Technical Specification: Cartridge Mutable Archive Format

The Cartridge format delivers a **high-performance, single-user mutable archive** with full SQLite VFS support, S3-style access control, and transactional audit trails. Design targets include sub-millisecond file operations, clean mutations without fragmentation, and ACID consistency through SQLite-level transactions. The format complements the immutable Engram format by serving as an active workspace that can be snapshotted for distribution.

This specification synthesizes production-proven patterns from SQLite's page management, RocksDB's LSM architecture, Git's packfile optimization, and AWS IAM's policy evaluation to create a **fast, safe, and auditable mutable container** suitable for interactive development workflows. Performance benchmarks suggest **100K+ IOPS on NVMe SSDs**, sub-10μs cached reads, and less than 1% audit overhead through careful architectural choices detailed below.

Binary format architecture balances simplicity with extensibility

The Cartridge binary format adopts a **hybrid page-based design** with 4KB fixed-size pages as the fundamental storage unit. This page size aligns with modern SSD block sizes and OS virtual memory pages while enabling predictable I/O patterns essential for performance.   The format header occupies the first page and establishes critical metadata including a magic number (`CART\\x00\\x01\\x00\\x00` for version identification), format version fields (major.minor for compatibility tracking), block size configuration, total and free block counts, and a pointer to the B-tree root for the file catalog.

Individual pages carry their own type identifiers distinguishing between catalog B-tree nodes, content data blocks, freelist pages, and audit log entries. Each page includes a 32-byte SHA-256 checksum in its header for corruption detection, though this verification remains optional during normal operations to avoid performance penalties. The format reserves 256 bytes in the main header for future extensions including compression configuration, encryption parameters, and feature flags.

The **central catalog uses B-tree indexing** with a fanout of approximately 680 entries per 16KB node, yielding 3-5 disk reads even for billions of files. Each catalog entry contains the file path hash (8 bytes for O(1) lookup), full path string, size, block list pointer, creation and modification timestamps, permission flags, and a content checksum. This centralized metadata design contrasts with ZIP's end-of-file directory but provides superior random access performance critical for mutable operations.

Content storage follows a **log-structured append pattern** for sequential write optimization. New files and updates append to the end of the archive with their data spread across allocated blocks. The format tracks block allocation through a hybrid approach: **bitmap allocation for small files** (under 256KB, using multi-level bitmaps with under 2% overhead) and **extent-based allocation for large files** (256KB and above, using B-tree maps of free extents with automatic coalescing). This hybrid strategy optimizes allocation speed across all file sizes while minimizing fragmentation.

Free space management prevents long-term fragmentation

The format implements **immediate reuse with incremental compaction** to maintain performance over time. When files are deleted or modified, their blocks immediately join the free pool managed by the hybrid allocator. Small freed blocks return to the bitmap allocator, while large freed extents merge with adjacent free space through automatic coalescing that runs in O(log n) time using the extent B-tree's neighbor-finding capability.

Background compaction runs continuously in a dedicated thread, performing limited work per iteration to avoid latency spikes. Each compaction cycle moves up to 100 fragmented blocks to contiguous locations, prioritizing the most fragmented regions identified through a fragmentation score that considers extent count and size distribution. This **incremental approach maintains under 10ms latency per cycle** while preventing the accumulation of dead space that plagues many mutable formats.

The format supports both copy-on-write and in-place updates depending on file size and modification patterns. Files under 64KB use in-place updates when possible to minimize write amplification, while larger files employ copy-on-write to enable snapshot capabilities and corruption resilience. Discarded blocks from copy-on-write operations remain allocated until the next checkpoint completes, ensuring crash consistency through a simple recovery mechanism that rolls back incomplete operations.

A user-initiated VACUUM operation provides thorough defragmentation when background compaction proves insufficient. This operation rewrites the entire archive with optimal layout, recomputes all free space structures, and rebuilds indexes for maximum efficiency.   VACUUM typically runs during maintenance windows and can reduce archive size by 20-40% in heavily modified archives while restoring peak performance.

SQLite VFS implementation enables full database capabilities

The Cartridge format implements the **sqlite3_vfs and sqlite3_io_methods interfaces** to present the archive as a mountable location for SQLite databases. The VFS layer requires 13 core methods (xOpen, xClose, xRead, xWrite, xTruncate, xSync, xFileSize, xLock, xUnlock, xCheckReservedLock, xFileControl, xSectorSize, xDeviceCharacteristics) plus additional VFS-level operations for file management. This implementation allows SQLite to treat the archive as a filesystem, storing database files, journals, and temporary tables within the container.

