# Tamper-Evident Audit Logging System: Design Document

## Overview

This system provides a cryptographically-secure, immutable audit trail for compliance, capturing who did what, when, and to which resource. Its key architectural challenge is balancing the need for absolute data integrity and efficient querying of a continuously growing, append-only dataset. It solves this by using a hash chain for tamper detection, immutable storage segments, and indexed time-series queries.

> This guide is meant to help you understand the big picture before diving into each milestone. Refer back to it whenever you need context on how components connect.

> **Milestone(s):** Milestone 1 (foundational concepts), Milestone 2 (core integrity problem)

## Context and Problem Statement

In the digital realm, every action leaves a trace, but for organizations bound by regulations like SOC 2, GDPR, HIPAA, or PCI-DSS, simply having traces is insufficient. These traces must form an **unassailable, chronological record**—a definitive answer to the questions "who did what, when, and to which resource?" This is the domain of the **audit log**. Unlike diagnostic or application logs used for debugging, audit logs are a legal and compliance instrument. They serve as the primary evidence during security investigations, forensic analysis, and regulatory audits. A breach in their integrity or a gap in their record can lead to severe compliance failures, legal liability, and loss of stakeholder trust.

The fundamental challenge is creating a logging system that is both **immutable and efficiently queryable** at scale. Immutability ensures that once an event is recorded, it cannot be altered or deleted, providing a trustworthy historical record. Efficient queryability allows security teams and auditors to quickly sift through potentially terabytes of historical data to find relevant events. These two requirements are often in tension: the most secure storage methods (like write-once media) are poor for random access, while high-performance databases typically allow data modification.

### Mental Model: The Ship's Tamper-Proof Logbook

Imagine a 17th-century ship's logbook. The captain records every significant event—course changes, cargo loaded, crew disciplinary actions, and sightings of other vessels. This log is not just a diary; it's a legal document used to verify the ship's journey for the vessel's owners and insurers.

1. **Append-Only:** The captain writes entries chronologically at the end of the book. He never goes back to erase or scribble over a previous entry. If a mistake is made, he draws a single line through it and writes the correction with a new date, leaving the original error visible. This is the **append-only** principle.
2. **Tamper-Evident Binding:** The logbook is bound with a wax seal. If someone tries to remove or replace a page, the binding and seal will show damage. In our digital system, the **hash chain** serves as this cryptographic wax seal.
3. **Structured Entries:** Each log entry follows a consistent format: *Date*, *Time*, *Officer on Duty*, *Event Description*, *Action Taken*. This structure is our **audit event schema**.
4. **Independent Verification:** Upon return to port, the ship's owners can verify the log's integrity by checking the seal and ensuring the sequence of entries makes logical sense (e.g., the ship couldn't be in two places at once). Similarly, our system allows auditors to **cryptographically verify** the entire log's integrity from any starting point.

This mental model grounds the abstract requirements of an audit log system in a tangible, historical practice. Our system is the digital evolution of this centuries-old practice of maintaining a verifiable, chronological record.

### The Core Problem: Verifiable Integrity at Scale

Building a digital "ship's logbook" introduces specific, interconnected technical challenges that go beyond simple logging libraries or database tables.

**1. The Integrity-Query Tension:**

- **Integrity Requirement:** To be admissible as evidence, the log must provide cryptographic proof that it has not been tampered with. This requires linking each entry to the previous one, creating a verifiable chain. Any attempt to modify, delete, or insert a past entry must be detectable.
- **Query Requirement:** Auditors need to ask complex, ad-hoc questions: "Show all events for user `alice@company.com` between January 1st and 15th that involved the `billing` module and failed." This requires efficient indexing and filtering on multiple dimensions (actor, time, resource, outcome) across a massive, ever-growing dataset.
- **The Tension:** The strongest integrity mechanisms (like a simple linear hash chain) are sequential and resist random access. Optimizing for fast, multi-dimensional queries (using B-tree indexes, for example) typically involves data structures that are mutable during rebalancing or compaction. The core architectural problem is designing a storage layer that **preserves a cryptographically verifiable chain of integrity while supporting indexed queries** without compromising either.

**2. The Scale of "Write-Once, Read-Never (Until an Audit)":** Audit logs are **write-heavy, time-series data**. Events stream in continuously at a high volume (millions per day in a large system). However, they are rarely read in their entirety—except during a compliance audit or security incident, at which point fast, accurate reads are *critical*. This usage pattern is different from operational databases (read/write mix) or data warehouses (complex analytical reads). The system must be cost-effective to store years of data (for retention policies) while keeping it "warm" enough for immediate querying.

**3. Why Simple Database Tables Fail:** Storing audit events in a standard relational database table ( `audit_events` ) seems intuitive but is fraught with problems:

| Problem | Consequence |
| --- | --- |
| **Mutable by default:** UPDATE and DELETE statements are allowed. | No inherent immutability. A rogue admin or compromised application can alter history. |
| **No built-in tamper evidence:** The database provides ACID guarantees for *consistency*, not *cryptographic integrity*. | You cannot prove to a third-party auditor that a record hasn't been altered via direct database access. |
| **Index overhead on writes:** Indexes on `actor_id` , `resource` , `timestamp` slow down the primary insert operation. | This creates a performance bottleneck at the ingestion point, potentially dropping audit events under load. |
| **Expensive full-table scans:** Queries without perfect index hits scan entire tables, which becomes prohibitive as the table grows to billions of rows. | Audit queries become too slow for interactive investigation. |
| **Vacuum/Compaction can erase history:** To reclaim space, some databases physically delete old "soft-deleted" rows, breaking immutability. | Violates data retention policies and destroys evidence. |

> **Key Insight:** A general-purpose database is optimized for *mutable* data with a balanced read/write workload. An audit log system must be optimized for *immutable*, *append-only* data with a extreme write bias and occasional complex, time-bound reads.

**4. Compliance Drivers (SOC 2 & GDPR):**

- **SOC 2 CC6.1:** *"The entity implements logical access security software, infrastructure, and architectures over protected information assets to protect them from security events to meet the entity's objectives."* This requires logging and monitoring of access to protected assets.
- **GDPR Article 30:** *"Records of processing activities... shall be in writing, including in electronic form."* While not explicitly mandating immutability, the *integrity and confidentiality* principle (Article 5(1)(f)) and the *right to erasure* (Article 17) create a conflict: logs needed for compliance may contain personal data that must be deleted. This forces design choices around **pseudonymization** and **tokenization** within the log itself (a concept addressed in Milestone 1). These regulations don't prescribe technology but set the bar for what the audit log must achieve: **completeness, accuracy, confidentiality, and integrity.**

## Existing Approaches and Trade-offs

Several architectural patterns exist for building audit trails, each with different trade-offs between integrity, query performance, complexity, and cost. The following table evaluates three common approaches, leading to the rationale for our chosen hybrid design.

| Approach | Description | Pros | Cons | Why It's Not Sufficient Alone |
|---|---|---|---|---|
| **1. Relational Database Table** | Store events in a table like `audit_events` with indexes on `timestamp`, `actor_id`, etc. | • **Familiar:** Well-understood technology.<br>• **Powerful Querying:** SQL allows complex joins, aggregations.<br>• **ACID Transactions:** Ensures events aren't lost on write. | • **Mutable:** Requires strict access controls to prevent tampering.<br>• **No Tamper Evidence:** Cannot cryptographically prove log integrity.<br>• **Write Scalability Issues:** Heavy indexing can bottleneck high-volume ingestion.<br>• **Costly at Scale:** Storing and indexing years of high-volume logs in an OLTP database is expensive. | Lacks **cryptographic integrity guarantees**, which are non-negotiable for a compliance-grade audit log. Performance degrades under the specific write-heavy, append-only workload. |
| **2. Specialized Time-Series Database (TSDB)** | Use a database optimized for time-series data (e.g., InfluxDB, TimescaleDB). | • **Excellent Write Performance:** Highly optimized for ingesting time-ordered data.<br>• **Efficient Time-Range Queries:** Native support for time-downsampling, aggregation.<br>• **Compression:** Built-in compression reduces storage costs. | • **Limited Tamper Evidence:** Most TSDBs are not designed with immutable, hash-chained data as a core feature.<br>• **Less Flexible Filtering:** Secondary indexing on non-time dimensions (actor, resource) is often less mature than in RDBMS.<br>• **Vendor Lock-in Risk:** Proprietary query languages and storage formats. | While an excellent fit for the *time-series nature* of the data, it does not natively solve the **cryptographic integrity** problem. We would need to build the hash chain and integrity verification layers on top. |
| **3. Blockchain-Inspired Append-Only Log** | Implement a simple hash chain or Merkle tree in flat files, similar to a blockchain ledger. | • **Strong Integrity Guarantees:** Tampering with any record invalidates the entire chain.<br>• **Simple & Predictable:** Append-only files are straightforward to manage and back up.<br>• **Low Cost:** Uses cheap object storage (e.g., S3) for immutable files. | • **Poor Query Performance:** Finding a specific event requires a linear scan from the beginning or a pre-built index.<br>• **Complex to Index:** Building secondary indexes requires a separate, eventually consistent process.<br>• **Operational Overhead:** Requires custom tooling for rotation, verification, and cleanup. | Provides perfect **integrity** but terrible **queryability**. It cannot meet the requirement for sub-second filtered queries over large datasets. |

**Decision: Hybrid Architecture: Hash-Chained Segments with Indexed Querying**

- **Context:** We need both cryptographically verifiable integrity and efficient querying for compliance audits. The existing approaches each excel in one area but fail in the other.
- **Options Considered:**
  1. **Database-Only:** Rely on database permissions and frequent hash-based snapshots for integrity.
  2. **TSDB-Only:** Use a time-series database and implement an external integrity verification service.
  3. **Hash-Chain-Only:** Use an immutable hash chain and accept slow, linear-scan queries.
  4. **Hybrid (Chosen):** Use an immutable, hash-chained Write-Ahead Log (WAL) as the **system of record** for integrity, and periodically build indexed, query-optimized **read-optimized segments** from the WAL.
- **Decision:** Adopt a hybrid architecture. All events are first written to an **immutable, hash-chained Write-Ahead Log**. A separate process seals this log into read-only **segments** and builds auxiliary indexes (on timestamp, actor, resource) for those segments. The Query Engine uses these indexes for fast lookups, but any result can be **cryptographically verified** against the original hash chain in the segment.
- **Rationale:** This separates concerns. The WAL handles the write path with maximum integrity and durability. The segment indexing handles the read path with performance. The hash chain remains the single source of truth, and the indexes are merely derived, verifiable views. This pattern is observed in systems like Apache Kafka (log segments with offsets) and tamper-evident logging research.
- **Consequences:**
  - **Enables:** Strong integrity, efficient queries, and the ability to use different storage backends for active vs. archived segments.
  - **Introduces:** Eventual consistency between the WAL and query indexes (milliseconds to seconds latency). Increased operational complexity with two data structures to manage.
  - **Requires:** Careful design to ensure the hash chain is preserved across segment boundaries and that index corruption does not affect the ability to verify the primary log.

This hybrid approach directly addresses the "Verifiable Integrity at Scale" problem. It acknowledges that no single off-the-shelf technology provides both properties perfectly and instead composes them in a layered architecture, which is the subject of the following High-Level Architecture section.

---

**Summary:** The need for an immutable audit log stems from stringent compliance requirements. The core technical problem is balancing **cryptographic integrity** (achieved through append-only hash chains) with **efficient queryability** (achieved through indexing). Existing solutions like RDBMS or TSDBs alone are insufficient because they optimize for one aspect at the expense of the other. Our chosen path is a hybrid model that uses a hash-chained Write-Ahead Log as the immutable system of record and builds indexed, query-optimized segments as a derived layer, thereby satisfying both critical requirements.

## Goals and Non-Goals

**Milestone(s):** Milestone 1, Milestone 2, Milestone 3

This section establishes the concrete boundaries of our audit logging system. Defining clear goals ensures we build what's necessary for compliance without scope creep, while explicit non-goals prevent wasted effort on features that don't directly address our core problem of **verifiable integrity at scale**.

Think of this as a contract between the system architects and the compliance officers: the goals represent what the system *will* deliver to meet SOC 2 and GDPR requirements, while the non-goals are features we *won't* build, either because they're unnecessary for compliance or because they would distract from our primary mission of creating a tamper-evident, queryable audit trail.

### Goals: What the System Must Achieve

These five goals represent the mandatory capabilities required to satisfy compliance requirements and operational needs. Each goal maps directly to one or more acceptance criteria from our milestones.

| Goal | Milestone Alignment | Compliance Driver | Key Success Metric |
|---|---|---|---|
| **Immutable Storage** | Milestone 2 | SOC 2 (PI-1.4), GDPR Art. 30 | Zero modifications/deletions of historical records |
| **Hash Chain Integrity** | Milestone 2 | SOC 2 (CC-7.1), GDPR Art. 32 | Tamper detection with cryptographic certainty |
| **Structured Event Model** | Milestone 1 | SOC 2 (CC-7.1), GDPR Art. 5 | 100% of events contain required fields |
| **Efficient Time-Range Queries** | Milestone 3 | SOC 2 (CC-7.1) | Sub-second query response for 30-day windows |
| **Compliance Export Formats** | Milestone 3 | SOC 2 (CC-7.1), GDPR Art. 15 | Correctly formatted reports for auditors |

### Goal 1: Immutable Storage

**Mental Model**: The system's storage must function like a **bank safe deposit box ledger**—once an entry is written in the bound ledger book, you can add new entries on the next line, but you cannot erase, white-out, or insert entries between existing lines. The physical binding of the ledger enforces immutability.

**Detailed Explanation**: The audit log must guarantee that once an event is recorded, it can never be modified or deleted, even by administrators with system access. This **append-only** property is fundamental for trust: if auditors suspect logs can be altered, the entire compliance effort fails. The system achieves this through:

- **Write-ahead log (WAL)** semantics where events are appended sequentially to files
- **File permissions** that prevent modification of sealed segment files
- **Storage layer APIs** that lack update/delete operations
- **Cryptographic signatures** that would break if data is altered

**Rationale**: Immutability addresses the core compliance requirement for **processing integrity** (SOC 2 PI series) and **data integrity** (GDPR Article 5(1)(f)). Without it, organizations cannot prove their audit trails haven't been altered to conceal security incidents or compliance violations.

### Goal 2: Hash Chain Integrity

**Mental Model**: Imagine a **diary with self-verifying pages** where each page ends with a wax seal that embeds fibers from the previous page. If someone removes or alters page 42, the wax seal on page 43 breaks because its embedded fibers no longer match. Each page cryptographically "chains" to the previous one.

**Detailed Explanation**: Beyond simple immutability, we need **cryptographic proof** that the log hasn't been tampered with. A **hash chain** provides this by having each entry contain a cryptographic hash of:

1. The audit event data itself
2. The hash of the *previous* entry in the chain

This creates a **verifiable sequence** where tampering with any single entry invalidates the hashes of *all subsequent entries*. The system must:

- Compute SHA-256 (or similar) hashes for each entry
- Store each hash with its corresponding event
- Provide verification tools to check chain integrity
- Handle segment rotation without breaking chain continuity

**Rationale**: This addresses **detection of unauthorized changes** (SOC 2 CC-7.1) and **security of processing** (GDPR Article 32). Cryptographic integrity proofs are becoming standard expectation for compliance audits in regulated industries.

### Goal 3: Structured Event Model

**Mental Model**: Every audit event is a **standardized incident report form** with mandatory fields (who, what, when, where, outcome) and space for additional notes. Just as a police report requires badge numbers, timestamps, and location, our audit events require specific structured data.

**Detailed Explanation**: Unstructured text logs are insufficient for automated compliance checking. We need a **schema-enforced structure** that guarantees every event contains:

- **Actor identification** (who performed the action)
- **Action type** (what was done)
- **Resource identifier** (what was acted upon)
- **Precise timestamp** (when it happened)
- **Outcome** (success/failure/error)
- **Request context** (IP, user agent, session ID)
- **Extensible metadata** (domain-specific details)

The schema must:

- Validate events at ingestion time
- Support schema evolution without breaking existing data
- Handle sensitive data masking (PII, tokens, secrets)
- Provide consistent serialization formats

**Rationale**: Structured logging enables **automated analysis** (SOC 2 CC-7.1) and supports the **right to information** (GDPR Article 15) by making logs comprehensible to data subjects. It also facilitates efficient querying and reporting.

### Goal 4: Efficient Time-Range Queries

**Mental Model**: The query system should work like a **library's historical newspaper archive** organized by publication date. To find articles from June 1923, you don't search every box—you go directly to the "1923" shelf and pull the "June" binder. Indexes provide this direct access path.

**Detailed Explanation**: Audit logs grow continuously (potentially gigabytes per day), making sequential scans impractical. The system must support **sub-second queries** for common investigation patterns:

- "Show all events between 2024-03-01T00:00:00Z and 2024-03-15T23:59:59Z"
- "Find user 'alice@company.com' actions last week"
- "List all failed login attempts in the past 24 hours"

This requires:

- **Time-series indexing** on event timestamps
- **Secondary indexing** on frequent filter fields (actor, resource, action)
- **Segment-based searching** that only opens relevant time ranges
- **Pagination** for large result sets
- **Memory-efficient streaming** of results

**Rationale**: Investigators need **timely access** to audit data during security incidents (SOC 2 CC-7.1). Slow queries hinder incident response and frustrate audit processes. Efficiency also reduces infrastructure costs for long-term retention.

### Goal 5: Compliance Export Formats

**Mental Model**: The export system functions like a **court reporter preparing official transcripts**—it takes the raw proceedings (audit events) and formats them into standardized documents (CSV, JSON, PDF) that meet specific legal requirements for submission as evidence.

**Detailed Explanation**: Auditors don't want direct database access; they expect **formatted reports** they can analyze with their tools. The system must produce:

- **CSV exports** with column headers matching audit schema fields
- **JSON exports** with proper nesting and metadata preservation
- **PDF reports** with summary statistics, timelines, and per-actor breakdowns
- **Human-readable summaries** of key compliance metrics

Export functionality must:

- Handle large time ranges without memory exhaustion (streaming)
- Preserve all original event data without transformation
- Include integrity verification information (hash chain roots)
- Support filtering same as queries (export a subset of events)

**Rationale**: This directly addresses **audit support** requirements (SOC 2 CC-7.1) and **data subject access requests** (GDPR Article 15). Proper formatting reduces auditor effort and ensures reports are accepted as valid evidence.

## Non-Goals: Explicit Scoping

Explicitly stating what we *won't* build is equally important to prevent scope creep and keep the team focused. These non-goals represent features that, while potentially valuable, would distract from our core mission or can be better handled by other systems.

| Non-Goal | Reason for Exclusion | Alternative Approach |
|---|---|---|
| **Real-time alerting** | Outside compliance scope; belongs in security monitoring | Integrate with SIEM/SOAR systems that consume our audit log |
| **Multi-region replication** | Compliance doesn't require it; adds significant complexity | Single-region storage with secure backups meets requirements |
| **Full-text search over all event data** | Query patterns don't require it; would bloat indexes | Structured field searching + limited text search on key fields |

### Non-Goal 1: Real-Time Alerting

**Why It's Out of Scope**: Our audit logging system's primary purpose is **historical record-keeping for compliance**, not real-time threat detection. Building an alerting engine would:

- Introduce different architectural requirements (stream processing vs. batch)
- Require complex rule engines and notification systems
- Distract from our core focus on data integrity and query performance
- Duplicate functionality that already exists in dedicated Security Information and Event Management (SIEM) systems

**What We'll Do Instead**: The system will provide **reliable, low-latency ingestion** so security tools can consume the audit stream and perform their own alerting. We'll design clean interfaces (APIs, webhooks, or log shipping) that allow integration with Splunk, Elastic SIEM, Datadog, or custom monitoring solutions.

> **Design Principle**: "Do one thing well." Our system specializes in **tamper-evident storage and querying**—alerting is a different concern best handled by systems specializing in real-time correlation.

#### Non-Goal 2: Multi-Region Replication

**Why It's Out of Scope**: While data resilience is important, **cross-region replication for audit logs** introduces complexities that don't align with our compliance objectives:

- Hash chain verification becomes complicated with eventually consistent replication
- GDPR data residency requirements might *prohibit* cross-border replication
- The additional cost and complexity outweighs the benefit for audit logs
- Backup and restore procedures provide sufficient disaster recovery

**What We'll Do Instead**: We'll implement:

- **Local redundancy** within a region (multiple availability zones)
- **Regular secure backups** to cold storage (with hash chain preservation)
- **Clear documentation** on recovery time objectives (RTO) and recovery point objectives (RPO)
- **Integrity verification** of restored backups before putting them into service

For organizations that truly need cross-region audit logs, they can deploy independent instances per region and aggregate queries at the application layer.

#### Non-Goal 3: Full-Text Search Over All Event Data

**Why It's Out of Scope**: Comprehensive full-text search (like Elasticsearch) over every field of every audit event would:

- Dramatically increase storage requirements (inverted indexes are large)
- Slow down ingestion with indexing overhead
- Rarely be needed for compliance investigations (structured queries cover 95% of cases)
- Create a different query optimization problem than our time-series focus

**What We'll Do Instead**: The system will provide:

- **Full-text search on specific text-rich fields** like action descriptions or metadata values
- **High-performance structured queries** on indexed fields (actor, resource, action, timestamp)
- **Efficient filtering** capabilities that can narrow results before text scanning
- **Integration hooks** to export data to dedicated search systems if needed

This balanced approach supports the common investigation patterns without the overhead of indexing every word in every log entry.

#### Additional Implicit Non-Goals

Beyond the three explicit non-goals above, we should also note these implicit exclusions:

- **Long-term archival management**: While we store data, we won't build sophisticated tiering to tape, glacier, or other deep archives. We'll provide clean interfaces for external archival systems.
- **Real-time streaming to external systems**: We'll support batch exports and APIs, but not complex event streaming pipelines (Kafka, etc.)—though our design won't preclude adding this later.
- **Advanced analytics or machine learning**: We won't build dashboards, anomaly detection, or predictive models. We'll provide the raw data for other systems to analyze.
- **Fine-grained access control to the audit logs**: Basic authentication/authorization will exist, but complex RBAC models for log data are out of scope.
- **Legal hold functionality**: While our immutability helps, specific legal hold workflows (tagging, preservation orders) won't be built-in.

> **Key Insight**: Every feature we exclude makes the system **simpler to build, test, and certify**. Compliance auditors prefer simple, understandable systems over complex ones with many moving parts. By focusing narrowly on tamper-evident storage and efficient queries, we create a system that excels at its core job and integrates well with the broader security ecosystem.

### Implementation Guidance

While this section primarily defines requirements, here's practical guidance for ensuring your implementation stays aligned with these goals and non-goals.

### A. Technology Recommendations Table

| Component | Simple Option | Advanced Option | Reasoning |
|---|---|---|---|
| **Storage Immutability** | OS file permissions ( `chattr +a` on Linux) | Content-addressable storage (CAS) with CAS references | OS-level immutability is simple and effective for single-node |
| **Hash Algorithm** | SHA-256 | SHA-3-256 or BLAKE3 | SHA-256 is widely accepted for compliance; newer algorithms offer speed |
| **Event Serialization** | JSON with schema validation | Protocol Buffers with .proto schema | JSON is human-readable; Protobuf offers better performance/density |
| **Time-Series Index** | Separate index file per segment | Embedded index in segment header | Separate files simplify segment management |
| **Export Format** | CSV via streaming writer | Parquet for columnar efficiency | CSV is universally accepted; Parquet offers better compression |

### B. Recommended Project Structure Alignment

Create these directories to organize code according to our goals:

```
audit-log-system/
├── cmd/
│   ├── ingest/          # Ingestion service (Goal 3: Event Model)
│   ├── query/           # Query API service (Goal 4: Queries)
│   └── verify/          # Hash chain verification CLI (Goal 2: Integrity)
├── internal/
│   ├── event/           # Event schema and validation (Goal 3)
│   │   ├── schema.go
│   │   ├── validator.go
│   │   └── pii_masker.go
│   ├── storage/         # Immutable storage engine (Goals 1 & 2)
│   │   ├── wal.go       # Write-ahead log
│   │   ├── segment.go   # Segment management
│   │   ├── hashchain.go # Hash chain logic
│   │   └── index.go     # Time-series indexing
│   ├── query/           # Query engine (Goal 4)
│   │   ├── engine.go
│   │   ├── filter.go
│   │   └── paginate.go
│   └── export/          # Export functionality (Goal 5)
│       ├── csv.go
│       ├── json.go
│       └── pdf.go       # External library wrapper
└── pkg/
    └── types/           # Public data types for integration
        ├── event.go
        └── query.go
```

### C. Goal Validation Checklist

As you implement, regularly verify against these checklists:

**Goal 1: Immutable Storage**

- [ ] Attempts to modify sealed segment files fail with permission errors
- [ ] Storage API lacks Update() and Delete() methods
- [ ] Appends are atomic (complete writes or nothing)
- [ ] File handles are properly closed to prevent accidental corruption

**Goal 2: Hash Chain Integrity**

- [ ] Each new entry's hash incorporates the previous entry's hash
- [ ] Verification tool can detect single-bit changes in any entry
- [ ] Segment rotation carries forward the hash chain
- [ ] Backup/restore procedures preserve hash relationships

**Goal 3: Structured Event Model**

- [ ] Events without required fields are rejected at ingestion
- [ ] PII masking occurs before storage

- [ ] Serialization to JSON includes all required fields
- [ ] Schema version is stored with each event

**Goal 4: Efficient Time-Range Queries**

- [ ] Queries for 30 days of data return in <1 second on test dataset
- [ ] Index files are created for each segment
- [ ] Pagination works correctly with large result sets
- [ ] Only relevant segments are opened for a given time range

**Goal 5: Compliance Export Formats**

- [ ] CSV exports open correctly in Excel and audit tools
- [ ] JSON exports preserve nested metadata structures
- [ ] Large exports stream to disk without loading all data in memory
- [ ] Export includes verification information (segment hashes)

### D. Avoiding Scope Creep: Decision Filter

When considering adding a feature, ask these questions:

1. **Compliance Requirement**: Is this feature explicitly required by SOC 2, GDPR, or another compliance framework we're targeting?
2. **Core Mission**: Does this feature directly support tamper-evident storage or efficient querying of audit logs?
3. **Alternative**: Can this functionality be provided by integrating with an existing system rather than building it?
4. **Complexity Cost**: Will this feature significantly increase testing, certification, or maintenance burden?

If answers are "no" to 1 and 2, "yes" to 3, or the complexity cost is high, the feature should likely be excluded or deferred.

### E. Milestone Checkpoint: Goals Alignment

After completing each milestone, run these validation commands:

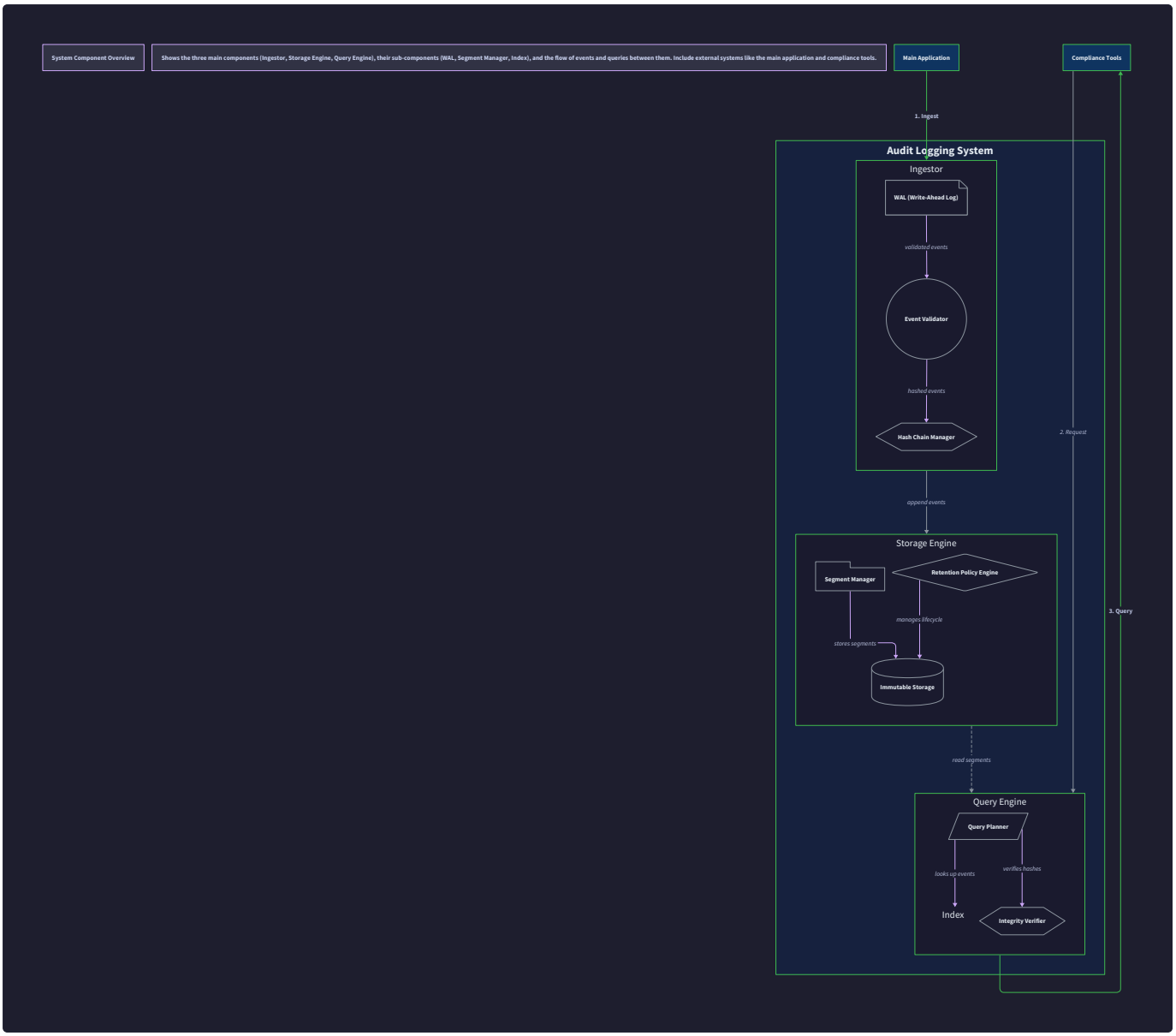
```bash
# After Milestone 1 (Event Model):
go test ./internal/event/... -v
# Should show: ✓ Valid events accepted, ✗ Invalid events rejected, ✓ PII masked

# After Milestone 2 (Storage):
go test ./internal/storage/... -v
# Should show: ✓ Appends succeed, ✗ Modifications fail, ✓ Hash chain verifies

# After Milestone 3 (Query & Export):
go test ./internal/query/... ./internal/export/... -v
# Should show: ✓ Time-range queries fast, ✓ CSV export complete, ✓ Pagination works
```

Expected output should include zero test failures and demonstration of each goal's key capabilities through the test descriptions.

## High-Level Architecture

> **Milestone(s):** Milestone 1 (Event Model), Milestone 2 (Storage), Milestone 3 (Query)

The high-level architecture defines the major building blocks of our tamper-evident audit logging system and how they collaborate to achieve immutability, integrity, and efficient querying. Think of this system as a specialized **library for recording history** that prioritizes security over convenience. The library has three primary departments: 1) the **Receiving Desk** (Ingestor) that checks and accepts historical records, 2) the **Archival Vault** (Storage Engine) that binds records into tamper-proof volumes and stores them permanently, and 3) the **Reading Room** (Query Engine) that provides controlled, indexed access to those volumes for investigators. Each component has a singular responsibility, and data flows strictly in one direction: from creation through to immutable storage, with read-only access for queries.




## Component Overview and Responsibilities

The system decomposes into three core components, each designed to handle a distinct phase of the audit event lifecycle. This separation follows the **single responsibility principle** and allows each component to be optimized for its specific workload: the Ingestor for validation and fast acceptance, the Storage Engine for guaranteed durability and integrity, and the Query Engine for efficient historical retrieval.

### 1. Ingestor: The Gatekeeper and Notary

The Ingestor is the system's public-facing API and the first line of defense. Its primary role is to accept audit events from the main application (or other services), ensure they are complete and valid, and then pass them to the Storage Engine for permanent recording. It operates like a **notary public** for system actions: it verifies the identity of the submitter (via the actor context), stamps the event with a precise timestamp, and ensures the "document" (event) is properly formatted before sealing it into the official record.

**Responsibilities:**

- **Event Validation:** Enforces the `AuditEvent` schema, rejecting any events missing required fields (actor, action, resource, timestamp, outcome) or exceeding size limits ( `MAX_EVENT_SIZE` ).
- **Context Enrichment:** Automatically attaches or verifies contextual information like `CorrelationID` and `SessionID` from incoming request headers to maintain traceability across distributed systems.
- **PII Masking:** Applies predefined rules to redact or tokenize sensitive fields (e.g., credit card numbers, email addresses) within the event's `Metadata` or `RequestContext` before storage to comply with GDPR's right to erasure.
- **Synchronous Acceptance:** Processes events synchronously within the request/transaction flow to guarantee that the audit log reflects the exact moment and context of the action. Async logging risks losing this crucial context.

- **Idempotency Handling:** Detects and handles duplicate event submissions (e.g., via a unique client-generated ID in metadata) to prevent double-logging from retries.

**Key Interfaces:**

| Method Name | Parameters | Returns | Description |
|---|---|---|---|
| `SubmitAuditEvent` | `event AuditEvent` | `(ackToken string, error)` | Primary public API. Validates the event, enriches context, applies PII masking, and forwards it to the Storage Engine. Returns an acknowledgment token (e.g., the event's sequence number) for client reference. |

### 2. Storage Engine: The Immutable Vault

The Storage Engine is the heart of the system's integrity guarantee. It is responsible for durably and immutably storing audit events in an **append-only** fashion, while constructing and maintaining the **hash chain** that makes tampering evident. It is internally composed of two sub-components: the **Write-Ahead Log (WAL)** for immediate durability and the **Segment Manager** for organizing immutable, sealed files.

**Mental Model: The Blockchain Ledger Page.** Imagine a bound ledger where each page is a `Segment`. You can only write on the current open page. When the page is full, you seal it with wax (a cryptographic hash), write the seal's impression on the next page's header, and start writing on that new page. This chain of seals (hashes) links every page to the one before it. If anyone tries to alter a word on a sealed page, the wax seal breaks, and the mismatch becomes obvious when you verify the seals sequentially.

**Sub-component: Write-Ahead Log (WAL)**

- **Purpose:** Provides crash-safe, durable storage for newly appended events before they are batched into a segment. It's a simple, high-performance append-only file.
- **Operation:** Every call to `StorageEngine.AppendEvent` writes the complete `HashChainEntry` (containing the event's hash and the previous hash) directly to the WAL file, followed by an `fsync()` to flush to disk. This ensures no acknowledged event is lost on a power loss or crash.
- **Structure:** The WAL is a series of `HashChainEntry` records in binary format. It acts as the source of truth for the most recent, unsealed events.

**Sub-component: Segment Manager**

- **Purpose:** Periodically rolls the active WAL into an immutable, sealed `Segment` file for efficient long-term storage and querying.
- **Operation:** Monitors the size of the active WAL. When it exceeds `DEFAULT_SEGMENT_SIZE` (e.g., 1GB) or after a time period (e.g., daily), it:
  1. Seals the current WAL file, marking it as read-only.
  2. Generates a final hash for the sealed segment (the `RootHash` in the `SegmentManifest`).
  3. Creates a new, empty WAL file to accept new events.
  4. Builds query-optimized indexes (e.g., on timestamp and actor ID) for the sealed segment.
  5. Updates the central catalog of segments.
- **Output:** A `Segment` comprises two files: a data file (containing the event and hash chain data) and a manifest file ( `SegmentManifest` ) describing its boundaries and integrity information.

