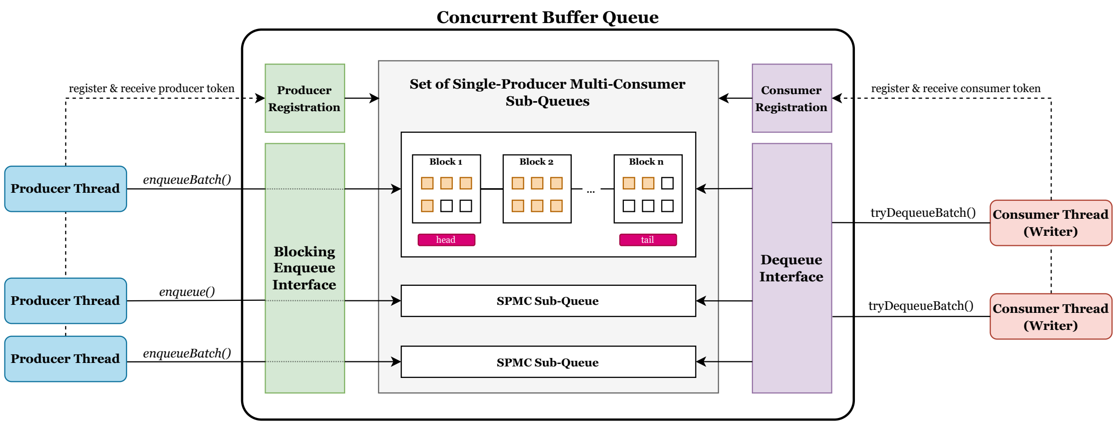

Figure 4.2: Buffer queue internal component breakdown

Internal Architecture and Data Structures

The queue's core architecture consists of multiple single-producer, multi-consumer sub-queues that work together to form a cohesive concurrent system. Rather than implementing a single shared queue that would require complex synchronization, the design partitions the data structure into dedicated sub-queues, each owned by a specific producer thread. This partitioning eliminates producer-to-producer contention entirely, as each producer operates exclusively within its own memory space.

Each sub-queue is implemented as a circular buffer with carefully designed memory layout to optimize cache performance. The sub-queues are allocated with appropriate padding and alignment to prevent false sharing between threads operating on adjacent memory regions. This architectural choice ensures that producers can write to their dedicated sub-queues without affecting the cache performance of other producers or consumers.

Token-Based Access Control

The system employs a sophisticated token mechanism that optimizes thread access patterns throughout the entire lifecycle of producer and consumer threads. Tokens are long-lived objects that threads acquire once during initial registration and retain for their entire operational lifetime, amortizing the cost of access path optimization across all subsequent operations.

For producer threads, tokens serve a direct routing function by maintaining a binding to the thread's designated sub-queue. When a producer thread performs enqueue operations, its token immediately directs the operation to the correct sub-queue without requiring any lookup or coordination overhead. This design ensures that producers achieve optimal performance by eliminating the need to search for available capacity across multiple sub-queues.

Consumer threads utilize tokens differently, as they must coordinate access across all available sub-queues to ensure fair consumption. The consumer token maintains state information that tracks the starting point for round-robin scanning across the collection of sub-queues. This mechanism enables multiple consumer threads to efficiently coordinate their access patterns, ensuring that no sub-queue is starved while preventing consumers from redundantly scanning the same sub-queues. The round-robin approach distributes the scanning workload evenly among consumers, optimizing overall system throughput.

Load Management and Batch Processing

When temporary capacity constraints occur, the system implements adaptive retry mechanisms with exponential backoff to balance persistence with resource conservation. The blocking enqueue operations provide bounded waiting guarantees, ensuring predictable behavior under stress while preventing indefinite stalls.

The queue supports batch operations natively, allowing both producers and consumers to operate on collections of log entries. This capability amortizes overhead across multiple items and proves particularly valuable for applications generating bursts of related log entries or optimizing the downstream compression, encryption and write operations

System Integration

The queue provides approximate sizing information for monitoring purposes without compromising operational performance. This design choice prioritizes efficiency over precision in non-critical scenarios. During system shutdown, the flush operation ensures complete queue drainage by coordinating with the lifecycle management to prevent new entries while consuming remaining data.

To illustrate the producer and consumer interaction with the `BufferQueue` we provide the following pseudo code algorithms:

Algorithm: Producer Enqueuing

```
producerToken = createProducerToken()

enqueueBatch(vector<QueueItem>, producerToken, timeout):
    subqueue = findDesignatedSubqueue(producerToken)

    if(sufficient space in subqueue):
        enqueueBatch(vector<QueueItem>)
        return true
    else if(timeout exceeded):
        return false
    else:
        retryExponentialBackoff()
```

Algorithm: Consumer Dequeuing

```
consumerToken = createConsumerToken()

dequeueBatch(maxItems, consumerToken):
    dequeued = []
    subqueue = findDesignatedSubqueue(consumerToken)

    while(true):
        if(subqueue contains items):
            newItems = subqueue.dequeue(maxItems - dequeued.size())
            dequeued.append(newItems)

            if(dequeued.size() == maxItems):
                return and process(dequeued)

        subqueue = next subqueue
        if(subqueue already checked):
            break;
```

```
if(dequeued.empty()):
    return
else:
    return and process(dequeued)
```

4.1.5 Writer Workers

Writer workers are the core execution units responsible for asynchronously consuming batches of buffered log entries and persisting them to disk in a secure and space-efficient manner. They operate independently and are orchestrated by the `Logging Manager`, allowing the logging architecture to offload intensive I/O and cryptographic operations from client threads. Each `Writer` instance is assigned to its own dedicated worker and manages the full lifecycle of data retrieval, processing, and dispatch. The number of active writers is configured via the system configuration passed to the `Logging Manager` parent component.

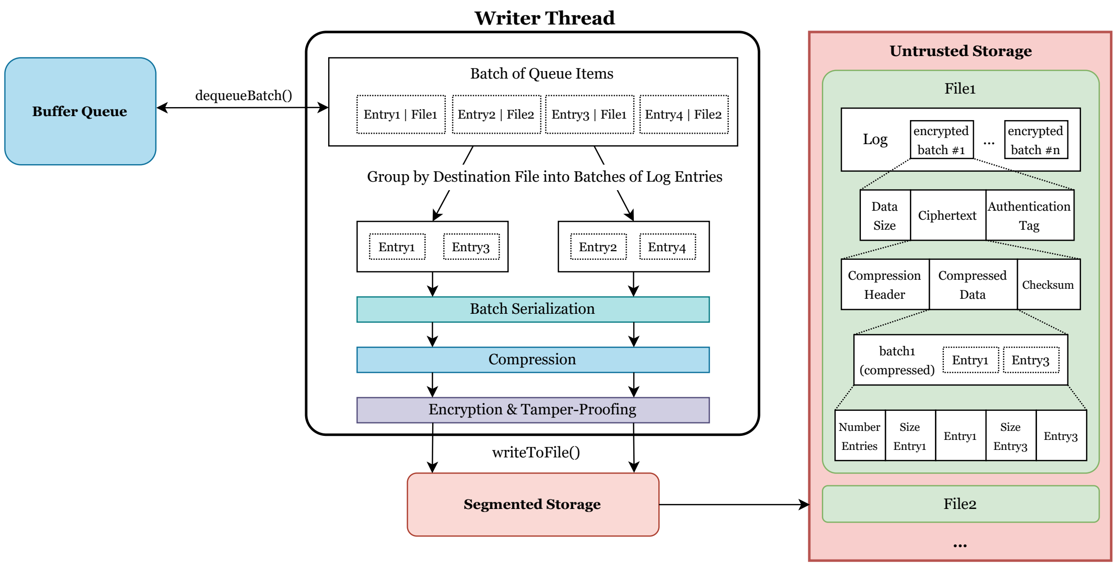

Figure 4.3: Internal component breakdown of the writers' pipeline in our logging system

Processing Pipeline Design

Writer workers implement a multi-stage processing pipeline that efficiently transforms raw log entries into optimized storage formats. The pipeline begins by retrieving batches of log entries from the buffer queue, taking advantage of bulk operations to reduce overhead. When the queue is empty, writers pause briefly to avoid wasting CPU resources while remaining responsive to new data. Once entries are retrieved, the writer groups them by their intended destination file. This grouping approach optimizes the processing pipeline by handling related entries together, which improves efficiency and supports flexible routing to different log files or storage targets.

Data Processing Stages

The transformation process consists of several key stages that prepare log entries for storage. First, the entries are serialized into a compact binary format that reduces storage space while maintaining all necessary information. Next, the compression stage provides significant storage savings by reducing the size of the serialized data. The system configuration allows to configure compression levels to balance between processing speed and storage efficiency, adapting to different deployment requirements and hardware capabilities. When security is required, an encryption stage protects the compressed data using industry-standard authenticated encryption algorithms. This ensures that sensitive log information remains confidential even if storage media is compromised. The final stage of the pipeline handles writing the processed data to persistent storage. Writers route data to specific files based on the corresponding log entry metadata, supporting flexible logging architectures that separate different types of information or implement custom retention policies.

Persistent Storage Format

The data written to persistent storage follows a structured format that reflects the multi-stage processing pipeline. Each stored unit represents an encrypted batch of log entries, beginning with the size of the encrypted data block, followed by the ciphertext and an authentication tag that ensures data integrity. Within the ciphertext, a compression header indicates the compression parameters used, followed by the compressed data and a checksum for additional data validation. The compressed data itself contains the serialized batch of log entries, structured with an initial count indicating the number of entries in the batch, followed by each individual entry represented as a size value paired with the actual entry data. This hierarchical format enables efficient storage and retrieval while supporting the security and compression features of the system.

Worker Management

Writer workers are designed with careful attention to lifecycle management, ensuring clean startup and shutdown behavior. The design prevents resource leaks and guarantees that all pending log entries are processed before the system terminates. The worker management system integrates with the overall logging architecture, allowing the number of active writers to be adjusted based on the expected system load. This flexibility enables the system to adapt to changing performance requirements while maintaining stability.

The overall writer workflow is presented by the pseudo code below.

Algorithm: Writer Worker Processing Loop

```
function processLogEntries():
    while running:
        batch = queue.dequeueBatch(batchSize)
        if batch.empty():
            sleep(shortInterval)
            continue

        grouped = groupByFilename(batch)
        for (filename, entries) in grouped:
            buffer = serializeBatch(entries)
            if compressionEnabled:
                buffer = compress(buffer, config.compressionLevel)
            if encryptionEnabled:
                buffer = encrypt(buffer)

            storage.writeToFile(filename, buffer)
```

4.1.6 Segmented Storage - File System Management

The `SegmentedStorage` component is responsible for persisting log data to disk in an efficient, append-only, and segment-structured format. It supports concurrent access to the same file by multiple writer threads, enforces file size limits through rotation, and manages file descriptor reuse through an integrated Last Recently Used (LRU) cache. This section explains the key data structures, synchronization mechanisms, and file rotation strategy that make up this component. Figure 4.4 provides a high-level overview of the segmented storage architecture employed by the logging system.

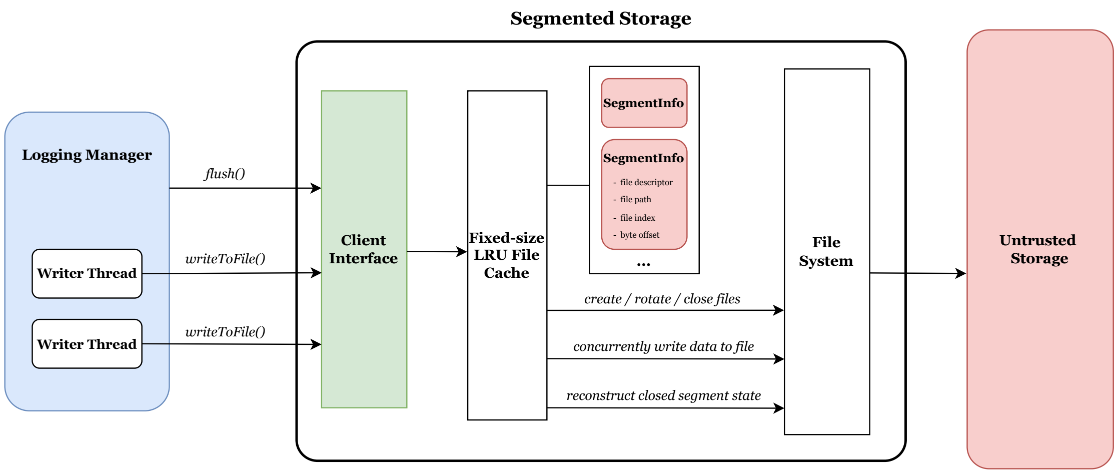

Figure 4.4: Overview of the segmented storage approach, adopted by our logging system

Design Philosophy and Requirements

The storage design addresses several competing requirements that shaped its architectural approach. The system must support high-throughput concurrent writes from multiple threads while maintaining strict append-only semantics that ensure data consistency. File size management prevents individual log files from growing without bounds, which would complicate backup, archival, and maintenance operations. Resource management represents another critical design consideration, as opening and closing file handles frequently would create significant performance overhead while keeping too many files open simultaneously could exhaust system resources. The design needed to balance these constraints while providing transparent access to multiple concurrent log files.

Segmentation Strategy

The core architectural principle is based on partitioning log data into discrete segments based on size thresholds rather than time boundaries. This approach provides predictable file sizes that simplify maintenance and processing while avoiding the coordination complexity that time-based segmentation would introduce in a multi-threaded environment. Each logical log stream maintains its own segmentation state independently, allowing different streams to rotate at different rates based on their individual activity levels. This independence ensures that high-volume log streams do not force premature rotation of low-volume streams, optimizing storage efficiency across diverse logging patterns. The segmentation design supports both default logging to a primary stream and targeted logging to specific named streams. This flexibility enables applications to separate different types of log data while maintaining unified management and consistent performance characteristics across all streams.

Concurrent Access Coordination

The storage system implements sophisticated coordination mechanisms that allow multiple writer threads to safely append data to the same logical stream without compromising data integrity or performance. The design employs a reservation-based approach where threads atomically reserve space in the current segment before performing their write operations. This reservation strategy prevents data corruption while minimizing coordination overhead between threads. When multiple threads target the same stream, they coordinate through lightweight atomic operations that scale effectively with the number of concurrent writers. The design ensures that write operations remain independent once space is reserved, allowing maximum parallelism during the actual I/O operations. Segment rotation introduces additional coordination challenges, as the system must ensure that ongoing writes complete successfully while transitioning to new storage segments. The design handles this transition through careful synchronization that prevents data loss while minimizing disruption to concurrent operations.

Resource Management and Caching

File descriptor management represents a critical performance optimization within the storage design. Opening files repeatedly would introduce significant overhead, while maintaining unlimited open file handles would eventually exhaust system resources. The solution employs an intelligent caching strategy that balances performance with resource constraints. The caching design implements a least-recently-used eviction policy that automatically manages the trade-off between keeping frequently accessed files readily available and respecting system resource limits. This approach ensures that active log streams benefit from optimized access patterns while inactive streams do not consume unnecessary resources. The cache integrates seamlessly with the broader storage operations, providing transparent file handle management that requires no coordination from higher-level components. When files must be evicted from the cache, the system ensures data durability through appropriate synchronization before releasing resources.

Rotation and Lifecycle Management

File rotation occurs dynamically based on size thresholds, ensuring predictable storage characteristics without requiring external coordination. The rotation process creates new segments with timestamp-based naming that provides natural ordering and supports operational procedures like archival and cleanup. The rotation design maintains consistency during the transition between segments, ensuring that no data is lost and that all writer threads can continue operating without interruption. New segments are created with appropriate permissions and initialized for immediate use, minimizing the time window during which rotation affects system performance. The naming strategy for rotated segments incorporates timestamp information that facilitates operational tasks while ensuring uniqueness across concurrent rotations. This approach supports both automated processing and manual operations that system administrators might need to perform.

Durability and Consistency Guarantees

The storage design prioritizes data durability through explicit synchronization operations that ensure written data reaches persistent storage. The system provides configurable durability guarantees that balance performance requirements with data protection needs, allowing operators to tune behavior based on their specific risk tolerance and performance requirements. Consistency guarantees ensure that concurrent operations do not interfere with each other and that partial writes cannot occur under normal operating conditions. The append-only nature of the storage simplifies consistency management while supporting the high-throughput requirements of the logging system. During system shutdown, the storage component ensures that all pending data is properly synchronized to persistent storage before resources are released. This shutdown coordination prevents data loss while maintaining reasonable termination times even under high load conditions. Through these design choices, the segmented storage component provides reliable, high-performance persistent storage that scales effectively with application demands while maintaining the operational characteristics required for production logging systems.

Write Granularity and Atomicity

The `SegmentedStorage` component accepts write operations of arbitrary size, from potentially single bytes up to the configured segment size limit. Unlike traditional block-based storage systems, writes are not required to align to sector or page boundaries. This flexibility simplifies the producer interface, allowing them to batch log entries without imposing artificial boundaries and avoiding the need for padding or buffering.