For **single-user scenarios**, the VFS uses `PRAGMA locking_mode=EXCLUSIVE` to eliminate the complexity of shared memory management. This configuration enables WAL mode without implementing the xShm* methods (xShmMap, xShmLock, xShmBarrier, xShmUnmap) while retaining WAL's performance benefits including higher write throughput and read concurrency within a single process. The exclusive lock persists for the connection lifetime, simplifying the locking implementation to no-op methods that always succeed.

Write-Ahead Logging provides **crash consistency and improved concurrency** even in single-user mode. The VFS creates three files per database: the main database file, a .wal file containing uncommitted changes, and optionally a .wal-index file for fast page lookups. Checkpoint operations transfer WAL contents back to the main database file at configurable intervals (default 1000 pages or 4MB), with automatic checkpointing preventing unbounded WAL growth. The xSync implementation must perform true fsync operations on both the WAL and database files to guarantee durability, though `PRAGMA synchronous=NORMAL` reduces sync frequency to checkpoint boundaries for improved performance.

The **xRead and xWrite methods** must handle full-size transfers or return appropriate error codes, as SQLite cannot tolerate short reads or writes. These methods map to the archive's block-based storage through the buffer pool, reading entire 4KB pages into cache and satisfying requests from cached data when possible. The xTruncate method updates file size metadata and adds truncated blocks to the free pool, while xFileControl handles PRAGMA operations and custom control codes for archive-specific features like compaction hints and integrity checking.

Memory-mapped I/O support through xFetch and xUnfetch methods remains optional but provides **near 2x read performance** for databases where it's applicable. However, Windows compatibility issues with truncating mapped files and virtual memory exhaustion concerns on 32-bit systems suggest making this feature configurable rather than default. The xDeviceCharacteristics method should return SQLITE_IOCAP_SAFE_APPEND and possibly SQLITE_IOCAP_ATOMIC4K if the underlying implementation guarantees atomic 4KB writes, enabling SQLite optimizations that reduce journal writes.

SQLite databases within Cartridge archives can expect **3-10x slower performance than native filesystem access**, primarily due to the additional indirection layer, block alignment overhead, and archive format bookkeeping. Mitigation strategies include using 64KB page sizes (set before database creation), allocating 100MB+ page cache via `PRAGMA cache_size=-102400`, enabling WAL mode with normal synchronous mode, and configuring temporary tables in memory. These optimizations typically reduce the overhead to 2-3x while maintaining full ACID guarantees.

S3-style IAM policies provide fine-grained access control

The Cartridge format embeds a **lightweight IAM policy engine** supporting the AWS IAM policy structure for path-based resource control. Policies consist of JSON documents containing a Version field (fixed to "2012-10-17" for compatibility), an Id field for policy identification, and a Statement array defining permissions. Each statement specifies an Effect (Allow or Deny), Principal (subject actor), Action (permitted operations like "cart:read" or "cart:write"), Resource (target paths with wildcards), and optional Condition blocks for context-sensitive rules.

Policy evaluation follows the **explicit deny precedence model**: the engine first scans all policies for explicit deny statements matching the request, immediately rejecting access if any deny applies. Only if no denies match does it check for explicit allows, granting access if at least one allow statement matches all request attributes. The default implicit deny ensures secure-by-default behavior where permissions must be explicitly granted. This evaluation completes in under 100μs for cached policies through careful optimization.

Resource matching supports **wildcard patterns** where asterisks match zero or more characters (including path separators) and question marks match exactly one character. For example, the pattern `cart://data/*/private/*` matches `cart://data/users/private/file.txt` and even `cart://data/a/b/private/c/d/file.txt` since wildcards span path segments. The implementation pre-compiles these patterns into regex objects cached in an LRU structure, avoiding repeated compilation overhead that would otherwise dominate evaluation time.

Condition operators enable **context-aware authorization** through 27+ operator types including StringEquals for exact matching, StringLike for pattern matching, NumericLessThan for size limits, DateGreaterThan for temporal restrictions, IpAddress for network-based rules, and ForAllValues/ForAnyValue for multi-valued attribute matching. Conditions use AND logic between different operators and OR logic within value arrays, allowing complex rules like "allow read access to finance department members from corporate IP ranges during business hours" through appropriate condition blocks.