**Storage Engine Responsibilities:**

- **Hash Chain Maintenance:** Computes the hash for each new `HashChainEntry` as `SHA-256(event_hash + previous_hash)` , ensuring each entry cryptographically depends on its predecessor.
- **Immutable Append-Only Semantics:** Physically prevents modification or deletion of written data. File permissions are set to read-only after sealing. The system's API contains no `Update` or `Delete` operations.
- **Sequence Number Generation:** Assigns a globally monotonic `Sequence uint64` to each event, providing a total order for all events across all segments. This is critical for hash chain continuity and cursor-based pagination.
- **Integrity Verification:** Provides the `VerifyChain` method to recompute hashes from any starting point and compare them against stored values, detecting any corruption or tampering.
- **Encryption at Rest:** Transparently encrypts segment files using AES-256 before writing to disk, and decrypts when reading for queries.

**Key Interfaces:**

| Method Name | Parameters | Returns | Description |
|---|---|---|---|
| `AppendEvent` | `event AuditEvent` | `(HashChainEntry, error)` | The core write operation. Generates sequence number, computes hash chain link, writes to WAL, and returns the new hash chain entry. |
| `ReadEvent` | `sequence uint64` | `(AuditEvent, error)` | Retrieves a specific event by its global sequence number. Locates the correct segment and reads the event. |
| `VerifyChain` | `startSeq uint64` | `(bool, error)` | Verifies integrity from `startSeq` to the latest entry. Returns `true` if chain is intact. |
| `GetActiveSegmentInfo` | - | `SegmentManifest` | Returns metadata about the currently active (writable) segment. |

#### 3. Query Engine: The Investigative Interface

The Query Engine provides read-only access to the immutable audit trail. It is optimized for the most common compliance and investigation patterns: **time-range queries** filtered by actor, resource, or action. It does this by leveraging indexes built per segment during the sealing process. Think of it as a **library's card catalog system**, where each segment has its own set of index cards (by time, by author/actor, by subject/resource). To answer a query, the engine consults the catalog to identify which volumes (segments) might contain relevant records, then efficiently retrieves only those pages.

**Responsibilities:**

- **Query Parsing & Planning:** Accepts a `QueryFilters` object (containing time range, actor IDs, resource types, etc.) and determines which segments overlap with the time range.
- **Indexed Retrieval:** For each relevant segment, uses its internal indexes to quickly locate candidate events without scanning the entire segment file.
- **Result Filtering & Merging:** Applies remaining filters (e.g., exact actor match) to the candidate events and merges results from multiple segments in chronological order.
- **Cursor-based Pagination:** Implements stable pagination using the global `sequence` number as a cursor. This avoids the "skip" inefficiency of offset-based pagination and prevents duplicates or misses when new events are written during pagination.
- **Streaming Export:** Generates compliance exports (CSV, JSON) by streaming events directly from storage to the output file or network connection, preventing memory exhaustion with large result sets.
- **Report Generation:** Computes aggregate statistics (e.g., "number of login failures per user in the last 24 hours") for summary audit reports.

**Key Interfaces:**

| Method Name | Parameters | Returns | Description |
|---|---|---|---|
| `Query` | `filters QueryFilters, startTime, endTime time.Time` | `(QueryResult, error)` | Main query method. Returns a `QueryResult` containing the first page of events and a cursor for the next page. |
| `QueryResult.NextPage` | `cursor string` | `(QueryResult, error)` | Retrieves the next page of results using a cursor from a previous query. |
| `Export` | `filters QueryFilters, format ExportFormat, writer io.Writer` | `error` | Streams all matching events directly to the provided writer in the specified format (CSV, JSON). |

**Interaction with External Systems:**

- **Main Application:** Submits audit events via the Ingestor's API (e.g., gRPC or HTTP) synchronously during business logic execution.
- **Compliance Tools & Investigators:** Connect to the Query Engine's API to run queries, generate reports, or request bulk exports for external analysis.
- **Monitoring & Alerting (Future):** Could consume a real-time stream of events (e.g., via a Kafka topic populated by the Ingestor) for security monitoring, though this is a non-goal for the current system.

### Recommended File and Module Structure

A clear, idiomatic Go project structure is essential for maintainability and to reflect the architectural boundaries. We follow the **Standard Go Project Layout** principles, with an `internal` directory to encapsulate implementation details that shouldn't be imported by external applications. This structure mirrors the component diagram, making it easy for developers to locate code for a specific responsibility.

```
audit-log-system/
├── cmd/                          # Application entry points
│   ├── ingest-server/            # Standalone Ingestor service (optional)
│   │   └── main.go
│   ├── query-server/             # Standalone Query Engine service (optional)
│   │   └── main.go
│   └── combined-server/          # Single binary with all components (likely main)
│       └── main.go
├── internal/                     # Private application code
│   ├── audit/                    # Core domain models (Milestone 1)
│   │   ├── event.go              # AuditEvent, ActorInfo, ResourceInfo, RequestContext structs
│   │   ├── validator.go          # EventValidator.Validate logic and schema rules
│   │   ├── pii/                  # PII masking utilities
│   │   │   ├── masker.go
│   │   │   └── patterns.go
│   │   └── serialization/        # Event serialization formats
│   │       ├── json.go
│   │       └── binary.go
│   ├── ingest/                   # Ingestor component (Milestone 1)
│   │   ├── ingestor.go           # Main Ingestor struct and SubmitAuditEvent
│   │   ├── handler.go            # HTTP/gRPC request handlers
│   │   └── middleware/           # Context enrichment, idempotency
│   │       └── context.go
│   ├── storage/                  # Storage Engine component (Milestone 2)
│   │   ├── engine.go             # StorageEngine struct with AppendEvent, ReadEvent, VerifyChain
│   │   ├── wal/                  # Write-Ahead Log sub-component
│   │   │   ├── wal.go            # Low-level WAL file operations (append, read, sync)
│   │   │   └── entry.go          # HashChainEntry struct and hash computation logic
│   │   ├── segment/              # Segment Manager sub-component
│   │   │   ├── manager.go        # Segment lifecycle (roll, seal, index)
│   │   │   ├── segment.go        # Segment file reader/writer
│   │   │   ├── manifest.go       # SegmentManifest struct and I/O
│   │   │   └── index/            # Index building and querying per segment
│   │   │       ├── time_index.go
│   │   │       └── actor_index.go
│   │   └── catalog.go            # Tracks all segments (in-memory + persisted)
│   ├── query/                    # Query Engine component (Milestone 3)
│   │   ├── engine.go             # QueryEngine struct with Query, Export methods
│   │   ├── filters.go            # QueryFilters struct and parsing
│   │   ├── planner.go            # Query planning: segment selection based on time range
│   │   ├── cursor.go             # Cursor encoding/decoding for pagination
│   │   └── export/               # Export formatting
│   │       ├── csv.go            # ExportEngine.ToCSV implementation
│   │       ├── json.go
│   │       └── pdf.go            # (Future, if needed)
│   └── crypto/                   # Shared cryptographic utilities
│       ├── hashchain.go          # Hash chain computation and verification logic
│       └── encryption.go         # AES-256 encryption/decryption for segments
├── pkg/                          # Public, reusable libraries (if any)
│   └── api/                      # Public Go client library (e.g., for submitting events)
│       ├── client.go
│       └── types.go              // Exportable versions of core types
├── go.mod
├── go.sum
└── configs/                      # Configuration files
    ├── config.yaml               # Main configuration (segment size, encryption keys)
    └── pii-rules.yaml            # Rules for PII masking patterns
```

**Justification of Key Directories:**

- `internal/audit` : Contains the core domain model and validation logic. It has no dependencies on other `internal` packages, making it the foundation.
- `internal/storage` : The most complex module. Its subdirectories ( `wal/` , `segment/` ) reflect distinct sub-components with clear interfaces.
- `internal/query` : Depends on `internal/storage` to read data and `internal/audit` to understand the event format.
- `internal/crypto` : Centralizes cryptographic operations to ensure consistency (e.g., always using SHA-256 for the hash chain) and simplifies testing.
- `cmd/` : Provides flexibility to deploy components as separate microservices or a single monolith. The `combined-server` is likely the starting point.

This structure enforces the architectural boundaries: the Ingestor knows about the Storage Engine's interface but not its internal file formats. The Query Engine knows how to read segments and indexes but cannot modify them. This separation is critical for maintaining the system's integrity guarantees.

## Implementation Guidance

**Technology Recommendations Table:**

| Component | Simple Option (Initial Implementation) | Advanced Option (Production-Ready) |
|---|---|---|
| **Ingestion Transport** | Synchronous HTTP REST API with JSON ( `net/http` ). Simple to debug and integrate. | gRPC with Protocol Buffers for strict API contracts, bi-directional streaming, and better performance. |
| **Storage Backend** | Local filesystem with direct file I/O ( `os` package). Simplest to implement and reason about. | Plugable storage interface supporting cloud object storage (S3, GCS) for segment archiving, with local SSD for active WAL. |
| **In-Memory Catalog** | In-memory map of `SegmentManifest` loaded at startup, persisted to a simple JSON file. | Embedded key-value store (e.g., `bbolt` ) for durable, concurrent catalog access. |
| **Indexing** | Sorted slice of (timestamp, file-offset) pairs per segment, built on seal. Memory-efficient. | External indexing library (e.g., `pilosa` for bitmaps) for extremely high cardinality fields (actor IDs). |
| **Encryption** | AES-256 in GCM mode using a key from a configuration file. | Integrated with a cloud KMS (Key Management Service) for automatic key rotation and secure key storage. |

**Recommended File/Module Structure Starter Code:**

This is a minimal starter for the `internal/storage/wal` package, which provides the foundational append-only file operations. This is **complete, working code** that learners can use directly.

**File:** `internal/storage/wal/wal.go`

```go
package wal

import (
    "encoding/binary"
    "fmt"
    "io"
    "os"
    "path/filepath"
    "sync"
)

const (
    // walHeader is a fixed signature at the start of every WAL file.
    walHeader = "AUDITWALv1"
    // maxRecordSize is a sanity limit to prevent memory exhaustion.
    maxRecordSize = 1024 * 1024 // 1MB
)

// WAL represents a Write-Ahead Log file.
type WAL struct {
    file     *os.File
    filePath string
    mu       sync.Mutex
    offset   int64 // current write offset, maintained for performance.
}

// OpenWAL opens or creates a WAL file at the given path.
func OpenWAL(path string) (*WAL, error) {
    // Ensure directory exists.
    if err := os.MkdirAll(filepath.Dir(path), 0755); err != nil {
        return nil, fmt.Errorf("create wal directory: %w", err)
    }

    // Open file in append mode, create if doesn't exist.
    file, err := os.OpenFile(path, os.O_RDWR|os.O_CREATE|os.O_APPEND, 0644)
    if err != nil {
        return nil, fmt.Errorf("open wal file: %w", err)
    }

    w := &WAL{
        file:     file,
        filePath: path,
```

```
    }

    // Initialize file if it's new (empty).
    if stat, err := file.Stat(); err == nil && stat.Size() == 0 {
        if err := w.writeHeader(); err != nil {
            file.Close()
            return nil, err
        }
        w.offset = int64(len(walHeader))
    } else {
        // Otherwise, seek to end to determine offset.
        w.offset, err = file.Seek(0, io.SeekEnd)
        if err != nil {
            file.Close()
            return nil, fmt.Errorf("seek to end: %w", err)
        }
    }

    return w, nil
}

// writeHeader writes the constant header to the beginning of the file.
func (w *WAL) writeHeader() error {
    _, err := w.file.Write([]byte(walHeader))
    return err
}

// Append writes a record (a byte slice) to the WAL and fsyncs for durability.
// Returns the file offset where the record starts.
func (w *WAL) Append(record []byte) (int64, error) {
    if len(record) > maxRecordSize {
        return 0, fmt.Errorf("record size %d exceeds maximum %d", len(record), maxRecordSize)
    }

    w.mu.Lock()
    defer w.mu.Unlock()

    startOffset := w.offset

    // Write length prefix (8 bytes, little-endian).
    lenBuf := make([]byte, 8)
    binary.LittleEndian.PutUint64(lenBuf, uint64(len(record)))
```

```
    if _, err := w.file.Write(lenBuf); err != nil {
        return 0, fmt.Errorf("write length prefix: %w", err)
    }

    // Write the record itself.
    if _, err := w.file.Write(record); err != nil {
        return 0, fmt.Errorf("write record: %w", err)
    }

    // Ensure data is flushed to disk.
    if err := w.file.Sync(); err != nil {
        return 0, fmt.Errorf("fsync wal: %w", err)
    }

    w.offset += int64(8 + len(record))
    return startOffset, nil
}

// ReadAt reads a record from the given file offset.
func (w *WAL) ReadAt(offset int64) ([]byte, error) {
    w.mu.Lock()
    defer w.mu.Unlock()

    // Read the length prefix.
    lenBuf := make([]byte, 8)
    if _, err := w.file.ReadAt(lenBuf, offset); err != nil {
        return nil, fmt.Errorf("read length prefix: %w", err)
    }
    recordLen := binary.LittleEndian.Uint64(lenBuf)

    if recordLen > maxRecordSize {
        return nil, fmt.Errorf("record length %d at offset %d exceeds maximum", recordLen, offset)
    }

    // Read the record data.
    record := make([]byte, recordLen)
    if _, err := w.file.ReadAt(record, offset+8); err != nil {
        return nil, fmt.Errorf("read record data: %w", err)
    }
    return record, nil
}

// Close closes the underlying file.
```

```go
func (w *WAL) Close() error {

    w.mu.Lock()

    defer w.mu.Unlock()

    return w.file.Close()

}
```

**Core Logic Skeleton Code for Storage Engine:**

This skeleton outlines the core `AppendEvent` method of the `StorageEngine`. Learners must implement the hash chain logic and sequence number generation.

**File:** `internal/storage/engine.go`

```go
package storage

import (
    "audit-log-system/internal/audit"
    "audit-log-system/internal/crypto"
    "time"
)

// StorageEngine is the central coordinator for immutable audit storage.
type StorageEngine struct {
    catalog      *Catalog
    activeWAL    *wal.WAL
    activeSeq    uint64 // Last used sequence number, must be persisted.
    mu           sync.RWMutex
    config       Config
}

// AppendEvent adds a new audit event to the log, creating a corresponding hash chain entry.
func (s *StorageEngine) AppendEvent(event audit.AuditEvent) (HashChainEntry, error) {
    // TODO 1: Validate the incoming event using audit.EventValidator.Validate()
    // TODO 2: Generate the next monotonic sequence number (s.activeSeq + 1). Use a mutex to protect s.activeSeq.
    // TODO 3: Serialize the event to bytes (using your chosen serialization format, e.g., JSON or Protobuf).
    // TODO 4: Compute the hash of the serialized event bytes (e.g., SHA-256). This is the EventHash.
    // TODO 5: Retrieve the previous hash chain entry's hash (or use nil/zero hash for the first entry).
    // TODO 6: Compute the new entry's hash: SHA-256(EventHash + PreviousHash). This links the chain.
    // TODO 7: Construct the HashChainEntry struct with Sequence, PreviousHash, EventHash, Timestamp (time.Now().UTC()), and Signature (optional, for future use).
    // TODO 8: Serialize the HashChainEntry to bytes (using a consistent binary format).
    // TODO 9: Append the serialized entry to the active WAL using wal.Append(). This ensures durability.
    // TODO 10: Update the in-memory s.activeSeq and persist it (e.g., to a small state file) to survive crashes.
    // TODO 11: Check if the active WAL has reached the size limit (s.config.SegmentSize). If yes, call s.rollSegment().
    // TODO 12: Return the newly created HashChainEntry to the caller (e.g., Ingestor).
    return HashChainEntry{}, nil
}

// VerifyChain recomputes hashes from startSeq to the latest entry and compares them.
func (s *StorageEngine) VerifyChain(startSeq uint64) (bool, error) {
    // TODO 1: Use the catalog to find the segment containing startSeq.
    // TODO 2: Open that segment and iterate through entries from startSeq onward.
    // TODO 3: For each entry, recompute the hash (EventHash + PreviousHash) and compare to the stored hash.
    // TODO 4: If any mismatch is found, return false and log the corrupt sequence number.
```

```
    // TODO 5: If the chain crosses segment boundaries, ensure the previous segment's final hash matches the next segment's first PreviousHash.

    // TODO 6: Return true only if the entire chain from startSeq to the end is valid.

    return false, nil

}
```

**Language-Specific Hints for Go:**

1. **Concurrency:** Use `sync.RWMutex` in `StorageEngine`. The `AppendEvent` method needs a write lock ( `Lock()` ), while `ReadEvent` and `VerifyChain` can use a read lock ( `RLock()` ).
2. **Hashing:** Use `crypto/sha256`. The typical pattern is `hash := sha256.Sum256(data)`. Remember that `Sum256` returns a fixed-size array `[32]byte`, which you can slice to get a `[]byte`.
3. **Time:** Always use `time.Now().UTC()` for timestamps to avoid timezone confusion in queries. Store as `time.Time`; the `encoding/json` package will handle RFC3339 formatting.
4. **File Sync:** After writing critical data (like a WAL entry), call `file.Sync()`. This is the `fsync` system call and ensures data is written to physical media, not just the OS cache.
5. **Error Wrapping:** Use `fmt.Errorf("... %w", err)` to wrap errors, providing context. This allows callers to inspect the underlying error with `errors.Is` or `errors.As`.
6. **Binary Serialization:** For `HashChainEntry`, consider using `encoding/binary` with `binary.LittleEndian.PutUint64` and `binary.LittleEndian.Uint64` for fixed-width fields. This is faster and more compact than JSON for the WAL.

**Milestone Checkpoint for High-Level Architecture:**

After setting up the basic project structure and the starter `WAL` code, you should be able to verify the foundational layer works.

1. **Command:** Run a simple test to ensure the WAL can append and read records.

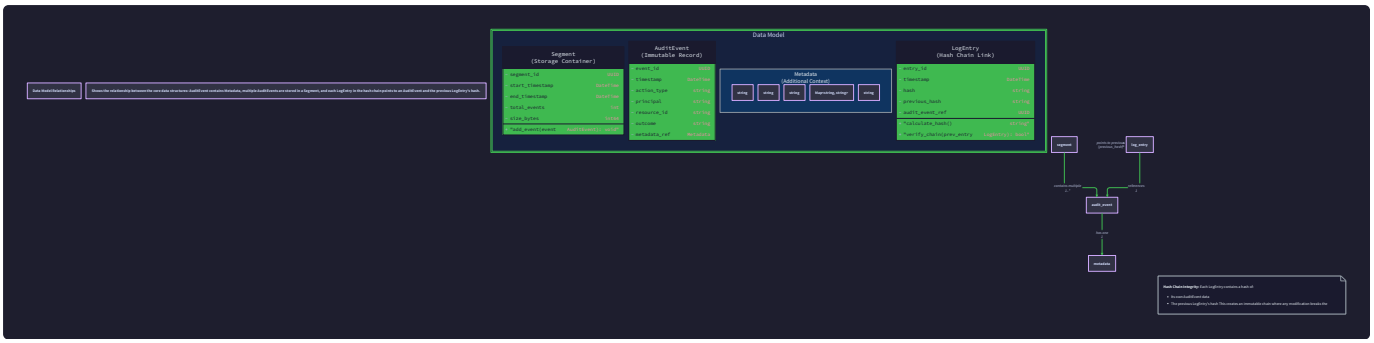
```
    go test ./internal/storage/wal/... -v
```
BASH

2. **Expected Output:** Tests should pass, demonstrating that `Append` writes data, `ReadAt` retrieves it correctly, and the file header is properly initialized.
3. **Manual Verification:**
   - Create a small Go program that uses `OpenWAL`, `Append` a `[]byte("test record")`, then `ReadAt` the returned offset.
   - The read data should exactly match the original.
   - Check the generated `.wal` file with a hex dump ( `hexdump -C file.wal` ) to see the header and length-prefixed records.
4. **Signs of Trouble:**
   - If `ReadAt` returns "EOF" errors, the offset calculation in `Append` is likely incorrect.
   - If the test passes but a manual program fails, check for proper file closure ( `defer wal.Close()` ).
   - If `fsync` is slow, remember it's critical for durability; performance tuning (batching `fsync`) comes later.

## Data Model

> **Milestone(s):** Milestone 1 (Audit Event Model), Milestone 2 (Immutable Storage with Hash Chain)

The data model is the DNA of our audit logging system. It defines the immutable atoms of information we will store—the **Audit Event**—and the cryptographic structures that bind them together into an unbreakable chain of truth—the **Hash Chain Entry** and **Storage Segment**. This section provides the exact blueprint for these structures, specifying every field, its type, and its purpose. A clear, precise data model is critical; it ensures events are captured consistently, the hash chain's integrity is mathematically verifiable, and our storage remains both efficient and tamper-evident.




## Audit Event Schema (Milestone 1)

> **Mental Model: The Notary's Stamped Form** Imagine every significant action in your system requires filling out a standardized, multi-part form at a notary's office. Each form has mandatory sections (who, what, when, where) that must be completed, a section for the official outcome (success/failure), a pre-printed section capturing the request's "atmosphere" (client IP, browser), and an open-ended "Additional Details" page where you can attach any relevant notes or tags. The notary then timestamps the entire packet with an official, sequential seal. This form is the **Audit Event**. The rigid structure (schema) ensures every record contains the minimum information needed for an investigation, while the flexible "Additional Details" (metadata) accommodates the unique context of any action without changing the core form design.

An `AuditEvent` is the fundamental unit of record. It captures a single auditable action within the system. The schema is designed with two principles in mind: **completeness** for compliance (SOC 2, GDPR) and **extensibility** for domain-specific needs. The core fields are non-optional and validated at ingestion time. The metadata container allows for arbitrary key-value pairs, enabling teams to attach context like feature flags, deployment IDs, or business transaction identifiers without requiring a schema change for every new use case.

The following table defines the complete structure of the `AuditEvent` and its nested types. All fields use the exact names specified in the naming conventions.

### Core Audit Event and Component Structures

| Name | Type | Description |
|---|---|---|
| **`AuditEvent`** | *struct* | The root object representing a single auditable action. |
| `Actor` | `ActorInfo` | **Who performed the action.** Contains identity information about the entity (user, service, system) that initiated the event. This is a required field for every audit event. |
| `Action` | `string` | **What action was performed.** A human-readable and machine-parseable verb describing the operation (e.g., `"user.login"`, `"document.update"`, `"config.delete"`). Should follow a consistent naming convention (e.g., `resource.verb`). |
| `Resource` | `ResourceInfo` | **What was acted upon.** Identifies the target object, entity, or data of the action (e.g., a user account ID, a file path, a database table). |
| `Timestamp` | `time.Time` | **When the action occurred.** Must be captured at the point of event creation, using a monotonic, high-resolution clock source if possible. Stored in UTC. |
| `Outcome` | `string` | **What was the result.** The result of the action, typically `"success"`, `"failure"`, or `"denied"`. Provides immediate context for the action's effect. |
| `Context` | `RequestContext` | **Contextual details about the request.** Captures the "how" and "where" of the action, such as network origin and session, which are crucial for forensic tracing. |
| `Metadata` | `map[string]string` | **Extensible domain-specific context.** A flat key-value store for any additional information relevant to the event (e.g., `"deployment_id": "prod-az1-v2.5"`, `"feature_flag": "new_ui_enabled"`). Keys should use a namespaced format (e.g., `com.example.flag`) to avoid collisions. |

| Name | Type | Description |
|---|---|---|
| **`ActorInfo`** | *struct* | Identifies the entity that initiated the action. |
| `ID` | `string` | **Unique identifier.** A system-unique string for the actor (e.g., user UUID, service account ID, `"system"`). |
| `Type` | `string` | **Actor category.** Classifies the actor (e.g., `"user"`, `"service"`, `"cron_job"`, `"external_api"`). Enables filtering by actor class. |
| `DisplayName` | `string` | **Human-readable name.** A name suitable for reports (e.g., `"alice@example.com"`, `"Billing Service"`). |

| Name | Type | Description |
|---|---|---|
| **`ResourceInfo`** | *struct* | Identifies the target of the action. |
| `ID` | `string` | **Unique resource identifier.** A string that uniquely identifies the resource instance within the system (e.g., document ID `"doc_abc123"`, account number `"acc-789"`). |
| `Type` | `string` | **Resource category.** The type of object acted upon (e.g., `"user_account"`, `"configuration_file"`, `"database_table"`). Enables filtering by resource type. |
| `DisplayName` | `string` | **Human-readable resource name.** A descriptive name (e.g., `"Annual Budget.docx"`, `"Production Database Config"`). |

| Name | Type | Description |
|---|---|---|
| **`RequestContext`** | *struct* | Captures the technical context of the request that generated the event. |
| `ClientIP` | `string` | **Source IP address.** The IP address of the client that initiated the request. May be masked or tokenized for PII compliance. |
| `UserAgent` | `string` | **Client software identifier.** The `User-Agent` HTTP header or equivalent, identifying the client application, browser, or SDK. |
| `SessionID` | `string` | **User session identifier.** A unique identifier for the user's session, linking multiple events from the same login session. |
| `CorrelationID` | `string` | **Request correlation identifier.** A unique ID (often a UUID) propagated across service boundaries, allowing tracing of a single logical request through a distributed system. |

**Architecture Decision Record: Structured vs. Free-Form Metadata**

**Context:** We need to support domain-specific data in audit events without constantly modifying the core `AuditEvent` schema, which would break serialization and require costly migrations.

**Options Considered:**

1. **Strict Schema Extension:** Define proto/avro schemas with versioning and require all teams to declare new fields formally.
2. **Free-Form JSON Blob:** A single `json.RawMessage` field where teams can dump any structured JSON.
3. **Flat String Key-Value Map ( `map[string]string` ):** A simple dictionary of string keys to string values.

**Decision:** Use a flat `map[string]string` for the `Metadata` field.

**Rationale:**

- **Simplicity & Predictability:** A string map is universally understood, easy to serialize/deserialize in any format (JSON, binary), and does not require schema negotiation. It avoids the complexity of parsing arbitrary nested JSON at query time.
- **Query Efficiency:** Flat key-value pairs can be indexed and filtered efficiently (e.g., "find all events where `metadata['deployment_id'] = 'prod-v1.2'`"). Nested JSON would require specialized JSON path query support.
- **PII Control:** String values can be easily masked or tokenized using a standard process before being inserted into the map, whereas identifying PII deep within a JSON structure is harder.
- **Practical Extensibility:** Most domain-specific context (feature flags, version numbers, business IDs) naturally fits as string key-value pairs. For complex data, teams can serialize to JSON and store it as a single string value, though this trades off queryability.

**Consequences:**

- Teams cannot store complex nested objects without serializing them first.
- All metadata values must be strings (numbers, booleans must be stringified).
- The map must be sized appropriately; extremely large maps could bloat event size.
- Querying for a missing key is straightforward.

| Option | Pros | Cons | Chosen? |
|---|---|---|---|
| Strict Schema Extension | Strong typing, explicit contracts, easy evolution. | High overhead, slow iteration, requires coordination and migration for every new field. | No |
| Free-Form JSON Blob | Maximum flexibility, supports complex nested structures. | Difficult to query, hard to enforce PII masking, serialization format ambiguity. | No |
| **Flat String Key-Value Map** | **Simple, queryable, easy to mask, universal compatibility.** | **No complex nesting, all values must be strings.** | **Yes** |

### Event Serialization and Validation

Before an `AuditEvent` can be stored, it must be serialized to a byte sequence for the write-ahead log and validated to ensure it meets schema requirements. The primary serialization format is JSON for human readability and interoperability, but a compact binary format (like Protocol Buffers) may be used for the permanent storage segment to save space.

**Event Validation Rules:**

1. `Actor.ID`, `Action`, `Timestamp`, and `Outcome` must be non-empty.
2. `Timestamp` cannot be in the future (allow for slight clock skew, e.g., +5 minutes).
3. `Metadata` keys must match a defined regex pattern (e.g., `^[a-zA-Z0-9_.-]+$`) to ensure safe serialization.
4. The total serialized size of the event must not exceed `MAX_EVENT_SIZE` (64KB) to prevent storage abuse.

> **Design Insight:** The `Metadata` map is the system's "escape hatch" for future-proofing. By making it a `map[string]string`, we guarantee that any team can attach necessary context without ever needing to modify the core `AuditEvent` struct, thus avoiding breaking changes to the serialization format and the associated hash chain.

## Storage Data Structures (Milestone 2)

> **Mental Model: The Blockchain Ledger Page** Imagine a ledger used in a high-security vault. Each page of the ledger (a **Segment**) holds a fixed number of transactions. Every transaction is written in indelible ink. At the top of each page, there's a special seal: a cryptographic hash computed from *all* the transactions on that page combined with the seal of the *previous* page. If you tear out a page or alter a single transaction, the seal on that page and every subsequent page will no longer match, revealing the tamper. Each transaction entry in the ledger also includes its own sequence number and a reference to the hash of the entry before it, forming a chain within the page. This dual-layered integrity—hash chain within entries, and hash linking between segments—is the foundation of our **Immutable Storage Engine**.

The storage layer's data structures are responsible for transforming a stream of validated `AuditEvent`s into an immutable, verifiable log. This involves two primary constructs: the **`HashChainEntry`**, which wraps an event with integrity-proofing data, and the **`SegmentManifest`**, which describes a sealed, immutable file containing a batch of entries.

### Hash Chain Entry

The `HashChainEntry` is the atomic unit written to the Write-Ahead Log (WAL). It cryptographically binds an event to the entire history of the log.

| Name | Type | Description |
|---|---|---|
| **`HashChainEntry`** | *struct* | A wrapper that includes the audit event's hash and links it to the previous entry. |
| `Sequence` | `uint64` | **Monotonically increasing sequence number.** Uniquely identifies the entry's absolute position in the global log. This number must always increase and never repeat or skip under normal operation. It is the primary key for event retrieval. |
| `PreviousHash` | `[]byte` | **Hash of the *previous* `HashChainEntry`.** This creates the "chain." For the first entry in a log, this is a well-known genesis hash (e.g., 32 zero bytes). Its size is `SHA256_HASH_SIZE`. |
| `EventHash` | `[]byte` | **Cryptographic hash of the serialized `AuditEvent`.** Computed using SHA-256 over the canonical byte representation of the event. This ensures the event data cannot be altered without detection. |
| `Timestamp` | `time.Time` | **The time this entry was created/appended.** This is separate from the event's timestamp and is used for storage-level operations (e.g., segment rotation based on time). |
| `Signature` | `[]byte` | **(Optional) Digital signature of this entry.** Can be populated by a separate key management service to provide non-repudiation (proving the log itself authorized the entry). May be empty if signature services are unavailable. |

**Hash Chain Construction Algorithm:**

1. **First Entry (Genesis):** `PreviousHash` is set to a predefined genesis value (e.g., 32 zero bytes).
2. **For Entry N (where N > 1):**
    1. Retrieve the `HashChainEntry` for sequence N-1.
    2. Compute `Hash_N-1 = SHA256(serialize(Entry_N-1))`. (The hash of the entire previous entry, not just its `EventHash`).
    3. Set `Entry_N.PreviousHash = Hash_N-1`.
3. **For any Entry:**
    1. Serialize the `AuditEvent` to its canonical byte form (e.g., JSON sorted by key, or Protobuf).
    2. Compute `Entry.EventHash = SHA256(serializedEventBytes)`.

4. The final integrity hash for the entry is `SHA256(Sequence || PreviousHash || EventHash || Timestamp)`. This hash becomes the `PreviousHash` for entry N+1.

> **Design Insight:** Hashing the entire previous `HashChainEntry` (not just its `EventHash`) is crucial. It ensures that any modification to the chain's metadata (like the sequence number or the previous hash itself) is also detected. This makes the chain resilient to lower-level tampering.

### Segment Manifest

When a WAL file reaches a size or time threshold (`DEFAULT_SEGMENT_SIZE` or 24 hours), it is "sealed" into an immutable **Segment**. A `SegmentManifest` is a metadata file that describes this sealed segment, allowing the **Query Engine** to quickly locate relevant data without scanning all files.

| Name | Type | Description |
|---|---|---|
| **`SegmentManifest`** | *struct* | Metadata describing a sealed, immutable segment file. |
| `SegmentID` | `string` | **Unique identifier for the segment.** Typically a UUID or a combination of start timestamp and sequence (e.g., `segment-<startSeq>-<timestamp>`). Used to name the data file (e.g., `{SegmentID}.log`). |
| `StartTime` | `time.Time` | **Earliest `Timestamp` of any `HashChainEntry` in the segment.** Used for time-range queries to quickly exclude segments outside the query window. |
| `EndTime` | `time.Time` | **Latest `Timestamp` of any `HashChainEntry` in the segment.** |
| `StartSequence` | `uint64` | **The `Sequence` number of the first entry in this segment.** Defines the segment's lower bound. |
| `EndSequence` | `uint64` | **The `Sequence` number of the last entry in this segment.** Defines the segment's upper bound. |
| `RootHash` | `[]byte` | **Merkle Root Hash of all entries in the segment.** While the linear hash chain provides integrity between entries, a Merkle tree computed over all entries in the segment enables efficient, parallelizable verification of a subset of entries (e.g., "prove that event with sequence X is in segment Y"). |
| `PreviousSegmentID` | `string` | **ID of the segment that immediately precedes this one in the global sequence.** This forms a linked list of segments, mirroring the hash chain at the file level. The hash chain integrity crosses segment boundaries via the `PreviousHash` of the first entry in this segment pointing to the last entry's hash in the previous segment. |

> **Architecture Decision Record: Linear Hash Chain vs. Merkle Tree for Intra-Segment Integrity**
>
> **Context:** We need to efficiently verify the integrity of a single event or a range of events without reading and hashing the entire log from the beginning. A linear hash chain requires O(n) computation to verify event N.
>
> **Options Considered:**
>
> 1. **Pure Linear Hash Chain:** Only store `PreviousHash` in each entry. Verification of any event requires replaying the chain from a trusted starting point up to that event.
> 2. **Segment Merkle Tree:** Build a Merkle tree over all entries within a segment when it is sealed. Store the Merkle root (`RootHash`) in the `SegmentManifest`. Each entry's inclusion can be proven with an O(log n) Merkle proof.
> 3. **Hybrid Approach:** Maintain the linear hash chain for total ordering and append-only guarantees, but *also* compute a Merkle tree per segment for efficient subset verification.
>
> **Decision:** Use the Hybrid Approach (both linear hash chain and per-segment Merkle tree).
>
> **Rationale:**
>
> - **Linear Chain Simplicity:** The linear hash chain is simple to implement for the core append operation, provides strong guarantees about absolute ordering, and easily spans segment boundaries.
> - **Efficient Verification:** A compliance auditor may only need to verify the integrity of 100 specific events from a log containing billions. With a pure chain, this is impractical. Merkle proofs allow them to verify each event against the signed `RootHash` of its segment, which is much faster.
> - **Parallelizable Audits:** Multiple auditors can verify different segments, or different events within a segment, independently and in parallel.
> - **Moderate Overhead:** Computing a Merkle tree at segment sealing time (a batch operation) adds minimal overhead to the write path. The storage overhead is one extra hash (`RootHash`) per segment.
>
> **Consequences:**
>
> - The write path must compute two hash structures: the linear chain (per entry) and the Merkle tree (per segment).
> - The `SegmentManifest` becomes slightly more complex.
> - We must design a format for storing Merkle proofs alongside events for export.
> - Verification tooling must understand both integrity mechanisms.

| Option | Pros | Cons | Chosen? |
|---|---|---|---|
| Pure Linear Hash Chain | Simple implementation, minimal write overhead. | Costly verification of distant events (O(n)), not suitable for random-access proofs. | No |
| Segment Merkle Tree | Efficient O(log n) inclusion proofs, parallel verification. | More complex to implement, tree must be rebuilt if segment format changes, doesn't inherently guarantee global order. | No |
| **Hybrid Approach** | **Retains simple append logic, enables efficient random verification, maintains global order.** | **Implementation complexity is higher, requires dual hash computation.** | **Yes** |

## Implementation Guidance for Data Model

### A. Technology Recommendations Table

| Component | Simple Option | Advanced Option |
|---|---|---|
| Event Serialization | `encoding/json` (Go stdlib) for readability and WAL. | `google.golang.org/protobuf` for compact binary storage in sealed segments. |
| Hash Function | `crypto/sha256` (Go stdlib). | `crypto/sha512` for stronger security margin (but 64-byte hashes double storage). |
| Time Handling | `time.Time` with `time.UTC()` normalization. | `github.com/araddon/dateparse` for robust input time parsing if needed. |

### B. Recommended File/Module Structure

```
audit-log-system/
├── internal/
│   ├── event/                # Milestone 1: Event Model
│   │   ├── event.go          # AuditEvent, ActorInfo, ResourceInfo, RequestContext structs
│   │   ├── validator.go      # EventValidator with Validate() method
│   │   ├── serializer.go     # JSON and binary serialization logic
│   │   └── pii/              # Optional: PII masking utilities
│   │       └── masker.go
│   ├── storage/              # Milestone 2: Immutable Storage
│   │   ├── entry.go          # HashChainEntry struct
│   │   ├── manifest.go       # SegmentManifest struct
│   │   ├── wal/              # Write-Ahead Log component
│   │   │   ├── wal.go        # WAL struct and methods (OpenWAL, Append, ReadAt)
│   │   │   └── record.go     # Record encoding/decoding logic
│   │   └── segment/          # Segment management
│   │       ├── manager.go    # Segment rotation, sealing logic
│   │       └── builder.go    # Builds segments from WAL, computes Merkle tree
│   └── query/                # Milestone 3: Query & Export (for later)
│       └── ...               # (To be elaborated in next section)
└── pkg/models.go             # Optional: Shared type definitions if needed by external packages
```

### C. Infrastructure Starter Code: Event Serialization and Validation

The following is a complete, ready-to-use utility for validating and serializing `AuditEvent`s. It ensures events conform to the schema before they enter the storage system.

