HBCMS: A Web-native Hierarchical Blockchain CMS for OTT – Contract-aware APIs, Edge-cache Consistency, and HTTP-level SLOs

Suhwan Bae[1], Jinsook Bong[2], Uijin Jang[3] and Yongtae Shin[4,*]

[1]*Department of Computer Science & Engineering, Soongsil university, Republic of Korea*
[2]*Baird College of General Education, Soongsil University, Republic of Korea*
[3]*Spartan SW Education Center, Soongsil University, Republic of Korea*
[4]*School of Computer Science and Engineering, Soongsil University, Republic of Korea*
E-mail: shbae0213@soongsil.ac.kr; jsbong@ssu.ac.kr; neon7624@ssu.ac.kr; shin@ssu.ac.kr
[*]*Corresponding Author*

Received 02 December 2025; Accepted 15 January 2026

Abstract

Over-the-top (OTT) platforms must expose metadata and digital rights from numerous content providers (CPs) through the web while maintaining low latency and verifiable integrity. This paper presents HBCMS, a hierarchical blockchain-based content management system that separates contract-aware governance in the upper layer from high-rate metadata management in CP-owned lower chains. The web layer is realized through a contract-aware API, a consistency model aligned with edge and browser caching, and HTTP-level service-level objectives (SLOs) linking blockchain verification to observable web behavior. The upper and lower chains are connected via Merkle-root

Journal of Web Engineering, Vol. 25_4, 539–582.
doi: 10.13052/jwe1540-9589.2544
© 2026 River Publishers

anchoring, and verification proceeds through contract validation, anchor matching, and Merkle proof verification exposed as RESTful endpoints. Anchoring is modeled as a Poisson process that determines the rate required to satisfy verification windows and guides content delivery network (CDN) cache-control policies. In large-scale experiments with up to 1000 CPs, HBCMS achieved about 2.6 k transactions per second (TPS), 0.185 s end-to-end latency, and 99.4% verification success, with lower-chain queries dominating delay. These results provide reproducible guidance for API versioning, cache invalidation, and observability in scalable OTT web architectures.

Keywords: Blockchain, OTT, digital rights management, cross-layer verification, Merkle root anchoring, web engineering.

1 Introduction

OTT platforms increasingly rely on distributed CPs for ingestion, metadata refresh, and license synchronization across heterogeneous regions [1, 5, 6]. Global subscription growth and the rapid expansion of multi-territory licensing have amplified both the scale and complexity of content governance. For instance, nearly half of Netflix's catalog demand now originates from licensed, non-exclusive titles [7], and Prime Video's exclusivity share declined from 41% to 24% between 2023 and 2025 [6]. This CP-dependent reality demands not only trustworthy metadata synchronization but also web-layer consistency – ensuring that cached objects, license scopes, and verifiable anchors remain coherent at the scale of millions of distributed HTTP requests.

Traditional digital rights management (DRM) and monolithic content management systems (CMS) struggle to meet these dual requirements. Policy servers and centralized DRM frameworks, typically based on ODRL or permissioned blockchains, introduce single points of policy failure and hinder real-time license enforcement across regions [3, 9]. At the same time, conventional web-engineering practices such as TTL-based caching and conditional validation with entity tags (ETags) [16–18] operate independently of contract or anchoring events. As a result, cache invalidation and content freshness in CDNs often lag behind actual blockchain-verified state changes, leading to transient inconsistencies between on-chain records and delivered media. Thus, the absence of an explicit link between blockchain verification and HTTP cache behavior thus becomes a significant obstacle to both performance and auditability.

To address these challenges, this paper presents HBCMS – a web-native architecture that couples hierarchical blockchain verification with contract-aware web delivery and HTTP- SLOs. HBCMS separates concerns across two interoperable layers:

(1) An *upper governance layer,* which enforces contract validity and records anchored Merkle roots and (2) *autonomous lower CP chains*, which finalize high-rate metadata and content fingerprints.

Cross-layer Merkle-root anchoring connects these layers, while the web layer translates verification results into cache-control semantics, such as bounding CDN/browser TTLs by $TTL_{cdn} \leq k \cdot w$, where w is the verification window and k is an empirically tuned constant ($\sim$1/3). In this way, the synchronization reliability derived from the Poisson-modeled anchor rate (μ) directly informs the caching behavior, purge triggers, and observable web-layer latency.

Compared with prior blockchain-based DRM or metadata frameworks [4, 10, 13], the proposed design explicitly integrates HTTP-level consistency, contract observability, and probabilistic freshness guarantees. The verification results are presented through a RESTful Access API (/access/verify) that performs contract validation, anchor matching, and Merkle-proof verification, returning verifiable storage locators (storage_addr), cache validators (etag), and synchronization hints (ttl_hint). The API also supports idempotent retries and unified error taxonomies aligned with HTTP status codes [20, 21]. Together, these components bridge blockchain auditability with standard web protocols – enabling deterministic access control while maintaining CDN-level efficiency and client-perceived responsiveness.

This work makes four contributions:

(1) Hierarchical OTT governance: Formalizing a two-layer architecture in which contract-aware upper-layer validation regulates access to lower-layer verified metadata.

(2) SLO-driven synchronization: Modeling anchor arrivals as a Poisson process that links the expected verification window (w) to both blockchain and HTTP caching parameters.

(3) Web-layer integration: Embedding blockchain verification into RESTful APIs and cache-control policies, achieving HTTP-observable integrity and freshness coupling.

(4) Empirical validation: Demonstrating at scale (up to 1000 CPs) that HBCMS sustains $\sim$2.6 k TPS with 0.185 s end-to-end latency and 99.4% integrity verification, which is consistent with OTT SLO requirements.

The remainder of this paper is organized as follows. Section 2 reviews related work on blockchain-based DRM, hierarchical multi-chain systems, and web-engineering models for OTT delivery. Section 3 details the system design and formalizes the cross-layer verification pipeline and the HTTP alignment. Section 4 presents the performance evaluation and scalability results, and Section 5 concludes the study with limitations and directions for future research.

2 Related Works

This section surveys research on blockchain-based DRM, blockchain applications for OTT content management, and hierarchical/multi-chain architectures. We highlight the structural, governance, and verification gaps that motivate the proposed HBCMS model. Whereas prior works rely on heuristic or ad-hoc efficiency measures, we clarify their limitations and outline the more reproducible metrics adopted in this study.

2.1 Blockchain-based Digital Rights Management

Blockchain has been widely explored for DRM because of its immutability, traceability, and auditability [3, 9]. Typical approaches hash content/rights/timestamp tuples to create tamper-evident records, where C_i denotes a content identifier, R_i denotes a license record, and T_i a timestamp [3].

$$H_i = SHA256(C_i \| R_i \| T_i) \tag{1}$$

Such models provide integrity guarantees and practical verification latencies in consortium settings, yet throughput commonly degrades beyond $\mathrm{O}(10^2)$ tps as participant counts increase [10]. Policy-driven frameworks based on ODRL and permissioned ledgers improve expressiveness but often re-introduce semi-centralized control planes (e.g., policy servers), reducing provider autonomy and complicating independent audits. Formal latency decompositions of contract validation, explain practical bottlenecks observed in permissioned deployments [10].

$$T_v = T_c + T_s + T_n \tag{2}$$

where T_c, T_s, and T_n denote the contract parsing, signature verification, and network propagation, respectively. Collectively, these DRM-focused studies

validate blockchain's potential while exposing scalability and governance limits that become acute in multi-provider OTT ecosystems.

2.2 Blockchain OTT Content Management

Applying blockchain to OTT content management aims to reduce metadata fragmentation, licensing misalignment, and cross-regional verification delays [7]. Empirical models of repository health suggest that the probability of duplicate registrations increases with the number of CPs with $P_d > 0.75$ when $N > 50$ under representative λ [8].

$$P_d = 1 - e^{-\lambda N} \tag{3}$$

This indicates that monolithic CMS designs are ill-suited for large, heterogeneous OTT pipelines [5, 6]. Prototype systems integrating blockchains have improved traceability and audit trails, but reported metadata operation rates ($\sim$15–20 tps) fall short of commercial OTT needs ($>$2 k tps sustained) and generally lack contract-aware access control synchronized with the on-chain state [3, 8]. These observations motivate architectures that preserve CP autonomy while enabling real-time, verifiable access governance [13].