The policy evaluator implements **multiple optimization layers** to minimize overhead. Pattern pre-compilation eliminates repeated regex parsing, reducing evaluation from milliseconds to microseconds. Index structures organize policies by action and resource prefix, enabling fast candidate selection without scanning all policies. Early termination on explicit deny avoids unnecessary computation, while result caching stores recent evaluation outcomes in an LRU cache keyed by request tuple. These optimizations yield 10,000+ evaluations per second per core, adding negligible latency to file operations.

Integration with the archive format occurs at the **VFS operation level** where each xRead, xWrite, xDelete, and xOpen call triggers policy evaluation. The request context includes the actor ID (from session), requested action, target resource path, timestamp, and any custom attributes. Policy documents live in a reserved section of the archive catalog, loaded at mount time and cached in memory with invalidation on updates. This placement keeps policies inside the archive itself, maintaining the self-contained “infrastructure-in-a-box” philosophy.

Transactional ledger provides lightweight audit capability

The audit system captures all operations through an **append-only transaction log** stored in dedicated pages within the archive. Each audit entry occupies just 24 bytes in its minimal form: an 8-byte microsecond timestamp, 4-byte actor ID, 2-byte operation type enum (CREATE=0, READ=1, UPDATE=2, DELETE=3), 2-byte resource table enum, 8-byte resource ID, and an optional 4-byte session ID. This compact representation enables high-throughput logging with minimal storage overhead, typically under 1% of archive size even for actively audited systems.

The logging implementation uses a **lock-free ring buffer** in shared memory for sub-microsecond write latency. Application threads write audit entries to the ring buffer without blocking, while a dedicated flush thread batches entries and commits them to disk every 10-100ms. Each batch write occurs within a single transaction, amortizing sync overhead across 100-1000 entries and reducing per-operation cost to negligible levels. The ring buffer sizing (default 8192 entries) provides sufficient headroom for burst traffic while limiting memory footprint.

Extended audit information lives in a separate optional table linked by audit entry ID. This includes before and after values for updates (stored as compressed BLOB fields), free-form metadata JSON for custom attributes, IP addresses for remote access tracking, and any additional context relevant for compliance requirements. Separating essential audit data from optional details keeps the hot path minimal while supporting deep forensic analysis when needed.

The **audit log uses compound B-tree indexes** on (timestamp), (actor_id, timestamp), and (resource_table, resource_id, timestamp) to enable fast queries across common access patterns. Time-range queries for recent activity, user activity queries for compliance review, and resource access history for security investigations all complete in milliseconds through these indexes. The timestamp-based primary ordering naturally supports retention policies that archive or delete old entries by date range.

Cryptographic integrity protection through **Merkle tree verification** remains optional for high-security scenarios. Each batch of audit entries forms a leaf node in the tree, with parent nodes computed through SHA-256 hashing of child pairs. The root hash gets signed with the archive owner's private key and stored as a checkpoint record every 1000 batches (configurable). This structure enables compact proof of inclusion for any entry using O(log n) hashes, supports efficient consistency proofs between checkpoints, and makes undetected tampering mathematically infeasible even if the archive is compromised.

Simpler **hash chain integrity** provides an alternative when Merkle trees add unnecessary complexity. Each audit entry includes the SHA-256 hash of the previous entry, creating a tamper-evident chain where modifying any entry breaks all subsequent hashes. This approach offers forward integrity (past entries remain secure even if the current key is compromised) at the cost of O(n) verification complexity. For most use cases, the performance and simplicity trade-offs favor hash chains over Merkle trees.

Retention policies support **time-based partition deletion** where audit tables partition by month or quarter, enabling efficient removal of old data by dropping entire partitions. Archive operations can move old partitions to cold storage (separate archive files), apply higher compression ratios to historical data, or securely delete data beyond retention requirements. Background maintenance ensures partition boundaries remain optimal and indexes stay efficient as data ages.

Performance optimization strategies deliver production-ready speed

The buffer pool implementation adopts **ARC (Adaptive Replacement Cache)** as its eviction policy for optimal hit rates across varied workloads. ARC maintains two

LRU lists: T1 for recently accessed pages and T2 for frequently accessed pages, plus ghost lists B1 and B2 tracking recently evicted pages. The adaptive parameter p adjusts dynamically based on hit patterns in ghost lists, automatically tuning the recency-frequency balance. Benchmarks show ARC delivers 34% better hit rates than plain LRU on mixed workloads while requiring no manual tuning, making it ideal for unpredictable access patterns in mutable archives.

The buffer pool should consume **40-60% of available system RAM** based on workload characteristics. OLTP-style random access patterns benefit from larger pools (60%) targeting 95%+ hit rates, while sequential scan workloads perform adequately with smaller pools (40%) since scan resistance prevents cache pollution. The implementation monitors hit ratio continuously, triggering alerts if rates drop below 90% for sustained periods. Page life expectancy should exceed 300 seconds and eviction rates should remain under 1000 pages per second in stable operation.