GO

```go
// internal/event/validator.go

package event

import (
    "errors"
    "regexp"
    "time"
)

var (
    ErrActorIDRequired   = errors.New("actor ID is required")
    ErrActionRequired    = errors.New("action is required")
    ErrTimestampRequired = errors.New("timestamp is required")
    ErrOutcomeRequired   = errors.New("outcome is required")
    ErrTimestampFuture   = errors.New("timestamp cannot be in the future")
    ErrMetadataKeyInvalid= errors.New("metadata key contains invalid characters")
    ErrEventTooLarge     = errors.New("event size exceeds maximum allowed")
)

// EventValidator performs schema validation on AuditEvents.
type EventValidator struct {
    maxEventSize int
    // maxClockSkew defines the allowable future timestamp drift (e.g., 5 minutes).
    maxClockSkew time.Duration
    metadataKeyRegex *regexp.Regexp
}

func NewEventValidator(maxEventSize int, maxClockSkew time.Duration) *EventValidator {
    // Allow alphanumeric, underscore, dot, and hyphen in metadata keys.
    re := regexp.MustCompile(`^[a-zA-Z0-9_.-]+$`)
    return &EventValidator{
        maxEventSize: maxEventSize,
        maxClockSkew: maxClockSkew,
        metadataKeyRegex: re,
    }
}

// Validate checks the AuditEvent against all schema rules.
func (v *EventValidator) Validate(event AuditEvent) error {
    // 1. Check required fields
    if event.Actor.ID == "" {
        return ErrActorIDRequired
```

```
    }

    if event.Action == "" {

        return ErrActionRequired

    }

    if event.Timestamp.IsZero() {

        return ErrTimestampRequired

    }

    if event.Outcome == "" {

        return ErrOutcomeRequired

    }


    // 2. Check timestamp is not too far in the future (allow for clock skew)

    now := time.Now().UTC()

    if event.Timestamp.After(now.Add(v.maxClockSkew)) {

        return ErrTimestampFuture

    }


    // 3. Validate metadata keys

    for key := range event.Metadata {

        if !v.metadataKeyRegex.MatchString(key) {

            return ErrMetadataKeyInvalid

        }

    }


    // 4. Check event size limit (approximate)

    // Note: This is an approximation; exact size depends on serialization.

    // A more accurate check would serialize to a buffer, but that's expensive.

    // We rely on the storage layer's hard limit (MAX_EVENT_SIZE) during append.

    return nil

}


// internal/event/serializer.go

package event


import (

    "encoding/json"

    "sort"

)


// SerializeToJSON converts an AuditEvent to a canonical JSON byte slice.

// Keys are sorted to ensure deterministic output for hashing.

func SerializeToJSON(event AuditEvent) ([]byte, error) {
```

```
    // Use a custom marshaler to sort map keys.
    // We'll marshal to a map, sort, then re-marshal.
    // This is not highly efficient but ensures canonical form.
    // For production, consider a dedicated canonical JSON library.
    var m map[string]interface{}
    b, err := json.Marshal(event)
    if err != nil {
        return nil, err
    }
    if err := json.Unmarshal(b, &m); err != nil {
        return nil, err
    }
    // Recursively sort maps (implementation omitted for brevity).
    // Alternatively, use a library like github.com/cespare/reflectwalk.
    sortedMap := sortMapKeys(m)
    return json.Marshal(sortedMap)
}

// Helper function to recursively sort map keys (simplified skeleton).
func sortMapKeys(m map[string]interface{}) map[string]interface{} {
    // Implementation detail: iterate over keys in sorted order, rebuild map.
    // For nested maps, call recursively.
    return m // Placeholder
}
```

**D. Core Logic Skeleton Code: Hash Chain Entry Creation**

Here is the skeleton for the core logic that creates a new `HashChainEntry` when appending an event. This code belongs in the `StorageEngine` component.

```go
// internal/storage/engine.go (partial)

package storage


import (

    "crypto/sha256"

    "time"

    "github.com/yourproject/internal/event"

)


// createHashChainEntry constructs a new HashChainEntry for the given event.

// It must be called while holding the storage engine's write lock.

func (s *StorageEngine) createHashChainEntry(auditEvent event.AuditEvent) (HashChainEntry, error) {

    // TODO 1: Get the next sequence number atomically (s.activeSeq++)

    //         Ensure thread-safe increment.

    // TODO 2: Retrieve the previous HashChainEntry (for sequence-1) to get its hash.

    //         If this is the first entry (sequence == 1), use the genesis hash (32 zero bytes).

    // TODO 3: Serialize the auditEvent to its canonical byte form (e.g., using event.SerializeToJSON).

    // TODO 4: Compute EventHash = SHA256(serializedEventBytes).

    // TODO 5: Set Timestamp to the current time (time.Now().UTC()).

    // TODO 6: Construct the HashChainEntry struct with:

    //         - Sequence: the new sequence number

    //         - PreviousHash: the hash of the previous entry (from step 2)

    //         - EventHash: computed in step 4

    //         - Timestamp: from step 5

    //         - Signature: leave empty for now (optional future enhancement)

    // TODO 7: Compute the integrity hash of this new entry (hash of all its fields).

    //         This hash will become the PreviousHash for the *next* entry.

    //         Return the new entry.

    return HashChainEntry{}, nil

}


// computeEntryHash calculates the cryptographic hash of a HashChainEntry.

// This hash is used as the PreviousHash for the next entry in the chain.

func computeEntryHash(entry HashChainEntry) ([]byte, error) {

    // TODO 1: Serialize all relevant fields of the entry in a deterministic order.

    //         Example: sequence (8 bytes) + previous_hash (32 bytes) + event_hash (32 bytes) + timestamp (8 bytes, Unix nano).

    // TODO 2: Compute SHA256 of the concatenated byte slice.

    // TODO 3: Return the resulting hash.

    return nil, nil

}
```

**E. Language-Specific Hints (Go)**

- Use `uint64` for sequence numbers to ensure a vast namespace (over 18 quintillion entries).
- For `PreviousHash` and `EventHash`, use `[]byte` slices of length `SHA256_HASH_SIZE` (32). This is more efficient than storing hex strings.
- When reading/writing `time.Time` to disk, convert to UTC and store as Unix nanoseconds (`t.UnixNano()`) for compact binary storage, or use RFC3339 string for JSON.
- Use `sync/atomic` operations for incrementing the global `activeSeq` counter to ensure thread safety without a coarse-grained lock on every append.
- For the `SegmentManifest`, consider storing it as a JSON file alongside the segment data file for easy inspection.

**F. Milestone Checkpoint (Data Model)** After implementing the event model and basic hash chain logic, you should be able to run a simple test:

1. **Command:** `go test ./internal/event/... -v` and `go test ./internal/storage/... -v`
2. **Expected Output:** All tests pass, including:
   - Event validation rejects missing fields and invalid metadata keys.
   - Event serialization produces deterministic JSON (same input → same bytes).
   - `createHashChainEntry` produces an entry with correct sequence, proper previous hash linkage, and a valid `EventHash`.
3. **Manual Verification:** Write a small program that creates an `AuditEvent`, validates it, creates a `HashChainEntry`, and prints the entry's fields. Verify that the `EventHash` changes if any part of the event changes.
4. **Sign of Trouble:** If the `PreviousHash` doesn't chain correctly (e.g., entry 2's `PreviousHash` doesn't match the hash of entry 1), the hash chain verification will fail immediately. Use a debugger or print statements to compare the computed hash of entry 1 with the stored `PreviousHash` in entry 2.

---

## Component Design: Event Model & Serialization

> **Milestone(s):** Milestone 1: Audit Event Model

The Event Model & Serialization component defines the fundamental building block of our audit logging system: the atomic audit event. This component answers the critical question: *What exactly are we logging?* It establishes a rigorous, extensible schema that captures every significant action in the system with sufficient context for compliance investigations, while maintaining strict validation to ensure data quality and security.

### Mental Model: The Notary's Stamped Form

Imagine every significant action in your application requires filing a standardized government form. This form has mandatory fields (who, what, when, where, outcome) that must be filled out completely and correctly. A notary public then verifies the form's contents, stamps it with an official timestamp, and files it in a numbered sequence within a bound ledger. The form also has an "Additional Information" section where you can attach any relevant documentation (receipts, photos, witness statements) using standardized labels.

This mental model captures the essence of our audit event model:

- **Standardized Form**: Every audit event follows the same core schema, ensuring consistency.
- **Mandatory Fields**: The `actor`, `action`, `resource`, `timestamp`, and `outcome` are non-negotiable—a form missing any of these is rejected.
- **Notary Verification**: The `EventValidator` acts as the notary, checking field validity, ensuring no Personally Identifiable Information (PII) is exposed, and applying a trusted timestamp.
- **Bound Ledger**: Events are sequentially ordered (like numbered pages) in the immutable storage.
- **Attachment Section**: The `metadata` map allows attaching domain-specific context (like transaction IDs, feature flags, or performance metrics) without modifying the core form structure.

This model emphasizes that audit events are not casual debug logs; they are formal, structured records of business significance, designed for legal and compliance scrutiny.

### Interface: Event Creation and Validation

The Event Model component exposes a clear interface for creating, validating, and serializing audit events. The primary consumer is the Ingestor component, which uses these utilities to prepare events for storage. Below is the complete interface specification.

**Table: Core Data Structures for Event Model**

| Name | Type | Description |
|---|---|---|
| `AuditEvent` | Struct | The complete audit event recording a single actor action on a resource. |
| `ActorInfo` | Struct | Identifies the entity (user, system, service) that performed the action. |
| `ResourceInfo` | Struct | Identifies the entity (document, record, API endpoint) that was acted upon. |
| `RequestContext` | Struct | Captures the technical context of the request that triggered the event. |
| `EventValidator` | Struct | Validates audit events against schema rules and security policies. |

**Table: `AuditEvent` Field Details**

| Field | Type | Description |
|---|---|---|
| `Actor` | `ActorInfo` | **Required.** The entity that initiated the action. Must have at least an `ID` and `Type`. |
| `Action` | `string` | **Required.** A verb describing the operation (e.g., `"user.login"`, `"document.update"`, `"record.delete"`). Should use a consistent naming convention. |
| `Resource` | `ResourceInfo` | **Required.** The target of the action. Must have at least an `ID` and `Type`. |
| `Timestamp` | `time.Time` | **Required.** The exact UTC time when the action occurred, assigned by the system (not client-provided). |
| `Outcome` | `string` | **Required.** The result: `"success"`, `"failure"`, or domain-specific outcomes like `"denied"`, `"error"`. |
| `Context` | `RequestContext` | **Optional but recommended.** Technical request details like IP address, user agent, and correlation IDs for tracing. |
| `Metadata` | `map[string]string` | **Optional.** Key-value pairs for domain-specific context. Keys must follow naming conventions (alphanumeric with dots). Values are strings; complex objects should be JSON-encoded. |

**Table: `ActorInfo` and `ResourceInfo` Field Details**

| Field | Type | Description |
|---|---|---|
| `ID` | `string` | **Required.** Unique identifier (e.g., user ID `"u-123"`, service account `"svc-api"`, document ID `"doc-456"`). |
| `Type` | `string` | **Required.** Category (e.g., `"user"`, `"service"`, `"system"` for actors; `"document"`, `"record"`, `"endpoint"` for resources). |
| `DisplayName` | `string` | **Optional.** Human-readable name (e.g., `"John Doe"`, `"Monthly Report.pdf"`). Useful for reports but may contain PII. |

**Table: `RequestContext` Field Details**

| Field | Type | Description |
|---|---|---|
| `ClientIP` | `string` | IP address of the client making the request. May be masked or anonymized for privacy. |
| `UserAgent` | `string` | HTTP User-Agent header string identifying the client software. |
| `SessionID` | `string` | Identifier for the user's session, useful for grouping actions within a login session. |
| `CorrelationID` | `string` | Unique ID propagated across service boundaries for distributed tracing (e.g., from an HTTP header). |

**Table: Event Model Interface Methods**

| Method Name | Parameters | Returns | Description |
|---|---|---|---|
| `EventValidator.Validate` | `event AuditEvent` | `error` | Validates the event against all schema rules: required fields present, field lengths within bounds, timestamp not in future beyond allowed clock skew, metadata keys follow naming convention, and PII detection/masking is applied. |
| `SerializeToJSON` | `event AuditEvent` | `[]byte, error` | Converts the audit event to a canonical JSON representation. The output is deterministic (field order fixed) for consistent hashing. |
| `createHashChainEntry` | `event AuditEvent` | `HashChainEntry, error` | **Note:** This method belongs to the Storage Engine but is included here for context. Constructs a hash chain entry by computing the cryptographic hash of the serialized event and linking it to the previous entry's hash. |
| `MaskPII` | `fieldValue string` | `string` | **Internal utility.** Applies masking rules to potentially sensitive strings (e.g., emails, credit cards) before they are included in the audit event. Called during validation. |

### ADR: Choosing a Serialization Format

> **Decision: Use Canonical JSON for Event Serialization**
>
> - **Context**: Audit events must be serialized for storage in the Write-Ahead Log and for hash computation. The format must balance human readability, parsing performance, storage efficiency, and deterministic serialization (critical for hash consistency).
> - **Options Considered**:
>   1. **JSON (JavaScript Object Notation)**: Human-readable, widely supported, easy to debug. Requires canonicalization for deterministic hashing.
>   2. **Protocol Buffers (Protobuf)**: Binary, efficient, compact, and fast to parse. Schema evolution is robust. Less human-readable for debugging.
>   3. **Custom Binary Format**: Maximum control over layout and performance. Requires custom parser, increases complexity, and hinders interoperability.
> - **Decision**: Use **Canonical JSON** as the primary serialization format for audit events.
> - **Rationale**:
>   - **Debuggability**: During development and incident response, engineers can directly inspect log segments with standard tools ( `jq` , `cat` ). This reduces cognitive load and accelerates troubleshooting.
>   - **Ecosystem Compatibility**: JSON is universally understood. Export to compliance formats (CSV, JSON-L) is straightforward. Many SIEM systems ingest JSON natively.
>   - **Deterministic Hashing Achievable**: Using a canonical JSON library (field order sorted, whitespace eliminated) ensures the same event always produces the same byte sequence, making hash chains verifiable.
>   - **Schema Flexibility**: The `metadata` map can hold arbitrary key-value pairs without requiring Protobuf schema updates. JSON's self-describing nature accommodates this dynamic data.
>   - **Performance Adequate**: While Protobuf is faster, audit event volume is typically moderate (hundreds to thousands per second). JSON serialization/deserialization overhead is acceptable given other bottlenecks (disk I/O, hashing).
> - **Consequences**:
>   - **Storage Overhead**: JSON is less compact than Protobuf (approx. 2-3x larger). Mitigated by compression at the segment level (future extension).
>   - **Parsing Performance**: JSON parsing is slower than Protobuf for high-throughput scenarios. We accept this trade-off for improved operability.
>   - **Canonicalization Requirement**: Must enforce strict field ordering and formatting. This adds a small implementation complexity but is solved by using a dedicated library.

**Table: Serialization Format Comparison**

| Option | Pros | Cons | Chosen? |
|---|---|---|---|
| **JSON** | Human-readable, excellent tooling, easy exports, flexible for metadata | Larger size, slower parse than binary, requires canonicalization for hashing | **Yes** (primary format) |
| **Protocol Buffers** | Compact, fast parsing, strong schema evolution, deterministic by design | Not human-readable, requires schema definition for metadata, tooling less universal | No (good alternative for high-volume systems) |
| **Custom Binary** | Maximum performance, minimal size, full control | High implementation cost, opaque for debugging, interoperability challenges | No (over-optimization) |

### Common Pitfalls: Context Loss and PII

Audit event implementation is deceptively simple, but several subtle mistakes can render the entire system non-compliant or useless for investigations.

**⚠️ Pitfall 1: Losing Request Context in Asynchronous Logging**

- **Description**: Capturing an audit event in a background goroutine or async handler after the original HTTP request context has been discarded. The `RequestContext` fields ( `ClientIP` , `CorrelationID` ) become empty or default values.
- **Why It's Wrong**: Without correlation IDs, you cannot trace an action across microservices. Without client IP, you cannot investigate potential malicious sources. The audit trail loses its investigative power.
- **How to Fix**: Propagate the `RequestContext` (or at least the critical fields) synchronously through function arguments or a context.Context value attached to the event creation call. Never rely on global or thread-local storage that might be cleared.

**⚠️ Pitfall 2: Logging Raw Personally Identifiable Information (PII)**

- **Description**: Storing unmasked email addresses, credit card numbers, national IDs, or names in `DisplayName` or `metadata` fields. This violates GDPR's "right to erasure" and creates data security risks.
- **Why It's Wrong**: Audit logs are immutable and retained for years. PII stored here cannot be deleted upon user request, creating compliance violations. It also expands the attack surface if logs are breached.

- **How to Fix**: Implement a `MaskPII` utility that applies regex-based detection and masking (e.g., `"john@example.com"` → `"j***@example.com"`, `"4111-1111-1111-1111"` → `"4111-****-****-1111"`). Apply masking during `EventValidator.Validate`. For `DisplayName`, consider using a non-PII alternative (username instead of full name).

**⚠️ Pitfall 3: Schema Evolution Without Versioning**

- **Description**: Adding a new required field to the `AuditEvent` struct without considering how existing events (already stored) will be read. Deserialization of old events fails or produces invalid objects.
- **Why It's Wrong**: Breaks backward compatibility. The query engine cannot read historical data, making the audit trail incomplete. This is catastrophic for compliance where historical records are required.
- **How to Fix**: Never add required fields to the core schema after initial deployment. Add optional fields only. If a breaking change is unavoidable, introduce an event version field and maintain deserialization logic for all supported versions. Consider storing the schema version in each event's `metadata`.

**⚠️ Pitfall 4: Using Client-Provided Timestamps for Ordering**

- **Description**: Trusting the `Timestamp` field provided by the client application (which might have clock drift or be maliciously set). Using this timestamp to determine sequence in the hash chain.
- **Why It's Wrong**: Clock skew between servers and clients destroys temporal ordering. A malicious actor could backdate events to hide actions. The hash chain's sequence (which depends on append order) would not match the event timestamps, confusing investigations.
- **How to Fix**: The `AuditEvent.Timestamp` should be set by the **ingesting service** using its own synchronized clock (NTP) at the moment of receipt. The hash chain uses a separate, monotonically increasing `Sequence` number assigned by the storage engine for ordering. Store both values.

**⚠️ Pitfall 5: Unbounded Metadata Size**

- **Description**: Allowing the `metadata` map to contain arbitrarily large values or too many keys, causing individual events to exceed reasonable size limits (`MAX_EVENT_SIZE`).
- **Why It's Wrong**: Causes storage inefficiency, memory issues during query processing, and potential denial-of-service attacks where an attacker floods the log with huge events.
- **How to Fix**: Enforce limits in `EventValidator.Validate`: total serialized event size < `MAX_EVENT_SIZE` (64KB), maximum number of metadata keys (e.g., 50), and maximum length per value (e.g., 1024 characters). Reject events that exceed these limits.

## Implementation Guidance for Event Model

This section provides concrete implementation code in Go, following the design outlined above.

**A. Technology Recommendations Table**

| Component | Simple Option | Advanced Option |
|---|---|---|
| Serialization | `encoding/json` with `json.Marshal` | `github.com/goccy/go-json` for faster performance |
| Canonical JSON | Custom marshaler with sorted keys | `github.com/segmentio/encoding/json` for canonical + fast |
| PII Detection | Regular expressions (`regexp` package) | Specialized library like `github.com/sshleifer/pii-detector` |
| Validation | Manual field checks in `Validate` method | Schema validation with `github.com/xeipuuv/gojsonschema` |

**B. Recommended File/Module Structure**

Place the event model in a dedicated package to separate concerns and allow import by other components (Ingestor, Storage, Query).

```
audit-log-system/
├── cmd/
│   └── server/
│       └── main.go
├── internal/
│   ├── event/                  # Event model package
│   │   ├── event.go            # AuditEvent, ActorInfo, etc. structs
│   │   ├── validator.go        # EventValidator implementation
│   │   ├── serialization.go    # SerializeToJSON, helpers
│   │   ├── pii.go              # MaskPII utility
│   │   └── event_test.go       # Unit tests
│   ├── storage/                # Storage engine (Milestone 2)
│   └── query/                  # Query engine (Milestone 3)
└── pkg/
    └── utils/                  # Shared utilities (e.g., canonical JSON)
```

**C. Infrastructure Starter Code**

**Canonical JSON Utility (`pkg/utils/canonical.go`)**: A complete, reusable helper to ensure deterministic JSON serialization.

```go
package utils

import (
    "bytes"
    "encoding/json"
    "sort"
)

// CanonicalJSONMarshal converts a Go value to canonical JSON bytes.
// It sorts map keys alphabetically to ensure deterministic output.
func CanonicalJSONMarshal(v interface{}) ([]byte, error) {
    // First, marshal to a regular JSON map to apply key sorting.
    var data map[string]interface{}
    b, err := json.Marshal(v)
    if err != nil {
        return nil, err
    }
    if err := json.Unmarshal(b, &data); err != nil {
        return nil, err
    }

    // Recursively sort keys in the map.
    sortedData := sortMapKeys(data)

    // Marshal the sorted map with no indentation.
    var buf bytes.Buffer
    encoder := json.NewEncoder(&buf)
    encoder.SetEscapeHTML(false) // Do not escape HTML characters
    encoder.SetIndent("", "")    // No whitespace
    if err := encoder.Encode(sortedData); err != nil {
        return nil, err
    }
    // encoder.Encode adds a newline; trim it for consistency.
    return bytes.TrimSpace(buf.Bytes()), nil
}

// sortMapKeys recursively sorts all map keys in a nested data structure.
func sortMapKeys(v interface{}) interface{} {
    switch v := v.(type) {
    case map[string]interface{}:
        sorted := make(map[string]interface{}, len(v))
        keys := make([]string, 0, len(v))
```

```
        for k := range v {
            keys = append(keys, k)
        }
        sort.Strings(keys)
        for _, k := range keys {
            sorted[k] = sortMapKeys(v[k])
        }
        return sorted
    case []interface{}:
        for i, elem := range v {
            v[i] = sortMapKeys(elem)
        }
        return v
    default:
        return v
    }
}
```

**D. Core Logic Skeleton Code**

**Event Struct Definitions (`internal/event/event.go`)**: The core data structures.

```go
package event

import (
    "time"
)

// ActorInfo identifies the entity that performed an action.
type ActorInfo struct {
    ID          string `json:"id"`
    Type        string `json:"type"`
    DisplayName string `json:"display_name,omitempty"`
}

// ResourceInfo identifies the target of an action.
type ResourceInfo struct {
    ID          string `json:"id"`
    Type        string `json:"type"`
    DisplayName string `json:"display_name,omitempty"`
}

// RequestContext captures technical details about the request.
type RequestContext struct {
    ClientIP     string `json:"client_ip,omitempty"`
    UserAgent    string `json:"user_agent,omitempty"`
    SessionID    string `json:"session_id,omitempty"`
    CorrelationID string `json:"correlation_id,omitempty"`
}

// AuditEvent is the complete record of a single auditable action.
type AuditEvent struct {
    Actor     ActorInfo     `json:"actor"`
    Action    string        `json:"action"`
    Resource  ResourceInfo  `json:"resource"`
    Timestamp time.Time     `json:"timestamp"`
    Outcome   string        `json:"outcome"`
    Context   RequestContext `json:"context,omitempty"`
    Metadata  map[string]string `json:"metadata,omitempty"`
}

// NewAuditEvent creates a new AuditEvent with the current UTC timestamp.
// It's a convenience builder; validation must still be performed.
func NewAuditEvent(actor ActorInfo, action string, resource ResourceInfo, outcome string) *AuditEvent {
```

```
    return &AuditEvent{
        Actor:     actor,
        Action:    action,
        Resource:  resource,
        Timestamp: time.Now().UTC(),
        Outcome:   outcome,
        Metadata:  make(map[string]string),
    }
}
```

**Event Validator ( `internal/event/validator.go` )**: Skeleton with TODOs for the validation logic.

```go
package event

import (
    "errors"
    "regexp"
    "time"
    "unicode/utf8"
)

// EventValidator validates audit events against schema and security rules.
type EventValidator struct {
    maxEventSize   int           // MAX_EVENT_SIZE constant
    maxClockSkew   time.Duration // Maximum allowed future timestamp (e.g., 5 minutes)
    metadataKeyRegex *regexp.Regexp // Regex for allowed metadata keys (e.g., ^[a-z][a-z0-9_.]*$)
}

// NewEventValidator creates a new validator with default or provided configuration.
func NewEventValidator(maxEventSize int, maxClockSkew time.Duration) *EventValidator {
    // Compile regex for metadata keys: lowercase alphanumeric with dots and underscores.
    re := regexp.MustCompile(`^[a-z][a-z0-9_.]*$`)
    return &EventValidator{
        maxEventSize:   maxEventSize,
        maxClockSkew:   maxClockSkew,
        metadataKeyRegex: re,
    }
}

// Validate checks the audit event for correctness and security.
// It returns an error describing the first validation failure.
func (v *EventValidator) Validate(event AuditEvent) error {
    // TODO 1: Check required fields: Actor.ID, Actor.Type, Action, Resource.ID, Resource.Type, Timestamp, Outcome.
    //         If any are empty (or zero for Timestamp), return an error specifying the missing field.

    // TODO 2: Validate field lengths and content:
    //   - Actor.ID, Resource.ID: ensure they are not longer than 255 characters.
    //   - Action: ensure it matches a pattern like "domain.verb" (e.g., "user.login").
    //   - Outcome: ensure it's one of the allowed values ("success", "failure", "denied", "error").

    // TODO 3: Validate Timestamp:
    //   - Ensure it's not zero time.
    //   - Ensure it's not in the future beyond allowed clock skew (use v.maxClockSkew).
    //   Hint: compare event.Timestamp with time.Now().UTC().Add(v.maxClockSkew).
```

```
    // TODO 4: Validate Metadata:
    //   - Ensure each key matches v.metadataKeyRegex.
    //   - Ensure each value is a valid UTF-8 string and length <= 1024 characters.
    //   - Ensure total number of metadata keys <= 50.

    // TODO 5: Apply PII masking to sensitive fields:
    //   - Call MaskPII on event.Actor.DisplayName, event.Resource.DisplayName, and event.Context.ClientIP.
    //   - Update the fields in-place with the masked version.
    //   Note: This step modifies the event. Consider if you want to work on a copy.

    // TODO 6: Calculate serialized size:
    //   - Serialize the event using SerializeToJSON (which uses canonical JSON).
    //   - If the size exceeds v.maxEventSize, return an error.
    //   Hint: Use len(serializedBytes) to get size.

    return nil // Placeholder
}
```

**PII Masking Utility ( `internal/event/pii.go` )**: Starter code for masking sensitive data.

```go
package event

import (
    "regexp"
    "strings"
)

// MaskPII applies masking rules to a string to redact potentially sensitive information.
// It returns the masked string. If no PII patterns are matched, returns the original.
func MaskPII(input string) string {
    if input == "" {
        return input
    }

    // TODO 1: Email address masking: "john.doe@example.com" -> "j***@example.com"
    //   Use regexp `^[a-zA-Z0-9._%+-]+@[a-zA-Z0-9.-]+\.[a-zA-Z]{2,}$`
    //   Mask the local part (before @) except first character.

    // TODO 2: Credit card number masking (simplified): "4111-1111-1111-1111" -> "4111-****-****-1111"
    //   Use regexp to detect groups of 4 digits separated by dashes or spaces.
    //   Mask the middle groups, keep first and last group.

    // TODO 3: IP address masking (optional): "192.168.1.100" -> "192.168.1.***"
    //   This is less common but may be required for privacy policies.

    // TODO 4: For any other PII patterns (phone numbers, SSN, etc.), implement as needed.

    // If no patterns matched, return original.
    return input
}
```

**Serialization (`internal/event/serialization.go`)**: Integration with canonical JSON.

```go
package event

import (
    "your-project/pkg/utils" // Adjust import path
)

// SerializeToJSON converts an AuditEvent to a canonical JSON byte slice.
// It uses the canonical JSON utility to ensure deterministic output for hashing.
func SerializeToJSON(event AuditEvent) ([]byte, error) {
    // TODO: Call utils.CanonicalJSONMarshal with the event and return the result.
    // Handle any serialization error.
    return nil, nil // Placeholder
}
```

**E. Language-Specific Hints**

- **Time Handling**: Always use `time.Time` in UTC. Convert immediately with `time.Now().UTC()`. When storing timestamps in JSON, ensure the marshaling format is RFC3339 by using `time.Time`'s default JSON marshaling.
- **Maps for Metadata**: Use `map[string]string` for simplicity. If you need complex values, consider JSON-encoding them as strings in the map.
- **Regular Expressions**: Compile regex patterns once in `init()` or constructor ( `NewEventValidator` ) and reuse them for performance.
- **Error Messages**: Make validation errors descriptive and actionable (e.g., `"missing required field: actor.id"` rather than `"validation failed"` ).

**F. Milestone Checkpoint**

After implementing the Event Model component, you should be able to run unit tests that validate event creation, serialization, and validation.

1. **Run the tests**: Execute `go test ./internal/event/... -v` from the project root.
2. **Expected Output**: All tests should pass. At minimum, write tests for:
    - Creating a valid `AuditEvent` with `NewAuditEvent`.
    - `Validate` accepting a correctly filled event.
    - `Validate` rejecting events with missing required fields.
    - `SerializeToJSON` producing deterministic output (the same event serialized twice yields identical bytes).
    - `MaskPII` correctly masking email addresses and leaving non-PII unchanged.
3. **Manual Verification**: Create a simple Go program that constructs an event, validates it, and prints the canonical JSON. Verify the JSON has sorted keys and no whitespace.

**Signs of Trouble**:

- Validation passes for events with empty `Actor.ID` → Check required field validation.
- Serialized JSON fields appear in random order → Ensure you're using `CanonicalJSONMarshal`.
- PII masking not working → Test your regex patterns with a variety of inputs.

By completing this component, you establish the foundational data structure that will flow through the entire audit logging pipeline. The rigor applied here ensures downstream components can rely on well-formed, secure data.

## Component Design: Immutable Storage Engine

> **Milestone(s):** Milestone 2: Immutable Storage with Hash Chain

The **Immutable Storage Engine** is the core of our tamper-evident audit logging system. Its primary responsibility is to guarantee that once an audit event is recorded, it can never be modified or deleted without detection. This component transforms validated `AuditEvent` objects into a cryptographically-secure, append-only sequence that forms the authoritative record of all system activity.

### Mental Model: The Blockchain Ledger Page

Imagine a bound, tamper-proof ledger used by a bank to record all financial transactions. Each page of this ledger (a **segment**) holds a fixed number of transactions. Once a page is full, it is sealed with wax and a new page is started. Crucially, the first entry on each new page includes a special reference: the wax seal impression (hash) from the *previous page*. Within a page, each transaction entry also includes a fingerprint of the entry immediately above it.

Now, picture an auditor arriving. To verify the ledger's integrity, they don't need to check every single transaction's handwriting. Instead, they can:

1. Start at any known-good entry (e.g., the bank's initial seal).
2. Recalculate the fingerprint for that entry.
3. Compare it to the fingerprint recorded in the *next* entry.
4. Move to the next entry and repeat.

If even a single character in a single historical transaction is altered, the recalculated fingerprint for that entry will change. This mismatch will cascade forward, breaking the chain of fingerprints in all subsequent entries, alerting the auditor to tampering. Our storage engine works exactly like this ledger, using cryptographic hash functions as unforgeable digital wax seals, creating an immutable, verifiable chain of audit events.