2.3 Hierarchical, Multi-chain, and Cross-chain Architectures

To overcome single-chain limits, previous studies have investigated multi-chain, sharded, and hierarchical topologies [14–16]. Cross-chain latency is often modeled as a weighted composition of intra-chain delays plus synchronization cost, showing that distributing workloads across heterogeneous chains can reduce end-to-end latency under mixed traffic. Sharding introduces effective per-shard complexity of $\mathrm{O}(N/k)$ with k shards, yielding multi-fold throughput gains in large networks [15]. However, inter-chain governance and verifiable linkage remain under-specified in many designs: interoperability protocols frequently trade security/finality for speed and hierarchical verification (e.g., Merkle-root anchoring with auditable finality across layers) is rarely formalized. For OTT, the absence of a contract-aware, cross-layer verification pipeline and explicit finality rules is a critical gap because providers must prove that off-chain or lower-chain states are both fresh and policy-compliant when access is granted [16].

2.4 Synthesis of Limitations and Motivation for HBCMS

A synthesis of the above literature reveals three persistent limitations that HBCMS is designed to address:

(1) Structural scalability. Single-chain DRM and monolithic CMS approaches suffer from O(N) coordination overheads as the CPs scales, whereas multi-chain designs without verifiable linkage cannot provide global auditability [14, 15]. A hierarchical approach that permits independent lower-layer finality and lightweight upper-layer anchoring is required to approach near-logarithmic throughput growth observed empirically observed under balanced loads.

(2) Governance and autonomy. Policy servers or centrally curated contracts in prior DRM deployments reduce CP autonomy and create single points of policy failure [3, 9]. A contract-aware governance layer that is on-chain, auditable, and decoupled from lower-layer ingestion is required to preserve autonomy while ensuring uniform, testable enforcement.

(3) Verification and finality. Many interoperability schemes lack provable cross-layer integrity and explicit finality semantics [14–16]. OTT settings demand a pipeline that (i) validates contract scope, (ii) checks upper-layer anchors against lower-layer Merkle roots, and (iii) verifies membership proofs – all within strict latency budgets. In addition, synchronization should be engineered as a probabilistic SLO rather than heuristic cadence; in later sections therefore ground anchoring in the Poisson model from (1) and reproducible efficiency metrics in lieu of ad hoc indices are adopted.

2.5 Web Engineering for OTT Delivery and Caching

Web engineering studies have examined HTTP caching, CDN consistency, and freshness semantics.

Techniques such as TTL-based cache expiry, conditional validation using ETags, and event-driven invalidation at the edge are well established [16–18].

These mechanisms enable scalable delivery and low-latency content reuse across geographically distributed clients.

However, OTT-specific research seldom integrates these cache-control primitives with the probabilistic or time-bounded synchronization models that characterize blockchain-anchored metadata.

Recent efforts in content-delivery optimization highlight the need for cache consistency models aware of dynamic contract or license updates [19].

This motivates the use of formally defined freshness and purge policies that can interoperate with verifiable anchors or provenance proofs, ensuring that distributed caches deliver content that remains both timely and policy compliant.

2.6 Contract-aware Web API and Observability

Web-native API design for decentralized media systems has received less systematic attention than consensus or ledger mechanisms.

Prior works discussed RESTful verification interfaces, but few specified contract-aware API semantics or unified error taxonomies for blockchain-backed OTT platforms [3].

Research on idempotent HTTP transactions and API versioning demonstrates their importance for reliable and replay-safe operations under fluctuating network conditions [20].

Likewise, recent literature on SLOs and web observability emphasizes percentile latency, error-budgeting, and cache hit ratio as measurable indicators of user-perceived performance [21, 22].

Finally, advances in browser-level DRM integration (e.g., EME/MSE) and policy expression via ODRL align with the notion of embedding verifiable rights and contractual metadata directly into web-layer tokens [9, 23].

These works collectively position the web layer – not merely as a delivery endpoint – but as a governable, measurable component of decentralized OTT architectures.

3 System Design

3.1 Architecture Overview

The HBCMS integrates blockchain-level integrity guarantees with web-layer consistency mechanisms to support contract-aware, low-latency OTT delivery. The system separates concerns across three interoperable layers: a web layer, an upper governance layer, and multiple lower CP chains, with off-chain storage used for large content objects. The overall architecture is shown in Figure 1.

(1) Web layer. At the entry point, the *web layer* exposes contract-aware verification and lookup functionalities through the *Access API* (e.g., */access/verify*, */access/search*). It mediates all HTTP-level interactions between clients and the blockchain, returning both content locators and

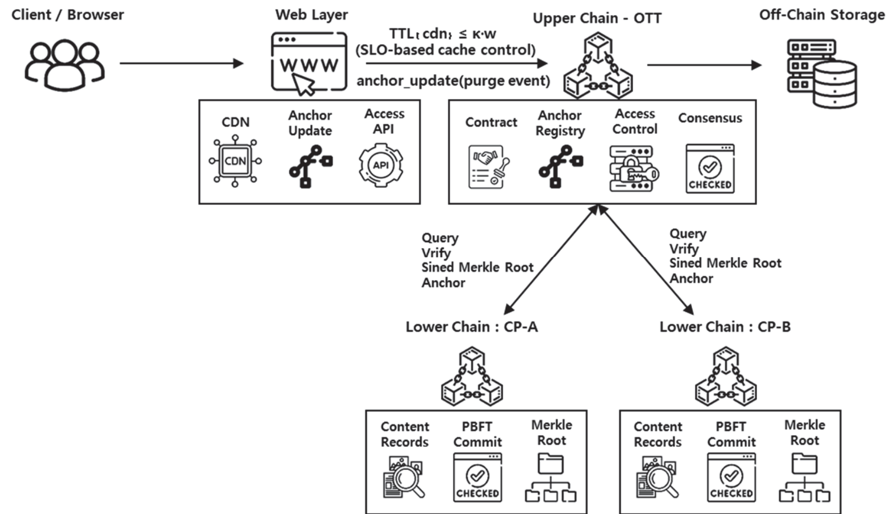

Figure 1 HBCMS system architecture.

integrity hints such as *ETag* and *TTL_hint*. A *CDN/edge Cache* is incorporated to accelerate content delivery while maintaining consistency under the synchronization SLO. Specifically, the cache lifetime is bounded by $TTL_{cdn} \leq k \cdot w$, where w is the verification window and k is an empirically determined constant (∼1/3). When a new anchor is accepted at the upper layer, an *anchor.updated* → *purge* event is triggered to invalidate outdated cached objects, ensuring alignment between the blockchain-verified state and the web-delivered content.

(2) Upper governance layer. The upper chain acts as the authoritative governance domain responsible for policy enforcement, contract validation, and cross-layer verification. It maintains a registry of contract metadata, an anchor registry that stores the latest signed Merkle roots from each CP, and an access control component that applies contract-scope filtering. Consensus is achieved through a Byzantine fault tolerance (BFT)/proof of authority (PoA) mechanism to provide deterministic finality, guaranteeing that accepted anchors and contract updates remain tamper-evident and auditable globally.

(3) Lower CP chains. Each CP operates an autonomous lower chain that finalizes high-frequency metadata and content fingerprints under a practical Byzantine fault tolerance (PBFT)-style consensus. Every block records

content records containing *content_id*, metadata, and *fingerprints*; its Merkle root (M_{lower}) is periodically signed and anchored to the upper layer along with a timestamp and signature tuple (M_{lower}, σ, t). These anchors establish cryptographic linkage between local commits and upper-layer auditability, while preserving provider autonomy.

(4) Off-chain storage. Actual multimedia objects are stored outside the blockchain in a cloud or object storage service. The blockchain retains only essential metadata, contract proofs, and integrity anchors to minimize on-chain overhead.

As depicted in Figure 1, this three-layer integration allows HBCMS to couple blockchain verification with observable web behavior. The web layer enforces freshness and consistency at the HTTP level, the upper layer ensures contract validity and global auditability, and the lower chains maintain deterministic finality for the CP-specific metadata commits. Table 1 summarizes the common notations used throughout Sections 3 and 4.