A **three-tier caching hierarchy** maximizes performance across the latency spectrum. L1 cache (1-5% of buffer pool capacity) stores hot metadata like file catalog entries and frequently accessed small files in an in-memory hash table with sub-microsecond access. L2 cache (90% of capacity) implements the main ARC buffer pool with 4KB pages and 1-10μs latency. Optional L3 SSD cache (remaining 5-10%) stores recently evicted blocks using simple LRU with 64-256KB blocks and 10-100μs latency, beneficial when working set exceeds RAM but fits on local SSD.

Memory-mapped I/O provides 2-3x faster read performance than traditional I/O for random access patterns when the working set fits in virtual memory. The implementation should use mmap for files under 4GB on 64-bit systems with sufficient address space, avoiding it for streaming transfers over 30GB where sequential buffered reads perform better. Linux-specific concerns include mmap scaling limited to 8 cores versus NVMe scaling to 80+ cores, suggesting hybrid approaches that combine mmap for hot data with io_uring for cold data and write operations.

The archive format mandates **io_uring on Linux systems** for all asynchronous I/O operations, delivering 2.6x faster performance than synchronous I/O and 1.75x faster than worker thread pools. io_uring's submission queue accepts batches of 8-16 operations submitted simultaneously, reducing system call overhead and enabling true async at the kernel level.   Buffer registration for zero-copy operations eliminates data copies between user and kernel space. The implementation should fall back to tokio::fs on non-Linux systems or tokio-uring when simpler API is preferred over raw io_uring control.

Batching writes reduces I/O overhead by 30-50% through amortization of sync operations. The format buffers small writes in a 4MB staging area, flushing when the buffer reaches capacity or 100ms elapses (whichever comes first). This batching occurs at the content write layer before SQLite VFS operations, allowing multiple file operations to share a single fsync. For SQLite databases within archives, batch commits (BEGIN IMMEDIATE, multiple inserts, COMMIT) similarly reduce fsync overhead from per-operation to per-transaction.

Sequential write layout through **log-structured append** delivers 100-200x faster writes on HDDs and 2-5x faster on SSDs compared to random writes. All new content appends to the archive end regardless of its logical position in the file tree, with the catalog maintaining the mapping from logical paths to physical block ranges. This approach maximizes SSD write lifespan through reduced write amplification, enables simple crash recovery through append-only semantics, and naturally supports copy-on-write snapshots by preserving old data until explicitly compacted.

Alignment optimizations prevent performance degradation from misaligned memory access. Cache line alignment (64 bytes) for hot data structures prevents false sharing in concurrent scenarios, yielding 10-30% improvements on frequently accessed counters and locks. Page alignment (4096 bytes) for large buffers enables direct I/O and memory mapping while satisfying OS requirements. Huge page support (2MB pages) reduces TLB pressure by 512x for buffer pools exceeding 1GB, though this requires explicit configuration on most systems.

Bloom filters eliminate 99% of negative lookups with just 10 bits per key (1% false positive rate). The catalog maintains bloom filters for file existence checks, avoiding expensive disk I/O for non-existent paths. Additional bloom filters on resource prefixes accelerate IAM policy evaluation by quickly eliminating non-matching policies. Filter construction occurs incrementally during catalog updates with minimal overhead, while queries complete in nanoseconds through bit array lookups and hash computations.

Rust implementation roadmap balances safety and performance

The recommended technology stack uses **Rust with NAPI-RS bindings** for Node.js integration, RocksDB for catalog storage, memmap2 for memory mapping, io_uring for async I/O, and ahash for fast hashing. This combination provides memory safety through Rust's ownership system, minimal runtime overhead through zero-cost abstractions, and excellent interoperability with JavaScript through NAPI-RS's automatic binding generation.

The **core architecture** separates concerns into distinct modules: a storage layer handling block allocation and I/O using io_uring, a catalog layer managing the file index and metadata with RocksDB, a VFS layer implementing sqlite3_vfs and sqlite3_io_methods interfaces, a policy engine evaluating IAM rules, an audit logger capturing operations through ring buffers, and an API layer exposing operations to JavaScript via NAPI-RS. This modular design enables independent optimization of each component and facilitates testing through clear boundaries.