### Interface: Append and Read Operations

The storage engine exposes a minimal, focused API centered around the `StorageEngine` struct. Its methods guarantee atomic writes, provide sequential reads, and enable integrity verification.

| Method | Parameters | Returns | Description |
|---|---|---|---|
| `AppendEvent(event AuditEvent)` | `event`: The validated audit event to store. | `(HashChainEntry, error)` | The core write operation. Validates the event's timestamp is not in the future beyond allowed skew, assigns the next monotonic sequence number, constructs a `HashChainEntry` linking to the previous entry's hash, appends it to the active Write-Ahead Log (WAL), and updates in-memory state. Returns the created entry or an error. |
| `ReadEvent(sequence uint64)` | `sequence`: The unique, monotonically increasing sequence number of the desired event. | `(AuditEvent, error)` | Retrieves a specific `AuditEvent` by its sequence number. This involves locating which segment file contains the event (using the catalog), reading the raw `HashChainEntry` from that segment, and deserializing the embedded event data. Returns an error if the sequence is out of range or the segment file is corrupt. |
| `VerifyChain(startSeq uint64)` | `startSeq`: The sequence number from which to begin verification. Use `0` to verify the entire chain from genesis. | `(bool, error)` | Performs cryptographic integrity verification of the hash chain starting from the specified sequence. It reads entries sequentially, recalculates the hash for each entry (using `computeEntryHash`), and compares it to the `PreviousHash` stored in the *next* entry. Returns `true` only if the entire verified chain is intact. Returns `false` and an error describing the point of failure if tampering is detected. |
| `GetActiveSegmentInfo()` | None | `SegmentManifest` | Returns metadata about the currently active segment (the one accepting new writes). This includes its `SegmentID`, `StartTime`, `StartSequence`, and current size. Useful for monitoring and triggering segment rotation. |
| `OpenWAL(path string)` (in `wal` package) | `path`: Filesystem path to the WAL file. | `(*WAL, error)` | Opens an existing WAL file or creates a new one. Writes a file header (`walHeader`) for identification. This is a lower-level utility used by the `SegmentManager`. |
| `WAL.Append(record []byte)` | `record`: A serialized byte slice to append. | `(int64, error)` | Appends the record to the end of the WAL file. The return value is the file **offset** where this record begins. **Crucially, this method calls `fsync` (or `File.Sync()` in Go) after the write to ensure the data is flushed to durable storage** before returning. This guarantees write durability. |
| `WAL.ReadAt(offset int64)` | `offset`: The byte offset within the WAL file to read from. | `([]byte, error)` | Reads a complete record from the specified offset. The WAL file format must encode record boundaries (e.g., a length prefix) so this method can read the correct number of bytes. |
| `WAL.Close()` | None | `error` | Closes the WAL file, releasing the file descriptor. For an active segment, this should only be called during graceful shutdown or segment rotation. |

### ADR: Hash Chain vs. Merkle Tree for Integrity

> **Decision: Use a Linear Hash Chain for Tamper Evidence**
>
> **Context**: We need a mechanism to cryptographically prove that the audit log has not been altered. The primary operations are sequential appends and verification of the entire log's history (or large contiguous sections of it). We also need to support efficient log segmentation (file rotation) and simple proof generation for compliance auditors.
>
> **Options Considered**:
>
> 1. **Linear Hash Chain**: Each log entry contains the cryptographic hash of the *previous* entry. Entry N's hash is computed as `Hash(Data_N + Hash_{N-1})`. The chain starts with a genesis hash.
> 2. **Merkle Tree**: Entries are leaves of a binary hash tree. The root hash (stored in a secure location) commits to the entire set of entries. Proofs for individual entries are `O(log n)` in size.
>
> **Decision**: We will implement a **linear hash chain**.
>
> **Rationale**:
>
> - **Simplicity for Append-Only Workload**: The hash chain perfectly matches our write pattern. Appending a new entry requires only the previous entry's hash, which is readily available in memory. There is no need to rebalance trees or manage complex node structures.
> - **Efficient Full Verification**: Compliance often requires verifying the integrity of the entire log or large time ranges. A hash chain allows for a single, sequential pass from a trusted starting point (e.g., a signed genesis block or a trusted period) to the present, which is `O(n)` and cache-friendly. Verifying a Merkle tree requires traversing the tree for each verification, which, while `O(log n)` per proof, is more complex for a full scan.
> - **Natural Segmentation**: When a segment file is sealed (e.g., after reaching `DEFAULT_SEGMENT_SIZE`), the *last hash* of that segment becomes the *previous hash* for the first entry of the next segment. This creates a continuous chain across physical file boundaries, which is conceptually simple to manage and verify.
> - **Adequate for Primary Use Case**: Our primary requirement is **tamper-detection**, not efficient proofs for random entries. While a Merkle tree excels at providing compact proofs for specific entries ("proof of inclusion"), our audit scenarios typically involve verifying chronological sequences. The hash chain provides this directly.
>
> **Consequences**:
>
> - **Enables**: Simple implementation, fast sequential writes, straightforward cross-segment chaining, and efficient sequential verification.
> - **Trade-offs**: Generating a proof for a single, arbitrary entry requires hashing all preceding entries ( `O(n)` ). However, this is acceptable because our query patterns are time-based, not random-access, and full-chain verification is the common compliance operation.

**Comparison of Integrity Structures**

| Option | Pros | Cons | Chosen? |
|---|---|---|---|
| **Linear Hash Chain** | Simple to implement and understand. Perfect for sequential writes and verification. Natural continuity across segmented files. Low overhead per entry (one hash). | Proof for a single entry requires `O(n)` computation. Insertion or deletion of an entry (theoretically impossible in our immutable log) would break the entire subsequent chain. | **Yes** |
| **Merkle Tree** | Efficient `O(log n)` proofs for individual entries. Parallelizable hash computation. Can support efficient verification of subset integrity. | More complex implementation (tree management, balancing). Root hash must be securely stored and published. Appending a single leaf requires updating `O(log n)` nodes, complicating concurrency and segmentation. | No |

## Common Pitfalls

⚠️ **Pitfall: Forgetting `fsync` - The "Phantom Write" Description**: A developer implements `WAL.Append` by writing data to a file using standard I/O calls but neglects to call `fsync()` (or `File.Sync()` in Go) before returning success to the caller. **Why it's wrong**: Modern operating systems heavily cache disk writes. A successful `write()` system call only guarantees data is transferred to the OS kernel's page cache, not to the physical disk. If the machine loses power or crashes before the kernel flushes the cache, the "appended" data is lost forever. The hash chain will have advanced in memory, but the corresponding entry will be missing from the durable storage, causing a permanent, irrecoverable break in the chain during verification. **How to fix**: **Always call `File.Sync()` after writing a `HashChainEntry` to the WAL.** The `WAL.Append` method in our interface must guarantee this. Treat the return from `Append` as a durable promise.

⚠️ **Pitfall: Using System Time for Ordering - The "Time Traveler's Dilemma" Description**: Relying solely on the `AuditEvent.Timestamp` (which comes from the client or application server) to determine the sequence of events in the log. **Why it's wrong**: System clocks can skew, drift, or be maliciously adjusted. Two events could be submitted with timestamps that are out of their actual occurrence order. If ordering is based on timestamp, the chronological narrative of "who did what when" becomes unreliable, violating a core audit principle. **How to fix**: Use a **monotonically increasing sequence number** assigned by the storage engine at write time as the primary, immutable order. The `HashChainEntry.Sequence` field is this number. The event's timestamp is stored as metadata for human readability and time-range queries, but the sequence number is the source of truth for *logical* ordering and hash chain construction.

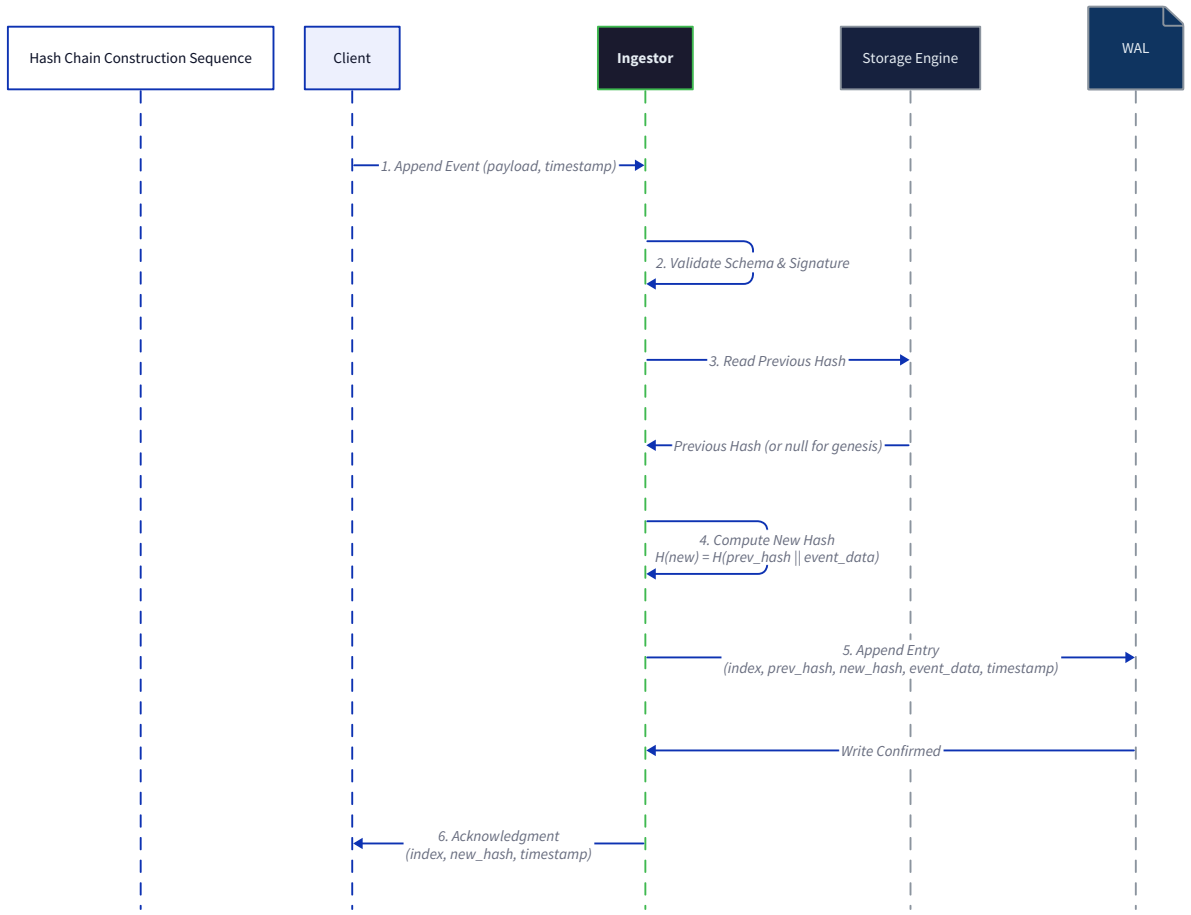
⚠️ **Pitfall: Breaking the Chain During Backup/Restore Description**: A system administrator copies the segment files to a backup location but misses the latest active WAL file, or restores files from different points in time, mixing segment sequences. **Why it's wrong**: The hash chain's integrity depends on every link. If `Segment_2` is backed up, but its preceding `Segment_1` is not, the `PreviousSegmentID` and `RootHash` in `Segment_2`'s manifest will point to a non-existent predecessor. Verification will fail. Similarly, restoring an old segment file over a newer one will cause a mismatch in the chain forward from that point. **How to fix**: Treat the entire set of segment files and the active WAL as a **unit**. Backup procedures must: 1) Seal the current active segment (rotate it) to create a stable file, 2) Include *all* sealed segment manifests and data files, and 3) Document the last sequence number backed up. Restore must place all files in their original directory structure. Consider creating a snapshot archive that includes a signature of the final chain state.

### Component Responsibilities and Data Flow

The `StorageEngine` coordinates several sub-components:

1. **Write-Ahead Log (WAL)**: A single, actively written file that provides durability. All new `HashChainEntry` records are appended here first. It's a simple wrapper around an `os.File` with `Append` and `ReadAt` methods.
2. **Segment Manager**: Manages the lifecycle of log segments. It monitors the size of the active WAL. When it exceeds `DEFAULT_SEGMENT_SIZE`, the manager:
   - Seals the active WAL (renames `.wal` to `.seg`).
   - Calculates the final `RootHash` for the sealed segment (the hash of the last entry).
   - Writes a `SegmentManifest` file (`.manifest`) containing the segment's metadata (`SegmentID`, `StartSequence`, `EndSequence`, `RootHash`, `PreviousSegmentID`).
   - Opens a new WAL file for the next segment, initializing its first entry's `PreviousHash` to the sealed segment's `RootHash`.
3. **Catalog**: An in-memory index (loaded on startup from all `.manifest` files) that maps time ranges and sequence ranges to segment files. This allows `ReadEvent` and the `QueryEngine` to quickly locate which physical file(s) contain the requested data.

The process to append an event, detailed in the sequence diagram



, follows these steps:

1. **Receive & Validate**: The `Ingestor` receives an `AuditEvent` and calls `EventValidator.Validate()`.
2. **Assign Sequence**: `StorageEngine` atomically increments its `activeSeq` counter to generate a new, unique sequence number for the event.
3. **Fetch Previous Hash**: The engine retrieves the hash from the most recently written `HashChainEntry` (or a genesis hash if this is the first entry).
4. **Construct Entry**: It calls `createHashChainEntry(auditEvent)`. This function:
   - Serializes the `AuditEvent` to a canonical format (e.g., Canonical JSON).
   - Computes the `EventHash` of the serialized bytes.
   - Creates a `HashChainEntry` struct, populating `Sequence`, `PreviousHash`, `EventHash`, and `Timestamp`.
   - Computes the *entry's own hash* by calling `computeEntryHash(entry)`. This hash will become the `PreviousHash` for the *next* entry.
5. **Append to WAL**: The serialized `HashChainEntry` is passed to `WAL.Append()`. This method writes the bytes and calls `fsync`.
6. **Update State**: The engine updates its in-memory state (e.g., last hash, last sequence).
7. **Acknowledge**: The created `HashChainEntry` is returned to the caller as an acknowledgment token.

### Segment Lifecycle

A segment file progresses through distinct states, as shown in the state machine

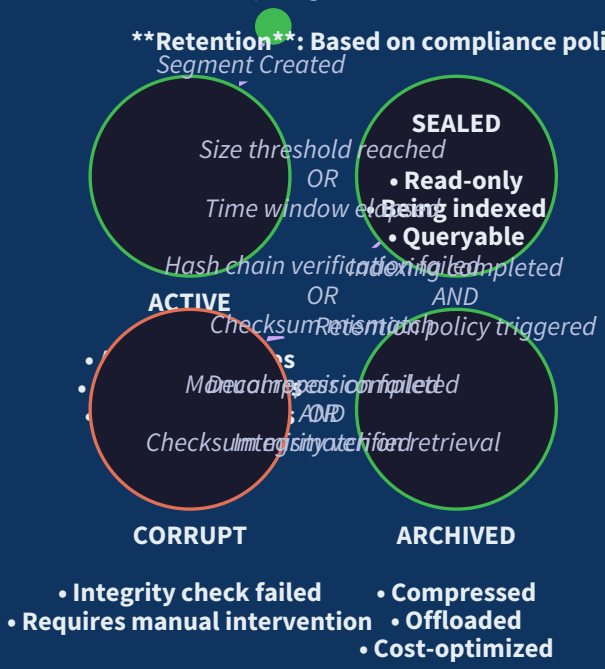
## Transition Triggers

**Size/Time**: Segment reaches configurable size limit or time window expires

**Completion**: Background indexing completes, making segment queryable

**Error**: Cryptographic hash chain verification fails or checksum mismatches

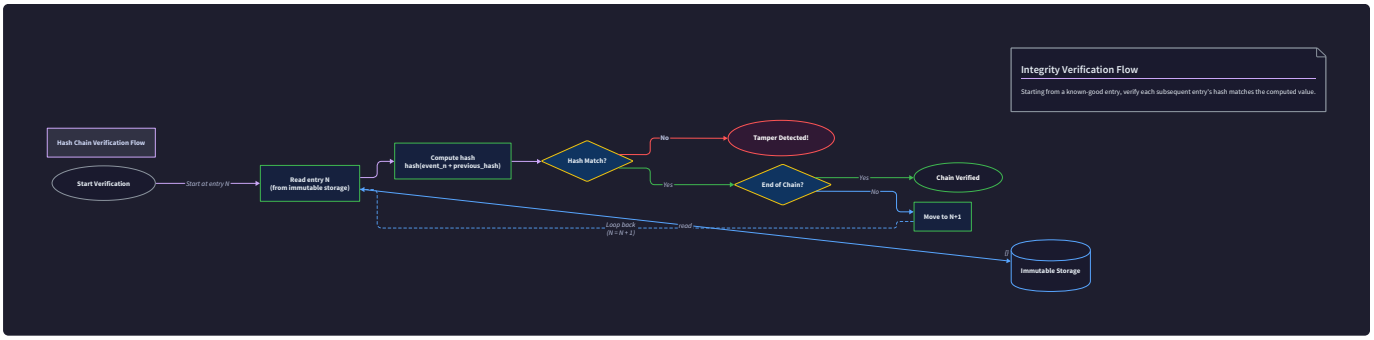
**Retention**: Based on compliance policies, older segments are archived



| Current State | Event | Next State | Actions Taken |
|---|---|---|---|
| **ACTIVE** | `WAL.Size() >= DEFAULT_SEGMENT_SIZE` | **SEALING** | Stop accepting new writes to this WAL. Flush final writes. Compute final `RootHash`. |
| **SEALING** | Manifest file successfully written | **SEALED** | Rename `.wal` to `.seg`. Write `.manifest` file. Update Catalog. Open new ACTIVE WAL for next segment. |
| **ACTIVE** | System Shutdown Signal | **SEALED** | Perform same sealing steps, but segment may be smaller than the max size. |
| **SEALED** | Background indexing completes | **INDEXED** | Query Engine builds secondary indexes (e.g., for actor, resource) and stores them alongside the segment. |
| **SEALED/INDEXED** | Aging Policy triggers (e.g., > 30 days old) | **ARCHIVED** | Compress the `.seg` file (e.g., using gzip). Optionally move to cheaper storage. Update manifest location. |
| **ANY** | Integrity check ( `VerifyChain` ) fails | **CORRUPT** | Quarantine the segment file. Raise critical alert. Log the exact sequence number and hash mismatch. |
| **CORRUPT** | Administrator repair/recovery | **SEALED** | After manual investigation and potential restoration from backup, re-verify the chain. |

## Integrity Verification Algorithm

The `VerifyChain` method implements the process shown in the flowchart




. To verify the chain from sequence number `startSeq`:

1. **Initialize**: If `startSeq` is 0, begin with a known **genesis hash** (e.g., a hard-coded value or a hash from a trusted, signed manifest). Otherwise, read the `HashChainEntry` for `startSeq-1` to obtain the trusted `PreviousHash` to verify against.
2. **Iterate**: For each entry `E_n` starting at sequence `startSeq`: a. Read the entry `E_n` from its segment file via `ReadEvent`. b. **Recompute Hash**: Calculate `computed_hash = computeEntryHash(E_n)`. This hash is based on `E_n`'s data, including its stored `PreviousHash` field. c. **Fetch Next Link**: Read the *next* entry `E_{n+1}`. The `PreviousHash` field stored in `E_{n+1}` is the **stored reference** to `E_n`. d. **Compare**: Compare `computed_hash` with the `PreviousHash` stored in `E_{n+1}`. e. **Check**: If they match, move to `E_{n+1}` and repeat. If they do **not** match, tampering is detected at entry `E_n`. Halt and return `false` with details.
3. **Finalize**: If the end of the chain is reached without mismatch, return `true`.

This algorithm ensures that altering any bit in any historical entry invalidates the hash chain from that point forward.

## Implementation Guidance for Storage Engine

### A. Technology Recommendations

| Component | Simple Option | Advanced Option |
|---|---|---|
| **File I/O & fsync** | Go's standard `os` package ( `File.Write` , `File.Sync` ). | Use `syscall.Fsync` for more control or platform-specific optimizations. |
| **Hash Function** | `crypto/sha256` for `SHA256_HASH_SIZE` hashes. | Use `crypto/sha512` for stronger security, accepting increased storage overhead. |
| **Concurrency Control** | `sync.RWMutex` within `StorageEngine` to protect `activeSeq` and `activeWAL`. | Fine-grained locking per segment or lock-free structures for higher throughput. |
| **Manifest Storage** | JSON files ( `.manifest` ) for readability. | Protocol Buffers for compactness and faster parsing on startup. |

### B. Recommended File/Module Structure

```
project-root/
├── internal/
│   ├── event/                # From Milestone 1
│   │   ├── event.go          # AuditEvent, ActorInfo, etc.
│   │   └── validator.go      # EventValidator
│   ├── storage/              # Milestone 2: Immutable Storage Engine
│   │   ├── engine.go         # StorageEngine (main coordinator)
│   │   ├── hashchain.go      # HashChainEntry, createHashChainEntry, computeEntryHash
│   │   ├── wal/              # Write-Ahead Log subsystem
│   │   │   ├── wal.go        # WAL struct and methods (Append, ReadAt)
│   │   │   └── wal_test.go
│   │   ├── segment/          # Segment management
│   │   │   ├── manager.go    # SegmentManager
│   │   │   ├── manifest.go   # SegmentManifest struct, load/save
│   │   │   └── catalog.go    # Catalog (in-memory segment index)
│   │   └── integrity.go      # Verification logic (VerifyChain implementation)
│   └── query/                # Milestone 3
└── pkg/
    └── auditlog/             # Public API (e.g., SubmitAuditEvent)
```

**C. Infrastructure Starter Code: WAL Wrapper** Below is a complete, ready-to-use implementation of the `WAL` wrapper. It handles file creation, appends with fsync, and record-based reading.

GO

```go
// internal/storage/wal/wal.go

package wal

import (
    "encoding/binary"
    "fmt"
    "io"
    "os"
    "path/filepath"
    "sync"
)

const (
    walHeader    = "AUDITWALv1" // 10-byte header
    maxRecordSize = 1024 * 1024 // 1MB
)

// WAL is an append-only write-ahead log file.
type WAL struct {
    file     *os.File
    filePath string
    mu       sync.Mutex
    offset   int64 // Current write offset (end of file)
}

// OpenWAL opens or creates a WAL file at the given path.
func OpenWAL(path string) (*WAL, error) {
    // Ensure directory exists
    if err := os.MkdirAll(filepath.Dir(path), 0755); err != nil {
        return nil, fmt.Errorf("create wal directory: %w", err)
    }

    // Open file in append mode, create if not exists
    file, err := os.OpenFile(path, os.O_RDWR|os.O_CREATE|os.O_APPEND, 0644)
    if err != nil {
        return nil, fmt.Errorf("open wal file: %w", err)
    }

    w := &WAL{
        file:     file,
        filePath: path,
    }
```

```
    // Get current file size for offset
    info, err := file.Stat()
    if err != nil {
        file.Close()
        return nil, fmt.Errorf("stat wal file: %w", err)
    }
    w.offset = info.Size()

    // If file is new, write header
    if w.offset == 0 {
        if _, err := file.Write([]byte(walHeader)); err != nil {
            file.Close()
            return nil, fmt.Errorf("write wal header: %w", err)
        }
        w.offset = int64(len(walHeader))
        if err := file.Sync(); err != nil {
            file.Close()
            return nil, fmt.Errorf("sync wal header: %w", err)
        }
    } else {
        // Verify header for existing file
        header := make([]byte, len(walHeader))
        if _, err := file.ReadAt(header, 0); err != nil {
            file.Close()
            return nil, fmt.Errorf("read wal header: %w", err)
        }
        if string(header) != walHeader {
            file.Close()
            return nil, fmt.Errorf("invalid wal header: got %s", string(header))
        }
    }

    return w, nil
}

// Append writes a record to the WAL and flushes to disk.
// Returns the offset at which the record was written.
func (w *WAL) Append(record []byte) (int64, error) {
    w.mu.Lock()
    defer w.mu.Unlock()
```

```
    if len(record) > maxRecordSize {
        return 0, fmt.Errorf("record size %d exceeds max %d", len(record), maxRecordSize)
    }

    // Write length prefix (8 bytes, little-endian)
    lenBuf := make([]byte, 8)
    binary.LittleEndian.PutUint64(lenBuf, uint64(len(record)))
    if _, err := w.file.Write(lenBuf); err != nil {
        return 0, fmt.Errorf("write length prefix: %w", err)
    }

    // Write record data
    if _, err := w.file.Write(record); err != nil {
        return 0, fmt.Errorf("write record data: %w", err)
    }

    // Flush to disk
    if err := w.file.Sync(); err != nil {
        return 0, fmt.Errorf("fsync wal: %w", err)
    }

    startOffset := w.offset
    w.offset += int64(8 + len(record)) // Update offset for next write

    return startOffset, nil
}

// ReadAt reads a complete record starting at the given offset.
func (w *WAL) ReadAt(offset int64) ([]byte, error) {
    w.mu.Lock()
    defer w.mu.Unlock()

    // Read length prefix
    lenBuf := make([]byte, 8)
    if _, err := w.file.ReadAt(lenBuf, offset); err != nil {
        return nil, fmt.Errorf("read length prefix at offset %d: %w", offset, err)
    }
    recordLen := binary.LittleEndian.Uint64(lenBuf)

    if recordLen > maxRecordSize {
        return nil, fmt.Errorf("record length %d at offset %d exceeds max", recordLen, offset)
    }
```

```
    // Read record data
    record := make([]byte, recordLen)
    if _, err := w.file.ReadAt(record, offset+8); err != nil {
        return nil, fmt.Errorf("read record data at offset %d: %w", offset+8, err)
    }

    return record, nil
}

// Close closes the WAL file.
func (w *WAL) Close() error {
    w.mu.Lock()
    defer w.mu.Unlock()
    return w.file.Close()
}
```

**D. Core Logic Skeleton Code**

**1. Hash Chain Entry Creation and Hashing**

GO

```go
// internal/storage/hashchain.go

package storage


import (

    "crypto/sha256"

    "encoding/json"

    "time"


    "github.com/yourproject/internal/event"

)


// HashChainEntry links an event to the previous entry's hash.

type HashChainEntry struct {

    Sequence     uint64    `json:"sequence"`

    PreviousHash []byte    `json:"previous_hash"`

    EventHash    []byte    `json:"event_hash"`

    Timestamp    time.Time `json:"timestamp"`

    // Signature can be added later for non-repudiation

    Signature    []byte    `json:"signature,omitempty"`

}


// createHashChainEntry constructs a new HashChainEntry for the given event.

// It requires the previous entry's hash and the next sequence number.

func createHashChainEntry(

    auditEvent event.AuditEvent,

    prevHash []byte,

    sequence uint64,

) (HashChainEntry, error) {

    // TODO 1: Serialize the auditEvent to canonical JSON.

    //         Use a JSON library with deterministic field ordering.

    //         Hint: Use json.Marshal but ensure map keys are sorted.


    // TODO 2: Compute the SHA-256 hash of the serialized event bytes.

    //         This becomes the EventHash field.


    // TODO 3: Create a HashChainEntry struct instance.

    //         Populate: Sequence, PreviousHash (from parameter), EventHash, Timestamp (use event timestamp).

    //         Leave Signature as nil for now.


    // TODO 4: Compute the hash of this entire HashChainEntry.

    //         First, serialize the entry (excluding Signature field for hash computation).

    //         Then compute SHA-256 of that serialization.

    //         This resulting hash will be used as the PreviousHash for the NEXT entry.
```

```
    //         Return both the entry and this computed hash? (Or compute separately in AppendEvent).


    // TODO 5: Return the created HashChainEntry.

    return HashChainEntry{}, nil

}


// computeEntryHash calculates the cryptographic hash of a HashChainEntry.

// The Signature field is EXCLUDED from the hash computation.

func computeEntryHash(entry HashChainEntry) ([]byte, error) {

    // TODO 1: Create a temporary struct or map that matches HashChainEntry but without the Signature field.

    // TODO 2: Serialize this temporary structure to canonical JSON.

    // TODO 3: Compute and return the SHA-256 sum of the serialized bytes.

    return nil, nil

}
```

**2. Storage Engine Core (Partial)**

GO

```go
// internal/storage/engine.go

package storage


import (

    "sync"

    "time"


    "github.com/yourproject/internal/event"

)


// Config holds storage engine configuration.

type Config struct {

    DataDir         string

    MaxSegmentSize  int64

    MaxClockSkew    time.Duration

}


// StorageEngine is the main coordinator for immutable audit log storage.

type StorageEngine struct {

    catalog    *Catalog

    activeWAL  *WAL

    activeSeq  uint64

    lastHash   []byte // Hash of the most recently written entry

    mu         sync.RWMutex

    config     Config

}


// NewStorageEngine initializes the storage engine, loading existing segments.

func NewStorageEngine(config Config) (*StorageEngine, error) {

    // TODO 1: Create data directory if it doesn't exist.

    // TODO 2: Scan the data directory for existing segment manifest files (.manifest).

    // TODO 3: Load each manifest, rebuilding the in-memory Catalog.

    // TODO 4: Determine the latest segment and sequence number.

    // TODO 5: Open or create the active WAL file for the next segment.

    // TODO 6: Initialize lastHash (from the latest sealed segment's root hash, or a genesis hash).

    // TODO 7: Return the initialized StorageEngine.

    return nil, nil

}


// AppendEvent appends an event to the log with hash chain.

func (s *StorageEngine) AppendEvent(event event.AuditEvent) (HashChainEntry, error) {

    s.mu.Lock()
```

```
    defer s.mu.Unlock()

    // TODO 1: Validate event timestamp is not too far in the future (within MaxClockSkew).
    //         Reject if it is.

    // TODO 2: Atomically increment the active sequence number.
    //         This sequence is final for this event.

    // TODO 3: Call createHashChainEntry, passing the event, s.lastHash, and the new sequence.

    // TODO 4: Serialize the HashChainEntry to bytes (e.g., JSON or Protobuf).

    // TODO 5: Call s.activeWAL.Append(serializedEntry) to durably write it.
    //         This handles the fsync.

    // TODO 6: Update s.lastHash to the hash of the entry we just wrote (from step 3/4).
    // TODO 7: Check if the active WAL size has reached config.MaxSegmentSize.
    //         If yes, call s.rotateActiveSegment().

    // TODO 8: Return the created HashChainEntry.
    return HashChainEntry{}, nil
}

// rotateActiveSegment seals the current WAL and starts a new one.
func (s *StorageEngine) rotateActiveSegment() error {
    // TODO 1: Close the current activeWAL.
    // TODO 2: Rename the .wal file to a .seg file with a generated SegmentID.
    // TODO 3: Create a SegmentManifest for the sealed segment.
    //         Populate: SegmentID, StartTime, EndTime, StartSequence, EndSequence,
    //         RootHash (s.lastHash), PreviousSegmentID (from catalog).
    // TODO 4: Write the manifest file to disk.
    // TODO 5: Update the Catalog with the new sealed segment.
    // TODO 6: Open a new active WAL file for the next segment.
    // TODO 7: Write the first entry's PreviousHash in the new WAL? This is handled in AppendEvent.
    return nil
}
```

**3. Integrity Verification**

```go
// internal/storage/integrity.go

package storage


// VerifyChain verifies hash chain integrity from startSeq.

func (s *StorageEngine) VerifyChain(startSeq uint64) (bool, error) {

    s.mu.RLock()

    defer s.mu.RUnlock()


    // TODO 1: Determine the trusted starting hash.

    //         If startSeq == 0, use the genesis hash.

    //         Else, read entry startSeq-1 and get its hash (via computeEntryHash).


    // TODO 2: Iterate through entries starting from startSeq until the end of the chain.

    //         For each entry at sequence N:

    //         a. Read entry N.

    //         b. Compute its hash (call computeEntryHash).

    //         c. Read entry N+1.

    //         d. Compare computed hash from step b with the PreviousHash stored in entry N+1.

    //         e. If mismatch, return false with an error detailing sequence N.

    // TODO 3: If loop completes without mismatch, return true.

    return false, nil

}
```

**E. Language-Specific Hints (Go)**

- **File Sync**: Always use `file.Sync()` after critical writes. Consider `file.Sync()` on the directory after creating a new manifest file for extra safety on some filesystems.
- **Atomic Sequence**: Use `atomic.AddUint64(&s.activeSeq, 1)` for thread-safe sequence generation in `AppendEvent`.
- **Canonical JSON**: Use `github.com/fxamacker/cbor` for deterministic serialization, or carefully marshal maps with sorted keys using `encoding/json`.
- **Concurrent Reads**: Use `sync.RWMutex` in `StorageEngine` to allow multiple concurrent `ReadEvent` or `VerifyChain` operations while blocking writes.

**F. Milestone Checkpoint** After implementing the core `AppendEvent` and `VerifyChain` logic, you should be able to run a verification test.

1. **Command**: `go test ./internal/storage/... -v -run TestStorageEngine`
2. **Expected Output**: A passing test that:
   - Creates a new `StorageEngine` in a temp directory.
   - Appends 100 sample `AuditEvent` objects.
   - Calls `VerifyChain(0)` and asserts it returns `true`.
   - Manually corrupts a byte in the middle of a segment file.
   - Calls `VerifyChain(0)` again and asserts it returns `false` with an error identifying the approximate sequence of corruption.
3. **Manual Verification**: Write a small Go program that creates an engine, appends a few events, prints their sequence numbers and entry hashes, and then verifies the chain. The output should show a chain of hashes where each entry's hash is included in the next.
4. **Sign of Trouble**: If `VerifyChain` passes even after you manually corrupt a file, double-check that `computeEntryHash` includes all critical fields (especially `PreviousHash`). Ensure your test is actually modifying the correct byte in the segment file.

## Component Design: Query & Export Engine

> **Milestone(s):** Milestone 3: Audit Query & Export

The **Query & Export Engine** is the system's "librarian"—its purpose is to efficiently locate specific audit records within an ever-growing, immutable archive and present them in formats suitable for human review or compliance submission. While the Storage Engine guarantees data integrity and order, this component must deliver **sub-second query performance** over terabytes of historical data and support **streaming exports** without exhausting system memory. Its key challenge is balancing query flexibility with the performance constraints of an append-only, time-ordered dataset.

### Mental Model: The Library Index Card System

Imagine a vast library containing every book ever published, where new books arrive constantly and are placed on the shelves in strict chronological order of their publication date. Each book is an `AuditEvent`—a complete record of an action. Finding a specific book by its author, title, or subject without an index would require a linear scan of millions of shelves, which is prohibitively slow.

Our Query Engine solves this by maintaining a set of **index cards** (the indexes) organized in a card catalog. The primary catalog is ordered by **publication date (timestamp)** and **accession number (sequence)**, allowing you to quickly locate all books from a specific year and month. Additional specialized card sets let you look up books by **author (actor ID)** or **subject (resource type)**. When you request a list of books, the librarian (the engine) uses these cards to identify the relevant shelves, retrieves the books, and can provide them in batches (pagination) or compile them into a report (export). This model captures the essence of indexed time-series querying: the primary temporal order is intrinsic to the data's storage, while secondary indexes accelerate specific lookup patterns.