Table 1 Notation for HBCMS verification and SLO binding

Category	Symbol	Unit	Description
Entities/scale	N	–	Number of CPs.
	n	–	Number of validators per lower CP chain.
Latency components	T_{upper}	s	Upper-layer processing latency.
	T_{lower}	s	Lower-layer query latency.
	T_{verif}	s	Cryptographic verification latency.
	T_{total}	s	End-to-end verification latency.
Latency statistics	μT	s	Mean of end-to-end latency T_{total}.
	σT	s	Standard deviation of T_{total}.
	P95, P99	s	95th/99th percentile of T_{total}.
SLO parameters	τ	s	Latency deadline for SLO.
	ρ	–	Target probability for SLO satisfaction.
Anchoring model	M_{upper}		Anchored Merkel root recorded at the upper layer.
	M_{lower}		Latest Merkle root obtained from the lower CP chain.
	μ	1/s	Anchor-arrival rate per CP
	w	s	Verification window length.
HTTP cache binding	k	–	Coupling constant between cache TTL and window w.
	TTL_{cdn}	s	CDN/edge cache TTL bound derived from w.
	$\text{TTL}_{\text{browser}}$	s	Browser TTL bound.

3.2 Functional Requirements and Design Goals

3.2.1 Functional requirements

To operate effectively in a heterogeneous OTT environment, HBCMS must satisfy:

(1) Distributed governance: Each CP autonomously finalizes its own lower-layer blockchain, whereas the upper governance layer manages only policies and anchored Merkle roots.

(2) Real-time verification: Policy compliance and integrity verification must be completed within the OTT verification window (<0.3 s) to support low-latency access control.

(3) Auditability: The system must enable post-hoc verification of consistency between the upper- and lower-layer states as well as contract enforcement histories.

(4) Scalability: As the number of CPs increases, the architecture should sustain near-logarithmic throughput growth without introducing global coordination bottlenecks.

3.2.2 Design goals

The following objectives are guided by system design:

(1) SLO-based synchronization: The anchoring frequency (μ) is dimensioned to satisfy a target verification window (w) according to the probabilistic SLO model.

(2) Explicit finality: The upper-layer consensus must define a formal rule for finality, whether via instantaneous deterministic commitment under BFT protocols or k-block probabilistic finality under proof of work (PoW) mechanisms.

(3) Consistency metrics: To ensure reproducible evaluation, the system adopts measurable efficiency indicators such as strong-scaling efficiency and logarithmic efficiency, unified with the experimental methodology presented in Section 4.

3.3 Data Model

3.3.1 Lower chain data model

The lower chain functions as an autonomous data-management layer operated by individual CPs. It handles high-frequency content updates and guarantees deterministic finality through PBFT-style consensus.

Each block (LowerBlock) contains multiple ContentRecords, and each record is included in Table 2.

Table 2 Data model of the lower chain

Field Name	Description	Role in System
content_id	Unique identifier for each content asset.	Serves as the primary key for content-level verification.
info	Descriptive metadata such as title, region, and tags.	Provides human-readable attributes for content lookup.
fingerprint	Cryptographic or perceptual hash of the content.	Ensures identity and integrity of the digital asset.
storage_addr	Off-chain storage URI or object storage key.	Points to the actual content stored outside the blockchain.
drm	Optional DRM or encryption metadata.	Stores license and protection details for rights management.
ts	Commit timestamp when the record was finalized.	Enables temporal tracking of content updates.
index	Sequential index of the lower block.	Maintains ordering in the CP's local chain.
prev_hash	Hash of the previous block.	Links blocks to maintain immutability.
timestamp	Creation time of the block.	Provides temporal traceability of commits.
merkle_root	Merkle root over all entries in the block.	Acts as the atomic anchor submitted to the upper chain.
block_hash	Hash of the current block header.	Uniquely identifies the lower block.

Each LowerBlock includes a Merkle root computed from its entries, along with its prev_hash, timestamp, and block_hash. The Merkle root acts as an atomic anchor that is periodically transmitted and verified by the upper chain. This implementation corresponds to the structural representation shown in Figure 2.

3.3.2 Upper chain data model

The upper chain serves as the governance layer that manages contract metadata, signed Merkle-root anchors, and cross-chain audit records.

Each block in this layer consists of one or more UpperRecord objects that encapsulate several metadata fields and linkage attributes (see Table 3).

Each UpperBlock stores multiple UpperRecord objects together with block metadata (index, prev_hash, timestamp, nonce, and block_hash). This structure enables the auditable recording of contract states and anchored lower-layer roots, ensuring traceability and trust across multiple CPs. The implemented structure is illustrated in Figure 3.

```
@dataclass
class ContentRecord:
    content_id: str
    info: Dict[str, Any]              # Descriptive metadata (e.g., title, region, tags)
    fingerprint: str                  # Content hash or perceptual fingerprint
    storage_addr: str                 # Off-chain storage URI or object store key
    drm: Dict[str, Any] | None = None # Optional DRM or encryption metadata
    ts: float = 0.0                   # Commit timestamp (in seconds since epoch)

@dataclass
class LowerBlock:
    index: int                        # Sequential block index
    prev_hash: str                    # Hash of the previous block
    timestamp: float                  # Block creation time
    entries: List[ContentRecord]      # List of committed content records
    merkle_root: str                  # Root hash of the Merkle tree over entries
    block_hash: str                   # Hash of the current block header
```

Figure 2 Lower chain data model.

Table 3 Upper chain block component

Field Name	Description	Role in System
content_id	Unique identifier for each content asset.	Serves as the primary key for content-level verification.
info	Descriptive metadata such as title, region, and tags.	Provides human-readable attributes for content lookup.
fingerprint	Cryptographic or perceptual hash of the content.	Ensures identity and integrity of the digital asset.
storage_addr	Off-chain storage URI or object storage key.	Points to the actual content stored outside the blockchain.
drm	Optional DRM or encryption metadata.	Stores license and protection details for rights management.
ts	Commit timestamp when the record was finalized.	Enables temporal tracking of content updates.
index	Sequential index of the lower block.	Maintains ordering in the CP's local chain.
prev_hash	Hash of the previous block.	Links blocks to maintain immutability.
timestamp	Creation time of the block.	Provides temporal traceability of commits.
merkle_root	Merkle root over all entries in the block.	Acts as the atomic anchor submitted to the upper chain.
block_hash	Hash of the current block header.	Uniquely identifies the lower block.

```
@dataclass
class ContractData:
    cp_id: str                          # Content Provider (CP) identifier
    expiry_ts: float                    # Contract expiration timestamp
    regions: List[str]                  # List of authorized regions
    allowed_content_ids: List[str]      # Access catalog: IDs of permitted content
    meta: Dict[str, Any] = field(default_factory=dict)  # Additional contract metadata

@dataclass
class UpperRecord:
    cp_id: str                          # Associated CP identifier
    contract_snapshot: Dict[str, Any] # Serialized snapshot of ContractData
    lower_root: str                     # Signed Merkle root anchored from lower chain
    access_catalog: List[str]           # Authorized content list (sorted)
    anchor_ts: float                    # Verified anchor timestamp from CP

@dataclass
class UpperBlock:
    index: int                          # Sequential block index
    prev_hash: str                      # Hash of the previous governance block
    timestamp: float                    # Block creation timestamp
    records: List[UpperRecord]          # List of governance records (per CP)
    nonce: int                          # PoW-style nonce (if difficulty > 0)
    block_hash: str                     # Hash of the current governance block
```

Figure 3 Upper chain data model.

3.3.3 API design, versioning, and error taxonomy

To facilitate interoperable access between CPs and the upper-layer governance chain, HBCMS provides a minimal contract-aware RESTful API that exposes verification and lookup functions through a single entry point */access/verify*. The API accepts a JSON payload containing *cp_id*, *content_id*, *region*, and timestamp *t_now*, returning both the content location (e.g., storage address) and integrity hints (e.g., *ETag* and *TTL* suggestion) if validation succeeds. Figure 4 illustrates the OpenAPI 3.0.3-based definition of the verification endpoint.

(1) API versioning and idempotency. To ensure reproducible validation across evolving policies, the interface adopts URL-based versioning combined with a message-level header (*X-Contract-Snapshot-Version*), which identifies the contract snapshot used during verification. This dual mechanism preserves interoperability and audit traceability across CP updates. Replay safety is achieved through *Idempotency-Key* headers that bind each request to its previous consistent response. Upon retries caused by network latency or client congestion, the server reuses the cached outcome instead of re-executing the verification logic, eliminating redundant anchoring operations and ensuring idempotent behavior.

```
openapi: 3.0.3
info: { title: HBCMS Web API, version: v1 }
paths:
  /access/verify:
    post:
      summary: Contract-aware verification & content lookup
      requestBody:
        required: true
        content:
          application/json:
            schema:
              type: object
              required: [cp_id, content_id, region, tnow]
              properties:
                cp_id: {type: string}
                content_id: {type: string}
                region: {type: string}
                tnow: {type: string, format: date-time}
      responses:
        '200': { description: OK (returns storage_addr, etag, ttl_hint) }
        '403': { description: POLICY }
        '409': { description: ANCHOR_MISMATCH }
        '422': { description: PROOF_INVALID }
        '429': { description: Rate limited }
```

Figure 4 OpenAPI Sinppet.