At the system call level, the implementation employs a retry mechanism with exponential backoff for transient I/O failures. The system's configured parameters govern this retry behavior in terms of maximum number of retries and the base timeout for retrying, making the system robust to temporary I/O glitches such as resource contention or brief filesystem unavailability. Additionally, the implementation handles partial write scenarios where the operating system may accept only a portion of the requested data. In such cases, the component continues writing the remaining bytes until all data is transferred to the system's buffer cache.

The design prioritizes simplicity and performance over perfect atomicity guarantees, accepting that repeated failures may result in incomplete writes. While complex transaction mechanisms could enforce stronger crash consistency, they come at a significant performance cost. Alternatively, the parameters of the logging system–such as the base retry delay and the maximum number of reattempts–can be tuned to minimize data loss in the event of a failure, although also at the expense of performance.

Crash Consistency

The append-only design inherently provides a strong consistency model by ensuring that existing data is never modified or overwritten. Persistence is enforced through explicit synchronization calls that flush all dirty buffers to storage at critical points in the component lifecycle: before closing any segment during rotation or eviction from the file descriptor cache. However, this flushing strategy is simplistic—individual writes are not immediately synchronized, meaning data may remain in system buffers for extended periods. This could lead to significant data loss in the event of system failures, indicating that more aggressive flushing policies would be necessary for production deployment.

The current design presents two distinct failure scenarios in the window between space reservation and write completion. First, if a system crash occurs after reserving write space but before the write begins, the segment will contain a hole–an unwritten region where data was intended to be written. Second, if a crash occurs during the write operation itself, only partial data will be persisted. In the first scenario, the data is simply missing. In the second, since the log data is encrypted, the partial write contains incomplete ciphertext that cannot be decrypted. Both scenarios result in data loss, with subsequent writes beginning after the reserved space, preserving the append-only property but leaving gaps or corrupted segments in the log stream.

While unwritten regions can be detected during recovery and partial writes identified through decryption failures, the current implementation does not include any recovery mechanisms. However, a straightforward recovery strategy could involve discarding the last (potentially corrupted) entry after a crash and resuming writing from its original start offset, effectively filling the hole. In future iterations, incorporating monotonically increasing counters into the log format would then allow such missing entries to be detected reliably.

Overall, the `SegmentedStorage` component ensures high-performance, append-only, and thread-safe log persistence. It abstracts file rotation, retry logic, and descriptor caching behind a unified interface. By using atomic operations and lock hierarchies wisely, it supports concurrent writers while providing data durability and segment management guarantees. The persistence pipeline can be illustrated with this simplified algorithm:

Algorithm: Segmented Write & Rotation Logic

```
function writeToFile(filename, data):
    segment = tryFindSegment(filename, LRU cache)
    if(segment not found):
        evictOldestSegment(LRU cache)
        segment = restoreOrCreateSegment(filename)
        addLatestSegment(segment, LRU cache)

    if segment.size() + data.size() > config.maxSegmentSize:
        rotateSegment(segment)

    segment.reserveSpace(data.size())
    segment.writeToReservedSpace(data)
```

5 Implementation

This chapter outlines the concrete low-level implementation details of the logging system, whose design was described in detail in the previous chapters. While the design chapter focused on conceptual architecture and high-level decisions, this section delves into the technical realization of each component, highlighting internal logic, concurrency mechanisms, and storage strategies.

The system is implemented in `C++`, selected for its performance characteristics and fine-grained control over memory and concurrency. Several external dependencies are integrated to support core functionality: `zlib` for compression, `OpenSSL` for authenticated encryption using AES-GCM, and standard threading primitives for parallel processing. For efficient multi-threaded buffering, the system makes use of the `moodycamel::ConcurrentQueue`—an open-source, lock-free, multi-producer/multi-consumer queue well-suited for high-throughput scenarios.

Each subsection corresponds to one of the core components previously introduced and details how the architectural ideas were translated into performant, low-level implementations that meet the required design goals for GDPR-compliant logging.

5.1 Logging Manager

Initialization and Configuration

The `LoggingManager` initialization process involves setting up all essential components based on parameters provided through a structured configuration (`LoggingConfig`). The following parameters are explicitly supported:

- `appendTimeout`: Defines the maximum duration producers wait to enqueue log entries before the append operation fails, safeguarding against indefinite waits when the internal buffer queue reaches capacity.
- `queueCapacity`: Sets the fixed size of the internal buffer queue. Internally, this capacity is adjusted to the nearest power of two for performance optimization.
- `maxExplicitProducers`: Specifies the maximum number of producers allowed to

enqueue log entries. This parameter optimizes the internal layout and performance of the buffer queue.

- `batchSize`: Determines the maximum number of log entries a writer thread can dequeue and process in one operation, balancing throughput and memory usage.
- `numWriterThreads`: Indicates the number of dedicated writer threads concurrently operating to dequeue and persist batches of log entries.
- `useEncryption` and `compressionLevel`: `useEncryption` is a boolean flag enabling or disabling encryption. `compressionLevel` is an integer parameter controlling compression behavior, where `0` disables compression, and values `1-9` specify explicit compression levels with higher values prioritizing compression ratio over speed. These parameters are primarily for evaluating the performance overhead and effectiveness of encryption and different compression strategies, as both are required to meet the design goals.
- `basePath`: Specifies the filesystem directory path where log segments are created and stored.
- `baseFilename`: Sets a default filename for log entries when no explicit filename is provided by producers.
- `maxSegmentSize`: Limits the maximum size (in bytes) of a single log segment file before it triggers rotation, ensuring manageability of log segments.
- `maxAttempts`: Defines the maximum number of retries for persistent storage operations (pwrite) in case of transient failures, aiming to eliminate data loss
- `baseRetryDelay`: Sets the initial delay for the exponential backoff mechanism applied during storage retries.
- `maxOpenFiles`: Limits the maximum number of simultaneously open log files, employing a Least Recently Used (LRU) mechanism to close files when this limit is exceeded.

Upon instantiation, the system ensures the log storage directory exists, creating it if necessary. The `BufferQueue`, `SegmentedStorage` and `Logger` components are then initialized using these configuration parameters.

Resource Management and Thread Safety

Thread safety within `LoggingManager` is carefully managed to ensure consistent and safe operations during concurrent access and state transitions. Atomic flags (`m_running` and `m_acceptingEntries`) explicitly control critical system states:

- `m_running`: Ensures threads accurately detect if the logging system is active, preventing unintended restarts or inconsistent behavior.
- `m_acceptingEntries`: Controls the acceptance of new log entries, enabling an orderly shutdown where no new entries can be submitted while pending ones complete processing.

Runtime Management

The `LoggingManager` provides explicit methods for controlling its runtime lifecycle:

- `start()`: Initiates writer threads with the specified parameters (such as `numWriterThreads`, `batchSize`, `useEncryption`, `compressionLevel`) and sets internal flags (`m_running` and `m_acceptingEntries`) to true, indicating readiness for accepting log entries. Writer threads then start dequeuing and processing log entries.
- `stop()` : Initiates a graceful shutdown by setting the internal flag `m_acceptingEntries` to false, indicating no further log entries will be accepted. It then waits for the buffer queue to emptied by calling its `flush()` method, afterwards stops all writer threads, and flushes remaining buffered log data to persistent storage. Finally, it resets the Logger to a clean state.

Logging API Exposure

In addition to managing the runtime lifecycle, the `LoggingManager` exposes several methods of the underlying `Logger` component directly to producers. Specifically, these methods include:

- `createProducerToken()`: Registers a new producer with the system, returning a token that optimizes internal resource usage.
- `append()`: Allows producers to enqueue a single log entry.
- `appendBatch()`: Allows producers to enqueue multiple log entries at once.
- `exportLogs()`: A placeholder method intended for future implementation to support log retrieval and export functionality.

The LoggingManager acts as a facade, delegating the implementation of these methods directly to the Logger component.

In summary, the `LoggingManager` manages the lifecycle of all important system components, and in addition it efficiently orchestrates all logging-related operations, ensuring robust configuration, high-performance concurrent execution, and reliable data handling.

5.2 Log Entry

Fields and Purpose

The `LogEntry` class includes the following key fields:

- `ActionType`: An enumeration representing the type of data operation performed (e.g., CREATE, READ, UPDATE, DELETE).
- `dataLocation`: A string indicating where the data was accessed or modified (e.g., the key in a key-value store).
- `dataControllerId`: The identifier of the data controller responsible for determining the purposes and means of the operation.
- `dataProcessordId`: The identifier of the data processor responsible for carrying out the operation on behalf of the data controller.
- `dataSubjectId`: The identifier of the data subject whose personal data was involved.
- `timestamp`: Automatically assigned at construction time; records the system time when the entry was created.
- `payload`: A byte array containing additional context or data associated with the operation.

Serialization and Deserialization

The `LogEntry` class implements serialization and deserialization to convert entries into compact binary representations and back.

- `serialize()`: Transforms the entry into a byte vector. The structure includes all fields in a deterministic, length-prefixed format to facilitate efficient parsing.
- `deserialize()`: Reconstructs an entry from a byte vector. It performs integrity checks such as length validation and safe offset calculations to ensure robust recovery from malformed data.

The class also includes static methods for handling batches:

- `serializeBatch(vector of LogEntries)`: Serializes a vector of log entries into a compact binary batch, each prefixed with its size and preceded by the number of entries.
- `deserializeBatch(vector of bytes)`: Converts a binary batch back into individual `LogEntry` instances.

These batch methods are essential for efficient writer thread processing, reducing the overhead of per-entry metadata and enabling bulk compression and encryption.

5.3 Logger

The `Logger's` role is to act as a gateway to the concurrent buffer (`BufferQueue`) and mediate the access to it based on producer tokens and timeout constraints, while minimizing latency for clients.

Lifecycle and Initialization

Before use, the `Logger` must be initialized by passing in:

- A shared pointer to a pre-configured BufferQueue instance.
- A timeout value (`appendTimeout`) used to bound how long a producer may wait to enqueue a log entry when the queue is full.

Initialization is guarded to ensure the singleton is not re-initialized. If improperly used (e.g., called without initialization), all API methods will report errors and return safely. The initialization is carried out by its parent component `LoggingManager`.

Exposed Endpoints

As already mentioned beforehand, the following endpoints are available to external components, exposed through the `LoggingManager`:

- `createProducerToken()`: Registers a new producer with the buffer queue and returns a dedicated `ProducerToken`. Internally, this method delegates the request to the underlying `BufferQueue`. The returned token uniquely identifies a producer and is required for all subsequent append operations, thus the producer needs to register once before beginning his append operations and then needs to keep track of the token throughout its lifetime.
- `append(LogEntry, ProducerToken, optional filename)`: Submits a single log entry to the logging system. The entry is first wrapped into a `QueueItem`, the internal data structure of items in the buffer queue, which combines the `LogEntry` with an optional target filename. This `QueueItem` is then passed to the `BufferQueue` via a blocking enqueue operation. The operation will block up to `appendTimeout` if the queue is full, failing gracefully if the timeout is exceeded.

- `appendBatch(vector of LogEntries, ProducerToken, optional filename)`: Submits a batch of log entries in one operation. Each entry is wrapped into a `QueueItem` as above, and the resulting vector of `QueueItems` is submitted to the buffer queue through a blocking batch enqueue call. The batching mechanism has the potential to drastically minimize contention and amortize the cost of synchronization on the buffer queue.
- `exportLogs(outputPath, Timestamp from, Timestamp to)`: Intended to support export of stored log data for audit or analysis. This method is currently a placeholder; its future implementation is expected to delegate filtering, decrypting, decompressing, and exporting logs to a dedicated component based on the specified timestamp range and output destination.

Internal Checks and Design Notes

- All methods verify that the `Logger` is properly initialized before proceeding, else the API methods emit a diagnostic error message and either fail gracefully or throw an exception.
- The `QueueItem` wrapper encapsulates a log entry and optionally associates it with a specific output file. This enables flexible routing of logs to different files while preserving batch semantics.
- The design of the `Logger` ensures full decoupling between log submission and persistence, helping maintain responsiveness even under heavy load.

5.4 Concurrent Buffer Queue

Foundation: moodycamel::ConcurrentQueue

At its core, the concurrent `BufferQueue` leverages the `moodycamel::ConcurrentQueue`, an open-source, lock-free, multi-producer, multi-consumer queue developed by Cameron Desrochers [4, 3]. This queue implementation is a popular library, renowned for its high performance and scalability in concurrent environments.

Its **key concepts and design principles** are:

- **Lock-Free and Wait-Free Semantics**: The queue achieves concurrency without relying on mutexes. Enqueue and dequeue operations use atomic Compare-and-Swap (CAS) primitives, ensuring non-blocking execution and preventing common locking overheads or contention bottlenecks.

- **Per-Producer Sub-Queues**: Each producer thread is assigned a dedicated sub-queue, implemented as a circular buffer. This reduces false sharing and allows producers to write without synchronizing with each other. Consumers cycle through these sub-queues to retrieve data in a round-robin or demand-driven fashion.
- **Cache-Friendly Layout**: Sub-queues are allocated with memory alignment and padding strategies that avoid cache-line contention between threads. This design dramatically improves cache locality and lowers latency.
- **Token-Based API**: The library introduces `ProducerToken` and `ConsumerToken` abstractions to bind thread-local metadata and optimize queue access paths. Tokens reduce overhead by caching lookups and reducing contention during queue traversal.
- **Bulk Enqueue and Dequeue**: Both enqueue and dequeue methods support batched operations. Internally, they avoid per-item atomic operations by allocating multiple slots ahead of time, enabling amortized synchronization costs and throughput scalability.

Custom Wrapper Code

Building upon the `moodycamel::ConcurrentQueue`, the `BufferQueue` serves as a specialized wrapper tailored to the logging system's requirements. It operates on `QueueItem` objects, which encapsulate individual log entries (`LogEntry`) along with optional target filenames.

Primary Responsibilities:

- **Failing Enqueue Operations (Internal):**
  - `enqueue(QueueItem, ProducerToken)`: A private method for submitting a single item non-blockingly. It returns true on success or false if the queue is full. This method is not exposed to external components and serves as a utility within the blocking wrapper.
  - `enqueueBatch(vector of QueueItems, ProducerToken)`: A private method for non-blocking submission of multiple entries. This allows efficient batch handling inside the wrapper and is similarly shielded from external access.
- **Blocking Enqueue Operations (Exposed):**
  - `enqueueBlocking(QueueItem, ProducerToken, Timeout)`: This publicly exposed method repeatedly attempts to enqueue a single `QueueItem` using exponential backoff until either success or timeout. It provides robust overload

handling for producers.

  - `enqueueBatchBlocking(vector of QueueItems, ProducerToken, Timeout)`: Exposed for bulk logging, this function applies the same retry mechanism to an entire batch of entries. It ensures that either all entries are enqueued or the operation fails cleanly within the given duration. By encapsulating retry logic in the wrapper, producers are relieved from dealing with this complexity directly.

- **Dequeue Operations:**
  - `dequeue(QueueItem, ConsumerToken)`: Retrieves one item from the queue if available. It returns a boolean indicating success and places the dequeued item in the provided reference.
  - `dequeueBatch(vector of QueueItems, ConsumerToken, maxItems)`: Attempts to dequeue up to `maxItems` entries, storing them into the provided vector. This method is optimized for writer thread consumption and minimizes synchronization costs.

- **Queue State Management:**
  - `size()`: Returns an approximate count of items currently buffered. This approximation arises from the internal implementation of the underlying `ConcurrentQueue`, which avoids global synchronization to maintain performance. Since producers and consumers operate independently with minimal coordination, intermediate states (e.g., items in transit between sub-queues) cannot be fully captured without introducing locking, which the library explicitly avoids. Therefore, the value is a performant but non-atomic estimate and should be used only for coarse-grained system diagnostics or monitoring.
  - `flush()`: Blocks until the queue is completely drained. Used primarily during shutdown to ensure all log data is processed. Importantly, the `flush()` method is only invoked after the system has prevented any further enqueue operations by not accepting entries from producers anymore. This ensures that no new entries are added during flushing. In this exclusive consumption state, the queue becomes drain-only, which guarantees that polling until the approximate `size()` reaches zero is logically safe and does not risk dropping entries.

5.5 Writer Threads

Writer threads are the core execution units responsible for asynchronously consuming batches of buffered log entries and persisting them to disk in a secure and space-efficient manner. They operate independently and are orchestrated by the `LoggingManager`. The number of active writers is configured via the `numWriterThreads` parameter of the `LoggingManager` parent component.

Configuration Parameters:

- `BufferQueue`: Reference to the shared concurrent queue from which log entries will be consumed.
- `SegmentedStorage`: Handle to the segmented, append-only file backend.
- `batchSize`: Maximum number of entries to dequeue and process in a single cycle.
- `useEncryption`: Flag to determine if AES-GCM encryption is enabled.
- `compressionLevel`: Integer parameter controlling compression behavior. `0` disables compression, `1-9` specify compression levels (higher values favor compression ratio over speed).