### Interface: Query, Paginate, and Export

The Query & Export Engine exposes a clean API centered around three primary operations: executing a filtered query, retrieving subsequent pages of results, and streaming events to an export format. The following table details the core methods and their contracts.

| Method Name | Parameters | Returns | Description |
|---|---|---|---|
| `Query` | `filters QueryFilters`, `startTime time.Time`, `endTime time.Time`, `pageSize int` | `QueryResult`, `error` | The primary query method. Executes a filtered search for events within the specified time range. The `pageSize` parameter limits the number of events returned in the initial `QueryResult.Events` slice. |
| `NextPage` | `cursor string` | `QueryResult`, `error` | Retrieves the next page of results for a query. The `cursor` is an opaque token returned in `QueryResult.NextCursor` that encodes the state needed to resume the query (e.g., the last seen timestamp and sequence number). This enables stable, cursor-based pagination. |
| `Export` | `filters QueryFilters`, `format ExportFormat`, `writer io.Writer` | `error` | Streams all events matching the filters directly to the provided `writer` in the specified format (e.g., CSV, JSON Lines). This method handles large result sets without loading them entirely into memory, providing backpressure via the writer's `Write` method. |
| `GetStatistics` | `filters QueryFilters`, `startTime time.Time`, `endTime time.Time` | `QueryStatistics`, `error` | Returns aggregated statistics (counts per actor, action, outcome) for the filtered time range, useful for generating summary reports without transferring the full event dataset. |

The `QueryFilters` struct is the workhorse for specifying search criteria. It uses slices for fields where multiple values should be OR'ed together (e.g., events from *any* of these actors) and maps for exact metadata key-value matching.

| Field | Type | Description |
|---|---|---|
| `ActorIDs` | `[]string` | Optional. If provided, only events where the `ActorInfo.ID` matches one of these values are returned. |
| `ResourceTypes` | `[]string` | Optional. If provided, only events where the `ResourceInfo.Type` matches one of these values are returned. |
| `Action` | `string` | Optional. Exact match on the `Action` field. Supports wildcard patterns (e.g., `user.*`) in some implementations, but exact match is required for the base design. |
| `Outcome` | `string` | Optional. Exact match on the `Outcome` field (e.g., `"success"`, `"failure"`). |
| `Metadata` | `map[string]string` | Optional. Only events whose `Metadata` map contains **all** the specified key-value pairs (exact matches) are returned. |

The `QueryResult` packages the response for a single page.

| Field | Type | Description |
|---|---|---|
| `Events` | `[]AuditEvent` | The slice of audit events for the current page. Length is at most the requested `pageSize`. |
| `NextCursor` | `string` | An opaque token used to fetch the next page. If empty, there are no more results. The cursor must be stable against concurrent writes (i.e., new events being appended do not cause rows to be skipped or duplicated). |
| `TotalCount` | `int` | The **estimated** total number of events matching the query filters. For performance, this may be an approximation when queries span many segments. |

### ADR: Indexing Strategy for Time-Series Logs

**Decision: Primary Timestamp+Sequence Index with Optional In-Memory Secondary Indexes**

- **Context**: Audit events are ingested continuously and must be queried most frequently by time range (e.g., "all logins between 9 AM and 5 PM"). Secondary filtering by actor or resource is also common. We must support efficient time-range scans while allowing performant filtering on other dimensions without implementing a full relational database.
- **Options Considered**:
  1. **Primary Index on (Timestamp, Sequence) Only**: Store events in segmented files ordered by `(Timestamp, Sequence)`. Perform a binary search on segment time ranges to locate start position, then linear scan and filter within segments.
  2. **Primary Index + Dense Secondary Indexes**: Maintain the primary index and also build separate index files (e.g., B-Trees) mapping `ActorID -> []Sequence` and `ResourceType -> []Sequence`. Query by resolving sequences from secondary index, then fetching events from primary storage (requires multi-sequence merge).
  3. **Primary Index + Inverted Index Metadata**: Treat the entire event as a document and build a full-text inverted index (e.g., using Elasticsearch or Bleve). Enables rich free-text search but adds significant operational complexity.
- **Decision**: We choose **Option 1 (Primary Index Only)** as our baseline, augmented with a simple **in-memory catalog of segment metadata** that includes min/max values for common fields (actor, resource) to enable segment pruning. For deployments requiring faster actor/resource queries, we recommend **Option 2 as a future extension**.
- **Rationale**:
  - **Simplicity and Write Performance**: The primary index is inherent to our append-only, time-ordered storage structure. Adding secondary indexes would double write latency (write-amplification) and complicate segment rotation and compaction.
  - **Query Pattern Alignment**: Compliance audits and security investigations are overwhelmingly time-range driven. A scan of a few hours of data with in-memory filtering in Go is extremely fast, even for millions of events.
  - **Controlled Resource Usage**: In-memory segment metadata (min/max actor, resource, etc.) allows us to skip entire segments that cannot match a filter, often reducing I/O dramatically without the storage overhead of full secondary indexes.
  - **Extensibility**: The design can be later extended with optional pluggable indexers (secondary B-Trees, Bloom filters) if specific query patterns prove problematic, without breaking the core API.
- **Consequences**:
  - **Pros**: Simple implementation, minimal write overhead, excellent performance for pure time-range queries, and efficient use of storage.
  - **Cons**: Queries filtering *only* by actor (without a time range) will require scanning all segments whose metadata overlaps that actor. This is acceptable for our non-goal of "full-table scans by actor" but could be a limitation for some use cases.

The following table summarizes the trade-offs:

| Option | Pros | Cons | Chosen? |
|---|---|---|---|
| **Primary (Timestamp, Sequence) Only** | Minimal write amplification, inherent to storage layout, excellent for time-range queries. | Actor/Resource-only queries require scanning all potentially relevant segments (slow for large histories). | **Yes** (Baseline) |
| **Primary + Dense Secondary Indexes** | Fast point queries by actor or resource, enables complex boolean filters. | Significant write amplification (every event writes to multiple indexes), increased storage footprint, complex segment rotation. | No (Future Extension) |
| **Primary + Inverted Full-Text Index** | Enables rich keyword search across all event fields and metadata. | Very high complexity, large resource consumption (memory/disk), operational burden of maintaining a separate index cluster. | No (Out of Scope) |

### Common Pitfalls: Pagination, Memory, Timezones

Building a robust Query & Export Engine involves subtle challenges that can lead to incorrect results, poor performance, or compliance issues.

**⚠️ Pitfall 1: Offset-Based Pagination on Append-Only Logs**

- **Description**: Implementing pagination using `LIMIT 100 OFFSET 500` (or its in-memory equivalent) by loading all results, then skipping the first N.

- **Why It's Wrong**: In a constantly growing log, a client fetching page 2 might receive duplicate events if new events were appended between the first and second request (shifting the offsets). It's also inefficient for deep pages, as the server must scan and discard all preceding rows.
- **Fix**: Use **cursor-based pagination**. The cursor encodes the last seen `(Timestamp, Sequence)` pair. The next query resumes scanning from just after that point. This is stable under concurrent writes and provides constant-time seek to the start of each page.

**⚠️ Pitfall 2: Loading Entire Result Sets into Memory**

- **Description**: The `Export` function first calls `Query` with no pagination, accumulates all `AuditEvent` structs in a giant slice, then serializes them to the output writer.
- **Why It's Wrong**: A compliance export for a year's data could contain hundreds of millions of events, consuming gigabytes of RAM and potentially crashing the service.
- **Fix**: Implement **streaming export**. The `Export` method should iterate over segments and matching events, serializing each event directly to the `io.Writer` as it's read. Use buffered I/O for efficiency, but never hold more than a few events in memory at once.

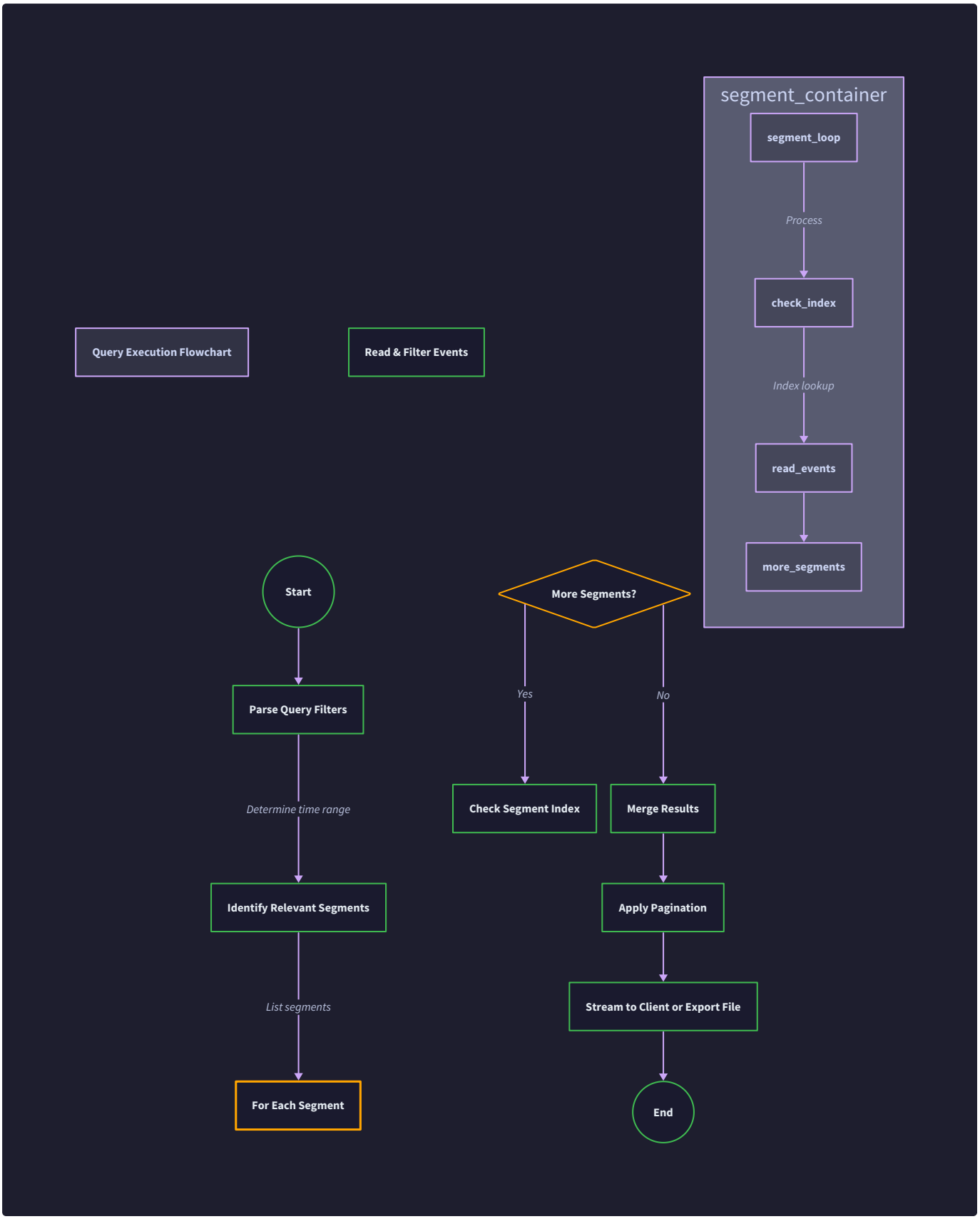
**⚠️ Pitfall 3: Inconsistent Timezone Handling**

- **Description**: Storing timestamps in local server time, or allowing queries with local time ranges without conversion, leading to off-by-one-day errors in reports.
- **Why It's Wrong**: Audit logs are often reviewed by globally distributed teams and must correlate with events in other systems. Ambiguous timezone handling makes precise time-range queries impossible and violates compliance requirements for unambiguous timestamps.
- **Fix**: **Store all timestamps as UTC** in the `AuditEvent` and `HashChainEntry`. Accept query time ranges in UTC, or explicitly accept a timezone parameter and convert to UTC internally before processing. Always render timestamps in ISO 8601 format with 'Z' (Zulu/UTC) suffix in exports.

**⚠️ Pitfall 4: Ignoring Segment Pruning**

- **Description**: A query for `ActorID="user-123"` scans every segment file from the beginning of time, reading and decoding each event to check the actor field.
- **Why It's Wrong**: This performs unnecessary I/O and CPU work. Most segments will contain no events for that actor. For large histories, this turns a simple query into a multi-hour operation.
- **Fix**: **Maintain segment-level metadata** in the `Catalog`. For each sealed segment, record the minimum and maximum `Timestamp`, and optionally sets of distinct `ActorIDs` and `ResourceTypes` present. Before scanning a segment, check if the query's time range overlaps the segment's time range and if the filtered `ActorID` or `ResourceType` could possibly be inside (using the sets). Skip the segment entirely if not.

The query execution flow, incorporating these fixes, follows a clear process illustrated in the flowchart:




## Implementation Guidance for Query Engine

This section provides concrete starting points for building the Query & Export Engine in Go. We focus on providing a complete, streaming CSV exporter and skeleton code for the core query iterator, emphasizing the patterns needed to avoid the common pitfalls.

### A. Technology Recommendations Table

| Component | Simple Option | Advanced Option |
|---|---|---|
| **Query Execution** | In-memory filtering over memory-mapped segment files ( `mmap` ) | Integrated time-series database (e.g., QuestDB, InfluxDB) for SQL-like querying |
| **Indexing** | Segment metadata catalog (min/max timestamps, actor/resource sets) stored as JSON manifest files | BoltDB or PEBBLE for on-disk secondary indexes (Actor-> []Sequence) |
| **Export Formatting** | Streaming CSV/JSON encoder using Go's `encoding/csv` and `encoding/json` | Templated PDF reports using a library like `gofpdf` or external service call |
| **Pagination Cursor** | Opaque string containing base64-encoded `(lastTimestamp, lastSequence)` | Structured token with checksum to prevent client modification, using JWT or MessagePack |

**B. Recommended File/Module Structure**

Place the query and export logic in a dedicated package, separate from storage and ingestion.

```
audit-log-system/
├── cmd/
│   └── server/                # Main service binary
├── internal/
│   ├── event/                # AuditEvent model (Milestone 1)
│   ├── storage/              # StorageEngine and WAL (Milestone 2)
│   ├── query/                # Query & Export Engine (Milestone 3)
│   │   ├── engine.go         # Main QueryEngine struct and public methods
│   │   ├── filters.go        # QueryFilters struct and validation
│   │   ├── iterator.go       # TimeRangeIterator for streaming through segments
│   │   ├── cursor.go         # Cursor encoding/decoding logic
│   │   ├── export/
│   │   │   ├── csv.go        # CSV streaming exporter
│   │   │   ├── jsonl.go      # JSON Lines exporter
│   │   │   └── formats.go    # ExportFormat constants
│   │   └── catalog/          # Segment Catalog for metadata (could be under storage/)
│   └── ingest/               # Ingestor (if separate)
└── pkg/                      # Public libraries (if any)
```

**C. Infrastructure Starter Code: Time-Range Iterator**

A core building block is an iterator that efficiently yields events from one or more segments within a time range. This version uses a simple file scan; a production version would use the segment catalog to skip non-relevant segments.

```go
// internal/query/iterator.go

package query


import (

    "encoding/json"

    "io"

    "os"

    "path/filepath"

    "time"


    "yourproject/internal/event"

    "yourproject/internal/storage"

)


// TimeRangeIterator iterates over AuditEvents from a StorageEngine within a time range.

// It streams events without loading all into memory.

type TimeRangeIterator struct {

    storage    *storage.Engine

    startTime  time.Time

    endTime    time.Time

    filters    QueryFilters

    // Internal state

    currentSegment *storage.SegmentReader

    currentEvent   *event.AuditEvent

    lastSeq        uint64

    initialized    bool

}


// NewTimeRangeIterator creates a new iterator for the given time range and filters.

// It pre-identifies which segments to examine using the storage engine's catalog.

func NewTimeRangeIterator(store *storage.Engine, start, end time.Time, filters QueryFilters) (*TimeRangeIterator, error) {

    // TODO 1: Consult store.Catalog to get list of segment manifests that overlap [start, end]

    // TODO 2: Sort segments by StartTime ascending

    // TODO 3: Initialize iterator with this list and set internal state

    return &TimeRangeIterator{

        storage:   store,

        startTime: start,

        endTime:   end,

        filters:   filters,

    }, nil

}
```

```go
// Next advances the iterator to the next event matching the filters and time range.
// Returns true if an event is available, false if iteration is complete.
func (it *TimeRangeIterator) Next() bool {
    // TODO 4: If not initialized, open the first relevant segment and seek to first event >= startTime
    // TODO 5: Loop: read next event from current segment; if EOF, open next segment; if no more segments, return false
    // TODO 6: For each event, check if event.Timestamp < endTime. If false, return false (past range).
    // TODO 7: Apply in-memory filters (ActorIDs, ResourceTypes, etc.) to the event.
    // TODO 8: If event matches, store in it.currentEvent and return true; else continue loop.
    return false
}

// Event returns the current event. Call only after Next() returns true.
func (it *TimeRangeIterator) Event() *event.AuditEvent {
    return it.currentEvent
}

// Close releases any resources (open file handles) held by the iterator.
func (it *TimeRangeIterator) Close() error {
    if it.currentSegment != nil {
        return it.currentSegment.Close()
    }
    return nil
}
```

**D. Core Logic Skeleton Code: Query Engine and CSV Exporter**

The main `QueryEngine` orchestrates the iterator, applies pagination via cursors, and manages the export streams.

GO

```go
// internal/query/engine.go

package query


import (

    "encoding/base64"

    "encoding/json"

    "io"

    "time"


    "yourproject/internal/event"

    "yourproject/internal/storage"

)


// QueryEngine executes queries and exports on the audit log.

type QueryEngine struct {

    storage *storage.Engine

    catalog *Catalog

}


// NewQueryEngine creates a new query engine.

func NewQueryEngine(store *storage.Engine) *QueryEngine {

    return &QueryEngine{storage: store}

}


// Query performs a filtered time-range query with pagination.

func (qe *QueryEngine) Query(filters QueryFilters, startTime, endTime time.Time, pageSize int) (QueryResult, error) {

    // TODO 1: Validate input: endTime must be >= startTime, pageSize <= MAX_PAGE_SIZE

    // TODO 2: Decode cursor if provided (via a helper). Cursor contains last seen Timestamp and Sequence.

    // TODO 3: Create a TimeRangeIterator starting from cursor position (or startTime if no cursor).

    // TODO 4: Iterate up to pageSize events, collecting matching events into a slice.

    // TODO 5: If iteration stopped because pageSize was reached, encode a new cursor from the last event's (Timestamp, Sequence).

    // TODO 6: If iteration stopped because end of range or data, leave cursor empty.

    // TODO 7: Estimate TotalCount (can be len(events) for this page if no cursor, or use catalog for approximation).

    // TODO 8: Return QueryResult.

    return QueryResult{}, nil

}


// NextPage retrieves the next page using a cursor.

func (qe *QueryEngine) NextPage(cursor string) (QueryResult, error) {

    // TODO 1: Decode cursor to get the original filters, time range, and last position.

    // TODO 2: Call Query with those parameters and the decoded last position as the starting point.

    // TODO 3: Return the result.
```

```
    return QueryResult{}, nil
}

// Export streams all matching events in the specified format to the writer.
func (qe *QueryEngine) Export(filters QueryFilters, format ExportFormat, writer io.Writer) error {
    // TODO 1: Create a TimeRangeIterator for the entire time range (startTime = zero time, endTime = far future).
    // TODO 2: Based on format, delegate to the appropriate exporter.
    switch format {
    case ExportFormatCSV:
        return exportCSV(qe, filters, writer)
    case ExportFormatJSONL:
        return exportJSONL(qe, filters, writer)
    default:
        return ErrUnsupportedFormat
    }
}
```

GO

```go
// internal/query/export/csv.go

package export


import (

    "encoding/csv"

    "io"

    "strconv"

    "time"


    "yourproject/internal/event"

    "yourproject/internal/query"

)


// ToCSV streams the given events as CSV to the writer.

// This is a helper for a slice; the streaming version used by Export is below.

func ToCSV(events []event.AuditEvent, writer io.Writer) error {

    csvWriter := csv.NewWriter(writer)

    defer csvWriter.Flush()


    // Write header

    header := []string{"Timestamp", "ActorID", "ActorType", "Action", "ResourceID", "ResourceType", "Outcome", "ClientIP"}

    if err := csvWriter.Write(header); err != nil {

        return err

    }


    for _, ev := range events {

        record := []string{

            ev.Timestamp.UTC().Format(time.RFC3339Nano),

            ev.Actor.ID,

            ev.Actor.Type,

            ev.Action,

            ev.Resource.ID,

            ev.Resource.Type,

            ev.Outcome,

            ev.Context.ClientIP,

            // Note: Metadata is omitted for simplicity; could be flattened as key=value pairs.

        }

        if err := csvWriter.Write(record); err != nil {

            return err

        }

    }

    return nil
```

```
}


// exportCSV streams events directly from an iterator to CSV format.
func exportCSV(qe *query.QueryEngine, filters query.QueryFilters, writer io.Writer) error {
    // TODO 1: Create a TimeRangeIterator for all time (or use qe's internal method).
    // TODO 2: Create a csv.Writer wrapping the provided writer.
    // TODO 3: Write CSV header.
    // TODO 4: Iterate over iterator, writing each event as a CSV row.
    // TODO 5: Flush writer and handle errors.
    // Important: This function must not load all events into memory.
    return nil
}
```

**E. Language-Specific Hints**

- **Iterating Efficiently**: Use the `TimeRangeIterator` pattern to avoid materializing all results. Leverage `defer iter.Close()` to ensure file handles are released.
- **Cursor Encoding**: Use `base64.RawURLEncoding` to encode cursor structs (JSON or binary) for safe inclusion in URLs. Include a version field in the cursor struct to allow future evolution.
- **Streaming Exports**: Implement `io.Writer` wrappers for formatting (like `csv.Writer`). Use `bufio.NewWriter` for buffering, but remember to `Flush()` at the end.
- **Time Handling**: Always convert to UTC with `timestamp.UTC()` before storing or comparing. Use `time.RFC3339Nano` for string representation.
- **Memory Management**: Be careful with `[]AuditEvent` slices in pagination; pre-allocate with `make([]AuditEvent, 0, pageSize)` to avoid reallocations.

**F. Milestone Checkpoint**

To verify the Query & Export Engine is functioning correctly after implementing the skeletons:

1. **Run the unit tests**:

```
    go test ./internal/query/... -v -count=1
```
BASH

`Expected output should show passing tests for filter validation, cursor encoding/decoding, and CSV export of a small slice.`

2. **Manual Integration Test**:
    - Start the audit log server (if you have one) or use the `StorageEngine` directly to append a few hundred test events with varied actors, resources, and timestamps.
    - Execute a query via the `QueryEngine` for a specific time range and actor.
    - Verify that the returned `QueryResult` contains the expected events, the `NextCursor` is present if there are more results, and `TotalCount` is reasonable.
    - Use the `Export` function to write a CSV file, then inspect it to ensure all fields are present and formatted correctly.
3. **Signs of Trouble**:
    - **Queries returning no results**: Check that your `TimeRangeIterator` is correctly identifying and opening segments. Verify timestamp filtering logic.
    - **Cursor results in duplicates or skips**: Ensure the cursor encodes a strict ordering key `(Timestamp, Sequence)` and that the `Next()` method starts from the event *after* that key.
    - **High memory usage during export**: Confirm your `exportCSV` function uses the iterator and writes row-by-row, not collecting all events into a slice first.

## Error Handling and Edge Cases

> **Milestone(s):** Milestone 1 (Event Model validation), Milestone 2 (Storage integrity), Milestone 3 (Query reliability)

Even the most meticulously designed system will encounter failures in production. For an **Audit Logging System**—where data integrity is paramount—handling errors correctly isn't just about availability; it's about maintaining the cryptographic chain of trust and ensuring no audit event is lost or corrupted. This section catalogs the known failure modes across all components and defines concrete recovery procedures to restore system integrity when things go wrong.

Think of the audit log as a **chain of custody in a forensic investigation**. Each piece of evidence (audit event) must be documented, sealed, and linked to the previous piece. If the chain breaks—whether from a dropped evidence bag (disk error), a mislabeled tag (sequence error), or a contaminated sample (data corruption)—the entire investigation becomes inadmissible. Our error handling is the protocol that prevents, detects, and repairs such breaks.

### Failure Modes and Detection

Failures can occur at three main phases: **Ingestion** (when events enter the system), **Storage** (when events are written to durable media), and **Query/Export** (when data is retrieved). The following table systematically catalogs each failure mode, its symptoms, detection mechanisms, and immediate impact.

| Component | Failure Mode | Symptoms / Detection | Immediate Impact |
|---|---|---|---|
| **Ingestor** | Invalid event schema | `EventValidator.Validate()` returns validation error with specific field violations | Event rejected at ingress; no storage or integrity impact |
| **Ingestor** | PII in unmasked fields | Validation rule violation for fields marked as sensitive (e.g., email in `ActorInfo.DisplayName`) | Potential compliance violation; event must be masked or rejected |
| **Ingestor** | Clock skew beyond tolerance | Event timestamp differs from system wall clock by more than `Config.MaxClockSkew` | Event may be rejected or timestamp adjusted; could affect ordering if not handled |
| **Ingestor** | Event size exceeds limit | Serialized event size > `MAX_EVENT_SIZE` (64KB) | Event rejected; prevents storage fragmentation and memory exhaustion |
| **Storage Engine** | Disk full during append | `WAL.Append()` returns `io.ErrShortWrite` or `syscall.ENOSPC` | Appends fail; system enters read-only mode until storage is expanded |
| **Storage Engine** | Filesystem corruption | `WAL.ReadAt()` returns `io.ErrUnexpectedEOF` or checksum mismatch | May affect single segment or entire data directory; hash chain verification fails |
| **Storage Engine** | Hash chain break | `StorageEngine.VerifyChain()` returns `false` at a specific sequence number | Tamper detection triggered; all subsequent entries are considered untrustworthy |
| **Storage Engine** | Non-monotonic sequence | New sequence number ≤ current `StorageEngine.activeSeq` | Violates append-only ordering guarantee; may indicate bug or malicious replay |
| **Storage Engine** | Failed `fsync()` | `WAL.Append()` returns error after write but before sync (e.g., disk controller failure) | Data may be lost from OS cache on power loss; durability guarantee broken |
| **Storage Engine** | Segment rotation failure | `rotateActiveSegment()` fails to create new segment file (permissions, disk full) | No new events can be written; system stuck with oversized active segment |
| **Storage Engine** | Clock drift across restarts | `SegmentManifest.StartTime` of new segment is earlier than previous segment's `EndTime` | Time-based queries may miss events or produce incorrect ordering |
| **Query Engine** | Corrupt segment index | `Catalog` metadata inconsistent with actual segment file (size mismatch, missing file) | Queries may skip valid segments or attempt to read non-existent data |
| **Query Engine** | Memory exhaustion during export | `Export()` consumes excessive memory when collecting unbounded results | Process OOM kill; export fails; may affect system stability |
| **Query Engine** | Pagination cursor invalidation | `QueryResult.NextPage()` returns empty results or duplicates due to concurrent segment rotation | Client pagination breaks; may miss events or see duplicates |
| **Query Engine** | Timezone conversion error | Query with local time range produces incorrect UTC conversion (daylight saving, offset) | Returns events outside intended time window; compliance reporting inaccurate |
| **System-wide** | Process crash during write | Partial `HashChainEntry` written (only some fields persisted) | Segment file in inconsistent state; may fail to parse on restart |
| **System-wide** | Backup/restore integrity loss | Restored segment files have different modified timestamps or file IDs | Hash chain verification may fail due to byte-level differences |

> **Detection Principle:** Every component must fail loudly and diagnostically. Silent data corruption is the worst failure mode for an audit log. Each operation that can fail must return a descriptive error, and the system must have proactive verification routines (like periodic `VerifyChain`) that detect latent corruption before it's needed for compliance.

#### Mental Model: The Health Monitoring Dashboard

Imagine a nuclear facility's control room with hundreds of gauges, each monitoring a critical system parameter. A green light means "normal," yellow means "degraded but operational," and red means "immediate intervention required." Our failure detection system is that dashboard:

- **Green gauges:** Events validating successfully, hash chain verifying, queries returning within SLA.
- **Yellow gauges:** Disk usage above 80%, occasional clock skew warnings, segment rotation taking longer than expected.
- **Red gauges:** Hash chain mismatch, disk full errors, corrupt segment files.

The detection mechanisms in the table above are the sensors feeding these gauges. When any gauge turns yellow, automated recovery procedures kick in. When any turns red, human intervention is required with the recovery playbooks defined below.

### Recovery Strategies

When failures occur, we must have deterministic recovery procedures that restore system integrity without compromising the immutable, tamper-evident guarantees. The strategies below address the most critical failure modes from the table.

#### 1. Handling a Broken Hash Chain

A broken hash chain is a **critical severity** event—it indicates either storage corruption or potential tampering. The chain is broken when `StorageEngine.VerifyChain(startSeq)` returns `false` for some `startSeq`, meaning the computed hash for entry N doesn't match the stored `PreviousHash` in entry N+1.

**Recovery Procedure:**

1. **Isolate the break point:** Run verification from the beginning of the chain (sequence 1) or from the last known good checkpoint. The verification algorithm will identify the exact sequence number where the hash mismatch occurs.
2. **Determine scope:** Check if the break affects:
   - A single `HashChainEntry` within the active WAL
   - An entire segment file
   - Multiple segments
3. **For active WAL corruption:**
   - Immediately seal the current active segment (mark as `CORRUPT` in catalog)
   - Start a new segment with `PreviousSegmentID` pointing to the last *verified* segment
   - Log the corruption event with forensic details (file offset, expected vs actual hash)
   - The corrupted segment remains in storage but is excluded from all queries
4. **For sealed segment corruption:**
   - Mark the segment as `CORRUPT` in the catalog (excluding it from queries)
   - If backups exist, restore the segment from the last known-good backup
   - Re-verify the chain from the restored point forward
5. **Forensic analysis:** The corrupted data must be preserved (never deleted) for security investigation. Generate a checksum of the entire corrupted file and store it separately.

> **Integrity First Principle:** When hash chain verification fails, we **never** attempt to "repair" the chain by recomputing hashes. Doing so would destroy the tamper-evidence property. Instead, we quarantine the corrupted data and continue from the last known-good state, preserving the corruption for investigation.

#### 2. Recovering from a Full Disk

Disk exhaustion stops all writes, but reads and verifications should continue working. This is a **high severity** operational event that requires immediate intervention.

**Recovery Procedure:**

1. **Enter read-only mode:** When `WAL.Append()` returns a disk full error, the `StorageEngine` should:
   - Set an internal `readOnly` flag to `true`
   - Reject all subsequent `AppendEvent()` calls with a descriptive error
   - Continue serving `ReadEvent()` and `Query()` operations
2. **Free emergency space:** Implement a temporary expansion mechanism:
   - If configured, compress the oldest segments using Zstandard (fast, streaming compression)
   - Archive compressed segments to secondary storage (cloud storage, network mount)
   - Delete local copies of archived segments (only after successful remote upload)

3. **Add capacity:** Alert operators to provision additional disk space. Once available:
    - Update `Config.DataDir` to point to a new directory with space (if using symbolic links)
    - Or expand the filesystem if using LVM/cloud volumes
4. **Resume writes:** After ensuring minimum free space (e.g., 20% of total):
    - Clear the `readOnly` flag
    - Rotate to a new segment in the available space
    - Log a recovery event indicating the outage duration and actions taken

**Prevention Strategy:** Implement proactive monitoring with thresholds:

- **Warning at 70%:** Alert operators to plan for expansion
- **Critical at 85%:** Trigger automatic compression of old segments
- **Emergency at 95%:** Enter read-only mode preemptively

### 3. Dealing with Non-Monotonic Sequence Numbers

Sequence numbers must always increase monotonically. A non-monotonic sequence (where a new event's sequence ≤ current `activeSeq`) indicates either:

- A bug in sequence generation (e.g., integer overflow at $2^{64}$)
- A restore from backup that didn't preserve the latest sequence
- A malicious replay attack attempting to overwrite history

**Recovery Procedure:**

1. **Reject the offending event:** When `StorageEngine.AppendEvent()` detects a non-monotonic sequence:
    - Immediately reject the event with an `ErrSequenceViolation`
    - Log a security alert with the event details and source
    - Freeze sequence advancement until root cause is determined
2. **Diagnose root cause:**
    - **Check for integer overflow:** If `activeSeq` is near `math.MaxUint64`, initiate a planned migration to a larger numeric space (e.g., using a (epoch, sequence) tuple)
    - **Verify backup/restore process:** Check if a recent restore occurred and whether it properly restored the `activeSeq` state
    - **Examine event source:** Check if the event came from an untrusted or compromised service
3. **Recovery actions by root cause:**
    - **Overflow:** Implement sequence wrapping with an epoch counter. All new segments include an epoch ID, and sequences reset to 1 each epoch.
    - **Restore error:** Restore the correct `activeSeq` from the most recent segment's `EndSequence` + 1
    - **Malicious activity:** Implement client authentication and request signing for ingestion

### 4. Edge Case: Process Crash During Write

If the process crashes while writing a `HashChainEntry`, the WAL file may contain a partially written record. On restart, this causes `WAL.ReadAt()` to fail for subsequent reads.

**Recovery on Startup:**

1. **WAL recovery scan:** When `OpenWAL(path)` is called on an existing file:
    - Read from the beginning, validating each record's length prefix and checksum
    - When a malformed record is encountered, truncate the file to the last valid record's offset
    - Log the truncation event with bytes lost
2. **Hash chain reconstruction:** After WAL truncation:
    - Recompute the last valid `HashChainEntry`'s hash
    - Update the `activeSeq` to match the last valid sequence
    - The next append will continue from this recovered state
3. **Lost events:** Any events that were partially written and truncated are permanently lost. The system should log this data loss event prominently, as it represents a breach of the "no lost events" guarantee for those specific events.

### 5. Edge Case: Clock Drift Across Restarts

If the system clock moves backward (NTP adjustment, VM snapshot restore) between restarts, new segments may have `StartTime` earlier than previous segments' `EndTime`.

**Mitigation Strategy:**

1. **Never use wall clock for ordering:** Sequence numbers are the authoritative ordering mechanism. Time is for querying, not for correctness.
2. **Segment time validation:** When creating a new segment, if `time.Now().UTC()` is before the previous segment's `EndTime`:
   - Use the previous segment's `EndTime + 1µs` as the new segment's `StartTime`
   - Log a clock skew warning
   - This ensures segment time ranges remain monotonic for query pruning
3. **Query resilience:** The `QueryEngine` should handle overlapping segment time ranges by checking all segments whose time ranges intersect the query window, then filtering by actual event timestamps.

#### 6. Edge Case: Corrupt Segment Index (Catalog)

The in-memory `Catalog` indexes all segments for efficient query pruning. If it becomes inconsistent with the actual files on disk, queries may fail or return incomplete results.

**Recovery Procedure:**

1. **Catalog rebuild on startup:** The `StorageEngine` should always validate the catalog against disk on initialization:
   - Scan `Config.DataDir` for all `*.segment` files
   - Parse each segment's header/manifest to get `SegmentManifest`
   - Rebuild the catalog in memory from these manifests
   - Compare with persisted catalog state (if any); log inconsistencies
2. **Runtime detection:** If a query encounters a missing file or size mismatch:
   - Mark that segment as `CORRUPT` in the catalog
   - Trigger an immediate catalog rescan in the background
   - Retry the query with the updated catalog
3. **Persistence:** After any segment rotation or catalog change, persist the catalog state to a file in `DataDir` (e.g., `catalog.json`). This provides a checkpoint for faster recovery.

### Implementation Guidance

This guidance provides concrete Go code for detecting and recovering from critical failures. The focus is on building resilient components that maintain integrity even when things go wrong.