(2) Error handling and taxonomy. Logical errors are standardized through compact mapping of HTTP status codes, facilitating automated policy handling and system observability. Specifically, *POLICY* violations (expired or out-of-scope contracts) return *403 Forbidden*; *ANCHOR_MISMATCH* (inconsistent upper/lower Merkle anchors) returns *409 Conflict*; and *PROOF_INVALID* (malformed or unverifiable Merkle proof) returns *422 Unprocessable Content*. These categories, further analyzed in Section 4.4, enable clients to interpret responses uniformly while supporting monitoring and policy analytics across heterogeneous CP environments.

3.4 Upper Governance Layer

3.4.1 Consensus and finality

The upper layer ensures reliable governance and auditable recordkeeping by employing a consensus mechanism with explicit finality guarantees.

(1) Default mode – BFT/PoA (authorized consensus). When operated under a BFT or PoA, blocks are finalized immediately within a trusted validator set, offering deterministic finality and predictable latency.

```
class UpperChain:
    def __init__(self, difficulty: int = 0):
        # difficulty = 0 → deterministic finality mode (e.g., BFT/PoA-like)
        # difficulty > 0 → PoW-style sealing (open participation)
        self.difficulty = max(0, difficulty)
        genesis = UpperBlock(
            index=0, prev_hash="0"*64, timestamp=time.time(),
            records=[], nonce=0, block_hash=sha256_hex(b"genesis")
        )
        self.blocks: List[UpperBlock] = [genesis]

    def _seal_block_hash(self, prev_hash: str, ts: float, records: List[UpperRecord]) -> Tuple[str, int]:
        """
        If difficulty == 0: return a deterministic hash (no nonce search).
        If difficulty  > 0: perform a toy PoW search for a hash with leading zeros.
        """
        payload = {"prev_hash": prev_hash, "timestamp": ts, "records": [r.__dict__ for r in records]}
        base = _json_canonical(payload)

        if self.difficulty <= 0:
            # Deterministic sealing: models BFT/PoA immediate finality at the upper layer
            return sha256_hex(base), 0

        # PoW-style sealing: models probabilistic finality with k-confirmations
        nonce = 0
        prefix = "0" * self.difficulty
        while True:
            h = sha256_hex(base + str(nonce).encode("utf-8"))
            if h.startswith(prefix):
                return h, nonce
            nonce += 1

    def create_block(self, recs: List[UpperRecord]) -> UpperBlock:
        """Create and append a governance block from prepared records."""
        if not recs:
            raise ValueError("no CP records to include")
        prev_hash = self.blocks[-1].block_hash
        ts = time.time()
        block_hash, nonce = self._seal_block_hash(prev_hash, ts, recs)
        blk = UpperBlock(index=len(self.blocks), prev_hash=prev_hash, timestamp=ts,
                         records=recs, nonce=nonce, block_hash=block_hash)
        self.blocks.append(blk)
        return blk
```

Figure 5 Consensus and finality.

System parameters such as the validator count and timeout thresholds are configured to maintain stable throughput and low-latency confirmation in the upper governance layer.

(2) Alternative mode – PoW (open participation). For open participation scenarios, the upper layer can adopt a PoW mechanism.

The expected block generation time is determined by the global hash rate H and network difficulty D follows:

$$\mathrm{E}[T_{Upper}] = \frac{2^D}{H} \tag{4}$$

A k-confirmation rule is applied such that an anchor becomes valid only after a depth of k blocks, ensuring probabilistic finality and mitigating

reorganization risks. In PoW mode, *k* directly affects the time until an anchor can be treated as final and therefore the freshness the web layer can be safely exposed. Let *B* denote the expected PoW block interval (from (4)). A simple approximation for added finality lag is $L_k \approx k \cdot B$. If the deployment targets observing at least one *finalized* anchor within window *w*, the effective window becomes $w_{\text{eff}} = w - L_{\text{k}}(w > L_{\text{k}})$, and the required rate increases to $\mu \geq -\ln(1-\rho)/w_{\text{eff}}$. Consequently, TTL bounds should be computed against w_{eff} (or w increased) to avoid serving objects whose anchor is not yet final. For example, if $B = 10$ s and $w = 60$ s, then $k = 1$ yields $w_{\text{eff}} \approx 50$ s, while $k = 6$ yields $w_{\text{eff}} \approx 0$, requiring either longer w or higher μ. The implemented structure is illustrated in Figure 5.

HBCMS assumes a permissioned upper governance layer operated by the OTT platform (or a consortium) with an explicitly managed validator set. In the default BFT/PoA mode, we assume a standard Byzantine setting in which the protocol's safety/liveness guarantees hold within the trust boundary of the validator set (e.g., fewer than a threshold fraction of validators are compromised at a time). Under this assumption, contract updates and accepted anchors are final and globally auditable.

We consider adversaries who control (i) one or more CPs (forged anchors, replayed submissions, inconsistent lower-chain reporting), (ii) the network/CDN caches (stale objects), and/or (iii) off-chain storage (tampering). HBCMS mitigates these threats through signature/HMAC checks, cross-layer Merkle-root matching (Mupper vs. Mlower), Merkle proof verification, and SLO-bounded TTL/purge coupling. If the upper validator set itself is compromised beyond the trust threshold (e.g., colluding authorities), the upper layer may accept invalid governance updates; such failures are out of scope for cross-layer verification and must be mitigated operationally (validator hardening, audit logging, and governance controls).

3.4.2 Governance layer

The upper layer maintains an on-chain registry of content-provider (CP) contracts, serving as the authoritative source for policy enforcement and access control. The structure of the governance layer is illustrated in Figure 6, and its implementation can be realized as shown in Figure 7.

(1) Contract lifecycle management. Each CP's contract state is managed explicitly through *registration*, *renewal*, and *revocation* events. These operations define how the upper layer governs active and expired contracts. All contract expirations must reference future timestamps to prevent premature invalidation or rollback of the state of governance.

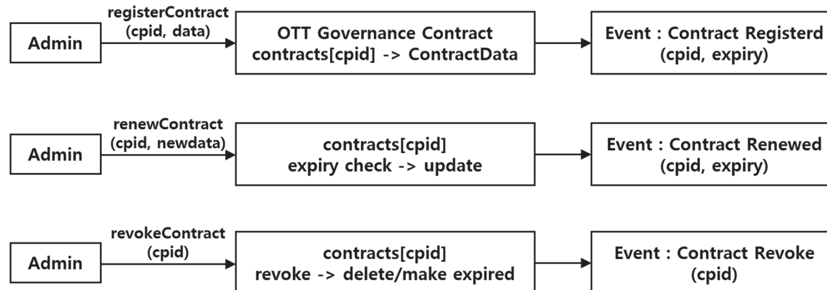

Figure 6 Upper layer governance: Contract event.

```
class UpperChainContractsMixin:
    """
    Contract registry and basic policy helpers for access control decisions.
    Intended to be mixed into UpperChain or used as a delegated component.
    """
    def __init__(self):
        self.contracts: Dict[str, ContractData] = {}

    # --- Contract lifecycle ---

    def register_contract(self, c: ContractData) -> None:
        # Admin-side registration. Assumes expiry_ts is validated as a future timestamp.
        self.contracts[c.cp_id] = c

    def renew_contract(self, c: ContractData) -> None:
        # Admin-side renewal. Fails if the contract is missing.
        if c.cp_id not in self.contracts:
            raise KeyError("contract not found")
        self.contracts[c.cp_id] = c

    # --- Policy helpers used during verification ---

    def _contract_valid(self, cp_id: str, when_ts: Optional[float] = None) -> bool:
        c = self.contracts.get(cp_id)
        ts = when_ts if when_ts is not None else __import__("time").time()
        return bool(c and ts < c.expiry_ts)

    def _allowed(self, cp_id: str, content_id: str) -> bool:
        c = self.contracts.get(cp_id)
        return bool(c and content_id in c.allowed_content_ids)
```

Figure 7 Governance contract.