Lifecycle Management:

- `start()`: Initializes and launches the internal thread, which enters the main loop `processLogEntries()`. A flag ensures that a writer cannot be started more than once.
- `stop()`: Signals the thread to halt by updating the atomic `m_running` flag. If the thread is active, it is joined to ensure graceful shutdown.
- `isRunning()`: Simple accessor to check the operational status of the writer thread.

Main Execution Loop:

The `processLogEntries()` method executes continuously while the thread is active. Its responsibilities are as follows:

1. **Dequeuing Items:** uses `dequeueBatch()` with the configured batch size to fetch multiple `QueueItems` at once. If the queue is empty, the thread sleeps briefly (5ms) to reduce busy-waiting and CPU load.

2. **Grouping by Target File:** fetched `QueueItems` are grouped into buckets of log entries based on their optional destination filename.
3. **Batch Serialization:** each group of `LogEntry` objects is serialized as a batch into a binary buffer using `LogEntry::serializeBatch()`, ensuring a compact and consistent format.
4. **Optional Compression:** If enabled, the serialized buffer is passed to the Compression::compress() method along with the configured `compressionLevel` parameter, reducing the total disk footprint. The `Compression` component is a utility wrapper developed for this project that internally leverages the widely adopted `zlib` compression library.
5. **Optional Encryption:** If enabled, the resulting buffer is encrypted with AES-GCM via `Crypto::encrypt()`. The `Crypto` component is similarly a project-specific wrapper around the robust, industry-standard cryptographic backend OpenSSL. For the purpose of this prototype, the encryption is only using a pre-set key and Initialization Vector (IV). In production, this would be needed to be replaced with secure key management and unique IV generation.
6. **Appending to Disk:** If a target filename is specified, the writer dispatches the buffer to the `writeToFile()` method of the `SegmentedStorage` component. Otherwise, it uses the default `write()` method, that will write the data to the default destination file.
7. **Batch Cleanup:** After processing, the local batch container is cleared for reuse in the subsequent iteration.

5.6 Segmented Storage - File System Management

The `SegmentedStorage` component persists log data to disk in an efficient, append-only, and segment-structured format. It supports concurrent access to the same file by multiple writer threads, enforces file size limits through rotation, and manages file descriptor reuse through an integrated LRU cache.

Initialization and Configuration

Upon instantiation, `SegmentedStorage` accepts the following parameters:

- `basePath`: The root directory name where all log segments are stored.
- `baseFilename`: The default filename used when no explicit file is specified by producers.

- `maxSegmentSize`: Upper bound in bytes for each log file. Exceeding this threshold triggers file rotation.
- `maxAttempts`: Maximum number of retries for write operations.
- `baseRetryDelay`: Initial delay for the exponential backoff strategy used in write retry logic.
- `maxOpenFiles`: Maximum number of file descriptors that can remain open simultaneously. This value governs the size of the internal LRU cache.

The constructor ensures the target directory exists and preloads the segment metadata for the base filename.

Segment Lifecycle and Access Coordination

Each logical log stream (identified by filename) is tracked via a `SegmentInfo` structure, which contains:

- `currentOffset`: Atomic counter tracking the current write position in bytes in the active segment.
- `segmentIndex`: Monotonically increasing index used for naming rotated files.
- `currentSegmentPath`: Full filesystem path of the active file segment.
- `fileMutex`: Shared mutex to guard access to each segment's metadata and I/O operations.

Segment entries are lazily created using `getOrCreateSegment()`, and safely inserted into an internal map guarded by `m_mapMutex`.

Writing Data: Concurrency and Rotation

Data writes are handled through the `writeToFile(filename, data)` method. This function:

1. Checks the current segment's `currentOffset` to determine if a rotation is necessary.
2. If required, acquires an exclusive lock to perform the segment rotation. A new file path is generated using `generateSegmentPath()` based on the current timestamp and segment index.
3. After validating or rotating, the method safely reserves a byte range in the file by atomically incrementing `currentOffset`.

4. Ensures the segment path and file descriptor remain stable throughout the operation by acquiring a shared lock.
5. Performs a `pwrite()` to the reserved byte offset.

This loop-and-check logic ensures that concurrent writer threads can safely append to the same or different files without races, while maintaining segment boundaries.

File Descriptor Cache (`FdCache`)

Opening and closing file descriptors is expensive. To avoid this cost and respect system limits, `SegmentedStorage` integrates a custom LRU-based file descriptor cache called `FdCache`. It:

- Maintains an internal mapping of segment paths to file descriptors.
- Uses an LRU list to evict least-recently-used entries when `maxOpenFiles` is exceeded.
- Synchronizes access via a dedicated mutex.
- Ensures `fsync()` is called on a file before it is closed to guarantee durability.

The cache is used throughout `writeToFile`, `rotateSegment`, and `flush()` operations.

Flush Mechanism

The `flush()` method ensures all dirty file buffers are persisted to disk. It iterates over all known segment paths, retrieves or opens their descriptors via the cache, and calls `fsync()` on each. Access is synchronized using both the global `m_mapMutex` and each segment's `fileMutex` to ensure no concurrent writes or rotations interfere.

Segment Path Generation

File names are timestamped using the pattern `<basePath>/<filename>_YYYYMMDD_HHMMSS _<segmentIndex>.log`. This naming convention ensures uniqueness and supports future log audits and exports.

6 Evaluation

6.1 Research Questions and Evaluation Goals

The logging system developed in this thesis is designed to serve as a secure, tamper-evident, and high-throughput audit layer for GDPR-compliant data systems. As discussed in previous chapters, the core challenges addressed by this system include preserving performance in write-heavy environments and ensuring data integrity and confidentiality.

This evaluation chapter aims to systematically assess whether the system fulfills its intended goals in a verifiable and reproducible manner through empirical benchmarking and stress testing. To guide this evaluation, we define the following high-level research questions:

How does the system perform under under heavy workloads and varying configurations?
The primary design goal of the system is to achieve high-performance logging without bottlenecking upstream services. This question highlights the impact of different system configuration parameters and workload exposures on overall throughput, measured in both log entries per second and bytes written per second.

What is the achieved latency of the proposed logging system under various workloads and configurations?
While high throughput is essential, individual client operations must also return promptly to ensure responsive system behavior. This question evaluates the latency experienced by producers during `append()` operations, capturing both average and tail latencies under different system configurations. Insights from this metric are critical for real-time systems and interactive applications, where outlier delays can impact user experience and system reliability.

What is the effect of compression and encryption on the system's performance?
To comply with GDPR requirements for confidentiality and integrity, all log entries are encrypted before being persisted. Compression, on the other hand, is primarily motivated by practical concerns: it helps reduce the volume of data written to disk, which is critical for maintaining scalability in write-intensive environments. However,

both encryption and compression are computationally expensive operations that may degrade system performance. This question investigates the performance impact of enabling these features individually and in combination, compared to a baseline with both disabled.

How well does the system scale with increasing numbers of producer and writer threads?
Scalability is critical in high-concurrency environments, particularly in distributed systems where multiple sources generate logs simultaneously. This question assesses the system's ability to scale horizontally and sustain throughput that scales with the consumed resources as parallelism increases.

Is the system reliable and robust under adverse or edge-case conditions?
A compliance logging system must guarantee data persistence even under stress. This question tests the behavior of the system under scenarios such as full queues, segment rotations, and I/O errors—validating retry logic, graceful shutdown, and data flushing mechanisms.

6.2 Experimental Setup

To thoroughly evaluate the performance and verify the expected functionality, a series of benchmarks and microbenchmarks were conducted on a high-end benchmarking server. This section outlines the hardware environment, variable software configurations, and the methodology used for measurement.

6.2.1 Hardware Environment

All benchmarks were conducted on a dedicated dual-socket Intel Xeon Gold 6326 server with the following key specifications:

- CPU: 2 × Intel Xeon Gold 6326 (32 physical cores, 64 threads total)
- RAM: 320 GiB DDR4-3200 ECC
- Disk: Intel S4510 SSD (SATA, 960 GB)
- Filesystem: ZFS on Linux (NixOS 24.11), kernel version 6.12.12
- OS and Build Environment: NixOS with GCC 13.3.0, benchmarks compiled with optimization flags enabled (-O3)

All benchmarks were executed with threads pinned to NUMA node 0 using `numactl` `(-cpunodebind=0 -membind=0)` to ensure consistent memory access patterns and eliminate NUMA-related performance variability. The server was not concurrently used by other processes during benchmarking, ensuring consistent, uncontended performance results.

6.2.2 System Configuration

Since the logging system is highly configurable, each benchmark targeted a specific aspect of system behavior by varying one or two parameters while keeping others fixed. Key configurable parameters include:

- Number of threads (producer and consumer)
- Encryption usage flag
- Compression level
- Batch size (producer and consumer)
- Queue capacity
- Number of different destination files
- Size threshold for the LRU file cache
- Size threshold for log file rotation

6.2.3 Measurement Methodology

To gather performance metrics, the benchmarking suite embedded within the logging system recorded detailed performance data for each test run. The following measurement techniques and metrics were used:

- **Log Data Generation**: Our benchmarks begin by generating synthetic `LogEntry` data. Since the compressibility of submitted log entries plays a crucial role in both execution time and final disk footprint, we synthetically created log entries with realistic compression characteristics. While typical `zlib` compression ratios for natural language range from 2:1 to 5:1 [10, 34], log data is often highly repetitive and can achieve significantly higher compression ratios [34]. Therefore, the synthetic log data for our benchmarks was specifically designed to achieve the following range of compression ratios across `zlib` compression levels `0-9`:

zlib Level	Compression Ratio
0	1.00
1	9.15
2	10.88
3	13.15
4	13.74
5	15.54
6	17.44
7	18.18
8	19.55
9	20.65

Table 6.1: Compression ratios of synthetic benchmark log data by zlib level

Each generated batch contains a predefined number of entries with 4096-byte payloads and structured metadata designed to achieve these target compression ratios, along with round-robin file distribution. These batches are then concurrently submitted to the logging system using multiple asynchronous producer threads.

- **Time Measurement:** All timing measurements were performed using `std::chrono ::high_resolution_clock::now()` to capture precise wall-clock durations of benchmark phases. For throughput measurements, time was measured from just before log submission to the point where all data had been written to persistent storage. To avoid interference with these measurements, separate benchmark runs were conducted in which the duration of each individual `append()` call was timestamped independently to measure request latency. These measurements were taken in thread-local collectors to avoid synchronization overhead and later merged into a global set for statistical analysis.
- **Data Size Calculation:** Input data volume was computed by serializing each entry and multiplying the sum by the number of producers to determine total logical data volume.
- **On-Disk Footprint**: Final output directory size was obtained by querying all generated log segments and reporting total storage used, accounting for metadata, padding, encryption tags, and compression.
- **Key Metrics:** The key metrics recorded include **throughput**, **latency**, and **write amplification**. Throughput refers to the number of log entries written per second, as well as the effective throughput in gibibytes per second (GiB/s), calculated

using both the input size and the on-disk size. Latency includes the maximum, mean, and median latencies for `append()` calls, measured in milliseconds. Write amplification is defined as the ratio of physical storage used (after batched serialization, compression, and encryption) to the original serialized input size.

The benchmarking framework is lightweight and tightly integrated into the logging system itself, designed to minimize the effect of measurement interference. No external profilers or instrumentation were used during performance evaluation to ensure the observed behavior reflects true system performance.

6.3 Performance Benchmarks

The benchmarking phase of this thesis pursues two primary objectives: to determine the optimal configuration parameters for achieving maximum system performance, and to evaluate how the logging system behaves under diverse, realistic workloads. The methodology for all benchmarks is based on synthetic log entry generation and concurrent submission to the logging system using asynchronous producer threads. This setup ensures a controlled environment that isolates the system's internal behavior and response under known input conditions. Throughput, request latency, and write amplification are the key metrics captured in each case.

Effect of Writer Batch Size

One of the most crucial tuning parameters in the logging system is the writer batch size—i.e., the number of log entries that writer threads dequeue in a single cycle. This benchmark investigates how varying this parameter affects system throughput, latency, and write amplification. The objective is to empirically determine an optimal writer batch size that maximizes throughput and minimizes write amplification, without incurring significant overhead or diminishing returns. Small batch sizes are expected to suffer from frequent encryption, compression, and I/O invocations, while excessively large batches may diminish the effect of concurrency.

The following parameters were fixed for running this benchmark:

- **Producer Threads:** 16
- **Entries per Producer**: 2000000
- **Total Data Size**: 124.6 GiB
- **Producer Batch Size:** 4096
- **Files Targeted:** 256 different log file paths, assigned round-robin

- **Writer Threads:** 16
- **File Rotation:** disabled via maximum segment size of 500 MiB
- **Queue capacity:** 2,000,000 (rounded internally to 2,097,152)
- **File Cache Size:** 512
- **Encryption:** enabled
- **Compression Level:** 4, balanced fast

Batch sizes tested were all powers of two ranging from 1 to 131072 (i.e., 1, 2, 4, ..., 131072), for a total of 18 distinct values.

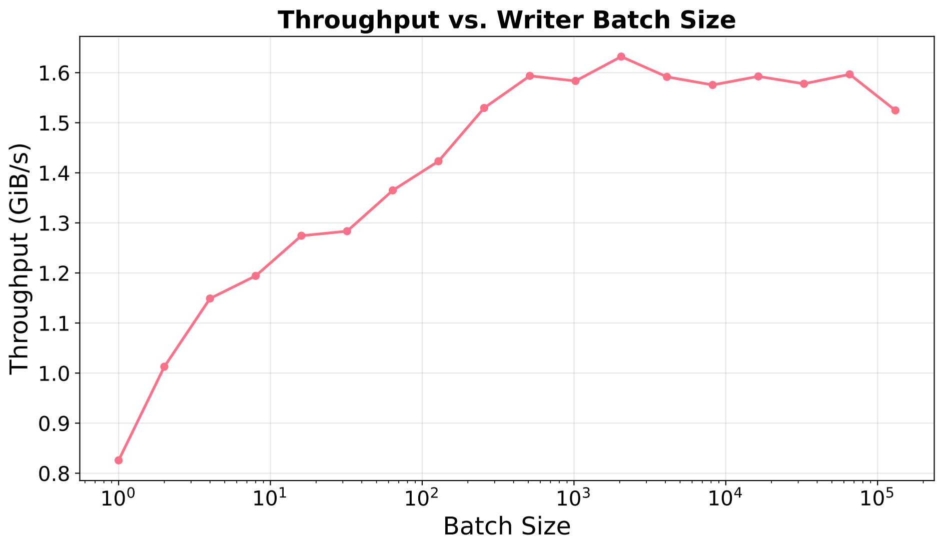

Figure 6.1: This figure shows the throughput in GiB per second for different batch sizes of the processing writer threads

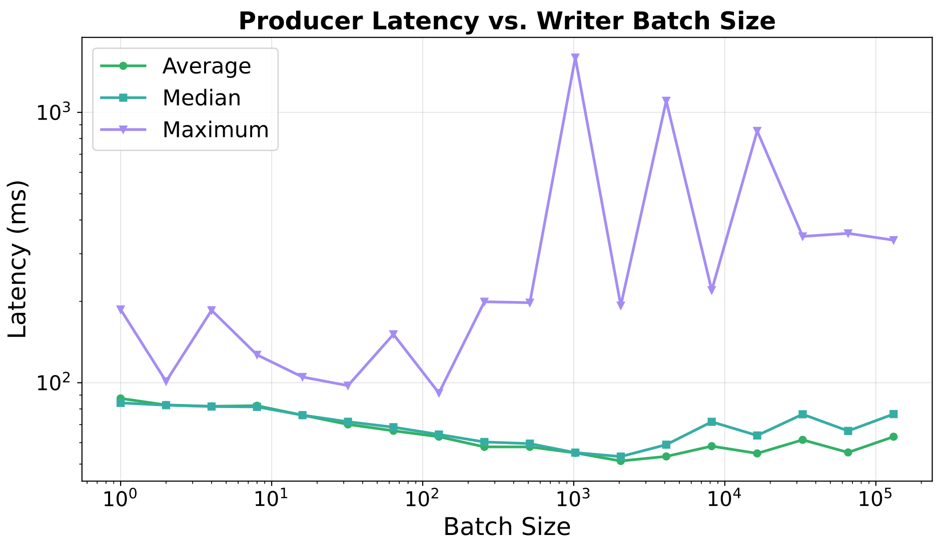

Figure 6.2: This figure shows the latency perceived by client threads (log producers) in milliseconds for different batch sizes of the processing writer threads