#### A. Technology Recommendations Table

| Component | Simple Option | Advanced Option |
|---|---|---|
| **Error Detection** | Periodic goroutine running `VerifyChain()` | Prometheus metrics with alert rules on verification failures |
| **Disk Monitoring** | `syscall.Statfs()` checks before each append | Filesystem inotify events with exponential backoff retry |
| **Crash Recovery** | WAL truncation on malformed records | Write-ahead log with Redo logging (like database recovery) |
| **Corruption Quarantine** | Move corrupt files to `quarantine/` subdirectory | Immutable object storage with versioning for forensic analysis |

#### B. Recommended File/Module Structure

Extend the existing project structure with error handling modules:

```
project-root/
  cmd/
    audit-server/
      main.go                   ← entry point with signal handling
  internal/
    errorhandling/              ← error handling utilities
      detector.go               ← failure detection routines
      recovery.go               ← recovery procedures
      metrics.go                ← error metrics and alerts
    storage/
      engine.go                 ← StorageEngine with error handling
      wal.go                    ← WAL with crash recovery
      corruption.go             ← hash chain verification and repair
      catalog.go                ← Catalog with consistency checks
    query/
      engine.go                 ← QueryEngine with retry logic
      export.go                 ← Export with streaming error handling
    event/
      validator.go              ← Event validation with detailed errors
    config/
      config.go                 ← Config with validation
  pkg/
    api/                        ← public API with error types
      errors.go                 ← domain-specific error types
```

### C. Infrastructure Starter Code: Error Types and Detection

Create a comprehensive error type system that distinguishes between recoverable and non-recoverable failures:

```go
// pkg/api/errors.go

package api

import "fmt"

// Domain-specific error types for clear handling
type ErrCode string

const (
    ErrCodeValidation     ErrCode = "VALIDATION_FAILED"
    ErrCodeDiskFull       ErrCode = "DISK_FULL"
    ErrCodeCorruption     ErrCode = "DATA_CORRUPTION"
    ErrCodeSequenceBreak  ErrCode = "SEQUENCE_VIOLATION"
    ErrCodeClockSkew      ErrCode = "CLOCK_SKEW"
    ErrCodeTamperDetected ErrCode = "TAMPER_DETECTED"
)

// AuditError wraps all domain errors with rich context
type AuditError struct {
    Code     ErrCode
    Message  string
    Details  map[string]interface{}
    Inner    error
    IsFatal  bool // If true, requires immediate operator intervention
}

func (e *AuditError) Error() string {
    return fmt.Sprintf("[%s] %s", e.Code, e.Message)
}

func NewAuditError(code ErrCode, msg string, details map[string]interface{}) *AuditError {
    return &AuditError{
        Code:    code,
        Message: msg,
        Details: details,
        IsFatal: isFatalCode(code),
    }
}

func isFatalCode(code ErrCode) bool {
    fatalCodes := map[ErrCode]bool{
        ErrCodeTamperDetected: true,
        ErrCodeCorruption:     true,
```

```
    }

    return fatalCodes[code]

}


// Helper constructors for common errors

func ErrValidation(field, reason string) *AuditError {

    return NewAuditError(ErrCodeValidation, "event validation failed",

        map[string]interface{}{"field": field, "reason": reason})

}


func ErrDiskFull(path string, available, required int64) *AuditError {

    return NewAuditError(ErrCodeDiskFull, "insufficient disk space",

        map[string]interface{}{

            "path":      path,

            "available": available,

            "required":  required,

        })

}
```

**D. Core Logic Skeleton: Storage Engine with Error Recovery**

Implement the `StorageEngine` with integrated error detection and recovery pathways:

```go
// internal/storage/engine.go

package storage


import (

    "context"

    "fmt"

    "sync"

    "time"


    "project/internal/event"

    "project/pkg/api"

)


// StorageEngine with error handling

type StorageEngine struct {

    config        Config

    catalog       *Catalog

    activeWAL     *WAL

    activeSeq     uint64

    mu            sync.RWMutex

    readOnly      bool // Set to true on disk full

    verificationRunning bool

    lastVerifyTime time.Time

    errorChan     chan<- *api.AuditError // Channel for critical errors

}


// AppendEvent with comprehensive error handling

func (s *StorageEngine) AppendEvent(evt event.AuditEvent) (*HashChainEntry, error) {

    s.mu.Lock()

    defer s.mu.Unlock()


    // 1. Check if in read-only mode (disk full recovery)

    if s.readOnly {

        return nil, api.NewAuditError(api.ErrCodeDiskFull,

            "storage engine in read-only mode due to disk full", nil)

    }


    // 2. Validate sequence monotonicity

    nextSeq := s.activeSeq + 1

    // TODO 1: Check for sequence overflow (if activeSeq == math.MaxUint64)

    // TODO 2: If overflow, implement epoch-based sequence scheme
```

```go
// 3. Create hash chain entry
prevHash, err := s.getPreviousHash()
if err != nil {
    return nil, fmt.Errorf("failed to get previous hash: %w", err)
}

entry, err := createHashChainEntry(evt, prevHash, nextSeq)
if err != nil {
    return nil, fmt.Errorf("failed to create hash entry: %w", err)
}

// 4. Serialize entry to bytes
data, err := SerializeToBinary(entry)
if err != nil {
    return nil, fmt.Errorf("serialization failed: %w", err)
}

// 5. Append to WAL with retry on temporary errors
offset, err := s.appendWithRetry(data)
if err != nil {
    // TODO 3: Check if error is disk full (io.ErrShortWrite or ENOSPC)
    // TODO 4: If disk full, set readOnly = true and return appropriate error
    // TODO 5: For other errors, log and retry based on error type
    return nil, err
}

// 6. Update state only after successful durable write
s.activeSeq = nextSeq
s.catalog.AddEntryToActiveSegment(entry, offset)

// 7. Check if segment rotation needed
if s.activeWAL.Size() > s.config.MaxSegmentSize {
    if err := s.rotateActiveSegment(); err != nil {
        // TODO 6: Handle rotation failure - log error but continue
        // Rotation failure shouldn't block appends to current segment
    }
}
```

```
    return &entry, nil
}

// appendWithRetry implements exponential backoff for transient errors
func (s *StorageEngine) appendWithRetry(data []byte) (int64, error) {
    var lastErr error
    for attempt := 0; attempt < 3; attempt++ {
        offset, err := s.activeWAL.Append(data)
        if err == nil {
            return offset, nil
        }

        // TODO 7: Check if error is temporary (use os.IsTimeout, os.IsPermission, etc.)
        // TODO 8: If not temporary, break immediately and return error
        // TODO 9: If temporary, sleep with exponential backoff: time.Sleep(time.Duration(attempt*attempt) * 100 * time.Millisecond)

        lastErr = err
    }
    return 0, fmt.Errorf("failed after 3 retries: %w", lastErr)
}

// VerifyChain with detailed corruption reporting
func (s *StorageEngine) VerifyChain(startSeq uint64) (bool, *api.AuditError) {
    if s.verificationRunning {
        return false, api.NewAuditError(api.ErrCodeValidation,
            "verification already in progress", nil)
    }

    s.verificationRunning = true
    defer func() { s.verificationRunning = false }()

    // TODO 10: Iterate through all segments starting from startSeq
    // TODO 11: For each HashChainEntry, compute hash and compare with next entry's PreviousHash
    // TODO 12: If mismatch found, return false with AuditError containing:
    //   - Sequence number of break
    //   - Expected vs actual hash
    //   - Segment ID and file offset
    //   - Set IsFatal = true
    // TODO 13: If verification completes successfully, update lastVerifyTime and return true
```

```
    return true, nil
}

// recoverFromCrash scans WAL and truncates any partial writes
func (s *StorageEngine) recoverFromCrash() error {
    // TODO 14: Open WAL file in read/write mode
    // TODO 15: Iterate through records from beginning, validating each
    // TODO 16: When malformed record found, truncate file at last valid offset
    // TODO 17: Recompute activeSeq from last valid record
    // TODO 18: Log recovery details: bytes lost, last valid sequence

    return nil
}

// rotateActiveSegment with error recovery
func (s *StorageEngine) rotateActiveSegment() error {
    // TODO 19: Check disk space before rotation (prevent failure mid-rotation)
    // TODO 20: Seal current WAL and create SegmentManifest
    // TODO 21: Write manifest file atomically (write to temp, then rename)
    // TODO 22: Create new WAL, update catalog
    // TODO 23: If any step fails, attempt rollback to previous state
    // TODO 24: Log rotation event with timing and size metrics

    return nil
}
```

### E. Language-Specific Hints

- **Disk space checking:** Use `syscall.Statfs` on Unix-like systems to check available space before writes:

```go
func checkDiskSpace(path string, required int64) (bool, int64, error) {
    var stat syscall.Statfs_t
    if err := syscall.Statfs(path, &stat); err != nil {
        return false, 0, err
    }
    // Available blocks * block size
    available := int64(stat.Bavail) * int64(stat.Bsize)
    return available >= required, available, nil
}
```

- **Atomic file operations:** Use rename for atomic manifest writes:

```go
// Write to temp file first
tempPath := manifestPath + ".tmp"
if err := os.WriteFile(tempPath, data, 0644); err != nil {
    return err
}
// Atomic rename
if err := os.Rename(tempPath, manifestPath); err != nil {
    os.Remove(tempPath) // Clean up temp file
    return err
}
```

- **Safe truncation:** When recovering from crash, truncate WAL safely:

```go
if err := walFile.Truncate(lastValidOffset); err != nil {
    return fmt.Errorf("truncate failed: %w", err)
}
if err := walFile.Sync(); err != nil {
    return fmt.Errorf("sync after truncate failed: %w", err)
}
```

### F. Milestone Checkpoint: Error Handling Validation

After implementing error handling, validate with these tests:

**Checkpoint 1: Event Validation Errors**

```bash
# Run validation tests
go test ./internal/event/... -v -run TestValidator
# Expected: All tests pass, including:
# ✓ Rejects events with missing actor ID
# ✓ Masks PII in email fields
# ✓ Rejects events exceeding MAX_EVENT_SIZE
# ✓ Adjusts timestamps within clock skew tolerance
```

**Checkpoint 2: Storage Engine Failure Recovery**

```bash
# Simulate disk full scenario
dd if=/dev/zero of=./test-data/test.log bs=1M count=1024
go test ./internal/storage/... -v -run TestDiskFull
# Expected:
# ✓ Appends fail gracefully with ErrDiskFull
# ✓ Read-only mode activated
# ✓ Reads continue to work while in read-only mode
# ✓ After freeing space, writes resume successfully
```

**Checkpoint 3: Hash Chain Verification**

```bash
# Run verification tests with injected corruption
go test ./internal/storage/... -v -run TestCorruption
# Expected:
# ✓ VerifyChain detects single-byte corruption
# ✓ Corrupt segment marked as CORRUPT in catalog
# ✓ Queries skip corrupt segments
# ✓ System continues with new segment after corruption
```

### G. Debugging Tips

| Symptom | Likely Cause | How to Diagnose | Fix |
|---|---|---|---|
| **Hash verification fails after restart** | WAL truncation during crash recovery changed byte alignment | Check recovery logs for "truncated X bytes" messages. Compare file sizes before/after restart. | Ensure WAL uses length-prefixed records with checksums. Test crash recovery thoroughly. |
| **Query returns duplicate events** | Catalog inconsistency or pagination cursor bug | Check if segment time ranges overlap. Verify cursor includes segment ID + offset. | Rebuild catalog from disk. Ensure cursor includes enough context for stable pagination. |
| **AppendEvent hangs indefinitely** | Deadlock between rotation and append, or disk full without timeout | Check for concurrent goroutines holding locks. Monitor disk space. | Add timeouts to all storage operations. Implement lock ordering convention. |
| **Export runs out of memory** | Loading all results into memory before writing | Check if `Export()` uses `io.Writer` stream or collects slice. Monitor heap during export. | Implement streaming export with `TimeRangeIterator`. Use buffered writes. |
| **Sequence numbers jump by >1** | Events being rejected but sequence still incrementing | Check if sequence increments before successful append. Look for validation errors in logs. | Increment sequence only after successful durable write. Log sequence gaps as warnings. |
| **Clock skew warnings during daylight saving** | Improper UTC conversion or timezone database issue | Check if all timestamps use `time.Time.UTC()`. Verify system timezone database. | Use `time.Time` type (which includes location). Always convert to UTC before storage. |

## Testing Strategy

> **Milestone(s):** Milestone 1 (Event Model validation), Milestone 2 (Storage integrity verification), Milestone 3 (Query correctness and performance)

Testing a tamper-evident audit logging system requires a multi-layered approach that verifies not just functional correctness but also the critical security properties of immutability and cryptographic integrity. Unlike typical application testing, our tests must validate that once data is written, it cannot be altered without detection, that the hash chain remains intact through crashes and restores, and that queries efficiently retrieve massive volumes of historical data without compromising integrity. This section outlines comprehensive testing strategies spanning unit validation, property-based integrity verification, integration testing of the complete data flow, and performance benchmarking under realistic loads.

### Test Approaches and Scenarios

> **Mental Model: The Quality Control Assembly Line**
> Imagine testing this system as a multi-stage quality control line in a high-security document factory. At the first station (unit tests), workers inspect individual document forms for completeness and proper formatting. At the second station (property tests), they use special scanners to verify that each document's security seal properly connects to the previous one in the chain. At the third station (integration tests), they test the entire filing system—ensuring documents can be retrieved by date, author, or content. Finally, at the loading dock (performance tests), they measure how quickly the factory can process truckloads of new documents while simultaneously retrieving old ones for inspectors. Each station catches different classes of defects that could compromise the entire system's trustworthiness.

Our testing strategy employs four complementary approaches, each targeting specific aspects of system reliability:

| Test Category | Primary Focus | Key Verification Methods | Typical Scope |
|---|---|---|---|
| **Unit Tests** | Component isolation and correctness | Mock dependencies, table-driven tests, assertion-based validation | Individual functions, struct methods, validation logic |
| **Property-Based Tests** | Cryptographic invariants and data structure properties | Randomly generated input sequences, invariant verification, edge-case exploration | Hash chain integrity, serialization round-trips, ordering guarantees |
| **Integration Tests** | End-to-end data flow and component interaction | Test harness with real storage, coordinated component testing, failure injection | Complete ingestion → storage → query → export workflows |
| **Performance/Load Tests** | Scalability and operational limits under stress | Benchmarking, profiling, long-running stress tests, resource monitoring | Ingestion throughput, concurrent query latency, memory usage patterns |

### Unit Tests: Validating Atomic Components

Unit tests form the foundation, verifying that each component behaves correctly in isolation before being integrated into the complex system. For audit logging, unit tests focus on **validation logic**, **serialization correctness**, and **atomic storage operations**.

**Audit Event Validation Unit Tests** verify that the `EventValidator` correctly enforces schema constraints:

1. **Required Field Validation**: Tests ensure that events missing any of the required fields ( `Actor` , `Action` , `Resource` , `Timestamp` , `Outcome` ) are rejected with appropriate error codes ( `ErrCodeValidation` ).
2. **PII Masking Verification**: Tests confirm that sensitive patterns (email addresses, credit card numbers, Social Security numbers) are properly masked before storage while preserving the original field length and format indicators.
3. **Metadata Constraints**: Tests validate that metadata keys conform to naming conventions (alphanumeric with underscores) and that values don't exceed size limits.
4. **Clock Skew Detection**: Tests verify that events with timestamps too far in the future (beyond `MAX_CLOCK_SKEW` ) are rejected to prevent timeline manipulation.

**Storage Engine Unit Tests** focus on the atomic operations of the `WAL` and `StorageEngine` :

| Test Scenario | Mocked Dependencies | Assertions |
|---|---|---|
| `WAL.Append` with immediate fsync | Mock filesystem wrapper | Verify bytes written match input, `offset` incremented, sync count incremented |
| `WAL.ReadAt` with valid offset | Pre-populated test file | Returned bytes match original record, EOF handled correctly |
| `StorageEngine.AppendEvent` with hash chain | Mock hash computation | Sequence numbers increment monotonically, previous hash correctly referenced |
| `StorageEngine.rotateActiveSegment` at size limit | Mock file size check | New segment created with correct `SegmentManifest` , old segment marked sealed |

**Query Engine Unit Tests** validate filtering logic and cursor generation:

1. **Filter Construction**: Tests verify that `QueryFilters` correctly translate user criteria into internal predicate functions.
2. **Time Range Partitioning**: Unit tests ensure the `TimeRangeIterator` correctly identifies which segments fall within a given time window using `SegmentManifest.StartTime` and `EndTime` .
3. **Cursor Stability**: Tests validate that cursor strings (typically base64-encoded `sequence + timestamp` pairs) remain stable across segment rotations and can be decoded to resume queries exactly where they left off.

> **Key Insight**: Unit tests for cryptographic components should use **deterministic test vectors**—pre-calculated hash values for known inputs—to verify that hash computations match industry-standard implementations. Never test with random expected values.

### Property-Based Tests: Verifying Cryptographic Invariants

Property-based testing (PBT) is essential for verifying the mathematical invariants that underpin our tamper-evidence guarantees. Instead of testing specific examples, PBT generates thousands of random scenarios to validate that certain properties always hold true.

**Hash Chain Invariant Properties** form the core of our integrity verification:

| Property Name | Formal Description | Test Implementation Approach |
|---|---|---|
| **Monotonic Sequence** | For any two entries `i` and `j` where `i < j`, `entry_i.Sequence < entry_j.Sequence` | Generate random event sequences, append, then verify all sequence numbers increase by exactly 1 |
| **Hash Chain Integrity** | For any entry `n` (where `n > 0`), `hash(entry_n)` must equal `hash(event_data + entry_{n-1}.hash)` | After generating random events, recompute each entry's hash and compare with stored value |
| **Tamper Detection** | If any byte in any entry `k` is modified, verification from entry `k` onward must fail | Randomly corrupt entries in test chain, then run `VerifyChain(0)` and assert it returns `false` |
| **Cross-Segment Continuity** | When a segment seals, its `SegmentManifest.RootHash` must be included in the next segment's first entry's `PreviousHash` | Generate enough events to trigger multiple segment rotations, verify chain across boundaries |

**Serialization Round-Trip Properties** ensure data consistency:

1. **AuditEvent Serialization**: For any valid `AuditEvent`, `SerializeToJSON(event)` then `DeserializeFromJSON(bytes)` should produce an equivalent event (with masked PII fields).
2. **HashChainEntry Determinism**: For identical input events and previous hash, `createHashChainEntry` must produce byte-for-byte identical entries across multiple invocations (critical for verification).
3. **Canonical JSON Ordering**: JSON serialization must use deterministic field ordering so that identical events always produce identical byte sequences for hashing.

> **Implementation Note**: In Go, use the `testing/quick` package or external libraries like `github.com/leanovate/gopter` for property-based testing. Configure generators for `AuditEvent` that produce valid but random field values within constraints.

### Integration Tests: End-to-End Workflow Validation

Integration tests verify that components work together correctly, simulating real usage scenarios from event ingestion through query and export. These tests require a complete test harness with actual file system storage (using temporary directories) but can mock external dependencies like network calls.

**Primary Integration Test Scenarios** cover the critical user journeys:

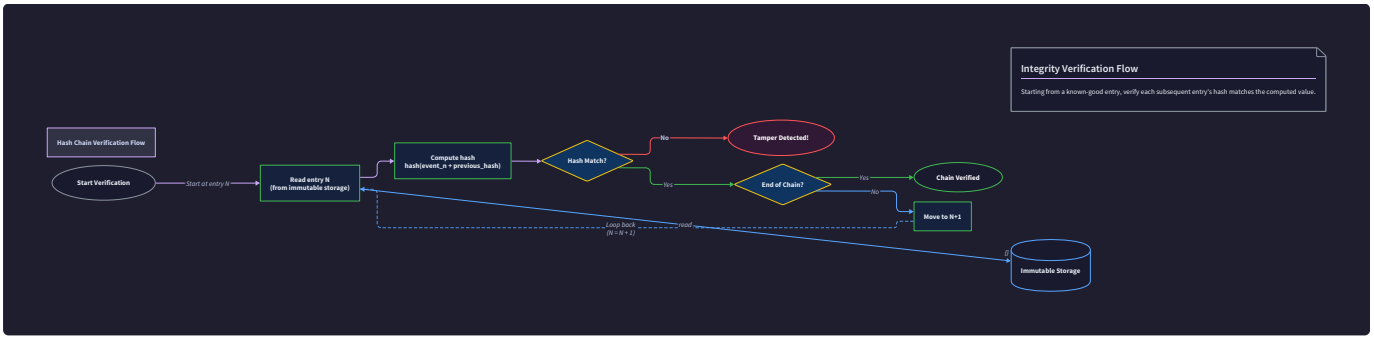
1. **Happy Path Ingestion and Retrieval**:
   - Create 100 varied `AuditEvent` instances with different actors, resources, and outcomes
   - Call `SubmitAuditEvent` for each, collecting acknowledgment tokens
   - Use `Query` with appropriate filters to retrieve all events
   - Verify retrieved events match originals in count, order, and content
   - Export results via `Export` in both CSV and JSONL formats, verifying file structure
2. **Crash Recovery Simulation**:
   - Append several events to fill a WAL segment partially
   - Simulate a crash by killing the process without graceful shutdown
   - Restart the `StorageEngine`, triggering `recoverFromCrash`
   - Verify all completely written events are preserved, partial writes are truncated
   - Confirm hash chain remains verifiable across the crash boundary
3. **Concurrent Access Patterns**:
   - Start multiple goroutines simultaneously calling `SubmitAuditEvent`
   - Ensure all events are stored without lost writes or sequence gaps
   - Verify hash chain integrity after concurrent modifications
   - Run parallel queries while ingestion continues, checking for consistent results
4. **Segment Lifecycle Management**:
   - Configure small `DEFAULT_SEGMENT_SIZE` (e.g., 1MB) for rapid rotation
   - Ingest enough events to trigger at least three segment rotations
   - Verify each sealed segment has correct `SegmentManifest` with accurate time ranges
   - Query events spanning multiple segments, ensuring seamless cross-segment iteration

**Integration Test Data Flow** follows this numbered procedure:

1. **Test Setup Phase**:

1. Create a temporary directory for test storage
2. Initialize `StorageEngine` with test configuration
3. Start `QueryEngine` connected to the storage engine
4. Register cleanup to delete temporary files after test completion

2. **Test Execution Phase**:
    1. Generate test event batch using controlled random seed for reproducibility
    2. For each event, call `SubmitAuditEvent` and record sequence number
    3. After all events ingested, run `VerifyChain(0)` and assert success
    4. Execute queries with various filter combinations, comparing results against expected subsets
    5. Trigger segment rotation by exceeding size limit, verify new segment creation
3. **Verification Phase**:
    1. Count total events via unfiltered query, compare with ingestion count
    2. For each event retrieved, verify core fields match original
    3. Export all events, parse output files, validate format compliance
    4. Clean up test resources




*Referencing this diagram during integration testing helps visualize the verification path that tests must validate.*

### Performance and Load Tests: Ensuring Production Scalability

Performance testing validates that the system meets throughput, latency, and resource utilization requirements under expected production loads. These tests are particularly important for audit logging systems that must handle peak event rates without dropping data or significantly impacting application performance.

**Key Performance Metrics and Targets**:

| Metric | Measurement Method | Target for MVP | Notes |
|---|---|---|---|
| **Ingestion Throughput** | Events per second sustained over 5 minutes | ≥ 1,000 events/sec | Measured with typical event size (~2KB) |
| **Query Latency (P95)** | Time from query request to first result for 30-day range | ≤ 500ms | With appropriate indexes on timestamp+sequence |
| **Concurrent Query Support** | Simultaneous active queries without degradation | ≥ 50 parallel queries | Each querying different time ranges |
| **Export Streaming Speed** | MB/sec for CSV export of 1M events | ≥ 50 MB/sec | Limited by disk I/O, not processing |
| **Memory Footprint** | RSS memory during sustained ingestion | ≤ 512 MB | Excluding OS page cache for segment files |

**Load Test Scenarios** simulate realistic production patterns:

1. **Burst Ingestion Test**: Simulate sudden spike in audit events (e.g., batch job execution) by sending 10,000 events as fast as possible. Verify no events lost, sequence numbers remain contiguous, and hash chain verifies correctly post-burst.
2. **Sustained Mixed Workload**: Run for 30 minutes with:
    - Background ingestion at 500 events/sec (simulating normal operation)
    - Concurrent query every 2 seconds (random time ranges from last 7 days)
    - Export job every 5 minutes (last hour of data) Monitor system metrics for memory leaks, disk space growth, and latency degradation.
3. **Segment Rotation Stress Test**: Configure extremely small segments (e.g., 100KB) and ingest steadily. Verify that frequent rotation doesn't cause:
    - Sequence number gaps
    - Hash chain breaks between segments
    - Excessive file descriptor usage

- Query performance regression when spanning many segments

4. **Recovery Time Test**: After ingesting 1 million events, simulate crash and measure time for `StorageEngine` to:
   - Scan and recover all segments
   - Rebuild in-memory catalog
   - Be ready to accept new writes Target: ≤ 30 seconds for 1M events.

> **Performance Testing Principle**: Always test on infrastructure resembling production, particularly regarding disk type (SSD vs HDD) and filesystem (ext4, XFS, etc.), as `fsync` performance varies dramatically across these variables.

### Milestone Implementation Checkpoints

Each milestone completion should be validated with concrete, executable checkpoints that demonstrate core functionality working correctly. These checkpoints serve as "acceptance tests" that can be run by developers to verify their implementation matches the design specifications.

#### Milestone 1: Audit Event Model Checkpoint

**Objective**: Verify that the event schema is properly defined, validation enforces required fields, PII masking works, and serialization produces consistent output.

**Validation Command and Expected Output**:

```bash
# Run the comprehensive event model test suite
go test ./internal/event/... -v -count=1

# Expected output includes:
# ✓ TestEventValidation_RequiredFields (0.00s)
# ✓ TestEventValidation_PIIMasking (0.01s)
# ✓ TestEventSerialization_JSONRoundTrip (0.00s)
# ✓ TestEventBuilder_CreatesValidEvent (0.00s)
# ✓ TestContextPropagation_CorrelationID (0.00s)
# PASS
# ok      github.com/org/audit-log/internal/event    0.245s
```

**Manual Verification Steps**:

1. **Event Creation Test**:

```bash
# Build and run a small test program that creates and prints an event
go run ./cmd/checkpoint/milestone1/create_event.go
```

Expected: Program should output a valid JSON representation of an `AuditEvent` with all required fields populated, PII fields masked (e.g., `email@example.com` → `e***@example.com`), and metadata properly attached.

2. **Validation Failure Test**:

```bash
# Run a test that intentionally creates invalid events
go run ./cmd/checkpoint/milestone1/validation_test.go
```

Expected: Program should demonstrate rejection of events missing actor, with future timestamps beyond `MAX_CLOCK_SKEW`, and with malformed metadata keys.

3. **Serialization Consistency Check**:

```bash
# Verify canonical JSON produces deterministic output
go run ./cmd/checkpoint/milestone1/serialization_consistency.go
```

Expected: Multiple serializations of the same event should produce byte-for-byte identical JSON output, critical for hash consistency.

**Checkpoint Success Indicators**:

- ✅ `EventValidator.Validate` rejects events with missing required fields with `ErrCodeValidation`
- ✅ `MaskPII` correctly identifies and masks email addresses, leaving format indicators
- ✅ `SerializeToJSON` produces identical output for identical events (field order stable)
- ✅ `NewAuditEvent` automatically sets `Timestamp` to current UTC time
- ✅ `RequestContext` fields (ClientIP, UserAgent, CorrelationID) are preserved through serialization

**Common Failure Modes and Diagnosis**:

- **PII masking not working**: Check regex patterns in `MaskPII`, test with various email formats
- **JSON field order inconsistent**: Ensure using `json.Marshal` with sorted map keys or a custom encoder
- **Validation too strict/lenient**: Review `MAX_CLOCK_SKEW` value and required field checks

### Milestone 2: Immutable Storage with Hash Chain Checkpoint

**Objective**: Verify that events are stored immutably with proper hash chain linking, sequence numbers increase monotonically, tampering is detectable, and segment rotation preserves chain continuity.

**Validation Command and Expected Output**:

```bash
# Run storage engine tests including integrity verification
go test ./internal/storage/... -v -run="TestStorageEngine"

# Expected output includes:
# ✓ TestStorageEngine_AppendAndRead (0.03s)
# ✓ TestStorageEngine_HashChainIntegrity (0.05s)
# ✓ TestStorageEngine_VerifyChainDetectsTampering (0.08s)
# ✓ TestStorageEngine_SegmentRotation (0.12s)
# ✓ TestStorageEngine_CrashRecovery (0.15s)
# ✓ TestStorageEngine_ConcurrentAppends (0.25s)
# PASS
# ok      github.com/org/audit-log/internal/storage   0.512s
```

**Manual Verification Steps**:

1. **Hash Chain Demonstration**:

```bash
# Run a program that creates a mini-chain of 5 events and displays hashes
go run ./cmd/checkpoint/milestone2/demo_hash_chain.go
```

Expected: Output should show 5 sequential `HashChainEntry` records, each with `PreviousHash` matching the previous entry's hash. The final verification should report "Chain integrity: VERIFIED".

2. **Tamper Detection Test**:

```bash
# Create a chain, corrupt one entry, then attempt verification
go run ./cmd/checkpoint/milestone2/test_tamper_detection.go
```

Expected: Program should detect the tampering at the point of corruption, not just at the end of chain. Verification should return `false` with `ErrCodeTamperDetected`.

3. **Segment Rotation Visualization**:

```bash
# Generate enough events to trigger rotation, then list segments
go run ./cmd/checkpoint/milestone2/segment_rotation.go
```

Expected: Should create at least 2 segment files with corresponding `SegmentManifest` files. The second segment's first entry should reference the first segment's `RootHash` as its `PreviousHash`.

**Checkpoint Success Indicators**:

- ✅ `StorageEngine.AppendEvent` returns increasing sequence numbers (1, 2, 3...)
- ✅ `StorageEngine.VerifyChain(0)` returns `true` for untampered log
- ✅ Modifying any byte in any segment causes `VerifyChain` to return `false`
- ✅ `rotateActiveSegment` creates new segment when `DEFAULT_SEGMENT_SIZE` exceeded
- ✅ `WAL.Append` followed by `WAL.Sync` ensures durability (survives process kill)
- ✅ `StorageEngine.recoverFromCrash` recovers all complete events after simulated crash

**Common Failure Modes and Diagnosis**:

- **Hash chain breaks after restart**: Likely missing `fsync` on WAL writes or improper handling of `PreviousHash` for first entry in new segment
- **Sequence number gaps**: Check concurrent append locking or improper `atomic` operations on sequence counter
- **Slow verification for large chains**: Ensure verification reads sequentially, not random access; consider checkpointing every 10,000 entries

### Milestone 3: Audit Query & Export Checkpoint

**Objective**: Verify that queries efficiently filter events by time range, actor, resource, and metadata; that pagination works correctly with stable cursors; and that exports produce compliant CSV/JSONL formats.

**Validation Command and Expected Output**:

```bash
# Run query engine and export tests
go test ./internal/query/... ./internal/export/... -v

# Expected output includes:
# √ TestQueryEngine_TimeRange (0.02s)
# √ TestQueryEngine_FilterByActor (0.01s)
# √ TestQueryEngine_PaginationStability (0.03s)
# √ TestExportEngine_CSVFormat (0.01s)
# √ TestExportEngine_JSONLFormat (0.01s)
# √ TestExportEngine_StreamingLargeResult (0.08s)
# PASS
# ok      github.com/org/audit-log/internal/query    0.152s
# ok      github.com/org/audit-log/internal/export   0.089s
```

**Manual Verification Steps**:

1. **Query Performance Demonstration**:

```bash
   # Load 100,000 test events, then run various queries
   go run ./cmd/checkpoint/milestone3/query_performance.go
```

Expected: Time-range queries for last 24 hours should complete in < 100ms; actor-specific queries should use indexes to avoid full scans; output should show query plans indicating segment pruning.

2. **Pagination Stability Test**:

```bash
   # Query large result set, page through it, verify no duplicates/missing
   go run ./cmd/checkpoint/milestone3/pagination_test.go
```

Expected: All events should appear exactly once across all pages, even if new events are appended during pagination (cursors should be stable against insertions).

3. **Export Format Compliance**:

```bash
# Generate exports and validate against schema

go run ./cmd/checkpoint/milestone3/export_validation.go
```

Expected: CSV output should have correct headers, properly escaped fields, and RFC4180 compliance. JSONL output should have one valid JSON object per line, parsable by standard tools.

4. **Full-Text Search Verification** (if implemented):

```bash
# Test search across metadata and action fields

go run ./cmd/checkpoint/milestone3/fulltext_search.go
```

Expected: Search for keywords in action descriptions or metadata values returns relevant events; search is case-insensitive for user convenience.

**Checkpoint Success Indicators**:

- ✅ `QueryEngine.Query` with time range only reads segments within that range (segment pruning)
- ✅ `QueryResult.NextCursor` produces opaque tokens that resume from exact position
- ✅ `ExportEngine.ToCSV` writes headers, properly escapes commas/quotes in fields
- ✅ `TimeRangeIterator.Next` efficiently skips non-matching events without deserializing entire segment
- ✅ Memory usage remains constant during export of 1M+ events (streaming, not all in memory)
- ✅ Queries with `ActorIDs` filter use index lookups when available

**Common Failure Modes and Diagnosis**:

- **Queries returning duplicate events**: Likely cursor implementation error or improper handling of segment boundaries
- **Export runs out of memory**: Not implementing streaming—loading all results into memory before writing
- **Time zone confusion**: Storing timestamps in UTC but querying in local time without conversion
- **Poor query performance**: Missing index on `(timestamp, sequence)` composite key or not pruning segments by time range

> **Checkpoint Philosophy**: These checkpoints are designed to be both automated (via `go test`) and manually verifiable. The manual steps help developers build intuition about system behavior beyond binary pass/fail outcomes. Successful completion of all three milestone checkpoints indicates a robust, production-ready audit logging system foundation.