(2) Audit events and recordkeeping. Each lifecycle event – *registration*, *renewal*, or *revocation* – is logged on-chain to ensure the transparency and post-hoc auditability of governance decisions. When a governance block is created, an *UpperRecord* object stores both the *contract_snapshot* and the corresponding anchor metadata, creating a verifiable trail between the contract events and anchored states. This process is depicted in Figure 6 as a contract-event flow.

(3) Implementation. The reference implementation, illustrated in Figure 7, instantiates these functions within the *governance contract*. The administrator executes contract lifecycle operations via smart-contract calls such as *set_cp_hmac_key()* and *receive_anchor()*. Each call appends immutable records to the governance ledger, guaranteeing deterministic replayability and preventing unauthorized overwrites.

Together, these mechanisms ensure that all governance actions – contract updates, key assignments, and anchor receptors – remain verifiable, auditable, and synchronized across the upper and lower layers.

HBCMS uses CP authentication material to prevent anchor forgery and replay. We support (i) a public-key signature setting, where each CP registers its public key at the upper layer and keeps its private signing key in a secure keystore/HSM; and (ii) a shared-secret HMAC setting for limited-trust deployments, where the secret is provisioned out-of-band and stored in a secure key manager while the upper chain records only key identifiers, validity windows, and the active key version for auditability.

(1) Key rotation and overlap. CP keys are versioned (cp_id $\rightarrow$ key_version, valid_from, valid_to, status). During a short overlap window, anchors signed under either the previous or new key version are accepted; verifiers select the newest non-revoked version.

(2) Compromise recovery. If compromise is suspected, the affected key version is revoked and the CP is quarantined until re-enrollment under a fresh key version. After recovery, the CP submits a fresh anchor and the existing anchor.updated $\rightarrow$ purge mechanism invalidates stale cached objects.

(3) Nonce-/timestamp-window replay defenses. In addition to signature/HMAC verification and strict timestamp monotonicity (t must strictly increase), the upper layer can enforce a bounded acceptance window for anchor timestamps and reject duplicates within a sliding window. For the Access API, replay safety is complemented by Idempotency-Key (server-side replay cache) and validation of client-supplied t_{now} within a bounded skew.

3.4.3 Anchor registry and upper block creation

The upper layer maintains a registry of the latest anchors reported by each CP, thereby providing the foundation for cross-layer integrity verification.

(1) Definition. For each content provider cp, the system stores the most recent anchor:

$$\mathrm{A[cp]} = < M_{lower}, \sigma, t > \tag{5}$$

where M_{lower} is the Merkle root of the lower chain, σ is the signature generated by the CP key, and t is the anchor timestamp.

(2) Verification rules. The signature σ must be verified correctly against the registered CP key, and the timestamp t must be strictly increased, preventing rollback or replay attacks.

Implementation: The receive_anchor() function validates both the signature integrity and monotonicity before updating the anchors[cp_id] = (root, ts, sig).

(3) Policy and exception handling. If an anchor mismatch persists for more than three consecutive synchronization attempts, the CP is quarantined and an operator alert is issued. This mechanism detects potential chain reorganization or faulty reporting.

Implementation suggestion: Introduce an ANCHOR_MISMATCH counter and quarantine flag within the monitoring subsystem.

(4) Cross-layer verification pipeline. During the access verification, the upper layer performs three checks:

(i) Contract validity.
(ii) Root equivalence between the upper anchor and the latest Merkle root of the lower chain.
(iii) Merkle proof verification of the requested content leaf.

The implemented structure is illustrated in Figure 8.

3.5 Lower CP Layer

The process of block generation and anchoring in HBCMS consists of two distinct yet interlinked layers: lower chain, responsible for deterministic content commits, and upper chain, responsible for governance-level anchoring and auditability.

The overall workflow is shown in Figure 9.

Each CP independently operates a LowerChain that deterministically finalizes content updates.

The process begins by collecting pending entries (ContentRecords), computing leaf hashes over canonical JSON objects using SHA-256, and generating a Merkle root representing the set of committed entries.

A new block is then sealed by computing a deterministic block_hash over the block header, which includes {index, prev_hash, timestamp, Merkle_root, cp_id}.

```
class UpperChainAnchorsMixin:
    """
    Anchor registry: stores the latest <lower_root, signature, timestamp> per CP,
    verifies CP signatures, and enforces strictly increasing timestamps.
    """
    def __init__(self):
        # cp_id -> (root_hex, ts, sig_hex)
        self.anchors: Dict[str, Tuple[str, float, str]] = {}
        # Demo key store: cp_id -> shared-secret bytes (replace with public-key crypto in prod)
        self.cp_hmac_keys: Dict[str, bytes] = {}

    # --- Demo key management (replace with ECDSA/Ed25519 in production) ---

    def set_cp_hmac_key(self, cp_id: str, secret: str) -> None:
        """Register CP's shared secret for HMAC-based anchor signing (demo only)."""
        self.cp_hmac_keys[cp_id] = secret.encode("utf-8")

    def _make_anchor_sig(self, cp_id: str, root: str, ts: float) -> str:
        """Compute HMAC-SHA256 over 'root|ts' using the CP's shared secret."""
        key = self.cp_hmac_keys[cp_id]
        return hmac.new(key, f"{root}|{ts}".encode("utf-8"), hashlib.sha256).hexdigest()

    # --- Anchor ingestion and verification ---

    def receive_anchor(self, cp_id: str, root: str, ts: Optional[float], sig: str) -> None:
        """
        Accept a new anchor from CP and verify:
          (1) signature correctness under the registered CP key,
          (2) strictly increasing timestamp to prevent rollback/replay.
        Stores the latest tuple as anchors[cp_id] = (root, ts, sig).
        """
        if cp_id not in self.cp_hmac_keys:
            raise PermissionError("unknown CP key")

        ts = float(ts) if ts is not None else time.time()
        expected = self._make_anchor_sig(cp_id, root, ts)
        if not hmac.compare_digest(expected, sig):
            raise PermissionError("invalid anchor signature")

        last = self.anchors.get(cp_id)
        if last and ts <= last[1]:
            raise ValueError("non-monotonic anchor timestamp")

        self.anchors[cp_id] = (root, ts, sig)

    # --- Optional: convenience accessors ---

    def get_anchored_root(self, cp_id: str) -> str:
        """Return the latest anchored Merkle root for a CP (or '' if missing)."""
        tup = self.anchors.get(cp_id)
        return tup[0] if tup else ""
```

Figure 8 Anchor registry.

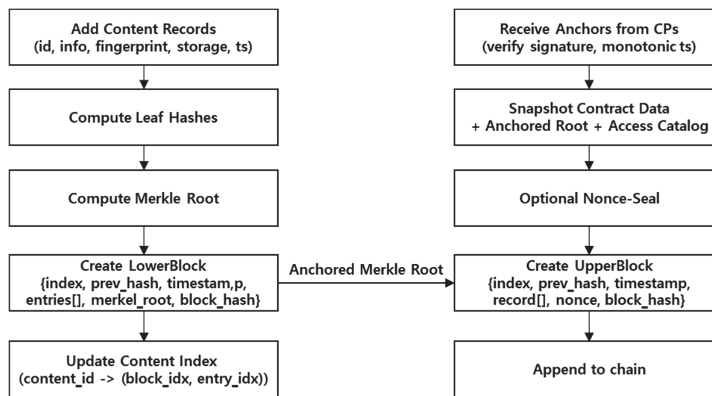

Figure 9 Lower chain workflow.

```
def finalize_block(self) -> LowerBlock:
    if not self.pending:
        raise ValueError("no pending content to commit")

    index = len(self.blocks)
    prev_hash = self.blocks[-1].block_hash
    entries = list(self.pending)

    # (1) Compute leaf hashes (SHA-256 over canonical JSON)
    leaf_hashes = [hash_content_record(e) for e in entries]

    # (2) Compute Merkle root
    root = merkle_root_hex(leaf_hashes)

    # (3) Seal block header
    header = {"index": index, "prev_hash": prev_hash,
              "timestamp": time.time(), "merkle_root": root, "cp_id": self.cp_id}
    block_hash = sha256_hex(_json_canonical(header))

    # (4) Create block and clear pending
    block = LowerBlock(index=index, prev_hash=prev_hash,
                       timestamp=header["timestamp"],
                       entries=entries, merkle_root=root,
                       block_hash=block_hash)
    self.blocks.append(block)
    self.pending.clear()

    # (5) Update content index
    for i, e in enumerate(entries):
        self._content_index[e.content_id] = (index, i)
    return block
```