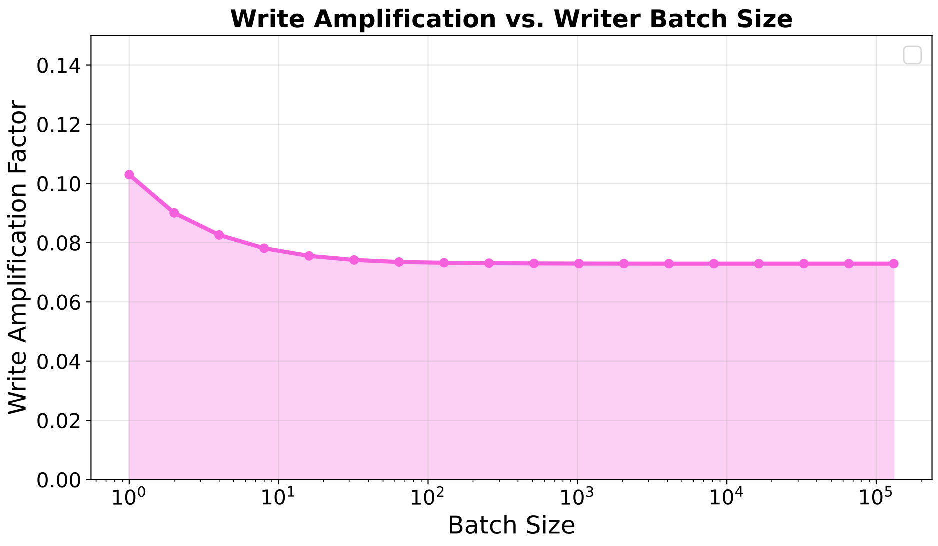

Figure 6.3: This figure shows the write amplification factor of the submitted log data for different batch sizes

The results, as shown in Figures 6.1, confirm a strong and consistent gain in throughput as the batch size increases—up to approximately 2.0× improvement when comparing batch size 1 (0.83 GiB/s) to the peak at batch size 2048 (1.63 GiB/s). This can mainly be attributed to the amortization of encryption and compression overhead across larger payload blocks. Beyond this peak, throughput decreases marginally but remains relatively stable, ranging between 1.52–1.60 GiB/s for larger batch sizes. Write amplification shows a decent reduction, dropping from 0.103 at batch size 1 to around 0.073 at batch size 64, where it stabilizes and remains essentially constant for all subsequent batch size increases (see Figure 6.3). Average and median latencies consistently decrease until reaching their optimal point at batch size 2048 (51.4 ms average, 53.3 ms median), after which they begin to increase slightly again for larger batch sizes. However, maximum latency exhibits a different pattern: it generally increases from lower to higher batch sizes (from 186.6 ms at batch size 1), with latency variance increasing noticeably as evidenced by occasional spikes exceeding 1,000 ms at intermediate batch sizes, though for very large batch sizes (32,768 and above) maximum latency stabilizes around 300–400 ms despite remaining elevated (see Figure 6.2). The analysis suggests an optimal operational point around batch size 2048, where the system achieves peak throughput, minimal average latency, and stabilized write amplification, while balancing maximum latency variance and memory usage considerations.

Impact of Writer Concurrency

This section evaluates how the number of writer threads influences performance. The goal is to measure the efficiency and scalability of parallel log writing (where efficiency is measured as actual throughput speedup relative to ideal linear scaling), and to identify at what point the system's performance begins to saturate—either due to CPU limits, synchronization overhead, or I/O contention. Two complementary benchmarks were conducted:

- **Fixed-Workload Concurrency Benchmark:** The input size is kept constant while varying the number of writer threads. This measures how the system scales its throughput as more threads are introduced to handle the same workload.
- **Scalability Benchmark:** The input size is scaled proportionally with the number of writer threads by adding more producer threads, where each producer submits the same number of entries. This simulates how the system performs under increasing load with proportional resource provisioning.

The following were the fixed test conditions for running these benchmark:

- **Entries per Producer**: 2000000
- **Producer Batch Size:** 4096
- **Writer Batch Size:** 2048

- **Files Targeted:** 256 different log file paths, assigned round-robin
- **File Rotation:** disabled via maximum segment size of 500 MiB
- **Queue capacity:** 2,000,000 (rounded internally to 2,097,152)
- **File Cache Size:** 512
- **Encryption:** enabled
- **Compression Level:** 4, balanced fast

For the **fixed-workload benchmark**, a total of 62.3 GiB input data was written, split evenly among 8 producers. The number of writer threads was increased from 1 to 16. For the **scalable-workload benchmark**, the number of producer threads and thus the total input data volume was scaled linearly with the number of writer threads from 1 to 16. The thread count ranges (1-16) were chosen based on the hardware constraints of our NUMA-pinned environment. With execution restricted to NUMA node 0, we have access to 16 physical cores. This setup allows us to analyze two distinct scaling regions. In the **linear scaling region (<=16 total threads)** the combined thread count of producers and writers does not exceed the available physical cores, enabling optimal resource utilization without context switching overhead. Beyond this threshold, in the **degradation region (>16 total threads)**, threads must compete for CPU resources, leading to context switching overhead and reduced scaling efficiency. For both benchmarks, we expect linear scaling of throughput until reaching 8 writer threads (totaling 16 active threads). Beyond this point (9-16 writer threads), the total thread count exceeds available physical cores, and we anticipate performance degradation due to CPU contention.

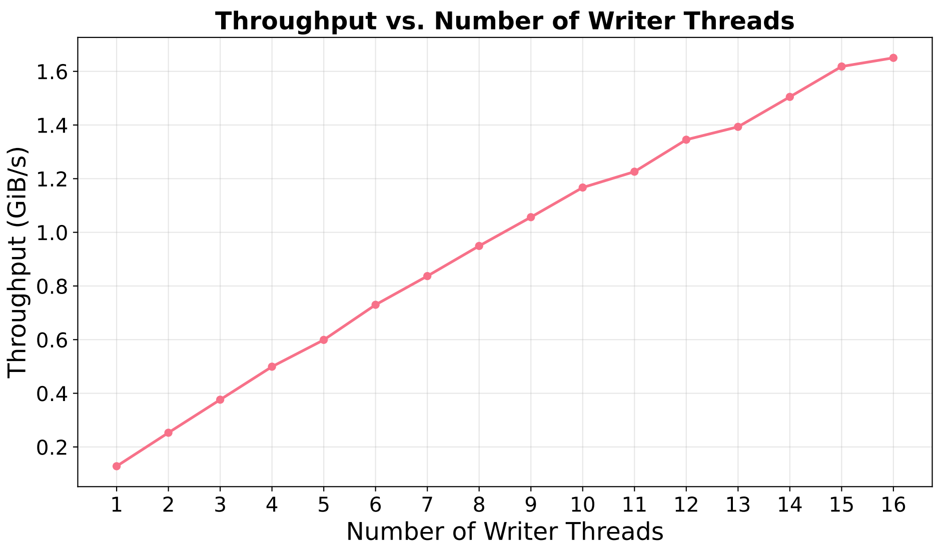

Figure 6.4: This figure shows the throughput in GiB per second for a varying number of writer threads (workload kept constant)

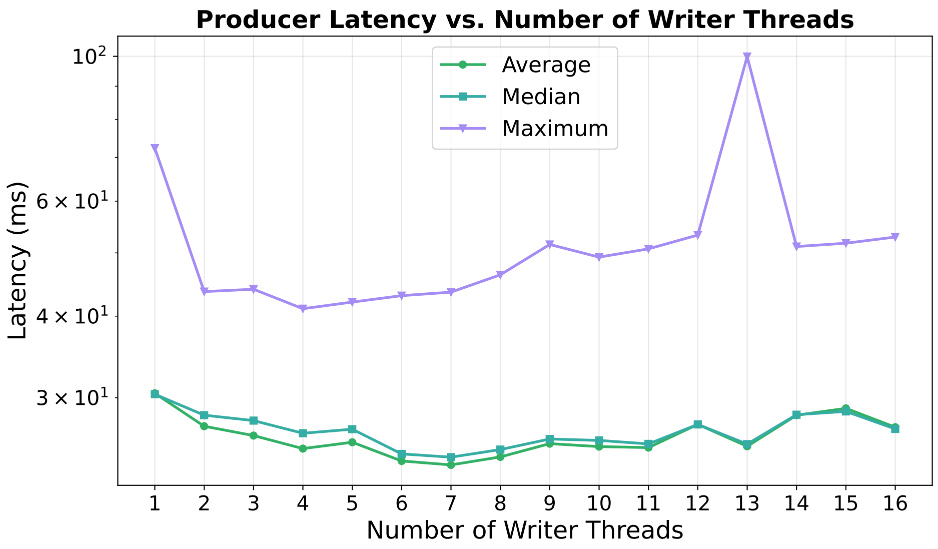

Figure 6.5: This figure shows the latency perceived by client threads (log producers) in milliseconds for a varying number of writer threads (workload kept constant)

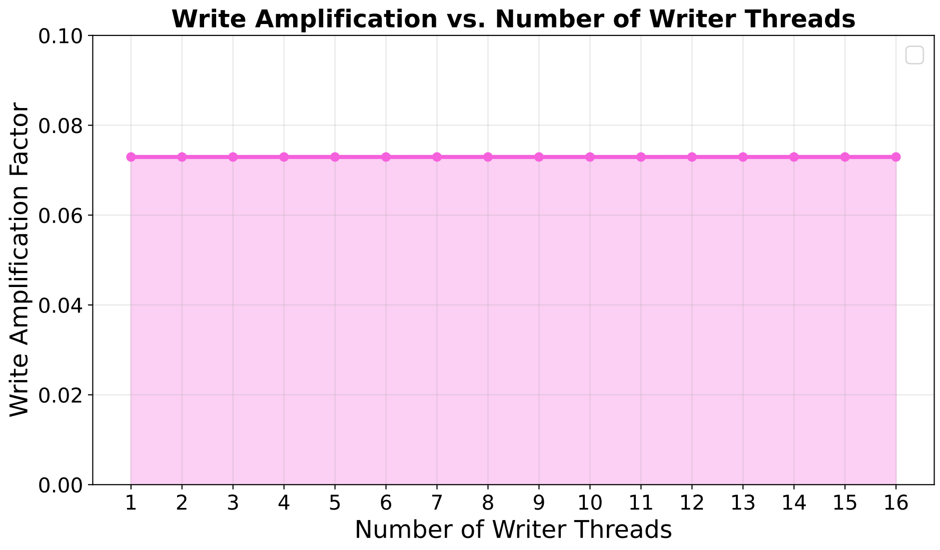

Figure 6.6: This figure shows the write amplification factor of the submitted log data for a varying number of writer threads (workload kept constant)

The fixed-workload concurrency benchmark shows good scalability up to 16 writer threads, achieving a 12.9× throughput improvement from 0.128 GiB/s to 1.65 GiB/s. The system demonstrates distinct performance characteristics across the two scaling regions. In the linear scaling region (1-8 writer threads, totaling <=16 threads), the system maintains near-optimal scaling efficiency, achieving 93% efficiency at 8 threads with minimal overhead. Beyond this threshold, in the degradation region (9-16 writer threads, >16 total threads), efficiency decreases but remains solid at 80% for 16 threads, with throughput continuing to increase steadily. Average latency improves with increased parallelization, decreasing from 30.5 ms at single-threaded operation to approximately 23-27 ms across most configurations, suggesting efficient load distribution. Maximum latency shows a more complex pattern: it improves significantly from 72.3 ms to the 41-46 ms range in the linear scaling region, but begins to increase in the degradation region, reaching 50-100 ms due to increased CPU contention and context switching overhead. Write amplification remains perfectly constant at 0.0729 throughout all configurations, demonstrating that the system maintains optimal efficiency regardless of concurrency level.

The fixed-workload concurrency benchmark shows good scalability up to 16 writer threads, achieving a 12.9× throughput improvement from 0.128 GiB/s to 1.65 GiB/s (see Figure 6.4). The system demonstrates distinct performance characteristics across the two scaling regions. In the linear scaling region (1-8 writer threads, totaling <=16

threads), the system achieves 93% efficiency at 8 threads, falling short of the expected perfect linear scaling but still demonstrating strong parallel performance. Beyond this threshold, in the degradation region (9-16 writer threads, >16 total threads), efficiency decreases further but remains solid at 80% for 16 threads, with throughput continuing to increase steadily. Average latency improves with increased parallelization, decreasing from 30.5 ms at single-threaded operation to approximately 23-27 ms across most configurations, suggesting efficient load distribution (see Figure 6.5). Maximum latency shows a more complex pattern: it improves significantly from 72.3 ms to the 41-46 ms range in the linear scaling region, but begins to increase in the degradation region, reaching 50-100 ms. Write amplification remains perfectly constant at 0.0729 throughout all configurations (see Figure 6.6), demonstrating that the system maintains optimal efficiency regardless of concurrency level.

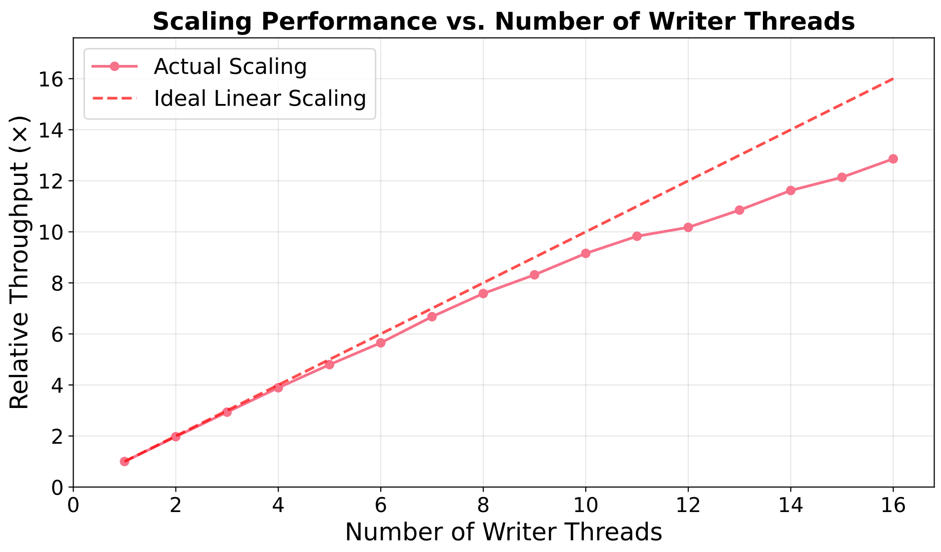

Figure 6.7: This figure shows the throughput in GiB per second for a varying number of writer threads (workload scaled accordingly)

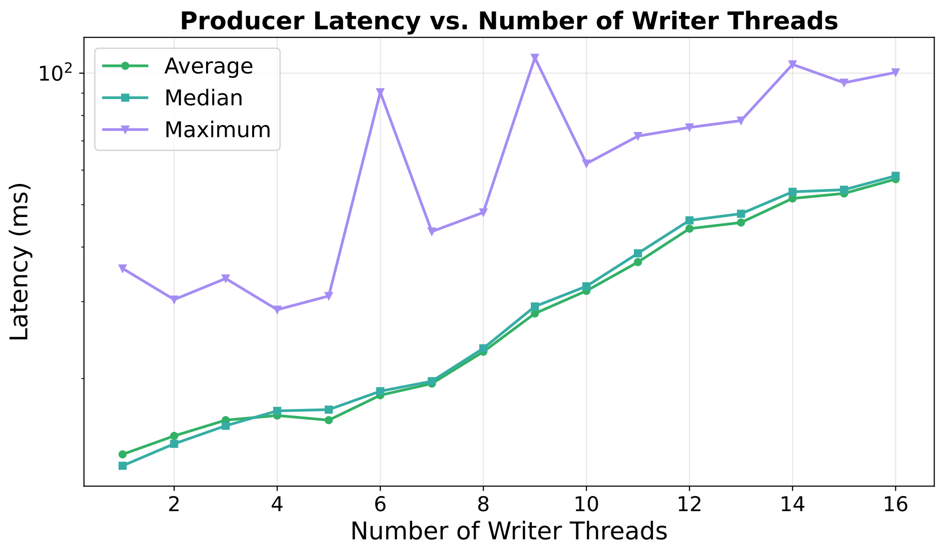

Figure 6.8: This figure shows the latency perceived by client threads (log producers) in milliseconds for a varying number of writer threads (workload scaled accordingly)

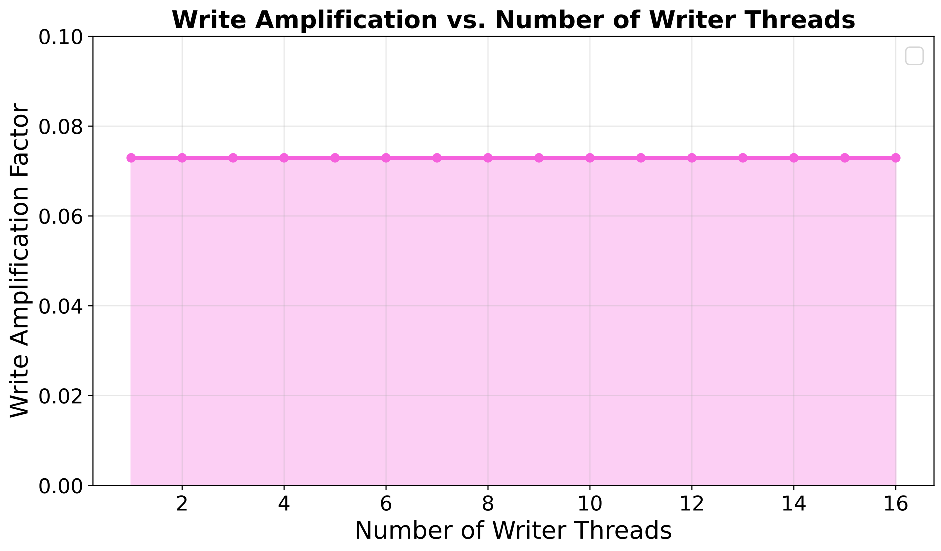

Figure 6.9: This figure shows the write amplification factor of the submitted log data for a varying number of writer threads (workload scaled accordingly)

The scalability benchmark similarly reveals good scaling characteristics across the tested range. The system maintains 95% scaling efficiency at 16 total threads (8 writers) and 80% efficiency at 32 total threads (16 writers), achieving approximately 12.9× relative performance improvement at that point. Likewise, some degradation in scaling efficiency becomes apparent beyond 16 threads in this benchmark as well due to thread scheduling overhead, synchronization costs, and I/O contention.