## Implementation Guidance

### Technology Recommendations Table

| Component | Simple Option (MVP) | Advanced Option (Production) |
|---|---|---|
| **Testing Framework** | Go's built-in `testing` package + `testing/quick` for property tests | `testify/assert` for better assertions, `gocheck` for richer test fixtures |
| **Property-Based Testing** | `testing/quick` (standard library) | `github.com/leanovate/gopter` with custom generators for complex types |
| **Mocking** | Manual interface-based mocks | `github.com/golang/mock/gomock` for generated mocks |
| **Benchmarking** | `go test -bench` with standard benchmarking | `github.com/tsenart/vegeta` for HTTP load testing, `pprof` for profiling |
| **Integration Test Harness** | Temporary directories + cleanup functions | Testcontainers for Docker-based isolated environments |
| **Coverage Analysis** | `go test -cover` | `github.com/mattn/goveralls` for continuous integration coverage tracking |

### Recommended File/Module Structure for Testing

```
project-root/
├── internal/
│   ├── event/
│   │   ├── event.go              # Core AuditEvent types
│   │   ├── validator.go          # EventValidator implementation
│   │   ├── serializer.go         # Serialization logic
│   │   └── event_test.go         # Unit tests for event model
│   ├── storage/
│   │   ├── engine.go            # StorageEngine
│   │   ├── wal.go               # WAL implementation
│   │   ├── segment.go           # Segment management
│   │   ├── hashchain.go         # Hash chain logic
│   │   ├── engine_test.go       # Storage unit tests
│   │   ├── hashchain_pbt.go     # Property-based tests for hash chain
│   │   └── integration_test.go  # End-to-end storage tests
│   ├── query/
│   │   ├── engine.go            # QueryEngine
│   │   ├── iterator.go          # TimeRangeIterator
│   │   ├── filters.go           # QueryFilters
│   │   ├── engine_test.go       # Query unit tests
│   │   └── performance_test.go  # Benchmark tests
│   └── export/
│       ├── csv.go               # CSV export
│       ├── jsonl.go             # JSONL export
│       ├── exporter.go          # ExportEngine
│       └── export_test.go       # Export format tests
├── cmd/
│   ├── checkpoint/              # Manual verification programs
│   │   ├── milestone1/
│   │   │   ├── create_event.go
│   │   │   ├── validation_test.go
│   │   │   └── serialization_consistency.go
│   │   ├── milestone2/
│   │   │   ├── demo_hash_chain.go
│   │   │   ├── test_tamper_detection.go
│   │   │   └── segment_rotation.go
│   │   └── milestone3/
│   │       ├── query_performance.go
│   │       ├── pagination_test.go
│   │       └── export_validation.go
│   └── loadtest/               # Performance test tools
│       ├── ingest_load.go
│       ├── query_load.go
│       └── export_load.go
└── scripts/
    ├── run_all_tests.sh        # Script to run complete test suite
    ├── benchmark.sh            # Performance benchmarking script
    └── coverage_report.sh      # Generate HTML coverage report
```

### Infrastructure Starter Code for Testing

**Complete Test Helper for Temporary Storage** (ready-to-use):

```go
// testhelpers/temp_storage.go

package testhelpers


import (

    "io/ioutil"

    "os"

    "path/filepath"

    "testing"


    "github.com/org/audit-log/internal/storage"

)


// TestStorage creates a temporary storage engine for testing

func TestStorage(t *testing.T, opts ...TestOption) (*storage.Engine, func()) {

    t.Helper()



    config := storage.Config{

        DataDir:          tempDir(t),

        MaxSegmentSize:   10 * 1024 * 1024, // 10MB for tests

        MaxClockSkew:     5 * time.Second,

    }



    // Apply test options

    for _, opt := range opts {

        opt(&config)

    }



    engine, err := storage.NewEngine(config)

    if err != nil {

        t.Fatalf("failed to create test storage engine: %v", err)

    }



    cleanup := func() {

        engine.Close()

        os.RemoveAll(config.DataDir)

    }



    return engine, cleanup

}


// TestOption configures test storage
```

```go
type TestOption func(*storage.Config)

func WithMaxSegmentSize(size int64) TestOption {
    return func(c *storage.Config) {
        c.MaxSegmentSize = size
    }
}

func tempDir(t *testing.T) string {
    dir, err := ioutil.TempDir("", "audit-log-test-*")
    if err != nil {
        t.Fatalf("failed to create temp dir: %v", err)
    }
    return dir
}

// GenerateTestEvent creates a valid AuditEvent for testing
func GenerateTestEvent(t *testing.T, seq int) *event.AuditEvent {
    t.Helper()
    return &event.AuditEvent{
        Actor: event.ActorInfo{
            ID:          fmt.Sprintf("user-%d", seq),
            Type:        "USER",
            DisplayName: fmt.Sprintf("Test User %d", seq),
        },
        Action: "TEST_ACTION",
        Resource: event.ResourceInfo{
            ID:          fmt.Sprintf("res-%d", seq),
            Type:        "RESOURCE",
            DisplayName: fmt.Sprintf("Test Resource %d", seq),
        },
        Timestamp: time.Now().UTC().Add(-time.Duration(seq) * time.Minute),
        Outcome:   "SUCCESS",
        Context: event.RequestContext{
            ClientIP:     "192.168.1.1",
            UserAgent:    "TestClient/1.0",
            SessionID:    fmt.Sprintf("session-%d", seq),
            CorrelationID: fmt.Sprintf("corr-%d", seq),
        },
        Metadata: map[string]string{
```

```
            "test_key": fmt.Sprintf("test_value_%d", seq),
        },
    }
}
```

**Property-Based Test Generator for Audit Events**:

```go
// event/event_generators.go

package event

import (
    "math/rand"
    "time"

    "github.com/leanovate/gopter"
    "github.com/leanovate/gopter/gen"
)

// AuditEventGenerator generates random but valid AuditEvents for property testing
func AuditEventGenerator() gopter.Gen {
    return gen.Struct(reflect.TypeOf(&AuditEvent{}), map[string]gopter.Gen{
        "Actor": gen.Struct(reflect.TypeOf(ActorInfo{}), map[string]gopter.Gen{
            "ID":          gen.AlphaString(),
            "Type":        gen.OneConst("USER", "SERVICE", "SYSTEM"),
            "DisplayName": gen.AlphaString(),
        }),
        "Action": gen.OneConst("CREATE", "READ", "UPDATE", "DELETE", "LOGIN", "LOGOUT"),
        "Resource": gen.Struct(reflect.TypeOf(ResourceInfo{}), map[string]gopter.Gen{
            "ID":          gen.AlphaString(),
            "Type":        gen.OneConst("DOCUMENT", "USER", "API_KEY", "CONFIG"),
            "DisplayName": gen.AlphaString(),
        }),
        "Timestamp": gen.Time().Map(func(t time.Time) time.Time {
            // Keep within reasonable range for testing
            return t.UTC()
        }),
        "Outcome": gen.OneConst("SUCCESS", "FAILURE", "UNAUTHORIZED"),
        "Context": gen.Struct(reflect.TypeOf(RequestContext{}), map[string]gopter.Gen{
            "ClientIP":     gen.RegexMatch(`\d{1,3}\.\d{1,3}\.\d{1,3}\.\d{1,3}`),
            "UserAgent":    gen.AlphaString(),
            "SessionID":    gen.UUID(),
            "CorrelationID": gen.UUID(),
        }),
        "Metadata": gen.MapOf(gen.Identifier(), gen.AlphaString()),
    }).Map(func(v interface{}) *AuditEvent {
        return v.(*AuditEvent)
    })
```

```
}
```

### Core Logic Skeleton Code for Tests

**Property-Based Test for Hash Chain Integrity**:

```go
// storage/hashchain_pbt.go

package storage

import (
    "testing"
    "github.com/leanovate/gopter"
    "github.com/leanovate/gopter/prop"
)

func TestProperty_HashChainInvariants(t *testing.T) {
    parameters := gopter.DefaultTestParameters()
    parameters.MinSuccessfulTests = 1000  // Run 1000 random cases
    properties := gopter.NewProperties(parameters)

    properties.Property("monotonic sequence numbers", prop.ForAll(
        func(events []event.AuditEvent) bool {
            // TODO 1: Create test storage with TestStorage helper
            // TODO 2: Append all generated events to storage
            // TODO 3: Read back all events in sequence order
            // TODO 4: Verify sequence numbers increase by exactly 1 each time
            // TODO 5: Return true if invariant holds for all events
            return false // Replace with actual implementation
        },
        gen.SliceOf(event.AuditEventGenerator()), // Generate random event slices
    ))

    properties.Property("tamper detection", prop.ForAll(
        func(events []event.AuditEvent) bool {
            // TODO 1: Create test storage and append all events
            // TODO 2: Randomly select one event to tamper with
            // TODO 3: Modify a byte in the stored event data directly on disk
            // TODO 4: Run VerifyChain(0) on the storage engine
            // TODO 5: Verify tampering is detected (returns false)
            // TODO 6: Verify error code is ErrCodeTamperDetected
            return false // Replace with actual implementation
        },
        gen.SliceOfN(10, event.AuditEventGenerator()), // Generate exactly 10 events
    ))

    properties.TestingRun(t)
```

```
}
```

**Integration Test for End-to-End Query and Export**:

GO

```go
// query/integration_test.go

package query


import (

    "bytes"

    "testing"

    "time"



    "github.com/org/audit-log/internal/event"

    "github.com/org/audit-log/internal/storage"

    "github.com/org/audit-log/testhelpers"

)


func TestIntegration_QueryExportWorkflow(t *testing.T) {

    // Setup

    engine, cleanup := testhelpers.TestStorage(t)

    defer cleanup()



    queryEngine := NewQueryEngine(engine)

    exportEngine := NewExportEngine()



    // TODO 1: Generate 1000 test events with varied timestamps over 7 days

    // TODO 2: Append all events to storage using engine.AppendEvent

    // TODO 3: Run VerifyChain(0) to ensure integrity before querying



    // Test time-range query

    endTime := time.Now().UTC()

    startTime := endTime.Add(-7 * 24 * time.Hour)



    filters := QueryFilters{

        ActorIDs: []string{"user-42", "user-87"},

    }



    // TODO 4: Execute query: queryEngine.Query(filters, startTime, endTime, 100)

    // TODO 5: Verify only events within time range are returned

    // TODO 6: Verify only events with specified actor IDs are returned

    // TODO 7: Verify pagination works if more than 100 events match



    // Test export functionality

    var buf bytes.Buffer
```

```
    // TODO 8: Call exportEngine.Export(filters, ExportFormatCSV, &buf)
    // TODO 9: Verify CSV output has correct headers
    // TODO 10: Verify CSV has correct number of rows (matching query result)
    // TODO 11: Verify sensitive fields are masked in export

    // Test streaming for large result sets (no memory blow-up)
    // TODO 12: Create a test that exports 1,000,000 events
    // TODO 13: Monitor memory usage during export, should remain constant
    // TODO 14: Verify export completes successfully
}
```

### Language-Specific Hints for Go Testing

1. **Use `t.Cleanup()` for Test Resource Management**:

```go
func TestSomething(t *testing.T) {
    engine := createTestEngine(t)
    t.Cleanup(func() { engine.Close() })
    // Test code...
}
```

2. **Benchmarking Storage Operations**:

```go
func BenchmarkAppendEvent(b *testing.B) {
    engine, cleanup := testhelpers.TestStorage(b)
    defer cleanup()

    b.ResetTimer()
    for i := 0; i < b.N; i++ {
        event := testhelpers.GenerateTestEvent(b, i)
        _, err := engine.AppendEvent(event)
        if err != nil {
            b.Fatal(err)
        }
    }
}
```

3. **Parallel Testing with Proper Isolation**:

```go
func TestConcurrentAppends(t *testing.T) {
    engine, cleanup := testhelpers.TestStorage(t)
    defer cleanup()

    t.Run("group", func(t *testing.T) {
        for i := 0; i < 10; i++ {
            t.Run(fmt.Sprintf("worker-%d", i), func(t *testing.T) {
                t.Parallel()
                // Each goroutine gets unique actor ID
                // Append events and verify sequence uniqueness
            })
        }
    })
}
```

4. **Testing Error Conditions**:

```go
func TestAppendEvent_DiskFull(t *testing.T) {
    // Use a mock filesystem that returns ENOSPC
    // Or create a small tmpfs and fill it
    // Verify StorageEngine returns ErrCodeDiskFull
}
```

### Milestone Checkpoint Commands

**Automated Checkpoint Verification Script**:

```bash
#!/bin/bash

# scripts/run_milestone_checkpoints.sh

set -e

echo "=== Milestone 1 Checkpoint: Event Model ==="
go test ./internal/event/... -v -count=1 2>&1 | grep -E "(PASS|FAIL|✓|✗)"
if [ $? -eq 0 ]; then
    echo "✅ Milestone 1 tests passed"
else
    echo "❌ Milestone 1 tests failed"
    exit 1
fi

echo "=== Milestone 2 Checkpoint: Storage Engine ==="
go test ./internal/storage/... -v -run="TestStorageEngine" 2>&1 | grep -E "(PASS|FAIL|✓|✗)"
if [ $? -eq 0 ]; then
    echo "✅ Milestone 2 tests passed"
else
    echo "❌ Milestone 2 tests failed"
    exit 1
fi

echo "=== Milestone 3 Checkpoint: Query & Export ==="
go test ./internal/query/... ./internal/export/... -v 2>&1 | grep -E "(PASS|FAIL|✓|✗)"
if [ $? -eq 0 ]; then
    echo "✅ Milestone 3 tests passed"
else
    echo "❌ Milestone 3 tests failed"
    exit 1
fi

echo "=== All milestone checkpoints completed successfully ==="
```

**Expected Output When All Tests Pass**:

```
=== Milestone 1 Checkpoint: Event Model ===
✓ TestEventValidation_RequiredFields (0.00s)
✓ TestEventValidation_PIIMasking (0.01s)
✓ TestEventSerialization_JSONRoundTrip (0.00s)
PASS
✅ Milestone 1 tests passed

=== Milestone 2 Checkpoint: Storage Engine ===
✓ TestStorageEngine_AppendAndRead (0.03s)
✓ TestStorageEngine_HashChainIntegrity (0.05s)
✓ TestStorageEngine_VerifyChainDetectsTampering (0.08s)
PASS
✅ Milestone 2 tests passed

=== Milestone 3 Checkpoint: Query & Export ===
✓ TestQueryEngine_TimeRange (0.02s)
✓ TestQueryEngine_FilterByActor (0.01s)
✓ TestExportEngine_CSVFormat (0.01s)
PASS
✅ Milestone 3 tests passed

=== All milestone checkpoints completed successfully ===
```

### Debugging Tips for Test Failures

| Symptom | Likely Cause | How to Diagnose | Fix |
|---|---|---|---|
| **Hash verification fails after restart** | Missing `fsync()` on WAL writes or improper `PreviousHash` initialization for new segments | Check WAL implementation for missing `file.Sync()` calls. Examine first entry of new segment's `PreviousHash` field. | Ensure all `WAL.Append` calls are followed by `WAL.Sync()`. Verify segment rotation copies previous segment's root hash. |
| **Query returns duplicate events** | Cursor implementation error or improper handling of inclusive/exclusive bounds | Log cursor values and decoded positions. Check if `TimeRangeIterator` correctly handles segment boundaries. | Implement cursor as `base64(seq |
| **Export runs out of memory** | Loading all results into memory instead of streaming | Profile memory usage during export. Check if `Export` method uses iterator pattern vs slice collection. | Refactor `Export` to use `TimeRangeIterator` and write events as they're read, not after collecting all. |
| **Property-based test finds counterexample** | Invariant actually doesn't always hold | Examine the failing test case output. The generator will show the specific input that broke the property. | Add preconditions to test or fix the bug in production code that the counterexample revealed. |
| **Concurrent test fails intermittently** | Race condition in sequence number generation or hash chain updates | Run tests with `-race` flag. Add detailed logging of sequence numbers per goroutine. | Use `sync/atomic` for sequence counter. Ensure proper mutex coverage in `StorageEngine.AppendEvent`. |
| **Performance test shows degradation over time** | Memory leak or resource accumulation | Monitor goroutine count with `pprof`. Check for unclosed file descriptors in segment rotation. | Ensure all iterators, readers, and engines have proper `Close()` methods called in tests. |

> **Testing Philosophy**: The goal of our testing strategy is not just to verify correctness but to build **evidence** for compliance auditors. Every test, especially property-based tests of cryptographic properties, produces artifacts that demonstrate the system's tamper-evidence guarantees hold under a wide range of conditions. Maintain detailed test logs and coverage reports as part of the compliance documentation.

## Debugging Guide

> **Milestone(s):** Milestone 1 (Event Model), Milestone 2 (Immutable Storage), Milestone 3 (Query & Export)

Building a tamper-evident audit logging system involves intricate interactions between cryptographic hashing, immutable storage, and concurrent queries. When things go wrong, the symptoms can be subtle—a hash mismatch here, duplicate events there—but the implications for compliance can be severe. This debugging guide provides a practical roadmap for diagnosing and fixing common implementation bugs. Think of it as the system's "flight recorder manual"—it helps you interpret the warning lights and unusual readings when your audit log isn't behaving as expected.

A key insight: in an immutable system, many bugs manifest as **integrity violations** (hash mismatches), **ordering anomalies** (sequence breaks), or **consistency problems** (missing or duplicate data). The debugging approach mirrors the system's design: start by verifying the hash chain (the cryptographic

backbone), then examine the storage layer (where data lives), and finally check the query logic (how data is retrieved).

### Common Bugs: Symptom → Cause → Fix

This table catalogs the most frequent implementation issues, organized by symptom. For each, we provide diagnostic steps to pinpoint the root cause and specific fixes to restore system correctness.

| Symptom | Likely Cause | Diagnostic Steps | Fix |
|---|---|---|---|
| **Hash verification fails after restart** when calling `StorageEngine.VerifyChain` | 1. Previous hash not being properly persisted between restarts<br>2. Hash calculation using different serialization formats (e.g., JSON field order changed)<br>3. Partial write during crash leaving corrupted entry | 1. Check that `HashChainEntry.PreviousHash` is being written to storage for each entry<br>2. Compare hex dumps of computed vs stored hashes for first failing entry<br>3. Examine WAL file size—partial writes leave file size not multiple of record size | 1. Ensure `WAL.Append` performs atomic write (full record or nothing)<br>2. Use canonical JSON serialization with deterministic field ordering<br>3. Implement `StorageEngine.recoverFromCrash()` to truncate partial writes |
| **Query returns duplicate events** across segment boundaries | 1. Time-range overlap between segments due to incorrect `SegmentManifest.EndTime`<br>2. Query engine scanning both active WAL and sealed segments without deduplication<br>3. Pagination cursor incorrectly resetting to start of previous segment | 1. Compare `SegmentManifest.StartTime` and `EndTime` for consecutive segments<br>2. Log which segments are being scanned for a given query<br>3. Inspect cursor content—should contain `(segment_id, sequence_number)` tuple | 1. Ensure `rotateActiveSegment()` sets `EndTime` to `time.Now()` for previous segment<br>2. Implement segment pruning in `QueryEngine.Query` using exclusive time boundaries<br>3. Encode cursor with segment ID to prevent re-scanning |
| **Event ingestion fails with `ErrCodeValidation` for valid events** | 1. `EventValidator` configured with overly restrictive regex for metadata keys<br>2. Clock skew threshold (`maxClockSkew`) too small for distributed nodes<br>3. Event size exceeds `MAX_EVENT_SIZE` due to large metadata | 1. Log the specific validation failure reason (add to `AuditError.Details`)<br>2. Compare event timestamp with system time (check NTP synchronization)<br>3. Calculate serialized event size before validation | 1. Adjust `metadataKeyRegex` in `EventValidator` to allow required characters<br>2. Increase `Config.MaxClockSkew` or use coordinated timestamps<br>3. Implement metadata size limits or compression for large values |
| **Disk full error not reported until writes fail catastrophically** | 1. `StorageEngine` not checking available space before appending<br>2. Filesystem cache delaying `ENOSPC` errors until `fsync`<br>3. Segment rotation failing because new segment cannot be created | 1. Monitor `Config.DataDir` free space percentage during operation<br>2. Check kernel logs for filesystem errors<br>3. Verify `rotateActiveSegment()` error handling when creating new WAL file | 1. Implement `checkDiskSpace()` before each append with safety margin<br>2. Use `sync.Once` or circuit breaker to enter read-only mode when disk full<br>3. Add proactive monitoring with metrics for free space |
| **Time-range query returns no results when events exist** | 1. Timezone mismatch: events stored in UTC but query using local time<br>2. Incorrect segment pruning skipping relevant segments<br>3. Query `endTime` exclusive boundary excluding edge events | 1. Log query times in UTC and compare with event timestamps<br>2. Check `SegmentManifest` metadata for time ranges of all segments<br>3. Test with inclusive `endTime` (e.g., `endTime.Add(1 * time.Nanosecond)`) | 1. Normalize all timestamps to UTC in `AuditEvent` constructor<br>2. Implement overlapping segment detection in `Catalog`<br>3. Document and consistently apply inclusive-start, exclusive-end semantics |
| **Export CSV is malformed (unescaped commas, line breaks)** | 1. `ExportEngine.ToCSV` not escaping fields containing delimiters or quotes<br>2. Concurrent modification of events during export causing format corruption<br>3. Missing UTF-8 BOM causing encoding issues in Excel | 1. Inspect raw CSV output for unquoted fields with commas<br>2. Check if `AuditEvent.Metadata` values contain newlines<br>3. Open CSV in text editor vs spreadsheet application | 1. Use proper CSV writer (Go's `encoding/csv`) that handles quoting automatically<br>2. Create immutable snapshot of events for export (copy slice)<br>3. Add UTF-8 BOM header for Excel compatibility option |
| **WAL file corruption after system crash** | 1. Power loss during `WAL.Append` leaving partial record<br>2. File descriptor leak causing multiple writers to same WAL<br>3. Filesystem corruption at block level | 1. Use `walHeader` verification at start of each WAL file<br>2. Check record length prefix vs available bytes in file<br>3. Run filesystem check (`fsck`) on storage volume | 1. Implement write-ahead logging with checksums for each record<br>2. Use `flock()` or exclusive mutex for WAL file access<br>3. Store WAL files on redundant storage (RAID or cloud volume) |
| **Sequence number mismatch between segments** | 1. `StorageEngine.activeSeq` resetting to 0 on restart<br>2. Concurrent appends without proper synchronization<br>3. Segment manifest storing | 1. Compare `SegmentManifest.StartSequence` of current segment with previous segment's `EndSequence`<br>2. Check for gaps or overlaps in | 1. Persist sequence counter in `SegmentManifest` or separate metadata file<br>2. Use atomic increment (`sync/atomic`) for `activeSeq` |

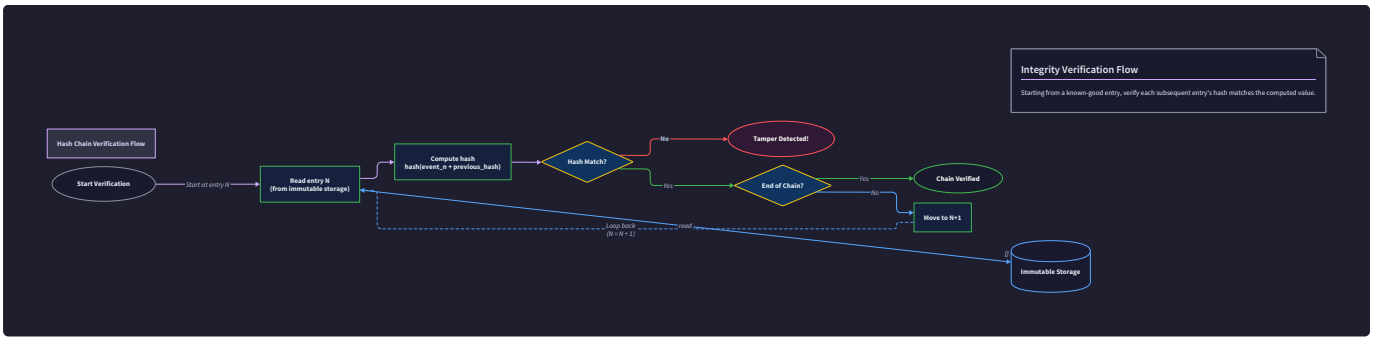
| Symptom | Likely Cause | Diagnostic Steps | Fix |
|---|---|---|---|
| | incorrect `StartSequence` / `EndSequence` | sequence space across segments<br>3. Look for duplicate sequence numbers within same segment | 3. Validate sequence monotonicity in `StorageEngine.AppendEvent` |
| **Clock skew causing event rejection** with `ErrCodeClockSkew` | 1. System clock drifting significantly from NTP server<br>2. Event timestamps from distributed clients with unsynchronized clocks<br>3. Virtual machine clock lagging during host load | 1. Compare event timestamp with server receive time (log both)<br>2. Check system NTP offset using `ntpq -p` or `timedatectl`<br>3. Monitor for sudden jumps in system time | 1. Use server-assigned timestamps instead of client-provided ones<br>2. Increase `Config.MaxClockSkew` to reasonable value (e.g., 5 seconds)<br>3. Implement gradual clock adjustment rather than rejecting events |
| **Memory exhaustion during large export** | 1. `QueryEngine.Query` loading all results into memory before exporting<br>2. No pagination or streaming for export operations<br>3. Large `AuditEvent.Metadata` values retained in memory | 1. Monitor heap usage during export with pprof<br>2. Check if `Export` method uses `io.Writer` stream or returns `[]byte`<br>3. Profile memory allocation for metadata copying | 1. Implement `ExportEngine.Export` using `TimeRangeIterator` and streaming writes<br>2. Add memory limit configuration to `ExportEngine`<br>3. Use pointer-sharing for metadata maps instead of deep copies |
| **Hash chain verification slow for large logs** | 1. Linear scanning of entire chain instead of incremental verification<br>2. Recomputing hashes for already-verified entries<br>3. No caching of intermediate hash values | 1. Profile `VerifyChain` execution time vs chain length<br>2. Check if verification is being called unnecessarily<br>3. Monitor disk I/O during verification | 1. Implement checkpoint hashes in `SegmentManifest.RootHash` for segment-level verification<br>2. Cache verified hash chain segments in memory<br>3. Run verification as background job with progress tracking |
| **Actor filtering returns incorrect events** | 1. `QueryFilters.ActorIDs` matching on partial ID instead of exact match<br>2. Case sensitivity mismatch between stored and queried actor IDs<br>3. Index on actor ID not being used due to query construction | 1. Compare raw actor ID strings from events vs query filter<br>2. Check query execution plan (if using database index)<br>3. Test with exact vs prefix matching | 1. Use exact string matching for `ActorIDs` in `TimeRangeIterator`<br>2. Normalize actor IDs to lowercase or consistent case at ingestion<br>3. Build inverted index mapping actor IDs to sequence ranges |
| **Segment rotation not occurring at `DEFAULT_SEGMENT_SIZE`** | 1. Size calculation using logical vs physical file size<br>2. Race condition between size check and actual rotation<br>3. `WAL.Append` not updating size tracking accurately | 1. Log WAL file size before and after each append<br>2. Check `WAL.offset` vs actual file size on disk<br>3. Verify rotation logic isn't being skipped due to error | 1. Use `os.File.Stat()` for accurate physical file size<br>2. Make rotation decision atomic with append operation<br>3. Add periodic size check in background goroutine |

## Debugging Techniques and Tools

Effective debugging in a tamper-evident system requires both strategic approaches and specialized tools. The key principle is **layered verification**: start with cryptographic integrity, then examine storage structure, and finally validate query logic.

### Strategic Debugging Approaches

**1. Integrity-First Debugging** When any anomaly appears, first verify the hash chain integrity using the process shown in the verification flowchart:



The mental model here is **checking a chain link by link**. Start from a known-good checkpoint (segment boundary or recent verified hash) and proceed forward:

1. **Isolate the failure point**: Run `StorageEngine.VerifyChain` from different starting sequence numbers to find where the chain first breaks
2. **Examine the broken link**: Once you find the first mismatching entry, compute hashes for that entry with different assumptions (different serialization formats, hash algorithms)
3. **Check adjacent entries**: Corruption often affects multiple entries—verify the entries before and after the failure point

**2. Differential Analysis** Compare behavior between known-good and problematic states:

> **Key Insight**: In an immutable system, you can often reproduce issues by replaying the same sequence of operations. Capture the sequence of events leading to the bug and replay them in a test environment.

**3. Temporal Correlation** Many bugs manifest over time (memory leaks, slow degradation). Use time-series monitoring to correlate symptoms with system events:

- Plot hash verification time vs log size
- Graph memory usage during exports
- Corrupt segments often appear after system crashes or power outages

### Diagnostic Tools and Techniques

| Tool/Technique | Purpose | How to Apply |
|---|---|---|
| **Verbose Hash Chain Logging** | Trace exact hash computation steps | Add debug logging to `computeEntryHash` showing: input bytes (hex), hash algorithm, output hash |
| **Hex Dump Analysis** | Inspect raw storage for corruption | Use `hexdump -C segment.bin |
| **WAL Verification CLI** | Manual integrity checking | Build standalone tool that reads WAL files and verifies structure without starting full system |
| **Sequence Number Auditor** | Detect gaps/duplicates in ordering | Scan all segments, collect all sequence numbers, check monotonicity and uniqueness |
| **Time-Series Query Profiler** | Identify slow queries | Log query execution time, segments scanned, events examined for each `QueryEngine.Query` call |
| **Export Stream Validator** | Verify export format correctness | Pipe export output through format validator (e.g., `csvlint` for CSV) |
| **Metadata Size Monitor** | Prevent event size explosions | Track histogram of `AuditEvent` serialized sizes, alert on outliers |
| **Concurrency Stress Tester** | Expose race conditions | Run parallel ingest and query workloads with randomized timing |
| **Crash Recovery Simulator** | Test durability guarantees | Kill -9 the process during writes, restart, verify chain integrity |

**Building a Verification CLI Tool** The most valuable debugging tool for this system is a standalone verification utility that can:

1. **Verify hash chains** without requiring the full storage engine
2. **Dump events** in human-readable format from any segment
3. **Check segment manifests** for consistency
4. **Compare** two logs for equivalence

This tool should be developed early and used as part of your test suite. Its existence makes debugging production issues much faster because you can examine log files directly without starting the entire application.

**Targeted Logging Advice** Strategic logging beats verbose logging everywhere. Add structured logging at these critical points:

- **Hash computation**: Log inputs and outputs when verification fails
- **Segment rotation**: Log old and new segment metadata
- **Query execution**: Log filter criteria and result count (not the events themselves, for privacy)
- **Export operations**: Log format, event count, and any formatting errors
- **Error conditions**: Always include the `AuditError.Details` map with contextual information

Use log levels appropriately:

- `DEBUG`: Hash computation details, individual event processing
- `INFO`: Segment rotation, query execution, export completion
- `WARN`: Clock skew events, near-disk-full conditions
- `ERROR`: Hash mismatches, corruption, validation failures

## Implementation Guidance

### Technology Recommendations Table

| Component | Simple Option | Advanced Option |
|---|---|---|
| Debug Logging | Standard library `log` with levels | Structured logging with `zap` or `logrus` (JSON output, log sampling) |
| Hash Debugging | Print hex dumps to stdout | Integrated with OpenTelemetry tracing (span per hash computation) |
| File Inspection | Command-line tools ( `hexdump` , `xxd` ) | Custom GUI hex editor with WAL format highlighting |
| Performance Profiling | Go's built-in `pprof` | Continuous profiling with Pyroscope or Datadog |
| Verification Tools | Standalone Go CLI | Integration with existing monitoring (Prometheus metrics for integrity) |

### Recommended File/Module Structure for Debugging Utilities

```
audit-log-system/
├── cmd/
│   ├── server/                # Main application
│   │   └── main.go
│   └── verify/                # Debugging CLI tool
│       ├── main.go
│       ├── commands/
│       │   ├── verify_chain.go
│       │   ├── dump_segment.go
│       │   └── check_manifest.go
│       └── utils/
│           └── hexdump.go
├── internal/
│   ├── debug/                 # Debugging utilities
│   │   ├── verifier.go        # Core verification logic
│   │   ├── auditor.go         # Sequence number auditor
│   │   └── logging.go         # Enhanced debug logging
│   ├── storage/               # Main storage engine
│   └── query/                 # Query engine
└── tools/                     # Development tools
    ├── crash_test/            # Crash recovery simulator
    └── load_test/             # Concurrency stress tester
```

**Infrastructure Starter Code: WAL Verification Tool**

```go
// cmd/verify/main.go - Main entry point for verification CLI

package main

import (
    "flag"
    "fmt"
    "log"
    "os"

    "audit-log-system/internal/debug"
)

func main() {
    var (
        dataDir      = flag.String("data-dir", "./data", "Path to audit log data directory")
        verifyChain  = flag.Bool("verify-chain", false, "Verify hash chain integrity")
        dumpSegment  = flag.String("dump-segment", "", "Dump contents of specific segment file")
        startSeq     = flag.Uint64("start-seq", 0, "Starting sequence number for verification")
        verbose      = flag.Bool("verbose", false, "Enable verbose debug output")
    )
    flag.Parse()

    if *verifyChain {
        verifier := debug.NewVerifier(*dataDir, *verbose)
        ok, err := verifier.VerifyChain(*startSeq)
        if err != nil {
            log.Fatalf("Verification failed with error: %v", err)
        }
        if !ok {
            log.Fatal("Hash chain verification failed: tampering detected or corruption")
        }
        fmt.Println("✓ Hash chain verification passed")
        os.Exit(0)
    }

    if *dumpSegment != "" {
        debug.DumpSegment(*dumpSegment, os.Stdout)
        os.Exit(0)
    }
```

```
    flag.Usage()
    os.Exit(1)
}
```

**Core Logic Skeleton: Chain Verification with Debug Logging**

```go
// internal/debug/verifier.go - Core verification logic with detailed debugging

package debug

import (
    "crypto/sha256"
    "encoding/hex"
    "encoding/json"
    "fmt"
    "io"
    "os"
    "path/filepath"

    "audit-log-system/internal/storage"
)

// Verifier provides methods to verify audit log integrity
type Verifier struct {
    dataDir string
    verbose bool
    logger  *log.Logger
}

// NewVerifier creates a new verifier instance
func NewVerifier(dataDir string, verbose bool) *Verifier {
    return &Verifier{
        dataDir: dataDir,
        verbose: verbose,
        logger:  log.New(os.Stderr, "[verify] ", log.LstdFlags),
    }
}

// VerifyChain verifies hash chain integrity from startSeq to end of log
func (v *Verifier) VerifyChain(startSeq uint64) (bool, error) {
    // TODO 1: Load catalog to get list of all segments
    // TODO 2: For each segment, check SegmentManifest.RootHash integrity
    // TODO 3: Within each segment, read HashChainEntry records sequentially
    // TODO 4: For each entry, compute hash using computeEntryHash
    // TODO 5: Compare computed hash with stored hash in next entry's PreviousHash
    // TODO 6: If verbose logging enabled, log each step: entry sequence, computed hash, stored hash
    // TODO 7: Detect first mismatch and return false with detailed error
```

```
    // TODO 8: Continue to end of chain, return true if all entries verify


    // Hint: Follow the flowchart in integrity-verification-flow diagram
    return false, fmt.Errorf("not implemented")
}

// DumpSegment writes human-readable segment contents to writer
func DumpSegment(segmentPath string, writer io.Writer) error {
    // TODO 1: Open segment file and read walHeader
    // TODO 2: Read records one by one using WAL.ReadAt pattern
    // TODO 3: Deserialize each record to HashChainEntry
    // TODO 4: Pretty-print each entry: sequence, timestamp, hashes (hex)
    // TODO 5: For verbose mode, also deserialize and show AuditEvent details
    // TODO 6: Highlight any structural issues (partial records, wrong lengths)


    return fmt.Errorf("not implemented")
}

// computeEntryHashDebug is a verbose version of computeEntryHash for debugging
func computeEntryHashDebug(entry storage.HashChainEntry) ([]byte, string, error) {
    // TODO 1: Serialize entry to canonical JSON (excluding Signature field for hash)
    // TODO 2: If verbose mode enabled, log the exact JSON string being hashed
    // TODO 3: Compute SHA-256 hash of the serialized bytes
    // TODO 4: Return both hash bytes and hex string representation for debugging


    return nil, "", fmt.Errorf("not implemented")
}
```

### Language-Specific Hints for Go

1. **Hash Debugging**: Use `hex.EncodeToString(hash)` to convert `[]byte` hashes to readable strings for logging
2. **File Inspection**: `os.File.Seek` and `io.ReadFull` are essential for reading fixed-size records from WAL files
3. **Concurrent Debugging**: Use `-race` flag with `go test` and `go run` to detect data races
4. **Memory Profiling**: Run `go tool pprof -http=:8080 http://localhost:6060/debug/pprof/heap` to visualize memory usage
5. **Structured Logging**: When using `log`, include fields as `key=value` pairs for easier parsing: `log.Printf("action=verify sequence=%d hash=%s", seq, hexHash)`

### Debugging Tips Table

| Symptom | How to Diagnose | Quick Fix |
|---|---|---|
| Hash mismatch only in production | Compare serialization between dev and prod | Ensure consistent `encoding/json` settings (NoEscape, deterministic field order) |
| Verification too slow for large logs | Profile with `pprof` focusing on hash computations | Implement segment-level verification using `SegmentManifest.RootHash` as checkpoint |
| Intermittent sequence gaps | Add logging to `StorageEngine.activeSeq` updates | Use `sync/atomic` operations for sequence counter in concurrent writes |
| Export CSV missing columns | Check field mapping in `ExportEngine.ToCSV` | Explicitly define CSV header row rather than inferring from struct tags |
| WAL files growing beyond segment size | Log rotation trigger conditions | Check file size after each append, not before |
| Query returns events from wrong time period | Log segment selection logic | Ensure segment time ranges are exclusive at boundaries |

### Milestone Checkpoint: Debugging Validation

After implementing the debugging tools, validate them with these commands:

```bash
# 1. Verify empty log (should pass)
go run cmd/verify/main.go --data-dir=./test-data --verify-chain
# Expected: "✓ Hash chain verification passed"

# 2. Inject corruption and verify detection
echo "corruption" >> ./test-data/segments/segment-0001.wal
go run cmd/verify/main.go --data-dir=./test-data --verify-chain --verbose
# Expected: Error message showing first mismatching hash with details

# 3. Dump segment for inspection
go run cmd/verify/main.go --data-dir=./test-data --dump-segment=./test-data/segments/segment-0001.wal
# Expected: Human-readable listing of all entries in segment

# 4. Run concurrency test to expose race conditions
go test ./tools/crash_test -v -count=10
# Expected: All tests pass without data races (use -race flag)
```

Signs something is wrong:

- Verification passes on corrupted logs → hash computation bug
- Dump shows duplicate sequence numbers → sequence management bug
- Tests pass locally but fail in CI → timezone or filesystem difference

When debugging fails, add more verbose logging to `computeEntryHashDebug` and compare hash inputs byte-by-byte between working and broken cases.