Figure 10 Deterministic block finalization.

```
def _get_nested(d: Dict[str, Any], dotted: str) -> Optional[Any]:
    cur = d
    for p in dotted.split("."):
        if isinstance(cur, dict) and p in cur:
            cur = cur[p]
        else:
            return None
    return cur

def _match(rec: ContentRecord, query: Dict[str, Any]) -> bool:
    for k, v in query.items():
        if k == "content_id":
            if rec.content_id != v: return False
        elif k.startswith("info."):
            val = _get_nested(rec.info, k.split(".", 1)[1])
            if val is None or str(v).lower() not in str(val).lower():
                return False
        elif k == "fingerprint":
            if rec.fingerprint != v: return False
        else:
            return False
    return True

def search(self, query: Dict[str, Any]) -> List[ContentRecord]:
    results = []
    for blk in self.blocks[1:]:  # skip genesis
        for e in blk.entries:
            if _match(e, query):
                results.append(e)
    return results
```

Figure 11 Nested-key search.

Finally, the content index is updated as content_id $\rightarrow$ (block_idx, entry_idx) to enable efficient lookup.

Search functionality is implemented through the search(query) method, which supports nested-key queries (e.g., “info.title”, “info.region”), allowing CPs to locate content efficiently using hierarchical metadata fields.

3.6 Cross-layer Anchoring and SLO Dimensioning

3.6.1 Anchoring protocol

Each CP signs its local Merkle root M_{lower} and submits it to the upper governance chain as an authenticated anchor, establishing cryptographic linkage between the lower-layer state and the governance ledger.

(1) Signature verification. Upon anchor receipt, the upper layer verifies the CP’s signature σ over $(M_{lower}|t)$ using the registered public key (or a shared secret in limited-trust settings). This validation ensures that the reported root originates from an authenticated CP and is not forged or altered.

(2) Monotonicity enforcement. To prevent replay or rollback, anchor timestamp *t* must be strictly increased relative to the previous submission. Any submission that violates this monotonicity rule is rejected, thereby preserving the temporal consistency of reported anchors.

If the check passes, the tuple (*root, ts, sig*) is stored as *anchors[cp_id]* in the governance registry.

Together, these mechanisms guarantee both the authenticity and temporal integrity of lower-chain submissions, forming the basis for cross-layer verification in subsequent steps.

3.6.2 Synchronization as an SLO

The anchoring frequency and verification latency in HBCMS are modeled as a probabilistic service-level objective (SLO) to balance real-time verification with the operational efficiency.

Assuming that anchors arrive according to a homogeneous Poisson process of rate μ, the probability of observing at least one anchor within a verification window *w* is $1 - \exp(-\mu w)$, which yields Equations (6) and (7) [24].

To achieve a target synchronization probability of 95%, the anchoring rate must satisfy:

$$\mu \geq -\frac{\ln(0.05)}{w} \tag{6}$$

For example, if $w = 60$ s, then

$$\mu \approx \frac{1}{20}\mathrm{s}^{-1} \tag{7}$$

Thus, maintaining one anchor every 20 s ensures a 95% probability of synchronization within a one-minute window.

This corrected the common misconception that a single anchor per hour would suffice for a 95% guarantee. In practice, HBCMS configures μ according to the verification latency requirements of the OTT services.

3.6.3 Aggregated anchoring

To enhance scalability, HBCMS supports aggregated anchoring by using a b-ary aggregation tree.

In this configuration, the per-CP verification path complexity is (8),

$$\mathrm{O}(\log_b N) \tag{8}$$

and the number of anchors published in the upper chain was reduced to (9).

$$\mathrm{O}\left(\frac{\mathrm{N}}{\mathrm{b}}\right) \tag{9}$$

This approach merges multiple CP Merkle roots into a single aggregated root for posting, while preserving per-CP signatures and verifiability.

The resulting hierarchical anchoring structure significantly reduces the global anchoring overhead without compromising the local independence or auditability.

3.6.4 SLO-aligned web caching and consistency

Anchoring arrivals per CP are modeled as a Poisson process with rate μ; by standard results for Poisson processes and exponential waiting times, the probability of observing at least one anchor within a window w is given by (10), and the required anchoring rate to meet reliability target ρ is given by (11) [24].

$$P(\text{anchor within } w) = 1 - e^{-\mu\omega} \tag{10}$$

$$\mu \geq -\frac{\ln(1-\rho)}{w} \tag{11}$$

For example, $w = 60$ s, $\rho = 0.95 \Rightarrow \mu \approx 1/20\ \mathrm{s}^{-1}$, that is, roughly one anchor every 20 s (the same setting as our running example). This keeps the synchronization SLO explicit rather than heuristic.

We bind the above SLO to the web caching layer by capping CDN/browser TTLs against the verification window as (12), and enforce $TTL_{browser} \leq \min(TTL_{cdn},$ license_expiry $- t_{\text{now}})$ when end-user licenses are time-bounded. Intuitively, a lower μ (slower anchoring) increases the policy–data drift risk, therefore TTL_{cdn} must shrink proportionally to w while event-driven invalidation closes the residual gap.

$$TTL_{cdn} \leq \kappa \cdot w \left(\text{empirically } \kappa \approx \frac{1}{3}\right) \tag{12}$$

Event-driven purge for anchor acceptance. When the upper layer accepts a new anchor, it emits an anchor.updated event and the CDN is purged by surrogate keys to bound inconsistency within *w*: Surrogate-Key: cp:{cp_id} content:{content_id}.

Integrity hints and conditional revalidation. To tie cached objects to the on-chain state we used content-addressed validators consistent with our data model:ETag: sha256(content_id|block_idx|entry_idx) and If-None-Match for conditional GET, plus Cache-Control/Surrogate-Control with the TTL bound. Segment delivery supports range requests for streaming.

3.7 Contract-aware Access and Cross-layer Verification (Web-layer Integration)

HBCMS implements a unified verification workflow between the governance (upper) and content (lower) layers so that access is both policy-compliant and cryptographically verifiable. We complement Figure 12, which instantiates each step as a concrete web interaction: a contract-aware Access API (POST /access/verify) orchestrates the three checks at the upper layer and returns the storage locator and cache hints that drive CDN/edge retrieval. This figure also illustrates an asynchronous cache-purge process triggered by anchor updates, which maintains consistency between browser and edge caches and the anchoring service-level objective (SLO).

3.7.1 Three-step pipeline as a web transaction

A client initiates an access request to the Access API with fields *cp_id, content_id, region*, and current time *t_now*. The request is processed using a deterministic three-step workflow that guarantees both policy compliance and cryptographic verifiability. Access is granted only when all three checks succeed; otherwise the request is denied with an explicit reason.

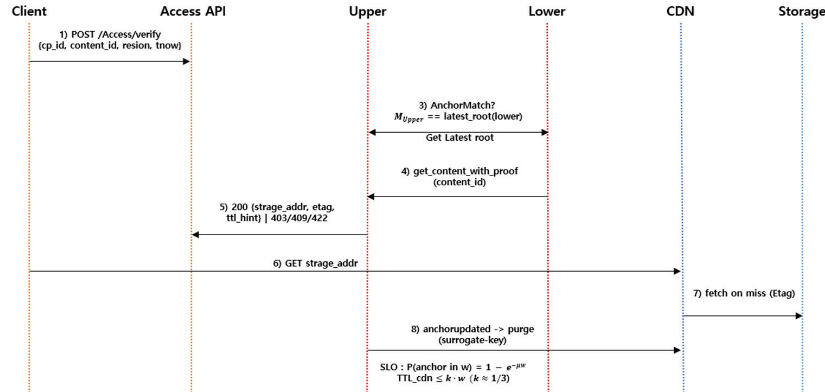

Figure 12 Cross layer searching with web-layer.

Step 1. Contract validity (upper)

The governance layer verifies that the content provider's contract is active and that the request parameters fall within the scope (allow-list, territory, and time). If the contract is invalid or the item is out of the scope, the transaction is rejected.