Critical implementation details include **panic guards in all FFI boundaries** using `std::panic::catch_unwind` to prevent unwinding into C code, Arc<Mutex<T>> for shared mutable state accessed across threads, careful lifetime management for 'static requirements in callbacks, proper error code translation from Result<T> to SQLITE_* constants, and aligned memory allocation for buffers used in direct I/O. These safety measures prevent undefined behavior while maintaining performance.

The **async handling pattern** wraps synchronous SQLite operations in `tokio::task::spawn_blocking` to avoid blocking the Node.js event loop. NAPI-RS methods take asynchronous signatures returning futures, but internally execute SQLite calls on a dedicated thread pool since SQLite itself is synchronous. This approach maintains responsiveness in JavaScript applications while fully utilizing Rust's efficient synchronization primitives.

Development phases should progress from **minimum viable VFS** (basic read/write with no-op locking, simple file operations, exclusive mode SQLite) through full VFS implementation (WAL support, proper checksums, comprehensive error handling), catalog and free space management (B-tree index, hybrid allocation, basic compaction), IAM policy engine (policy parsing, evaluation with caching, VFS integration), audit logging (ring buffer, async flush, index creation), to final optimization (mmap support, io_uring integration, performance tuning, benchmarking). Each phase delivers incremental value and remains testable independently.

Testing strategy encompasses **unit tests** for individual components (allocator correctness, policy evaluation accuracy, audit log integrity), integration tests against actual SQLite operations (schema creation, transactions, crash recovery, concurrent access), performance benchmarks measuring throughput and latency targets (100K IOPS, sub-10μs cached reads, 90%+ hit rates), and corruption testing simulating power loss scenarios. Fuzzing with cargo-fuzz helps identify edge cases in parsing and state management.

Performance targets establish measurable goals: **read latency under 10μs for cached data** and under 100μs for SSD-backed data, **write latency under 50μs including audit**, **buffer pool hit ratio exceeding 90%**, **throughput exceeding 100K IOPS on modern NVMe**, and **memory overhead under 20% of data size**. Continuous profiling with cargo-flamegraph identifies hotspots requiring optimization, while production telemetry tracks actual performance against targets.

Implementation recommendations prioritize practical deployment

The format should **default to performance-optimized SQLite configuration**: WAL journal mode for concurrency, NORMAL synchronous mode balancing durability and speed (fsync on checkpoint rather than every transaction), memory temporary storage avoiding disk I/O for temp tables, 64KB page size set before database creation for fewer page operations, 100MB page cache (PRAGMA cache_size=-102400) matching the buffer pool, and periodic optimization (PRAGMA optimize) run during maintenance windows.

For **production deployment**, configure 40-60% of system RAM for the buffer pool, enable background compaction with 60-second intervals, set audit flush interval to 100ms balancing latency and throughput, use io_uring on Linux systems with 256-entry submission queues, enable bloom filters with 10 bits per key for 1% false positive rates, configure checkpointing at 4MB WAL size (1000 pages), and implement monitoring for hit ratios, fragmentation levels, and audit lag.

The **snapshot-to-Engram workflow** enables distribution by freezing the mutable Cartridge at a point in time, running VACUUM to optimize layout and reclaim space, optionally applying compression to content blocks, building the immutable Engram format with append-only TOC, copying files with content-addressing for deduplication, and generating final checksums for integrity verification. This workflow typically completes in minutes for gigabyte-scale archives and supports incremental updates through delta compression between snapshots.

Error handling strategies should distinguish between recoverable errors (catalog inconsistency triggering rebuild, partial writes rolling back via WAL, policy evaluation failures defaulting to deny) and fatal errors (header corruption requiring restore from backup, disk full conditions halting operations, checksum failures in critical structures). Comprehensive error types using thiserror enable appropriate handling at each layer while maintaining good error messages for debugging.

Security considerations mandate **input validation on all paths** preventing directory traversal, size limits preventing resource exhaustion (maximum file size, maximum archive size, maximum audit entries), rate limiting on operations preventing abuse, secure default policies denying access unless explicitly allowed, and audit log protection preventing modification without detection through cryptographic integrity mechanisms.

The infrastructure-in-a-box philosophy means Cartridge archives should be **self-contained and portable**, embedding all necessary metadata, policies, and audit trails within the single file format. This enables simple deployment (copy one file), straightforward backup (archive the archive), transparent migration (move between systems without reconfiguration), and clear security boundaries (all access control and audit in one place). The format remains practical for archives from megabytes to terabytes through its scalable index structures and lazy loading patterns.

This specification provides a complete foundation for implementing a production-ready mutable archive format that balances performance, safety, and functionality for modern application development workflows.