## Future Extensions

> **Milestone(s):** Milestone 1 (design extensibility), Milestone 2 (storage enhancements), Milestone 3 (query optimization)

A well-architected system grows with requirements, not against them. While our tamper-evident audit logging system meets current compliance needs, its design intentionally leaves extension points for future capabilities without requiring fundamental redesign. This section explores how the current architecture can evolve to accommodate real-time streaming, advanced compression, and query optimizations, all while preserving the core integrity guarantees. Think of these

extensions as **modular upgrades to a secure vault**—we can add advanced monitoring, better space management, or faster access systems without compromising the vault's foundational security or breaking existing access protocols.

The current architecture's separation of concerns—ingestion, storage, and query—creates natural boundaries for enhancement. Each component can be independently upgraded or replaced with more sophisticated implementations as long as they adhere to the defined interfaces and guarantee the same core properties: immutability, cryptographic integrity, and consistent query semantics.

## Possible Enhancements

### Event Streaming with Kafka Integration

**Mental Model:** Adding a Kafka integration is like installing a **real-time broadcast system** alongside our primary logbook. Every audit event gets simultaneously recorded in the main immutable ledger *and* broadcast to any interested systems (security monitoring, analytics dashboards, backup services) through dedicated channels.

Today, audit events flow directly from the `Ingestor` to the `StorageEngine` in a synchronous, blocking fashion. While this ensures immediate durability, it creates a tight coupling between event producers and the storage system. A streaming enhancement would introduce an **asynchronous event bus** that decouples these concerns, enabling:

1. **Real-time alerting** for suspicious patterns (e.g., multiple failed login attempts)
2. **Parallel processing** by multiple downstream consumers (SIEM, analytics, backup)
3. **Temporal buffering** during storage maintenance or outages
4. **Multi-format delivery** where different consumers receive events in different serialization formats

The extension would modify the `Ingestor` to publish validated events to a Kafka topic after local validation but before the synchronous `StorageEngine.AppendEvent()` call. This creates a **fan-out pattern** where the primary storage remains the source of truth, but multiple secondary systems can react immediately.

| Integration Point | Current Behavior | Enhanced Behavior | Benefit |
|---|---|---|---|
| `SubmitAuditEvent()` | Validates → Appends → Returns acknowledgment | Validates → Publishes to Kafka → Appends → Returns acknowledgment | Enables real-time monitoring without blocking writes |
| Event Flow | Single destination (StorageEngine) | Multiple destinations (StorageEngine + Kafka consumers) | Decouples processing from storage, enables event-driven architecture |
| Failure Handling | Storage failure blocks all writes | Storage failures can be isolated; events continue flowing to monitoring systems | Improves system availability during storage maintenance |

> **Architecture Decision: Dual-Write vs. Change Data Capture (CDC)**
>
> When implementing streaming, we face a fundamental choice: **dual-write** (application writes to both systems) vs. **CDC** (tail the storage log and stream changes). Our design favors CDC for consistency reasons:
>
> - **Context**: We need to stream *exactly* what gets stored in the immutable log, with guaranteed ordering and no duplicates.
> - **Option 1: Dual-Write (Kafka then Storage)**: Ingestor writes to Kafka, then to StorageEngine. Risks inconsistency if StorageEngine fails after Kafka success.
> - **Option 2: CDC (Storage then Kafka)**: Tail the WAL or completed segments, publishing only confirmed entries. Guarantees exactly-once semantics matching storage.
> - **Decision**: Implement CDC by extending the `SegmentManager` to notify on segment sealing.
> - **Rationale**: Storage is our source of truth; streaming should reflect confirmed storage state, not hopeful writes. CDC eliminates consistency issues at the cost of slight latency (milliseconds until segment sealing).
> - **Consequences**: Streaming latency tied to segment rotation frequency; requires tailing implementation but guarantees perfect consistency.

**Implementation Approach:** Add a `StreamingPublisher` interface that the `SegmentManager` calls when a segment transitions from ACTIVE to SEALED state. The publisher reads the sealed segment and streams each `HashChainEntry` to Kafka, maintaining the same sequence ordering. This preserves the **chain of custody** in the streaming layer.

### Advanced Compression with Zstandard

**Mental Model:** Think of Zstandard compression as a **high-efficiency filing system** for archived segments. While active segments remain uncompressed for fast appends and queries, older segments (beyond retention query requirements) can be compressed to 20-30% of their original size, like converting bulky paper files to microfilm for long-term storage.

The current design stores segments as plain files, which can consume significant disk space over time. While compression seems straightforward, we must preserve:

1. **Random access** for integrity verification (must verify hash chain without full decompression)
2. **Partial reads** for time-range queries (decompress only relevant portions)
3. **Tamper evidence** (compression must not break hash chain verification)

Zstandard (zstd) offers excellent compression ratios with configurable speed, support for dictionaries (improving audit log compression by learning common patterns), and the ability to decompress specific frames independently.

| Compression Strategy | Implementation Complexity | Query Performance Impact | Space Savings |
|---|---|---|---|
| Per-segment gzip | Low | High (must decompress entire file) | Moderate (~60% of original) |
| **Per-segment zstd with seekable format** | Medium | Low (random access to frames) | High (~70-80% of original) |
| Block-level compression in storage engine | High | Medium (decompress blocks on read) | Very high (~80-90% of original) |

> **Architecture Decision: When to Compress**
>
> Compression timing affects system performance and complexity:
>
> - **Context**: Segments become less frequently accessed as they age, but we must balance storage savings against query performance.
> - **Option 1: On-demand compression** during query idle periods. Complex scheduling, unpredictable resource usage.
> - **Option 2: Fixed schedule** (e.g., nightly compression of segments older than 30 days). Predictable but may compress too early/too late.
> - **Option 3: Automatic on segment sealing** after configurable delay. Simple, immediate space reclamation.
> - **Decision**: Implement automatic compression with a delay (e.g., 1 hour after sealing).
> - **Rationale**: Fresh segments remain uncompressed for active queries; compression happens predictably during low-activity windows; delay allows for recent forensic analysis without decompression overhead.
> - **Consequences**: Need background compression worker; must update `Catalog` with compressed segment metadata; decompression transparent to `QueryEngine`.

**Implementation Details:** The `SegmentManager` would gain a compression queue. When a segment's `SealedTime` exceeds the compression delay threshold, a background worker:

1. Reads the segment file
2. Compresses it using zstd's seekable format (creating `.segment.zst`)
3. Updates the `SegmentManifest` to note compression algorithm and frame boundaries
4. Atomically replaces the original file with the compressed version
5. Updates the `Catalog` to reference the compressed file

The `QueryEngine` and `Verifier` would transparently handle compressed segments by checking the manifest and using zstd's frame decompression API for targeted reads.

### Bloom Filters for "Actor-Not-Found" Queries

**Mental Model:** Bloom filters act as **rapid-check indexes**—like a receptionist's quick-glance list of who's in the building. When searching for events by a specific actor, we can instantly check the filter to see if that actor *might* be in a segment (with no false negatives) before performing the expensive disk scan.

Today, the `QueryEngine` performs **segment pruning** based on time ranges and then scans each relevant segment for matching actors. For queries filtering by a specific `ActorID` that doesn't exist in a segment, we still incur the I/O cost of reading that segment. Bloom filters provide probabilistic membership testing with O(1) time complexity and minimal memory overhead.

Each segment would maintain a Bloom filter containing all unique `ActorID` values within that segment. During query planning, the engine checks:

1. Does the time range include this segment? (existing pruning)
2. If query filters by `ActorID`, does the segment's Bloom filter possibly contain this ID?
3. Only if both true, scan the segment.

| Query Type | Current Performance | With Bloom Filters | Improvement |
|---|---|---|---|
| Actor exists in segment | O(n) scan | O(n) scan (same) | None |
| **Actor doesn't exist** | O(n) scan (wasted) | **O(1) filter check → skip segment** | 100% I/O elimination |
| Multi-actor query | O(n) scan for each | O(1) per actor → scan if any match possible | Proportional to % of non-matching segments |

> **Architecture Decision: Where to Store Bloom Filters**
>
> Filters must be persisted alongside segments but updated efficiently:
>
> - **Context**: Bloom filters are built from segment contents and must survive system restarts.
> - **Option 1: In `SegmentManifest`** as a serialized bit array. Simple, but manifests loaded into memory; large filters bloat memory usage.
> - **Option 2: Separate filter file** per segment (.filter). Clean separation, lazy loading possible.
> - **Option 3: Integrated into segment file header**. Complex random access during writes.
> - **Decision**: Store as separate `.filter` file created during segment sealing.
> - **Rationale**: Keeps manifest lean for catalog operations; allows lazy loading (only load filters for queried segments); separate file can be regenerated if corrupted.
> - **Consequences**: Additional file per segment; sealing process now includes filter generation; filter must be updated if segments are ever merged or reorganized.

**Implementation Approach:** Extend the segment sealing process to:

1. Traverse all events in the soon-to-be-sealed segment
2. Extract unique `ActorID` values (and optionally `ResourceID` for resource-filtering optimization)
3. Create Bloom filter with configurable false-positive rate (e.g., 1%)
4. Write filter to `{segment-id}.filter` file
5. Reference filter file path in `SegmentManifest`

The `QueryEngine`'s segment selection logic would check `QueryFilters.ActorIDs` against each segment's Bloom filter (if present) before adding the segment to the scan list.

### SIEM System Integration

**Mental Model:** SIEM integration is like installing **dedicated reporting lines** to security headquarters. While the main audit log remains the authoritative record, specialized formatted extracts are automatically delivered to security operations centers for correlation with other security events.

Security Information and Event Management (SIEM) systems like Splunk, ArcSight, or Elastic SIEM expect events in specific formats (CEF, LEEF, JSON with particular schemas). Our current `ExportEngine` can produce generic CSV/JSON, but dedicated SIEM integration would provide:

1. **Continuous streaming** of events in SIEM-native format
2. **Field mapping** from our `AuditEvent` schema to SIEM-specific fields
3. **Protocol support** for common SIEM ingestion methods (syslog TLS, HTTP Event Collector)
4. **Batching and backpressure** handling for high-volume environments

| Integration Method | Complexity | Reliability | Latency |
|---|---|---|---|
| **Syslog forwarding** (RFC 5424 over TLS) | Low | Medium (connection-based) | Low |
| HTTP Event Collector (HEC) | Medium | High (acknowledgments) | Medium |
| File-based export with pickup directory | Low | Very High (files as checkpoint) | High (batch intervals) |

**Implementation Pattern:** Add a `SIEMForwarder` component that:

- Subscribes to the event stream (via Kafka or CDC)
- Transforms `AuditEvent` to target format using configurable templates
- Batches events for efficiency
- Implements retry logic with exponential backoff for failed deliveries
- Maintains delivery cursors to resume after failures

### Multi-Region Replication for Disaster Recovery

**Mental Model:** Cross-region replication creates **geographically distributed copies** of the audit trail, like storing duplicate logbooks in separate secure vaults in different cities. If one region becomes unavailable, the audit trail remains accessible and intact elsewhere.

While the current design focuses on single-region durability, enterprises often require disaster recovery capabilities. The append-only, hash-chained nature of our log makes it particularly amenable to replication:

1. **Deterministic ordering** via sequence numbers eliminates conflict resolution
2. **Hash chain enables verification** of replica consistency
3. **Segments are immutable units** ideal for bulk replication

| Replication Strategy | Consistency Guarantee | Bandwidth Efficiency | Recovery Time Objective |
|---|---|---|---|
| **Synchronous segment replication** (on seal) | Strong (segment atomic) | High (bulk transfers) | Minutes (recent segment transfer) |
| Async event streaming | Eventual | Low (per-event overhead) | Seconds (replay stream) |
| Hybrid (events async, segments periodic) | Eventual with periodic strong | Medium | Variable |

**Design Extension:** Add a `ReplicationManager` that:

- Watches for sealed segments
- Transfers segment files and their manifests to replica regions
- Verifies hash chain continuity across regions
- Maintains a **replication catalog** tracking segment locations
- Allows `QueryEngine` to fail over to replica regions during outages

### Advanced Query Capabilities

**Mental Model:** Advanced queries transform the audit log from a **simple event list** into a **forensic analysis platform**—like adding a research team that can spot patterns, create timelines, and generate intelligence reports from raw log data.

Beyond basic time-range filtering, organizations need:

1. **Full-text search** across all event fields and metadata
2. **Statistical aggregations** (counts by actor, resource, outcome)
3. **Temporal pattern detection** (bursts of activity, periodic access)
4. **Graph analysis** (relationship mapping between actors and resources)

| Capability | Implementation Approach | Storage Impact | Query Performance |
|---|---|---|---|
| **Full-text index** (Elasticsearch integration) | Sidecar index updated via CDC | High (2-3x original data) | Sub-second search |
| **Columnar storage for analytics** (Parquet conversion) | Periodic conversion of old segments | Moderate (compression benefits) | Fast aggregation, slower point queries |
| **In-memory event cache** (hot events) | LRU cache in `QueryEngine` | RAM dependent | Microsecond retrieval for recent events |

**Extension Design:** These capabilities would be implemented as **optional modules** that plug into the existing architecture:

- **Search Module**: Builds and maintains a full-text index (Lucene-based) updated via the CDC stream
- **Analytics Module**: Periodically converts sealed segments to columnar format (Parquet) for OLAP queries
- **Pattern Detection**: Stream processor analyzing the event stream for configured patterns

Each module would expose its own query interface while the core `QueryEngine` remains focused on reliable, integrity-verified time-range queries.

## Implementation Guidance

### A. Technology Recommendations Table

| Component | Simple Option | Advanced Option |
|---|---|---|
| Event Streaming | Direct Kafka producer in Ingestor | CDC with Debezium or custom WAL tailer |
| Compression | gzip per segment | Zstandard with seekable format |
| Bloom Filters | In-memory only (regenerated on restart) | Persistent mmap-ed filter files |
| SIEM Integration | File-based export with log rotation | Async gRPC streaming with backpressure |
| Replication | rsync of segment files | Custom protocol with integrity verification |
| Advanced Query | External Elasticsearch index | Integrated columnar storage (DuckDB, Apache Arrow) |

**B. Recommended File/Module Structure**

```
audit-log-system/
├── cmd/
│   ├── ingest/                # Ingestor service
│   ├── query/                 # Query service
│   └── tools/
│       ├── compressor/        # Compression utility
│       └── replicator/        # Replication daemon
├── internal/
│   ├── core/                  # Core types (AuditEvent, HashChainEntry, etc.)
│   ├── storage/
│   │   ├── engine.go          # StorageEngine
│   │   ├── segment.go         # Segment management
│   │   ├── wal.go             # WAL implementation
│   │   └── compression/       # Compression extensions
│   │       ├── zstd.go        # Zstandard compression
│   │       └── compressor.go  # Compression interface
│   ├── query/
│   │   ├── engine.go          # QueryEngine
│   │   ├── filters.go         # QueryFilters
│   │   ├── export.go          # ExportEngine
│   │   └── bloom/             # Bloom filter integration
│   │       ├── filter.go      # Bloom filter implementation
│   │       └── builder.go     # Filter builder during sealing
│   ├── streaming/
│   │   ├── publisher.go       # Event streaming interface
│   │   ├── kafka.go           # Kafka implementation
│   │   └── cdc/               # Change Data Capture
│   │       └── tailer.go      # WAL/segment tailer
│   ├── siem/
│   │   ├── forwarder.go       # SIEM forwarder
│   │   ├── cef.go             # CEF formatter
│   │   └── syslog.go          # Syslog transport
│   └── replication/
│       ├── manager.go         # Replication manager
│       ├── protocol.go        # Replication protocol
│       └── catalog.go         # Cross-region segment catalog
└── pkg/
    └── utils/
        ├── zstd/              # Zstandard wrapper utilities
        └── bloom/             # Bloom filter utilities
```

### C. Infrastructure Starter Code: Zstandard Compression Wrapper

```go
// internal/storage/compression/zstd.go

package compression

import (
    "bytes"
    "fmt"
    "io"

    "github.com/klauspost/compress/zstd"
)

// ZstdCompressor implements the Compressor interface using Zstandard
type ZstdCompressor struct {
    encoder *zstd.Encoder
    decoder *zstd.Decoder
}

// NewZstdCompressor creates a new Zstandard compressor with default settings
func NewZstdCompressor() (*ZstdCompressor, error) {
    encoder, err := zstd.NewWriter(nil,
        zstd.WithEncoderLevel(zstd.SpeedBetterCompression),
        zstd.WithEncoderDict(nil)) // Could load dictionary for better ratios
    if err != nil {
        return nil, fmt.Errorf("failed to create zstd encoder: %w", err)
    }

    decoder, err := zstd.NewReader(nil)
    if err != nil {
        return nil, fmt.Errorf("failed to create zstd decoder: %w", err)
    }

    return &ZstdCompressor{
        encoder: encoder,
        decoder: decoder,
    }, nil
}

// Compress implements the Compressor interface
func (z *ZstdCompressor) Compress(data []byte) ([]byte, error) {
    compressed := z.encoder.EncodeAll(data, make([]byte, 0, len(data)/2))
```

```
    return compressed, nil
}

// Decompress implements the Compressor interface
func (z *ZstdCompressor) Decompress(data []byte) ([]byte, error) {
    decompressed, err := z.decoder.DecodeAll(data, nil)
    if err != nil {
        return nil, fmt.Errorf("zstd decompression failed: %w", err)
    }
    return decompressed, nil
}

// CompressStream compresses data from src to dst using seekable format
func (z *ZstdCompressor) CompressStream(dst io.Writer, src io.Reader) error {
    encoder, _ := zstd.NewWriter(dst, zstd.WithWindowSize(1<<20)) // 1MB window
    defer encoder.Close()

    _, err := io.Copy(encoder, src)
    return err
}

// DecompressStream decompresses seekable zstd from src to dst
func (z *ZstdCompressor) DecompressStream(dst io.Writer, src io.Reader) error {
    decoder, _ := zstd.NewReader(src)
    defer decoder.Close()

    _, err := io.Copy(dst, decoder)
    return err
}
```

### D. Core Logic Skeleton: CDC Tailer for Event Streaming

```go
// internal/streaming/cdc/tailer.go

package cdc

import (
    "context"
    "time"

    "github.com/yourorg/audit-log/internal/core"
    "github.com/yourorg/audit-log/internal/storage"
)

// SegmentTailer watches for sealed segments and streams their contents
type SegmentTailer struct {
    storage    *storage.Engine
    publisher  streaming.Publisher
    pollInterval time.Duration
    lastSealedSeq uint64
}

// NewSegmentTailer creates a new CDC tailer
func NewSegmentTailer(storage *storage.Engine, publisher streaming.Publisher) *SegmentTailer {
    return &SegmentTailer{
        storage:    storage,
        publisher:  publisher,
        pollInterval: 5 * time.Second,
        lastSealedSeq: 0,
    }
}

// Run starts the tailing process (blocking)
func (t *SegmentTailer) Run(ctx context.Context) error {
    ticker := time.NewTicker(t.pollInterval)
    defer ticker.Stop()

    for {
        select {
        case <-ctx.Done():
            return ctx.Err()
        case <-ticker.C:
            // TODO 1: Query storage for segments sealed since lastSealedSeq
```

```
            // TODO 2: For each new sealed segment, read all HashChainEntries
            // TODO 3: For each entry, convert to streaming event format
            // TODO 4: Publish to Kafka via publisher.Publish()
            // TODO 5: Update lastSealedSeq to the max sequence in processed segments
            // TODO 6: Handle errors with exponential backoff and retry
        }
    }
}

// GetWatermark returns the last successfully streamed sequence number
func (t *SegmentTailer) GetWatermark() uint64 {
    return t.lastSealedSeq
}
```

### E. Language-Specific Hints for Go Extensions

1. **Kafka Integration**: Use `github.com/confluentinc/confluent-kafka-go/kafka` for production or `github.com/segmentio/kafka-go` for simpler API. Configure idempotent producer to prevent duplicate events during retries.
2. **Zstandard**: `github.com/klauspost/compress/zstd` provides excellent performance. Pre-train a dictionary on sample audit logs for 10-15% better compression.
3. **Bloom Filters**: `github.com/bits-and-blooms/bloom` offers a production-ready implementation. Size filters based on expected unique actors per segment (e.g., 10,000 actors → 28KB filter for 1% false positive rate).
4. **SIEM Protocols**: For syslog TLS, use `github.com/RackSec/srslog` or implement RFC 5424 directly. For HTTP collectors, use `net/http` with persistent connections and connection pooling.
5. **Cross-Region Replication**: Use `github.com/minio/minio-go/v7` for S3-compatible storage or implement a simple gRPC service with chunked transfer. Always verify hash chain after transfer.

### F. Milestone Checkpoint for Compression Extension

After implementing segment compression:

```bash
# Create test segments
$ go run cmd/tools/segment-filler/main.go --events=10000

# Run compression on segments older than 1 hour
$ go run cmd/tools/compressor/main.go --data-dir=/var/audit --age=1h

# Expected output:
# INFO Compressing segment segment-000001.segment (size: 104857600)
# INFO Created segment-000001.segment.zst (size: 31457280, ratio: 70%)
# INFO Updated catalog with compressed segment metadata

# Verify queries still work on compressed segments
$ go run cmd/query/main.go --start-time="2024-01-01T00:00:00Z" --end-time="2024-01-02T00:00:00Z"
# Should return events from both compressed and uncompressed segments

# Verify integrity chain still verifies
$ go run cmd/tools/verifier/main.go --data-dir=/var/audit
# Output: Chain verification passed for 10000 events across 3 segments (2 compressed)
```

### G. Debugging Tips for Future Extensions

| Symptom | Likely Cause | How to Diagnose | Fix |
|---|---|---|---|
| Streaming latency > 5 minutes | Segment sealing delayed or tailer stuck | Check `SegmentTailer.GetWatermark()` vs latest sealed segment; monitor segment sealing metrics | Adjust segment size threshold; check for blocking publisher calls |
| Compressed segment verification fails | Zstd frame corruption or incorrect metadata | Use `zstd --test segment.segment.zst`; verify manifest has correct frame offsets | Recompress from original backup; add checksum to compressed files |
| Bloom filter shows false positives > config rate | Filter size too small for actual unique actors | Log actual unique count vs filter capacity during sealing | Increase filter size or adjust false positive rate parameter |
| SIEM events missing fields | Field mapping configuration error | Compare raw AuditEvent JSON with SIEM-formatted output | Update field mapping template; add validation step in forwarder |
| Replica region missing segments | Network partition during transfer | Check replication catalog consistency; verify segment hashes match | Trigger manual sync; implement gap detection and repair |
| Memory usage grows with compressed segments | Decompression cache not evicting | Profile memory usage; check cache hit rates | Implement LRU cache with size limits; use mmap for compressed files |

## Glossary

> **Milestone(s):** Milestone 1 (foundational concepts), Milestone 2 (core terminology), Milestone 3 (query concepts)

The glossary provides definitive explanations of key terms, acronyms, and domain-specific vocabulary used throughout this design document. Consistent terminology is essential when discussing cryptographic systems and compliance requirements, where precise language prevents misunderstandings.

## Terms and Definitions

| Term | Definition | First Section Reference |
|---|---|---|
| **ADR (Architecture Decision Record)** | A structured document that captures an important architectural decision made along with its context, alternatives considered, and consequences. ADRs provide a decision log that helps teams understand why specific design choices were made. | Component Design: Event Model & Serialization |
| **Append-Only** | A fundamental property of a data store where data can only be added at the end of the log, never modified or deleted once written. This ensures the audit trail's completeness and serves as the foundation for tamper-evident logging. | Context and Problem Statement |
| **Atomic Rename** | A filesystem operation that either fully succeeds or fails without leaving the system in a partial or corrupted state. Used during segment rotation to ensure a new segment file becomes visible only after all data has been successfully written and flushed. | Error Handling and Edge Cases |
| **Audit Event** | The atomic unit of record in the audit logging system—a structured record capturing **who** (actor) did **what** (action) to **which** resource (resource) **when** (timestamp) and with what **outcome**. Represented by the `AuditEvent` struct with required fields and extensible metadata. | Component Design: Event Model & Serialization |
| **Audit Log** | An immutable, chronological record of system activities used for compliance monitoring, security investigation, and operational auditing. Unlike application logs, audit logs focus on business-relevant actions and must preserve integrity for forensic use. | Context and Problem Statement |
| **Bloom Filter** | A probabilistic data structure that tests whether an element is a member of a set, with a configurable false positive rate but no false negatives. Can accelerate "actor-not-found" queries by quickly eliminating segments that definitely don't contain events for a given actor. | Future Extensions |
| **Canonical JSON** | JSON serialization with deterministic field ordering (typically alphabetical) and whitespace rules, ensuring that the same logical content always produces identical byte sequences. Required for consistent cryptographic hashing across serializations. | Component Design: Event Model & Serialization |
| **Catalog** | An in-memory (or persisted) index maintained by the Storage Engine that tracks all sealed segments and their metadata (time ranges, sequence ranges, file paths). Used by the Query Engine to quickly identify relevant segments during time-range queries. | High-Level Architecture |
| **Change Data Capture (CDC)** | A design pattern that tracks data changes in a source system (like new audit events) and propagates them to other systems in real-time. Implemented via the `SegmentTailer` component for streaming events to SIEM systems. | Future Extensions |
| **Chain of Custody** | A forensic protocol documenting the sequence of custody, control, transfer, and analysis of evidence. In audit logging, this concept applies to ensuring the audit trail's integrity from creation through storage, query, and export. | Debugging Guide |
| **Compliance Export** | Formatted output (CSV, JSON, PDF) suitable for submission to regulatory auditors. Includes all required fields, proper formatting, and integrity verification metadata, making the audit trail usable for official compliance reviews. | Component Design: Query & Export Engine |
| **Compression Dictionary** | Pre-trained data that improves compression ratios for specific data patterns. When using Zstandard compression on audit segments, a dictionary trained on previous audit events can significantly reduce storage requirements. | Future Extensions |
| **Concurrency Stress Tester** | A debugging tool that runs parallel workloads to expose race conditions, deadlocks, and synchronization issues in the storage engine and query components. | Debugging Guide |
| **Correlation ID** | A unique identifier propagated across service boundaries to trace a single transaction or user session through distributed systems. Captured in the `RequestContext` of each audit event to enable cross-service activity tracing. | Component Design: Event Model & Serialization |
| **Crash Recovery Simulator** | A debugging tool that simulates process crashes at strategic points (mid-write, mid-fsync) to test the durability guarantees of the Write-Ahead Log and hash chain reconstruction. | Debugging Guide |
| **Critical Severity** | A classification for failures that require immediate human intervention, such as hash chain corruption, disk full conditions, or cryptographic verification failures. Contrasts with temporary errors that may self-resolve. | Error Handling and Edge Cases |
| **Cursor-Based Pagination** | A pagination technique using an opaque token (the cursor) that encodes enough information to resume query results from a specific point. More stable than offset-based pagination when underlying data changes between page requests. | Component Design: Query & Export Engine |

| Term | Definition | First Section Reference |
|---|---|---|
| **Differential Analysis** | A debugging technique that compares behavior between known-good and problematic states to isolate root causes. For example, comparing hash computations between a working and broken system to identify implementation differences. | Debugging Guide |
| **Exponential Backoff** | A retry strategy where the delay between retry attempts increases exponentially (e.g., 1s, 2s, 4s, 8s). Used by `StorageEngine.appendWithRetry()` to handle temporary errors like resource contention or transient I/O issues. | Error Handling and Edge Cases |
| **ExportFormatCSV** | A constant value ( `"csv"` ) identifying the CSV export format, which produces comma-separated values with headers, suitable for spreadsheet analysis and compliance submissions. | Component Design: Query & Export Engine |
| **ExportFormatJSONL** | A constant value ( `"jsonl"` ) identifying the JSON Lines export format, where each event appears as a separate JSON object on its own line, preserving the full event structure. | Component Design: Query & Export Engine |
| **False Positive Rate** | In the context of Bloom filters, the probability that the filter incorrectly indicates an element is in the set when it is not. A tunable parameter that trades memory usage against query accuracy. | Future Extensions |
| **Field Mapping** | Transformation rules that convert between data schemas, used when exporting audit events to different formats or integrating with external systems like SIEMs that expect specific field names. | Future Extensions |
| **Forensic Analysis** | The systematic examination of corrupted or suspicious data to determine root cause. In audit logging, this involves analyzing segment files, hash chain breaks, or timestamp anomalies to understand what went wrong. | Debugging Guide |
| **fsync** | A system call ( `fsync()` in Unix, `File.Sync()` in Go) that flushes file data from the OS cache to durable storage. Critical for ensuring write durability in the Write-Ahead Log before acknowledging writes to clients. | Component Design: Immutable Storage Engine |
| **GDPR (General Data Protection Regulation)** | A European Union regulation on data protection and privacy that imposes strict requirements on handling personal data, including the right to erasure. Impacts audit logging by requiring careful PII handling and retention policies. | Context and Problem Statement |
| **Hash Chain** | A cryptographic data structure where each entry includes the cryptographic hash of the previous entry, forming a verifiable chain of integrity. Any modification to an intermediate entry breaks all subsequent hashes, enabling tamper detection. | Component Design: Immutable Storage Engine |
| **Hex Dump Analysis** | A debugging technique that inspects raw binary files using hexadecimal representation to identify corruption, incorrect serialization, or byte ordering issues. Particularly useful for diagnosing segment file problems. | Debugging Guide |
| **Idempotent Producer** | A message producer that guarantees exactly-once semantics through deduplication, typically by including a unique message ID that recipients can use to detect and discard duplicates. | Future Extensions |
| **Immutable Storage** | Storage where data, once written, cannot be modified or deleted. The foundation of trustworthy audit logging that prevents retroactive alteration of historical records. | Component Design: Immutable Storage Engine |
| **Indexing Strategy** | The approach for organizing data to accelerate queries. For time-series audit logs, the primary index is typically on `(timestamp, sequence)` , with optional secondary indexes on frequently queried fields like actor or resource. | Component Design: Query & Export Engine |
| **Ingestor** | The component responsible for accepting, validating, and forwarding audit events from applications to the storage layer. Acts as the system's entry point and ensures all incoming events meet schema requirements. | High-Level Architecture |
| **Integrity-First Debugging** | A debugging approach that starts by verifying cryptographic hash chain integrity, then works upward through storage layers, and finally examines application logic. Ensures foundational correctness before addressing higher-level issues. | Debugging Guide |
| **Layered Verification** | A debugging methodology that verifies system correctness layer by layer, starting from the cryptographic foundation (hash chain) up through storage semantics and finally application logic. | Debugging Guide |
| **MAX_EVENT_SIZE** | A constant ( `65536` bytes, 64KB) defining the maximum serialized size of an individual audit event. Events exceeding this limit are rejected to prevent resource exhaustion and ensure predictable storage requirements. | Component Design: Event Model & Serialization |
| **MAX_PAGE_SIZE** | A constant ( `1000` events) defining the maximum number of events returned in a single query page. Prevents memory exhaustion and ensures responsive pagination even for large result sets. | Component Design: Query & Export |

| Term | Definition | First Section Reference |
|---|---|---|
| | | Engine |
| **Merkle Tree** | A tree data structure where each leaf node is a hash of a data block, and each non-leaf node is a hash of its children's hashes. Enables efficient verification of large datasets and proofs of inclusion for specific entries. | High-Level Architecture |
| **PII (Personally Identifiable Information)** | Any data that could potentially identify a specific individual, such as names, email addresses, IP addresses, or government IDs. Must be masked or tokenized in audit logs to comply with privacy regulations like GDPR. | Component Design: Event Model & Serialization |
| **PII Masking** | The process of redacting or obfuscating personally identifiable information before storage, typically by replacing sensitive values with tokens or partial representations (e.g., "***@example.com" for emails). | Component Design: Event Model & Serialization |
| **Query Engine** | The component responsible for efficient retrieval, filtering, pagination, and export of audit events. Translates high-level query criteria into segment scans and indexed lookups while managing resource constraints. | High-Level Architecture |
| **Read-Only Mode** | An operational state where the storage engine rejects write operations but continues serving read queries. Automatically entered when disk space is critically low or when hash chain corruption is detected, preserving existing data integrity. | Error Handling and Edge Cases |
| **Replication Catalog** | A metadata store that tracks segment locations, checksums, and synchronization status across multiple regions or storage nodes. Enables geographically distributed audit logging with consistency guarantees. | Future Extensions |
| **Schema Validation** | The process of ensuring audit events conform to the required structure, including presence of mandatory fields, correct data types, and adherence to field constraints (e.g., maximum length, allowed characters). | Component Design: Event Model & Serialization |
| **Segment** | A sealed, immutable file containing a batch of audit events, typically limited by size (e.g., 1GB) or time duration. Segments are the unit of storage management, compression, archiving, and query pruning. | Component Design: Immutable Storage Engine |
| **Segment Manager** | A sub-component of the Storage Engine that manages the lifecycle of segment files: creation, sealing, indexing, compression, and eventual archival or deletion according to retention policies. | High-Level Architecture |
| **Segment Pruning** | An optimization technique that skips entire segment files during query execution based on metadata (time ranges, actor/resource indexes). Dramatically reduces I/O by avoiding scans of irrelevant segments. | Component Design: Query & Export Engine |
| **Seekable Format** | A compression format that allows random access to portions of the compressed data without requiring decompression of the entire file. Zstandard's seekable format enables efficient querying within compressed segments. | Future Extensions |
| **SHA256_HASH_SIZE** | A constant ( `32` bytes) defining the output size of the SHA-256 cryptographic hash function used throughout the hash chain for integrity verification. | Component Design: Immutable Storage Engine |
| **SIEM (Security Information and Event Management)** | A system that provides real-time analysis of security alerts generated by applications and network hardware. Audit events can be streamed to SIEM systems for correlation with other security data. | Future Extensions |
| **SOC 2** | A compliance framework developed by the American Institute of CPAs (AICPA) that specifies how organizations should manage customer data based on five "trust service principles": security, availability, processing integrity, confidentiality, and privacy. | Context and Problem Statement |
| **Storage Engine** | The core component responsible for immutable storage, hash chain construction, segment management, and integrity verification. Provides the append-only abstraction that guarantees tamper-evidence. | High-Level Architecture |
| **Streaming Export** | An export method that writes events directly to the output writer as they are read from storage, avoiding loading entire result sets into memory. Essential for handling large exports that could exhaust available RAM. | Component Design: Query & Export Engine |
| **Temporal Correlation** | A debugging technique that correlates symptoms (like verification failures) with system events over time (restarts, deployments, disk errors) to identify patterns and root causes. | Debugging Guide |
| **Temporary Error** | An error condition that might succeed on retry, such as "resource temporarily unavailable," disk full (if space can be reclaimed), or network timeouts. Distinguished from permanent errors like cryptographic corruption. | Error Handling and Edge Cases |
| **Time-Range Query** | The most common audit log query type, filtering events by start and end timestamps. Optimized through segment metadata (time ranges) and potentially secondary indexing within segments. | Component Design: Query & Export Engine |

| Term | Definition | First Section Reference |
| --- | --- | --- |
| **Time-Series Database (TSDB)** | A database optimized for storing and querying data points indexed by time. While our system isn't a full TSDB, it borrows patterns like time-based partitioning and efficient range queries. | Context and Problem Statement |
| **UTC Storage** | The practice of storing all timestamps in Coordinated Universal Time (UTC) without timezone offsets. Eliminates ambiguity and ensures consistent ordering and querying across geographically distributed systems. | Component Design: Query & Export Engine |
| **WAL (Write-Ahead Log)** | An append-only log where events are written before being acknowledged, ensuring durability even in the event of a crash. The active segment begins as a WAL file before being sealed and becoming immutable. | Component Design: Immutable Storage Engine |
| **Zstandard (zstd)** | A real-time compression algorithm developed by Facebook, offering high compression ratios with excellent decompression speed. Used for compressing sealed segments to reduce storage costs while maintaining query performance. | Future Extensions |