Step 2. Anchor match (upper↔lower)

The anchored Merkle root recorded by the governance layer M_{upper} is compared with the latest Merkle root obtained from the provider's chain M_{lower}. Processing continues only if $M_{upper} = M_{lower}$; otherwise the request is denied because of an anchor mismatch.

Step 3. Merkle proof verification (upper↔lower)

Upon a successful anchor match, the system fetches the target record and its Merkle proof via *get_content_with_proof()* and verifies the inclusion up to M_{lower}. If the proof is valid and the content is permitted by the contract, the API returns:

{ "storage_addr": "...", "etag": "sha256:...", "ttl_hint": <derived from *w* and μ> }

The client subsequently retrieves the object from the CDN using a GET request to storage_addr. Cache behavior is governed by the ETag and ttl_hint, where ttl_hint is derived from the synchronization objective $P(anchor\ in\ w) = 1 - e^{-\mu w}$. To prevent stale content, an asynchronous purge signal triggered by anchor updates invalidates outdated objects at the edge.

3.7.2 Code-level and HTTP mapping

The verification logic is encapsulated in *UpperChain.cross_chain_get_by_id()* and *UpperChain.cross_chain_search()*, which orchestrate the three checks and expose consistent outcomes through the Access API.

The internal verification and API orchestration procedures are formalized below to ensure reproducibility and alignment with the three-step access pipeline. The overall flow operates as shown in Figures 13 and 14, which illustrate how the verification and HTTP layers interact during request processing.

```
Procedure CROSS_CHAIN_GET_BY_ID(cp_id, content_id, region, t_now, lower, w, κ)

IF NOT CONTRACT_VALID(cp_id, content_id, region, t_now) THEN
    RAISE AccessError(POLICY, "contract expired or request out of scope")
END IF

(M_upper, anchor_ts, _) ← UPPER.ANCHORED_ROOT(cp_id)
IF M_upper = ∅ THEN
    RAISE AccessError(ANCHOR_MISMATCH, "no anchor recorded for CP")
END IF

M_lower_live ← LOWER.LATEST_ROOT()
IF M_upper ≠ M_lower_live THEN
    RAISE AccessError(ANCHOR_MISMATCH,
        "upper anchor and lower latest root differ",
        {upper: M_upper, lower: M_lower_live})
END IF     // AnchorMatch = (M_upper = M_lower_live)

(rec, leaf, M_lower_proof, proof, meta) ← LOWER.GET_CONTENT_WITH_PROOF(content_id)
IF rec = ∅ OR leaf = ∅ OR proof = ∅ THEN
    RAISE AccessError(PROOF_INVALID, "content missing or proof unavailable")
END IF
IF M_lower_proof ≠ M_lower_live THEN
    RAISE AccessError(PROOF_INVALID, "proof built for a different root")
END IF
IF NOT VERIFY_MERKLE_PROOF(leaf, proof, M_lower_proof) THEN
    RAISE AccessError(PROOF_INVALID, "invalid Merkle proof")
END IF     // MerkleProofValid = true

IF NOT ALLOWED_BY_CONTRACT(cp_id, content_id) THEN
    RAISE AccessError(POLICY, "content not allowed by contract")
END IF

etag ← SHA256(content_id ॥ meta.block_idx ॥ meta.entry_idx)
TTL_cdn ← MAX(1, FLOOR(κ × w))      // TTL_cdn ≤ κ·w (SLO-bound)
storage_addr ← EXTRACT_STORAGE_ADDR(rec)
version ← UPPER.CONTRACT_SNAPSHOT_VERSION(cp_id)

RETURN AccessDecision(cp_id, content_id, storage_addr, etag, TTL_cdn, version)
End Procedure
```

Figure 13 Contract-aware cross-chain verification.

```
Procedure ACCESS_VERIFY_HTTP(req)

body ← JSON_PARSE(req)
idem_key ← req.headers["Idempotency-Key"] (nullable)
cached ← IDEMPOTENCY_GET("verify", idem_key)
IF cached ≠ ∅ THEN
    (data, status, headers) ← cached
    RETURN HTTP_RESPONSE(status, JSON(data), headers)
END IF

cp_id ← body["cp_id"]
content_id ← body["content_id"]
region ← body["region"]
t_now ← body["t_now"] OR NOW()

TRY
    dec ← CROSS_CHAIN_GET_BY_ID(cp_id, content_id, region, t_now, LOWER, w, κ)
    data ← { "storage_addr": dec.storage_addr,
             "etag": dec.etag,
             "ttl_hint": dec.ttl_hint }
    status ← 200
    headers ← {}
    IF dec.contract_snapshot_version ≠ ∅ THEN
        headers["X-Contract-Snapshot-Version"] ← STRING(dec.contract_snapshot_version)
    END IF
CATCH AccessError e
    data ← { "error": { "category": e.category, "detail": e.msg } }
    status ← MAP_STATUS(e.category)   // POLICY→403, ANCHOR_MISMATCH→409, PROOF_INVALID→422, NOT_FOUND→404
    headers ← {}
END TRY

IDEMPOTENCY_PUT("verify", idem_key, (data, status, headers), ttl = 300 s)
RETURN HTTP_RESPONSE(status, JSON(data), headers)
End Procedure
```

Figure 14 Access API and idempotency handling.

4 Evaluation

This section presents the results of the performance evaluation of the proposed HBCMS.

Four research questions are formulated and experimentally verified to validate the effectiveness of the proposed method.

The requirements and objectives of each research question are summarized in Table 4.

4.1 Testbed and Implementation

Experiments were conducted on a Python-based prototype of HBCMS, implemented as a Flask REST framework connecting the lower chain PBFT simulator, the upper governance chain, and off-chain object storage.

The governance layer operates in BFT/PoA mode by default to guarantee deterministic finality, and an alternative PoW mode is evaluated separately by adjusting the difficulty D and measuring the resulting block-time distribution. The references are summarized in Table 5.

Table 4 Research questions

RQ	Focus	Research Question
Throughput	Performance gain	How much throughput improvement does the proposed HBCMS achieve compared to the single-chain and non-anchored multi-chain baselines?
Latency	Verification delay	Does the end-to-end verification latency – including contract validation, anchor matching, and Merkle proof verification – satisfy the OTT requirement (<0.3 s)?
Integrity & synchronization	Reliability	Can the proposed system maintain integrity verification accuracy and synchronization reliability (SLO) even as the number of CPs increases?
Scalability	Efficiency & trend	Does the proposed architecture exhibit near-logarithmic scalability, and are efficiency metrics consistently reported as the CP count increases?

Table 5 Evaluation environment

Component	Description
Software stack	Python prototype (Flask REST API), PBFT-based lower-chain simulator, upper-chain governance (BFT/PoA mode, PoW for optional test), off-chain storage module for large-object management.
Hardware & network	Intel i7-12700K (3.6 GHz), 32 GB RAM, Ubuntu 22.04; RTT emulation 5–20 ms.
Consensus configuration	Lower-chain validators = 5–20 (PBFT); upper-chain = BFT deterministic mode for main experiments; PoW difficulty (D) varied in appendix tests.
Repetition & statistics	$\geq$5 runs per measurement; results reported as mean $\pm$ 95% confidence intervals. All scripts, random seeds, and table generators are included in a reproducibility package.

Our evaluation uses a Python/Flask prototype and a PBFT-style simulator to isolate architectural effects. Absolute throughput/latency may shift under production-grade implementations and heterogeneous WAN conditions. Nevertheless, the relative trends are architecture-driven: per-CP lower chains enable horizontal concurrency while the upper layer remains lightweight by anchoring and enforcing contract-aware governance. Implementation optimizations are expected to change constant factors, whereas the relative ordering and scaling trend should remain stable because they follow from system decomposition and the verification pipeline.

4.2 Workloads and Baselines

All models were evaluated under identical transaction generation and network conditions to ensure fairness.

Each transaction includes a content-metadata hash and signature verification, distributed to the respective chain according to the target architecture.

Common symbols such as N, T_{total}, and SLO-related parameters are defined in Table 1; Table 7 lists only evaluation-specific workload and measurement terms.

Table 6 Target architecture for evaluation

Model	Description
Single-chain	All CP transactions aggregated into a single blockchain.
Multi-chain (no-anchor)	Independent CP chains without any cross-layer anchoring or verification.
Proposed (HBCMS)	Per-CP PBFT lower chains + upper governance (BFT). Cross-layer verification is performed using signed Merkle-root anchors and contract-aware validation.