Effect of Queue Capacity

The queue capacity parameter controls the maximum number of log entries that can be buffered in memory between producer and writer threads. This benchmark investigates how varying queue sizes affects system performance, particularly focusing on the trade-offs between memory usage, throughput, and latency characteristics. Understanding optimal queue sizing is crucial for preventing producer thread blocking while avoiding excessive memory consumption in production deployments. Smaller queue capacities are expected to cause more frequent blocking of producer threads when the system cannot keep up with write demand, potentially leading to higher tail latencies. Conversely, larger queues should provide better buffering against bursty workloads but may increase memory pressure and average response times due to longer queuing delays. The following parameters were fixed for running this benchmark:

- **Producer Threads:** 16
- **Entries per Producer**: 2000000
- **Total Data Size**: 124.6 GiB
- **Producer Batch Size:** 4096
- **Writer Batch Size:** 2048
- **Files Targeted:** 256 different log file paths, assigned round-robin
- **Writer Threads:** 16
- **File Rotation:** disabled via maximum segment size of 500 MiB
- **File Cache Size:** 512
- **Encryption:** enabled
- **Compression Level:** 4, balanced fast

Queue capacities tested were all powers of two ranging from 8192 to 33554432, for a total of 13 distinct values.

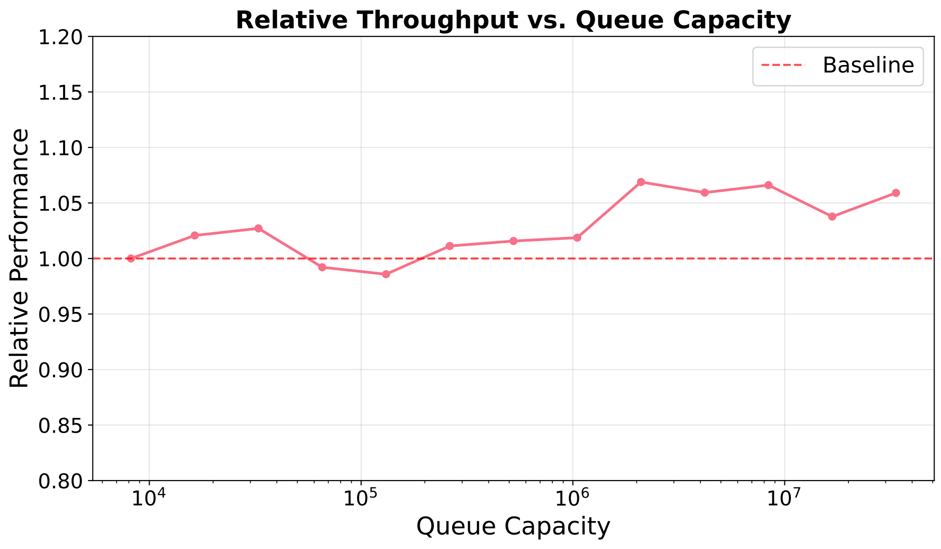

Figure 6.10: This figure shows the relative throughput in GiB per second for various queue capacities, with values normalized against the throughput achieved using the smallest queue capacity

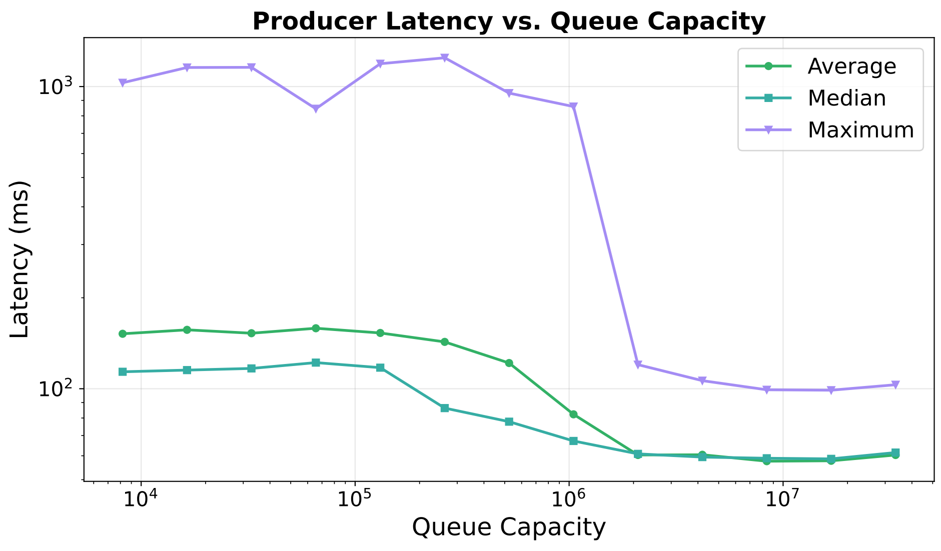

Figure 6.11: This figure shows the latency perceived by client threads (log producers) in milliseconds for different queue capacities

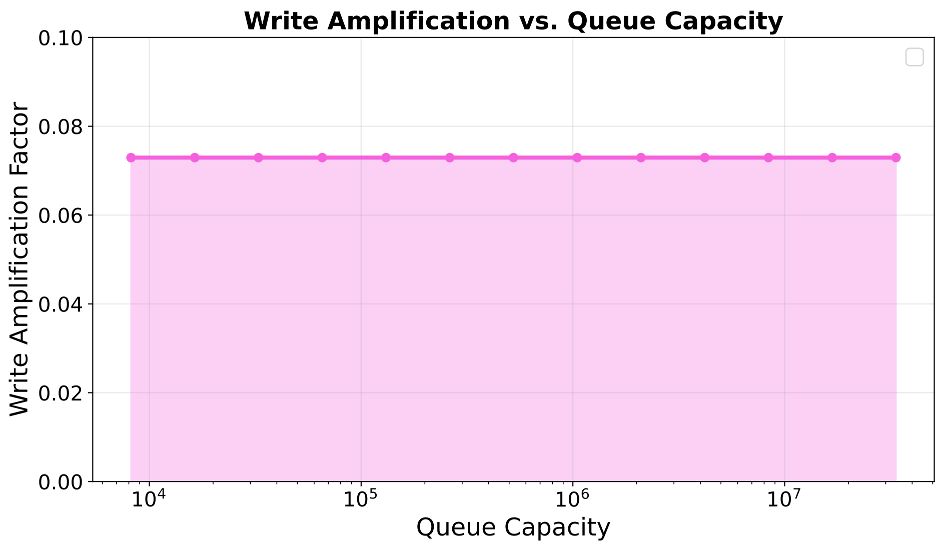

Figure 6.12: This figure shows the write amplification factor of the submitted log data for different queue capacities

The results, as shown in Figures 6.10, 6.11, 6.12, reveal several interesting characteristics of queue capacity tuning. Peak throughput is achieved with larger queue sizes, showing a modest but consistent improvement across the tested range—from 389,156 to 421,950 entries per second, representing an 8.4% performance gain from the smallest to the best-performing queue capacity. This suggests that while queue capacity is not a primary bottleneck for sustained throughput, larger capacities do provide measurable benefits under these workload conditions. Latency characteristics show more noticeable differences across queue sizes. Smaller queues (8,192-1,048,576 entries) exhibit significantly higher maximum latencies, ranging from 844ms to 1,245ms for worst-case append operations, indicating that insufficient buffering capacity causes producer threads to experience severe blocking when writer threads cannot keep pace with incoming data. A substantial transition occurs at a queue size of 2,097,152 entries, where larger queues demonstrate dramatically improved tail latency behavior, with maximum latencies dropping to under 120ms—a reduction of over 85% compared to smaller queue configurations. This improvement comes with the benefit of reduced average latency as well, dropping from approximately 152ms to 60ms, reflecting more efficient buffering and reduced thread blocking. Write amplification remains constant at 0.0729 across all queue sizes, confirming that buffering strategy does not affect compression and encryption efficiency. The optimal queue capacity appears to be in the range of 2-8 million entries, providing both peak throughput (6-7% above baseline) and excellent tail latency characteristics suitable for production environments where predictable response times are critical. Given the minimal performance differences within this optimal range, selecting a capacity closer to 2 million entries is recommended to minimize memory consumption while maintaining excellent performance characteristics.

Effect of File Rotation Frequency

File rotation is a critical aspect of the proposed logging system, as it determines when log segments are closed and new files are created. The maximum segment size parameter controls how frequently file rotation occurs, directly impacting both system performance and operational characteristics. This benchmark investigates the relationship between rotation frequency and system throughput, examining the overhead introduced by file creation and segment finalization operations. More frequent rotations (smaller segment sizes) are expected to reduce throughput due to increased overhead from file operations. However, smaller segments may offer operational advantages such as faster individual file processing and more granular log management. Understanding this trade-off is essential for optimizing the system for different deployment scenarios. The following parameters were fixed for running this benchmark:

- **Producer Threads:** 16
- **Entries per Producer**: 1000000
- **Total Data Size**: 62.3 GiB
- **Producer Batch Size:** 4096
- **Files Targeted:** Single log file (no file distribution)
- **Writer Threads:** 16
- **Writer Batch Size:** 2048
- **Queue capacity:** 2,000,000 (rounded internally to 2,097,152)
- **File Cache Size:** 512
- **Encryption:** disabled
- **Compression Level:** 0, disabled

We evaluated 10 distinct segment size thresholds: 64 GiB; 32 GiB; 16 GiB; 8 Gib; 4 Gib; 2 GiB; 1 GiB; 512 MiB; 256 MiB; and 128 MiB. In the following relative throughput measurements are reported with respect to the first threshold (64 GiB), where no file rotation occurred.

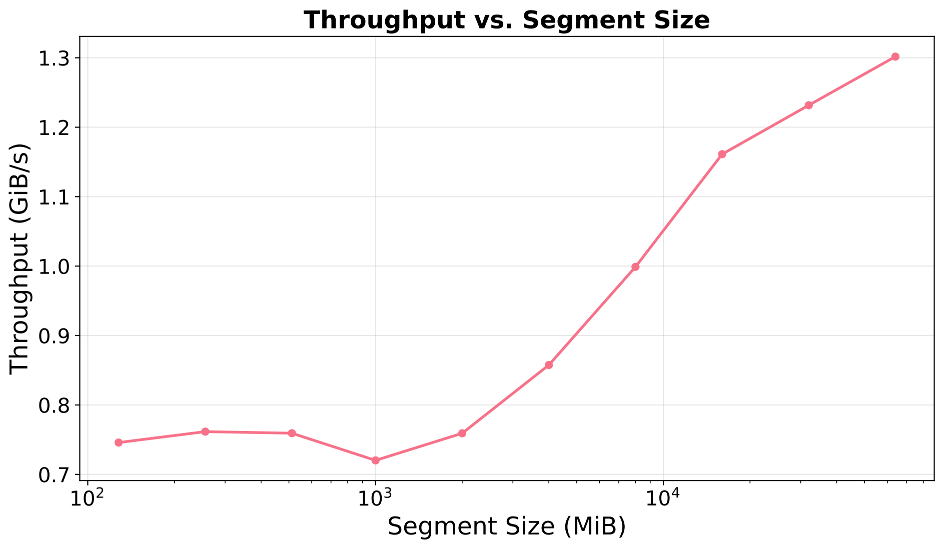

Figure 6.13: This figure shows the throughput in GiB per second for different segment sizes in MiB

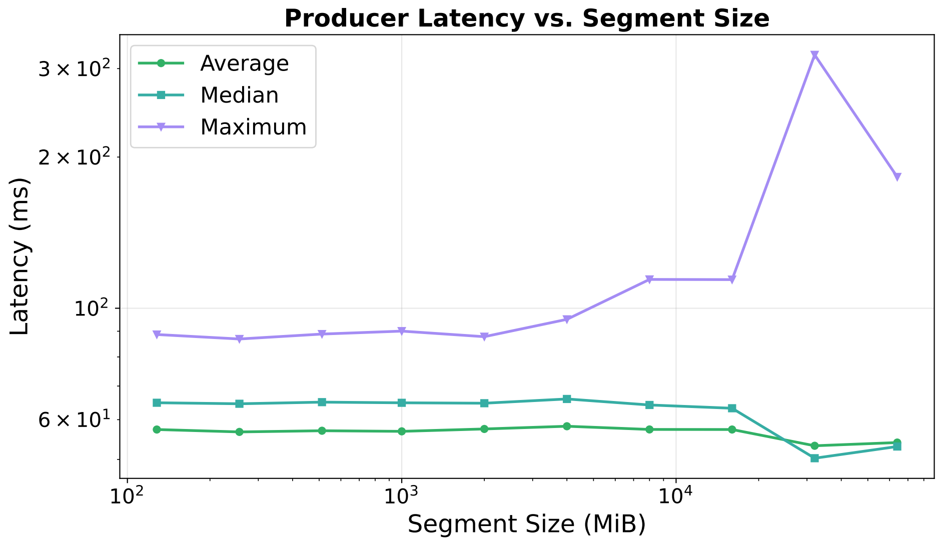

Figure 6.14: This figure shows the latency perceived by client threads (log producers) in milliseconds for different segment sizes in MiB

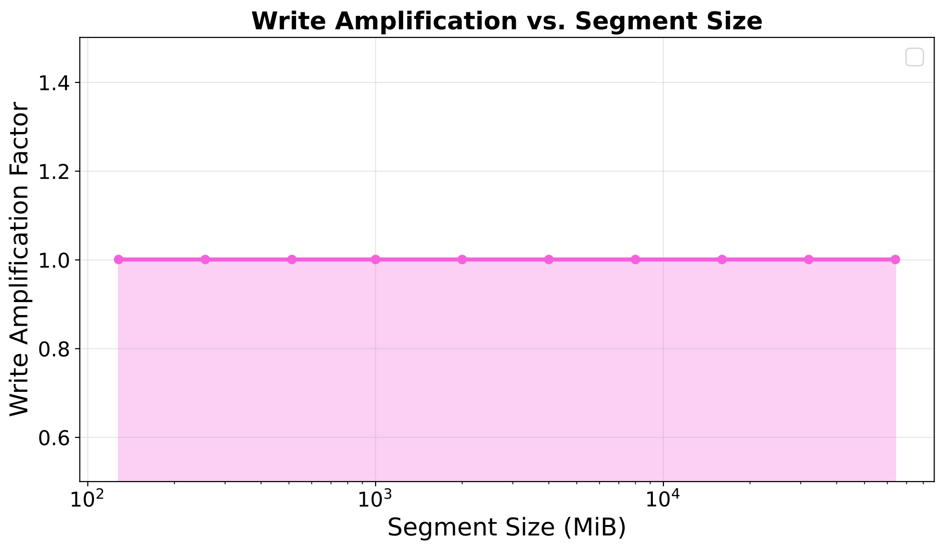

Figure 6.15: This figure shows the write amplification factor of the submitted log data for different segment sizes in MiB

The results of Figures 6.13, 6.14, and 6.15 demonstrate a clear positive relationship between segment size and logging throughput. Performance improves significantly as segment sizes increase, with throughput rising from 1.92 million entries per second for 128 MiB segments to 3.34 million entries per second for 64 GiB segments—a 74.5% increase in peak performance. The number of files created scales accordingly with rotation threshold, ranging from just 1 file for the largest segments to 521 files for the smallest. This exponential increase in file operations directly correlates with the observed performance degradation for smaller segments, confirming that file creation and finalization represent significant overhead in the logging pipeline. Write amplification remains constant across all segment sizes at approximately 1, which is expected since compression and encryption are disabled in this benchmark. Interestingly, latency characteristics remain relatively stable across different rotation frequencies, with average latencies consistently in the 53.2-58.2ms range and median latencies around 50.3-65.9ms. This suggests that while frequent rotation impacts overall throughput, individual append operations do not experience proportionally increased delays. The optimal segment size depends on the specific deployment requirements. In scenarios where log data is divided under a small number of destination files, in order to reach maximum throughput, larger segments (2 GiB or above) are preferable, achieving over 89% of peak performance while maintaining reasonable file counts. For environments requiring frequent log processing or archival under such a scenario, smaller segments

(512 MiB- 1 GiB) may be acceptable, sacrificing 15-45% of peak throughput for operational flexibility. In scenarios where log data is written to a large variety of different file names, rotation is expected to happen so rarely, that the segment size threshold will hardly have an impact on throughput and smaller segment sizes will be perfectly valid.

Effect of Encryption and Compression

The computational operations of encryption and compression introduce processing overhead that can significantly impact system performance. This benchmark systematically evaluates the individual and combined effects of encryption and compression on throughput, latency, and storage efficiency. The evaluation tests all combinations of encryption (enabled/disabled) and compression levels (0 - 9) to understand both the individual costs and potential synergies between these features. Compression level 0 represents no compression, while levels 1-9 represent increasing compression strength from speed-optimized to maximum compression ratio. Understanding these trade-offs is critical for selecting appropriate security configurations that balance compliance requirements with performance objectives. The following parameters were fixed for running this benchmark:

- **Producer Threads:** 16
- **Entries per Producer:** 300000
- **Total Data Size**: 18.69 GiB
- **Producer Batch Size:** 8192
- **Files Targeted:** 256 different log file paths, assigned round-robin
- **File Cache Size:** 512
- **Writer Threads:** 16
- **Writer Batch Size:** 2048
- **File Rotation:** 500 MiB segment size
- **Queue Capacity:** 2000000

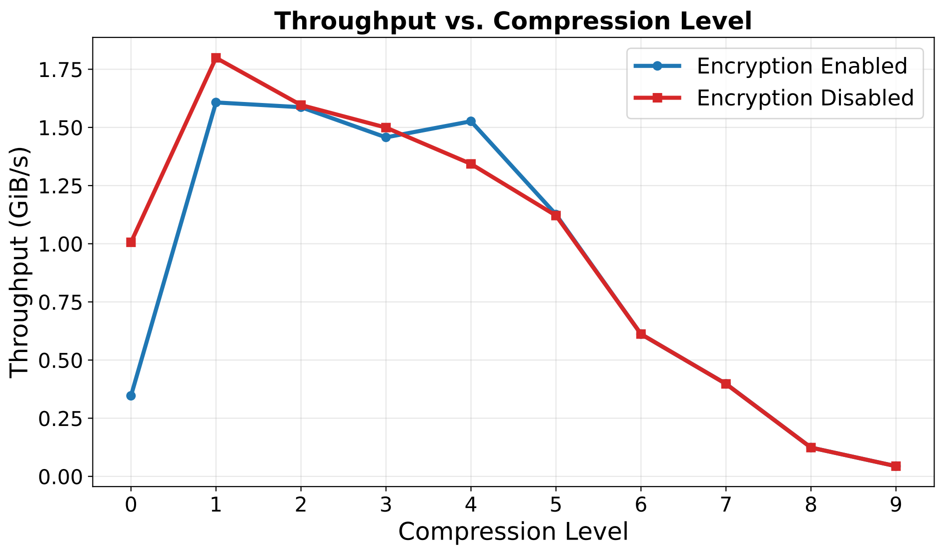

Figure 6.16: This figure shows the throughput in GiB per second for both encryption enabled and disabled for different zlib compression levels

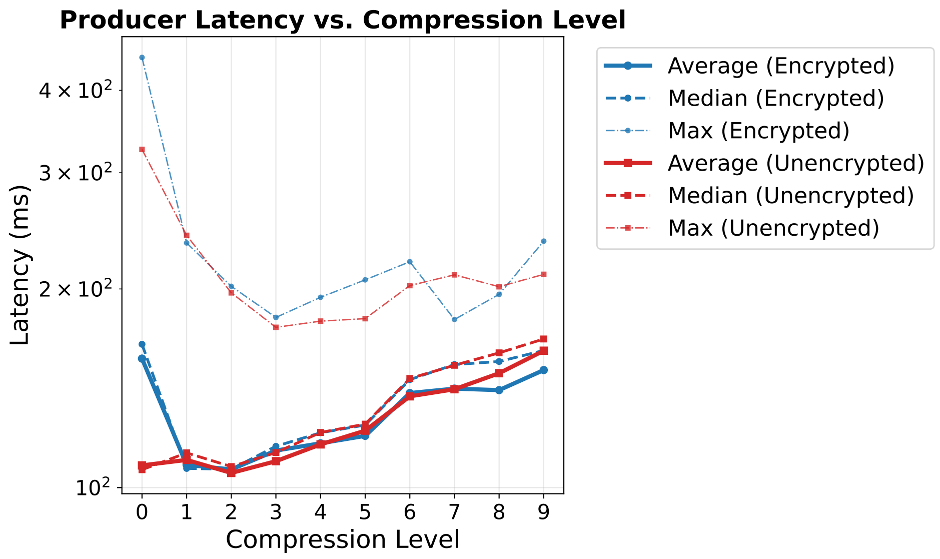

Figure 6.17: This figure shows the latency perceived by client threads (log producers) in milliseconds for both encryption enabled and disabled for different zlib compression levels

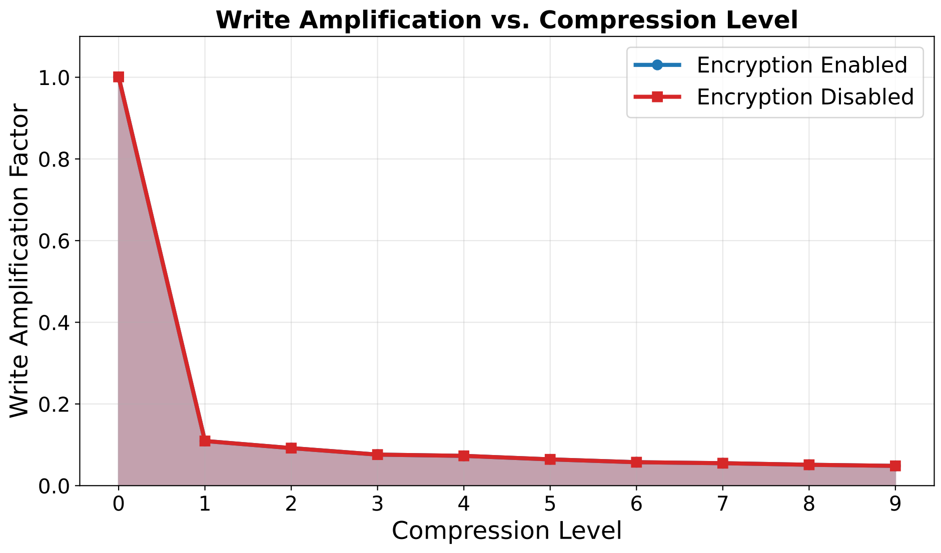

Figure 6.18: This figure shows the write amplification factor of the submitted log data for both encryption enabled and disabled for different zlib compression levels

The results, presented in Figures 6.16, 6.17, and 6.18, reveal several critical insights about the relationship between security features and system performance. The analysis demonstrates that while encryption introduces significant computational overhead, the application of compression can significantly mitigate these penalties and enable encrypted configurations to substantially gain throughput performance compared to uncompressed processing.

When analyzing configurations without compression (level 0), encryption imposes severe performance penalties. The throughput decreases by 65.6% from 1.006 GiB/s to 0.346 GiB/s when encryption is enabled. Average latency increases by 45.2% from 108.1ms to 156.8ms, while maximum latency spikes by 37.8% from 325.5ms to 448.6ms. These results underscore the significant computational burden of cryptographic operations when applied to raw, uncompressed data streams

The most significant finding emerges when examining compression's role as a performance multiplier. Compression level 1 achieves peak throughput in both configurations: 1.799 GiB/s without encryption and 1.607 GiB/s with encryption. While the encrypted peak performance remains 10.7% lower than the unencrypted peak, this represents a dramatic improvement compared to the 65.6% penalty observed without compression. This demonstrates that compression effectively reduces the computational workload for encryption by minimizing the data volume requiring cryptographic processing.

Write amplification analysis confirms compression's substantial storage benefits, with level 1 compression achieving an 89.1% reduction compared to no compression, and maximum compression (level 9) providing a 95.2% reduction. Importantly, write amplification values remain almost identical between encrypted and unencrypted configurations at equivalent compression levels, confirming that encryption overhead manifests primarily in computational complexity rather than storage efficiency.

The latency patterns exhibit complex behavior across compression levels. Without encryption, average latency increases progressively from 49.1ms to 80.1ms (63.1% increase) as compression level rises, reflecting the CPU-intensive nature of higher compression algorithms. However, with encryption enabled, average latency surprisingly decreases from 85.9ms at level 0 to 77.5ms at level 9 (9.8% improvement). This suggests that the reduced data volume from compression more than compensates for the additional compression overhead when combined with encryption, likely due to improved cache efficiency and reduced I/O operations.

The latency analysis reveals a distinctive bifurcation in behavior between uncompressed and compressed configurations. At compression level 0, encryption imposes substantial latency penalties with average latency increasing by 45.2% (108.1ms to 156.8ms) and maximum latency spiking by 37.8% (325.5ms to 448.6ms). However, this penalty effectively disappears when compression is applied. Across compression levels 1-9, encrypted and unencrypted configurations exhibit remarkably similar latency patterns, with an average difference of only 2.5% between corresponding compression levels.

Both encrypted and unencrypted configurations demonstrate consistent latency progression as compression intensity increases. Average latency rises from approximately 108-110ms at level 1 to 150-161ms at level 9, representing increases of 39.2% and 46.4% respectively. This progression follows a predictable pattern with a notable inflection point between levels 5 and 6, where latency jumps significantly from 122ms to 137.5ms (+12.7%), indicating the transition from speed-optimized to storage-optimized compression algorithms. Interestingly, maximum latency exhibits different behavior, remaining relatively stable or even decreasing slightly for unencrypted configurations (-12.7% from level 1 to 9) while showing minimal variation (+0.6%) for encrypted configurations. This suggests that compression helps stabilize worst-case latency spikes, particularly in unencrypted scenarios.

The results indicate that compression level 1 with encryption as the optimal security-performance balance, delivering 1.607 GiB/s throughput—a 59.7% improvement over the unoptimized baseline while maintaining security compliance. For applications prioritizing minimal latency with strong performance, compression level 2 with encryption offers an excellent alternative (1.587 GiB/s, 106.5ms average latency). Configurations beyond compression level 5 show diminishing throughput returns while providing only marginal storage benefits.

Effect of Number of Destination Files

In multi-tenant and distributed logging environments, log entries are typically written to numerous different files simultaneously, representing different services, users, or data categories. This benchmark evaluates how the system's performance scales with the number of distinct log files being written to concurrently. The evaluation is particularly important for testing the effectiveness of the LRU file cache mechanism, which manages file handles when the number of open files exceeds the configured maximum. When the number of target files exceeds the `maxOpenFiles` limit (configured at 512), the system must manage file handles by closing least recently used files and reopening them as needed. This introduces potential overhead from file operations while preventing resource exhaustion.

The following parameters were fixed for running this benchmark:

- **Producer Threads:** 16
- **Entries per Producer:** 500000
- **Total Data Size**: 31.15 GiB
- **Producer Batch Size:** 4096
- **Writer Threads:** 16
- **Writer Batch Size:** 2048
- **File Rotation:** disabled through 125 GiB segment size
- **Queue Capacity:** 2000000 (rounded internally to 2,097,152)
- **File Cache Size:** 512
- **Encryption & Compression:** both disabled

We evaluated 11 distinct numbers of destination files: 1; 16; 32; 64; 128; 256; 512; 1024; 2048; 4096; and 8192.

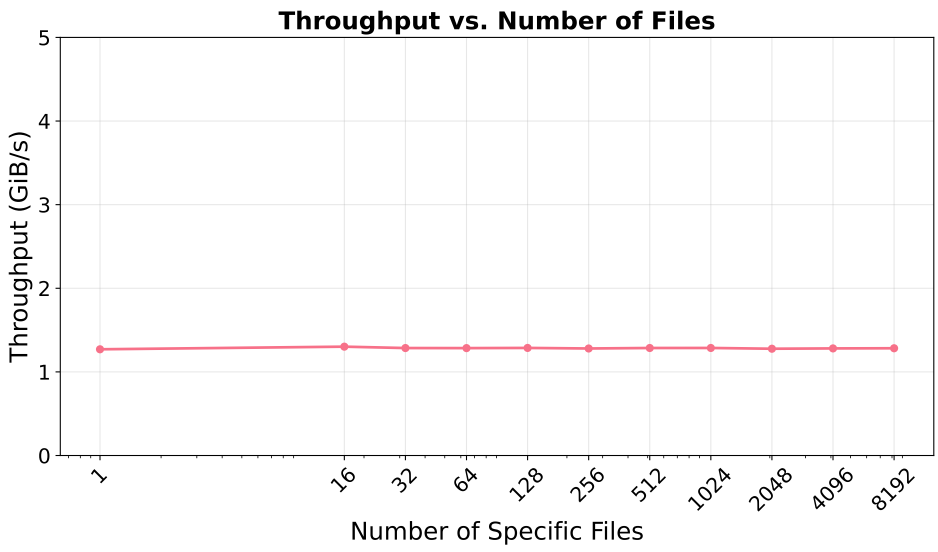

Figure 6.19: This figure shows the throughput in GiB per second for different number of destination log files

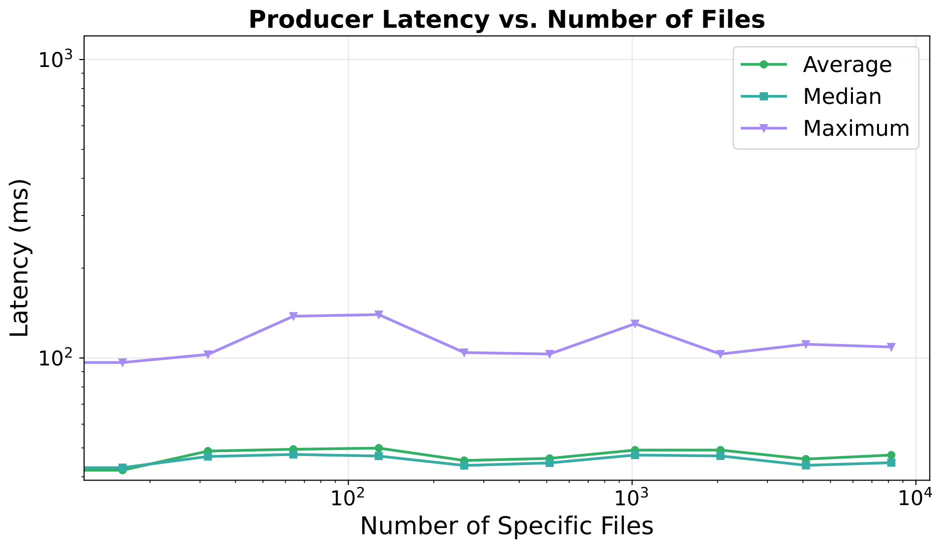

Figure 6.20: This figure shows the latency perceived by client threads (log producers) in milliseconds for different number of destination log files

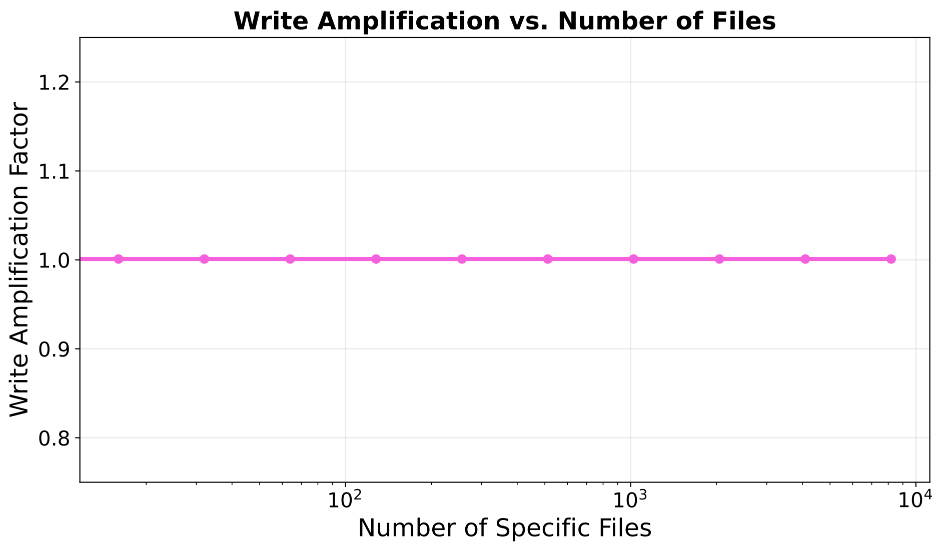

Figure 6.21: This figure shows the write amplification factor of the submitted log data for different number of destination log files

Figures 6.19, 6.20, 6.21 demonstrate strong scalability characteristics for number of destination files, with several noteworthy observations. Despite a 8192× increase in the number of target files—from 1 up to 8,192—the system maintains remarkably stable throughput, varying by only ±1.0%. This result highlights the effectiveness of the LRU cache, particularly under conditions where the number of files significantly exceeds the configured maximum of 512 open file descriptors. The consistent performance validates the cache design for high-file-count scenarios. Latency remains largely unaffected by the increase in file count, with average values holding steady at around 47 ms. This indicates that distributing log entries across a larger number of files does not introduce meaningful latency overhead from the perspective of the client. Additionally, the write amplification factor remains stable at around 1 across all tested configurations, since neither encryption nor compression were used in this benchmark.

These results validate the system's suitability for multi-tenant environments where log entries must be segregated across hundreds or thousands of different files. The LRU cache mechanism successfully prevents resource exhaustion while maintaining consistent performance, making the system practical for complex deployment scenarios with diverse file requirements. The linear scalability up to 8,192 files demonstrates robust architecture design capable of handling enterprise-scale workloads.

Main Benchmark

The main workload benchmark was designed to assess the logging system's peak sustained performance under a high-throughput production-like scenario. It serves as the primary performance indicator for the system and was executed using the best-performing parameters identified during the validation phase.

Benchmark Design:

- **System Configuration:**
  - 16 Writer Threads
  - Producer Batch Size: 2048 entries
  - Queue Capacity: 2,000,000 (rounded internally to 2,097,152)
  - Encryption: Enabled
  - Compression level: 1, very fast
  - Segment Size Limit: 500 MiB
  - Max Open Files: 512
- **Workload Characteristics:**
  - Producers: 16 asynchronous threads
  - Entries per Producer: 2,000,000
  - Average Entry Size: 4 KiB
  - Total Input Data Volume: 124.6 GiB
  - File Distribution: Batches targeted 1024 distinct log files in a round-robin manner

Each producer thread submitted pre-generated batches of log entries with pre-assigned destinations to simulate a large-scale, evenly distributed workload.

Results:

Metric	Value
Execution Time	59.95 seconds
Throughput (Entries)	∼533,711 entries/sec
Throughput (Data)	∼2.08 GiB/sec
Latency	Med.: 54.7ms, Avg.: 55.9ms, Max: 182.7ms
Write Amplification	0.109
Final Storage Footprint	∼13.62 GiB for ∼124.6 GiB input data

Table 6.2: This table displays the performance results summary of the logging system's main benchmark

The results shown in Table 6.2 demonstrate that the system, when run with optimized configuration and concurrency settings, is capable of sustaining decent log submission rates with minimal storage overhead and quite low latency. The observed write amplification was remarkably low due to the combined effects of efficient batching, compression, and minimal overhead from encryption metadata. This benchmark provides a reference performance ceiling for the system and validates that it can meet the demands of larger real-time log submission pipelines.

6.4 Research Question Evaluation

This section revisits the research questions introduced in Section 6.1 and evaluates them based on the collected benchmark data.

Throughput under heavy workloads and varying configurations

The benchmarking results show that the system sustains high throughput across a variety of configurations and workloads. In the main benchmark scenario, throughput reached approximately 533,711 entries per second (∼2.08 GiB/s), while specific parameter studies demonstrated throughput improvements through tuning writer batch sizes, queue capacities, and concurrency levels. These findings suggest that the system maintains stable and high performance under substantial load, provided configuration parameters are appropriately chosen.

Latency under various workloads and configurations

Measured latencies vary depending on system configuration, with average values typically between 50 ms and 160 ms across the tested scenarios. In configurations with limited queue capacity or small batch sizes, occasional latency spikes were observed, but these were reduced significantly when parameters were increased beyond specific thresholds. While no absolute latency target was defined, the observed values suggest behavior consistent with low-overhead append operations under most configurations.

Encryption and Compression effect on performance

The use of encryption alone introduces measurable overhead, reducing throughput and increasing latency in uncompressed scenarios. However, combining encryption with compression—particularly at lower compression levels—substantially improves throughput and mitigates latency increases. Compression reduces the volume of data to be encrypted, which in turn decreases the computational cost. Across all configurations, write amplification remained relatively low, and performance characteristics improved markedly with moderate compression.

Scalability

The system exhibits good scalability up to the point where thread counts approach the number of physical CPU cores. Benchmarks show near-linear throughput scaling in this region, with diminishing returns and increased contention beyond it. Both throughput and latency metrics indicate effective parallelization within the available hardware limits, and write amplification remained constant across scaling scenarios.

Reliability and Robustness

Benchmarks involving extreme queue pressure, high file counts, and frequent file rotation provide some indication of system behavior under stress. These tests showed that performance remains stable when appropriate buffer sizes and caching strategies are in place. In particular, the LRU-based file cache maintained consistent throughput even with thousands of concurrent log destinations, suggesting that the system's design accommodates high variability in workload structure.

Overall, the evaluation results indicate that the system exhibits performance characteristics and design trade-offs consistent with its intended use in high-throughput, security-conscious logging scenarios. While concrete deployment requirements may vary, the observed throughput, latency, and storage behavior across a broad range of configurations suggest that the system can serve as a viable basis for audit logging in demanding environments.

7 Related Work

7.1 High-Performance Logging Systems

Research on log management has produced numerous systems optimized for extreme write throughput and low latency, often by relaxing or delaying work that isn't immediately necessary for persistence. NanoLog is one example: a nanosecond-scale logging framework that minimizes runtime overhead by shifting log processing out of the critical execution path [32]. Instead of formatting and writing human-readable text at call time, NanoLog performs compile-time extraction of static log message components and defers expensive operations (like string formatting) to a post-execution stage [32]. The log entries recorded at runtime are compact binary records, which are later converted to text only when needed for analysis. This design yields tremendous speed-ups. NanoLog reports throughput up to tens of millions of log messages per second, with each log invocation adding only ∼8 nanoseconds of overhead [32]. By comparison, traditional logging libraries (such as Log4j2 or spdlog) achieve on the order of only a few million messages per second and incur latencies in the microsecond range [32]. The performance gains come with trade-offs: logs are not immediately human-readable and must be post-processed. Notably, NanoLog's aggressive optimizations omit any cryptographic features—entries are written in plaintext and assume a trusted storage foundation, meaning integrity or confidentiality guarantees are sacrificed for speed.

Another approach to high-throughput logging is to batch and sequence writes so as to maximize disk or network efficiency. Classic techniques like write-ahead logging inspired modern research that uses a dedicated log as the primary sink for all updates, which can later be propagated to their final destinations [28]. For example, Simha et al. built a disk-based logging system that first writes each update to a sequential log and only later applies it to the proper database or file location [28]. By converting random writes into linear append streams, their design can handle intense write loads that would overwhelm a conventional synchronous approach. In fact, their prototype delivered approximately 10^6 log records per second (each ∼256 bytes) with average write latencies under 1ms by carefully optimizing the disk access patterns. The key insight is that a well-tuned logger can serve as a high-speed buffer for the storage system: so long as appends are handled in large batches or continuous streams, even slow disks can sustain high throughput [28]. This philosophy extends to distributed settings as well. CORFU, for instance, is a cluster-level shared log that stripes append operations across many SSD-backed servers in parallel [1]. With 16–32 machines, CORFU can sustain around one million 4KB appends per second while keeping data replicated for fault tolerance [1]. Such systems highlight that logging can scale out in both throughput and capacity, serving as a fundamental building block for high-performance data stores and event streams.

The downside of these performance-first systems is that they don't treat logs as sensitive or auditable data. They typically lack encryption, access control, or tamper-evidence. NanoLog, for example, produces a binary log format that is extremely efficient but not designed to be cryptographically verifiable or secure against a malicious actor. Similarly, disk loggers and distributed log services assume the integrity of the log is guaranteed by the environment (or by simple checksums at best), and they do not protect the confidentiality of log contents. In a scenario with stringent security or compliance requirements—such as GDPR-regulated personal data processing—these systems would need substantial modifications. The challenge is that many optimizations used in high-speed loggers (custom binary formats, deferred processing, direct writes to storage buffers) are difficult to combine with on-the-fly encryption or per-entry authentication without adding latency. Thus, while high-performance logging frameworks demonstrate how to achieve significant gains in throughput via compile-time specialization [32], asynchronous batching, or parallel I/O, they leave open the question of how to simultaneously maintain strong security and privacy guarantees. Our work can be seen as bridging this gap, by drawing on the techniques of batched, non-blocking I/O and log structuring from these systems but adding transparent encryption and integrity checks suited for an auditing context.

7.2 Secure and Tamper-Evident Logging

Separate from the performance-focused work, there is a large amount of research on secure logging—making sure that once an event is logged, it can't be changed or deleted without being detected, even if an attacker later compromises the system. Early work by Schneier and Kelsey introduced the idea of forward-secure append-only logging on untrusted machines [25]. Their method employs evolving cryptographic keys and chained message authentication codes (MACs) so that log entries written prior to a security breach remain cryptographically sealed and cannot be undetectably modified or deleted by an attacker who gains control of the machine [25]. This concept treats logs as important security tools. Later systems built on these ideas to allow public verification of log integrity. Ma and Tsudik, for example, proposed Logcrypt and related schemes that combine forward-secure signatures with hashing chains to allow an external auditor to verify log integrity without sharing secret keys [14]. The evolution of these ideas led to constructs like Forward-Secure Sequential Aggregate (FssAgg) signatures and the work of Yavuz et al., who developed a family of schemes (BAF, FA-BAF, and LogFAS) that produce compact, tamper-evident logs suitable for resource-constrained environments [35]. In particular, LogFAS (Log Forward-secure Append-only Signature) achieves publicly verifiable audit logs with strong forward security while significantly reducing the overhead for verification [37]. It aggregates log entry signatures so that verifying a batch of L entries requires only a constant number of expensive operations (instead of linear in L) and each verifier needs to store only a single short public key [37]. This improves scalability when auditors must frequently check log integrity. The drawback, however, is that these cryptographic techniques introduce significant computation and complexity at both logging and verification time. Schemes like forward-secure signatures often assume a trusted component to periodically update keys or require careful key management to prevent any single compromise from exposing all past log entries. Moreover, purely cryptographic solutions do not by themselves guarantee availability — an attacker might not forge the log, but could still delete it entirely if there are not off-site backups or replication.

To improve tamper resistance and availability, researchers have also turned to blockchains and append-only ledgers for secure logging. Blockchains offer a way to create distributed, append-only logs protected by cryptographic hashes and consensus protocols. Modern systems often use permissioned blockchains run by multiple organizations to prevent any one party from hiding incidents [27]. In these systems, data is often encrypted before being added to the blockchain to maintain confidentiality. Since public blockchains are slow and expensive, many designs use a hybrid approach: storing full logs off-chain and writing only hashes or summaries to the blockchain [27]. This allows fast local logging while providing tamper-evidence through the chain. Systems like WedgeBlock follow this model, writing high-speed logs off-chain and periodically anchoring summaries on-chain [29]. This method balances performance and security but comes with complexity—some rely on special hardware and others add delay by waiting for blockchain confirmation [29]. Managing a blockchain is often too heavy for internal logs within a single organization. Our system borrows the idea of chaining log batches with hashes for tamper-evidence for future developments but avoids the complexity and slowdowns of using a full blockchain.

In short, secure logging methods offer many tools—like cryptographic operations, hash chains, and signatures—that can protect logs from tampering. These techniques shape how we design our integrity features. But many of these solutions are slow or complex.

Some add a lot of overhead, or require special verification tools. Our goal is to offer strong integrity without sacrificing speed or ease of use. We aim to merge the best of both worlds: the performance of fast logging systems with the trust guarantees of secure logs.

7.3 GDPR-Compliant Data Management Systems

The enforcement of the EU's General Data Protection Regulation has motivated research into systems that can prove compliance and manage personal data according to legal requirements. A naive approach to GDPR compliance might log every single access, change, or deletion of personal data for auditing, but this can be unacceptably slow. Recent studies highlight just how much performance can degrade if one retrofits strict auditing onto existing systems. For example, Shah et al. show that modifying an in-memory key-value store (Redis) to synchronously record every read and write operation in an audit log slowed throughput by a factor of 20× under a realistic workload [26]. The reason is that GDPR's mandate for comprehensive record-keeping (each "interaction" with personal data must be tracked) effectively turns read operations into write operations as well, adding expensive I/O on the critical path [26]. This highlights the challenge: adding accountability often hurts performance [26].

To solve these demands, researchers have proposed specialized architectures and techniques that feature compliance directly into data processing pipelines. One important direction is log pseudonymization and anonymization. GDPR encourages pseudonymization – replacing or masking personal identifiers – as a safeguard, since pseudonymized data, if sufficiently de-linked from identities, reduces privacy risk. Varanda et al. (2021) examine how this can be applied at the log level [30]. They point out that application logs frequently contain personal data (user IDs, IP addresses, emails, etc.), which in raw form would violate data-minimization principles and worsen breach impacts. Their system adds a pseudonymization step in the logging workflow: as logs are submitted, identifiable fields are transformed (for instance, replacing a username with a hash or random token) and the real identities are kept in a separate, access-restricted mapping store [30]. This way, the main log can be stored and analyzed without directly exposing personal information, yet if needed (for fulfilling a legal access request or debugging), the pseudonyms can be resolved by those with appropriate privileges. An interesting result from Varanda's experiments is that doing this transformation early, at submission time, has performance benefits in later processing stages. Although it incurs a slight delay on each log write (to compute hashes or look up tokens), it means that all subsequent processing and querying on the log can happen on sanitized data, which is faster than constantly having to filter or encrypt/decrypt sensitive fields on the fly [30]. In a prototype built on the Elastic Stack (a popular open-source log management suite), they achieved GDPR compliance through pseudonymization with only minimal overhead, and even improved query performance for certain analytics because the data indexed in Elasticsearch was uniformly pseudonymized and smaller in size [30]. In a follow-up work, they integrated these ideas into an open-source Security Information and Event Management solution, demonstrating that an enterprise can handle large volumes of security events in a GDPR-compliant manner by combining off-the-shelf tools (Elastic, Logstash, etc.) with custom pseudonymization plugins [30]. The system maintained good scalability and performance, showing that compliance measures like masking can be layered onto log infrastructure without severely violating operational requirements [30]. However, such a solution, while practical, is still quite specific: it assumes a centralized log pipeline and focuses on privacy (protecting identities in logs) more than on proving immutability or tamper-evidence of the logs themselves.

Another category of GDPR-oriented research looks at data provenance and deletion. One of GDPR's main principles is the "right to be forgotten", which implies that systems must be able to delete or disassociate personal data on request [7], posing a challenge for traditional append-only logs. Some recent works explore privacy-aware logging. For instance, techniques involving encrypting each log entry with an individual key have been proposed so that if a particular user must be removed, the corresponding log entries can be made unreadable (by destroying keys) without actually deleting entries and breaking the log's integrity chain [39]. While elegant in concept, these methods can introduce significant key-management complexity and performance cost, and they raise questions about what it means for a log to be "immutable." Indeed, a fully tamper-proof log seems at odds with a regulation that demands erasure of data; researchers note that a balance must be struck via encryption or abstraction layers that allow logical deletion without physical deletion [39]. In practice, many systems adopt a compromise by keeping records of actions (e.g., "User X's data was deleted") but not the personal data itself, making logs auditable without exposing sensitive information.

In summary, GDPR-compliant systems must combine privacy, security, and auditability with performance. Solutions range from simple filtering to advanced cryptographic protocols. The key insight is that compliance must be built into the system—it can't be added later without hurting speed [26]. Our work follows this principle. We combine techniques from fast logging (like batching and non-blocking writes) with secure logging and GDPR practices (like encryption and hash chaining) into one system. We aim to create a general-purpose logging tool that is fast, secure, and compliant, without the complexity of blockchain or the performance penalties of some secure logs.

We improve on pure performance systems by adding built-in integrity and encryption, and make secure logging easy to use.

8 Discussion & Future Work

While the implemented logging system demonstrates the feasibility of achieving secure, high-throughput audit logging under GDPR constraints, it **remains a prototype** and lacks several critical features required for production deployment. One of the most prominent limitations is the current lack of **export functionality**. Although the system is capable of efficiently ingesting, serializing, compressing, and encrypting log data, it does not yet support the inverse operations necessary for auditing. In a production setting, a complete audit trail must be retrievable in a human-understandable form. This includes the ability to decrypt, verify, decompress, and deserialize stored data into log segments. Adding such capabilities would be essential for responding to regulatory audits or subject access requests, though it would introduce some non-trivial complexity around efficient file scanning.

Another key limitation lies in the **encryption** mechanism. At present, the system uses a static, hardcoded key for all encryption tasks, which is acceptable for demonstration but absolutely insufficient for real-world use. Any production-grade deployment must incorporate a **robust key management** solution, potentially integrating with hardware security modules or trusted key vaults to ensure confidentiality and secure access control. Introducing proper key management will, however, increase the operational complexity and may slightly affect throughput due to additional key retrieval and validation steps during batch encryption.

In addition to the encryption key, the prototype also employs a placeholder **initialization vector (IV)** and does not persist this IV alongside the encrypted payload. For authenticated encryption modes such as AES-GCM, the IV must be unique for each encryption operation to maintain cryptographic security. A feasible approach for further developments could be to derive the IV from the monotonically increasing byte offset of the target file in combination with a hash of the filename. This method would guarantee uniqueness without the need to store the IV separately, simplifying export logic and preserving encryption strength. However, the computational overhead of hashing would slightly impact writer thread performance.

The current **tamperproofing** mechanism also presents limitations. While the current use of AES-GCM ensures per-batch integrity, there is no mechanism in place to **detect reordering, omission, or deletion** of batches. The implementation of a strict hash chain—where each log entry includes a cryptographic hash of the previous one—is not feasible under the current design due to the system's focus on high concurrency and throughput. Such chains require sequential write ordering, which conflicts with the parallelized, lock-free architecture. A potential compromise could involve introducing a global, atomic, monotonically increasing counter that is included in the encryption process. This counter, if properly managed, could serve as a lightweight anchor to verify the continuity of log segments without imposing serial bottlenecks. Still, the trade-off here would be added complexity and contention around atomic counter access.

In addition to detecting tampering within files, a production system must also guard against the **deletion of entire log files**, which could silently erase critical segments of the audit trail. Currently, the system has no mechanism to track the presence or integrity of full segment files once they have been rotated and closed. To address this, a possible enhancement could involve maintaining a cryptographically signed index or manifest of all expected log files, allowing the system to detect missing segments during verification or export. This measure would further strengthen audit guarantees but would require careful coordination with the file management logic and introduce overhead during rotation and verification operations.

Finally, the current **storage flushing policy** poses a risk to data durability. At present, log data is flushed to disk only upon segment rotation or when a file is evicted from the file descriptor cache. This behavior can lead to prolonged periods where large amounts of data remain buffered in memory and are vulnerable to loss in the event of a system crash. To mitigate this, **more aggressive flushing strategies** would need to be introduced, such as periodic syncs or thresholds based on data size or batch count. While this would enhance data safety, it would also increase I/O pressure and may reduce overall throughput, especially in write-intensive deployments.

In summary, while the presented prototype lays a solid foundation for secure and performant GDPR-compliant logging, a number of enhancements are required to ensure production readiness. Each proposed improvement entails trade-offs between security, reliability, and performance, and must be carefully balanced depending on the deployment context and threat model.

9 Summary and Conclusion

This bachelor's thesis addressed the critical challenge of implementing a high-performance, secure, and tamper-evident logging system for GDPR-compliant key-value store environments. Through the development of a lock-free, multi-threaded logging architecture, this work demonstrates that it is possible to achieve decent throughput and strong security guarantees without sacrificing the performance characteristics essential for larger deployments.

The system's design centers on asynchronous batch processing, where dedicated writer threads consume log entries from a concurrent buffer queue, apply serialization, compression, and authenticated encryption (AES-GCM), before atomically appending data to immutable, segmented storage files. This architecture successfully decouples logging overhead from client operations while maintaining strict append-only semantics and cryptographic integrity guarantees.

The empirical evaluation revealed several significant findings that advance our understanding of this system's high-performance secure logging capabilities. Most notably, the system achieves a log submission rate of approximately 2.08 GiB per second with a low write amplification factor of 0.109 under heavy concurrent workload, demonstrating solid performance and efficiency in minimizing storage footprint. Additionally, the system exhibited strong scalability characteristics, achieving nearly linear speedup up to 16 writer threads and maintaining consistent performance across diverse patterns of workloads.

The broader impact of this work lies in its potential to serve as a foundational component for GDPR-compliant infrastructures. By architecting the system as a standalone, modular component that integrates seamlessly into existing environments, organizations can establish secure, verifiable audit trails without intrusive modifications to their storage systems. This capability directly addresses the GDPR's accountability principle, enabling confident responses to regulatory audits and subject access requests while preserving the low-latency characteristics essential for high-performance applications.

Overall, this thesis establishes a robust foundation for future advancements in GDPR-compliant data management, proving that the seemingly conflicting requirements of regulatory compliance, security, and high performance can be successfully reconciled through careful architectural design and implementation. The insights gained from this work contribute to the growing body of knowledge on privacy-preserving systems and provide a practical pathway for organizations seeking to balance legal compliance with operational efficiency in high-performance distributed environments.

The complete source code implementation is available at https://github.com/chriskari/bachelor-thesis. While the current prototype demonstrates the feasibility and effectiveness of the proposed approach, future work should focus on implementing export functionality, robust key management, enhanced tamper-proofing mechanisms, and more aggressive data durability policies to achieve full production readiness.

Abbreviations

GDPR	General Data Protection Regulation
API	Application Programming Interface
AES-GCM	Advanced Encryption Standard - Galois/Counter Mode
IV	Initialization Vector
MAC	Message Authentication Code
LRU	Last Recently Used
CAS	Compare-and-Swap

List of Figures

List of Tables

Bibliography

[1] M. Balakrishnan, D. Malkhi, J. D. Davis, V. Prabhakaran, M. Wei, and T. Wobber. "CORFU: A Distributed Shared Log." In: *ACM Transactions on Computer Systems* 31.4 (Dec. 2013), 10:1–10:24. DOI: 10.1145/2535930.

[2] G. Cenikj, D. Jankovikj, and O. Dimitriov. "The challenges of key-value stores." In: May 2020.

[3] C. Desrochers. *A fast general-purpose lock-free queue for C++*. https://moodycamel.com/blog/2014/a-fast-general-purpose-lock-free-queue-for-c++. Accessed: 2025-06-15. 2014.

[4] C. Desrochers. *concurrentqueue: A fast multi-producer, multi-consumer lock-free concurrent queue for C++*. https://github.com/cameron314/concurrentqueue. Accessed: 2025-06-15.

[5] European Parliament and Council of the European Union. *General Data Protection Regulation (GDPR) – Article 4: Definitions*. https://eur-lex.europa.eu/eli/reg/2016/679/oj. Regulation (EU) 2016/679 of the European Parliament and of the Council of 27 April 2016. 2016.

[6] European Parliament and Council of the European Union. *General Data Protection Regulation (GDPR) – Article 5: Principles relating to processing of personal data*. https://eur-lex.europa.eu/eli/reg/2016/679/oj. Regulation (EU) 2016/679 of the European Parliament and of the Council of 27 April 2016. 2016.

[7] European Parliament and Council of the European Union. *General Data Protection Regulation (GDPR) – Articles 12–21: Rights of the data subject*. https://eur-lex.europa.eu/eli/reg/2016/679/oj. Regulation (EU) 2016/679 of the European Parliament and of the Council of 27 April 2016, Articles 12-21. 2016.

[8] European Parliament and Council of the European Union. *General Data Protection Regulation (GDPR) – Articles 31 & 32: Cooperation with the supervisory authority & Security of processing*. https://eur-lex.europa.eu/eli/reg/2016/679/oj. Regulation (EU) 2016/679 of the European Parliament and of the Council of 27 April 2016, Articles 31 and 32. 2016.

[9] O. G. Fakeyede, P. A. Okeleke, A. O. Hassan, U. Iwuanyanwu, O. R. Adaramodu, and O. O. Oyewole. "Navigating Data Privacy Through IT Audits: GDPR, CCPA, and Beyond." In: *International Journal of Research in Engineering and Science (IJRES)* 11.11 (Nov. 2023). ISSN (Print): 2320-9356, pp. 184–192. ISSN: 2320-9364.

[10] J.-l. Gailly and M. Adler. *zlib Technical Details*. Accessed: 2025-06-15.

[11] V. Geetha, N. Udaya Ponni, T. Devagi, and M. Nandhini Priya. "A Tamper-Proof Log Architecture for Cloud Forensics." In: *Advances in Natural and Applied Sciences* 9.6 (2015). Special Issue, pp. 722–727. ISSN: 1995-0772.

[12] D. Gonçalves-Ferreira, M. Leite, C. Santos-Pereira, M. E. Correia, L. Antunes, and R. Cruz-Correia. "HS.Register – An Audit-Trail Tool to Respond to the General Data Protection Regulation (GDPR)." In: *Building Continents of Knowledge in Oceans of Data: The Future of Co-Created eHealth*. Ed. by A. Ugon et al. Open Access under CC BY-NC 4.0. Published by the European Federation for Medical Informatics (EFMI). Amsterdam: IOS Press, 2018, pp. 81–85. DOI: `10.3233/978-1-61499-852-5-81`.

[13] Google. *Protocol Buffers*. `https://protobuf.dev/`. Accessed: 2025-06-15.

[14] J. Holt and K. Seamons. "Logcrypt: Forward Security and Public Verification for Secure Audit Logs." In: *IACR Cryptology ePrint Archive* 2005 (Jan. 2005), p. 2. DOI: `10.1145/1151828.1151852`.

[15] S. Idreos and M. Callaghan. "Key-Value Storage Engines." In: *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data*. SIGMOD '20. Portland, OR, USA: Association for Computing Machinery, 2020, pp. 2667–2672. ISBN: 9781450367356. DOI: `10.1145/3318464.3383133`.

[16] S. Idreos and M. Callaghan. "Key-Value Storage Engines." In: *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data*. SIGMOD '20. Portland, OR, USA: Association for Computing Machinery, 2020, pp. 2667–2672. ISBN: 9781450367356. DOI: `10.1145/3318464.3383133`.

[17] Impanix. *8 Requirements For Logging & Record Monitoring In GDPR*. Online; accessed 15-June-2025.

[18] H. Li, L. Yu, and W. He. "The Impact of GDPR on Global Technology Development." In: *Journal of Global Information Technology Management* 22.1 (2019), pp. 1–6. DOI: `10.1080/1097198X.2019.1569186`.

[19] D. Ma and G. Tsudik. "A New Approach to Secure Logging." In: (). Computer Science Department.

[20] D. Ma and G. Tsudik. "A new approach to secure logging." In: *ACM Trans. Storage* 5.1 (Mar. 2009). ISSN: 1553-3077. DOI: `10.1145/1502777.1502779`.

[21] National Institute of Standards and Technology. *Guide to Computer Security Log Management*. NIST Special Publication 800-92. Gaithersburg, MD: National Institute of Standards and Technology, Sept. 2006.

[22] National Institute of Standards and Technology. *Protecting Controlled Unclassified Information in Nonfederal Systems and Organizations*. NIST Special Publication 800-171 Revision 2. Gaithersburg, MD: National Institute of Standards and Technology, Feb. 2020.

[23] M. Patchappen, Y. M. Yassin, and E. K. Karuppiah. "Batch processing of multivariant AES cipher with GPU." In: *2015 Second International Conference on Computing Technology and Information Management (ICCTIM)*. 2015, pp. 32–36. DOI: 10.1109/ICCTIM.2015.7224589.

[24] B. Schneier and J. Kelsey. "Cryptographic Support for Secure Logs on Untrusted Machines." In: *Proceedings of the 7th USENIX Security Symposium*. USENIX Association. San Antonio, Texas: USENIX Press, Jan. 1998, pp. 53–62.

[25] B. Schneier and J. Kelsey. "Cryptographic Support for Secure Logs on Untrusted Machines." In: *The Seventh USENIX Security Symposium Proceedings*. San Antonio, Texas: USENIX Press, Jan. 1998, pp. 53–62.

[26] A. Shah, V. Banakar, S. Shastri, M. Wasserman, and V. Chidambaram. "Analyzing the Impact of GDPR on Storage Systems." In: *11th USENIX Workshop on Hot Topics in Storage and File Systems (HotStorage 19)*. Renton, WA: USENIX Association, July 2019.

[27] L. Shekhtman and E. Waisbard. "EngraveChain: A Blockchain-Based Tamper-Proof Distributed Log System." In: *Future Internet* 13.6 (2021). ISSN: 1999-5903. DOI: 10.3390/fi13060143.

[28] D. Simha, G. Rajagopalan, P. Bose, and t.-c. Chiueh. "High-throughput low-latency fine-grained disk logging." In: vol. 41. June 2013. DOI: 10.1145/2465529.2465552.

[29] A. Singh, Y. Zhou, S. Mehrotra, M. Sadoghi, S. Sharma, and F. Nawab. "Wedge-Block: An Off-Chain Secure Logging Platform for Blockchain Applications." In: *Proceedings of the 26th International Conference on Extending Database Technology (EDBT)*. EDBT. Ioannina, Greece: OpenProceedings.org, Mar. 2023, pp. 526–539. ISBN: 978-3-89318-092-9. DOI: 10.48786/edbt.2023.45.

[30] A. Varanda, L. Santos, R. L. Costa, A. Oliveira, and C. Rabadão. "The General Data Protection Regulation and Log Pseudonymization." In: May 2021, pp. 479–490. ISBN: 978-3-030-75077-0. DOI: 10.1007/978-3-030-75078-7_48.

[31] W. Wei, L. Na, Z. Lei, L. Fang, C. Hao, Y. Xiuying, H. Lei, Z. Min, W. Gang, Z. Jie, X. Jing, S. Tao, M. Li, Z. Qiang, H. Jun, G. Wei, H. Yong, G. Yuan, L. Dan, Z. Yi, and S. Li. *An Extensive Study on Text Serialization Formats and Methods*. 2025. arXiv: `2505.13478 [cs.PL]`.

[32] S. Yang. "NanoLog: A Nanosecond Scale Logging System." PhD dissertation. Stanford University, Mar. 2020.

[33] S. Yang, S. J. Park, and J. Ousterhout. "NanoLog: A Nanosecond Scale Logging System." In: *2018 USENIX Annual Technical Conference (USENIX ATC 18)*. Boston, MA: USENIX Association, 2018, pp. 335–350. ISBN: 978-1-939133-01-4.

[34] K. Yao, H. Li, W. Shang, and A. E. Hassan. "A study of the performance of general compressors on log files." In: *Empirical Software Engineering* 25 (2020), pp. 3043–3085. DOI: `10.1007/s10664-020-09822-x`.

[35] A. Yavuz and P. Ning. "BAF: An Efficient Publicly Verifiable Secure Audit Logging Scheme for Distributed Systems." In: Dec. 2009, pp. 219–228. DOI: `10.1109/ACSAC.2009.28`.

[36] A. A. Yavuz and P. Ning. "BAF: An Efficient Publicly Verifiable Secure Audit Logging Scheme for Distributed Systems." In: *2009 Annual Computer Security Applications Conference*. IEEE. Honolulu, HI, USA: IEEE, Dec. 2009, pp. 219–228. ISBN: 978-0-7695-3919-5. DOI: `10.1109/ACSAC.2009.28`.

[37] A. A. Yavuz, P. Ning, and M. K. Reiter. "Efficient, Compromise Resilient and Append-Only Cryptographic Schemes for Secure Audit Logging." In: *Financial Cryptography and Data Security*. Ed. by A. D. Keromytis. Berlin, Heidelberg: Springer Berlin Heidelberg, 2012, pp. 148–163. ISBN: 978-3-642-32946-3.

[38] R. N. Zaeem and K. S. Barber. "The Effect of the GDPR on Privacy Policies: Recent Progress and Future Promise." In: *ACM Trans. Manage. Inf. Syst.* 12.1 (Dec. 2020). ISSN: 2158-656X. DOI: `10.1145/3389685`.

[39] A. Zafar. "Reconciling blockchain technology and data protection laws: regulatory challenges, technical solutions, and practical pathways." In: *Journal of Cybersecurity* 11.1 (Feb. 2025), tyaf002. ISSN: 2057-2085. DOI: `10.1093/cybsec/tyaf002`. eprint: `https://academic.oup.com/cybersecurity/article-pdf/11/1/tyaf002/61986970/tyaf002.pdf`.

[40] W. Zhao, I. M. Aldyaflah, P. Gangwani, S. Joshi, H. Upadhyay, and L. Lagos. "A Blockchain-Facilitated Secure Sensing Data Processing and Logging System." In: *IEEE Access* 11 (2023), pp. 21712–21728. DOI: `10.1109/ACCESS.2023.3252030`.

[41] V. Zieglmeier and G. Loyola Daiqui. "GDPR-Compliant Use of Blockchain for Secure Usage Logs." In: *Proceedings of the 25th International Conference on Evaluation and Assessment in Software Engineering*. EASE '21. Trondheim, Norway: ACM, June 2021, pp. 313–320. ISBN: 978-1-4503-9053-8. DOI: 10.1145/3463274.3463349.