Table 7 Workload and metrics notation

Category	Symbol	Unit	Description
Workload parameters	N_{tx}	tx	Total number of application-level transactions in one run.
	d	ms	Inter-arrival delay between consecutive transactions.
	RTT	ms	Emulated network round-trip time.
Block/run-level sets	$\mathcal{B}$	–	Set of blocks created in a run.
	Entries(b)	tx/block	Number of transactions in block b.
	EndStamp(b)	s	Completion timestamp of block b.
	$L(b)$	s	Block-creation latency of block b.
Throughput curve	$T(N)$	tx/s	Measured throughput as a function of CP count N.
Log-linear fit	α, β	–	Regression parameters of the throughput fitting model.
	R^2	–	Coefficient of determination for the fit.
Correctness	Accuracy	–	Fraction of successful verifications over total requests.
Detection behavior	T_{detect}	s	Detection latency from fault injection to first mismatch observation.
	ν	1/s	Audit-probe rate in the detection model.
	Δ	s	Fixed overhead in the detection model.

4.3 Throughput

4.3.1 Throughput evaluation

Throughput evaluation was conducted to assess the scalability and efficiency of the proposed HBCMS architecture in comparison with the single-chain and multi-chain (no-anchor) baselines.

All the experiments were performed under identical workload and network conditions, using a unified transaction generator that performed metadata hashing and signature verification.

Network latency was emulated at 5–20 ms RTT, and each measurement point was repeated at least five times.

Results are reported as mean $\pm$ 95% confidence interval ($\leq\pm3\%$).

The number of CPs was varied as $N \in \{10, 100, 250, 500, 1000\}$ to evaluate scalability with increasing system load.

The experimental results are summarized in Table 8, and the corresponding trends are shown in Figure 15.

Across all the configurations, the proposed HBCMS consistently achieved the highest throughput.

As shown in Figure 15, the throughput of the HBCMS scales almost linearly on a logarithmic axis, demonstrating a near-logarithmic growth with respect to the number of CPs.

Table 8 Throughput comparison

Model	10 CPs	100 CPs	250 CPs	500 CPs	1000 CPs
Single-chain (baseline)	500	700	757	800	950
Multi-chain (no anchor)	700	1200	1371	1500	1700
Proposed HBCMS	900	1600	1885	2100	2600

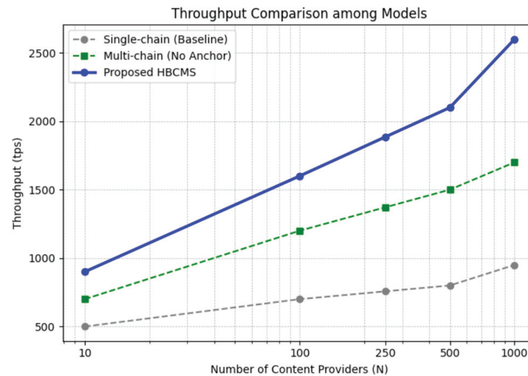

Figure 15 Throughput comparison.

At 1000 CPs, the proposed model achieved a $+172\%$ improvement over the single-chain baseline and $+53\%$ improvement over the multi-chain model. At 250 CPs, it maintained $+149\%$ and $+37.5\%$ gains, respectively.

This consistent trend confirms that the hierarchical anchoring and verification mechanism of HBCMS introduces no performance bottlenecks, even under large-scale operation.

The increment between $N = 500$ and $N = 1000$ appeared steeper than that at earlier intervals. This is an artifact of (i) increased parallelism across independent CP-owned lower chains and (ii) the amortization of fixed per-request overheads in the upper-layer access/verification pipeline. As N grows, more lower-chain instances can be exercised concurrently, so a larger fraction of wall-clock time is spent on the fully parallelizable lower-chain query path rather than on the constant setup and coordination costs. Importantly, the overall trend is still well captured by the single log-linear fit over all five points (Section 4.3.2), and each point is reported with tight confidence intervals ($\leq \pm 3\%$); therefore, local slope variations should be interpreted as implementation-level utilization effects rather than a protocol-level phase change.

4.3.2 Log-linear scalability model

The measured data were fitted to a log-linear empirical model as follows:

$$T_{model}(N) = \alpha_{model} + \beta_{model} \cdot \log_{10} N \tag{13}$$

where $T(N)$ is the average throughput and $\log_{10} N$ represents the logarithmic growth of the participating CPs.

All five data points $(N = 10, 100, 250, 500, 1000)$ were used in the regression for each model.

The parameters α (intercept) and β (slope) were estimated via ordinary least squares (OLS) using the standard closed-form solution for linear regression in (14) [25]:

$$\beta = \frac{\sum_i (x_i - \overline{x})(y_i - \overline{y})}{\sum_i (x_i - \overline{x})^2}, \quad \alpha = \overline{y} - \beta\overline{x} \tag{14}$$

The regression coefficients and coefficients of determination (R^2) are summarized in Table 9. Figure 16 visualizes the same results under (left) linear scaling and (right) the fitted log-linear model. The log-linear fit achieves high explanatory power ($R^2 \geq 0.95$), supporting the claim that throughput growth is approximately logarithmic in N within the tested range. The slope

Table 9 Long linear regression result (OLS, 5 data points)

Model	α (Intercept)	β (Slope)	R^2	Approximation
HBCMS	41.62	799.91	0.971	$T_H(N) \approx 41.62 + 799.91 \log_{10} N$
Multi-chain (no anchor)	210.99	488.05	0.997	$T_M(N) \approx 210.99 + 488.05 \log_{10} N$
Single-chain (baseline)	282.84	206.61	0.958	$T_B(N) \approx 282.84 + 206.61 \log_{10} N$

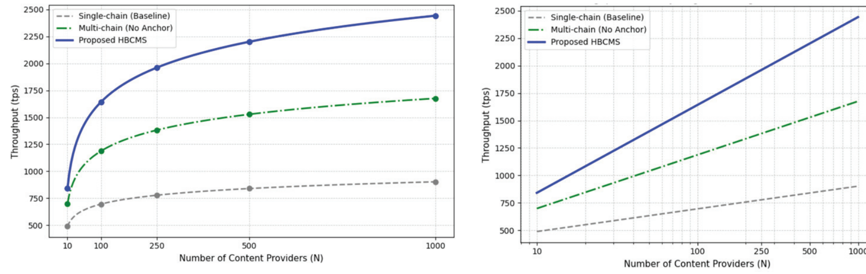

Figure 16 Throughput scalability (linear vs log-linear).

differences further indicate that HBCMS benefits more from increasing the CP counts than the baseline.

The slope β represents the throughput increment per decade increase in CP count. Because $\beta_{HBCMS} > \beta_{Multi\text{-}chain} > \beta_{Single\text{-}chain}$, the proposed HBCMS exhibits the steepest throughput growth rate, confirming its superior scalability.

For example, substituting $N = 250$ gives $T_H(250) \approx 1885$ tps, consistent with the experimental observation.

All regressions yielded $R^2 \geq 0.95$, demonstrating that the log-linear model accurately described the observed quasi-logarithmic throughput scaling across all configurations.

Hence, the HBCMS achieves high efficiency and predictable performance as the system scales, outperforming all baselines in terms of both magnitude and growth rate.

4.4 Latency

4.4.1 Measurement methodology

End-to-end verification latency, denoted as T_{total}, was measured along the full pipeline from request to response, following the timestamp sequence $t_0 \rightarrow t_3$.

Table 10 Definition of latency measurement segments

Segment	Symbol	Measured Scope/Function
Upper query start	t_0	Beginning of upper-layer processing.
Lower query dispatch	$t_1^{\uparrow}$	Start of lower-layer query latency (send phase).
Lower response reception	$t_1^{\downarrow}$	Completion of lower-layer query latency (receive phase).
Verification stage	t_2	Cryptographic verification delay.
Response delivery	t_3	End of total latency measurement.

Each measurement point was recorded in the upper layer, capturing the time consumed during the upper query processing, lower query retrieval, verification, and final response transmission.

The corresponding measurement segments and their functional definitions are presented in Table 10. This table delineates each latency component, including the separate send and receive timestamps ($t_1^{\uparrow}$ and $t_1^{\downarrow}$), which are used to quantify the contribution of each stage to the overall end-to-end delay.

Hence, the latency components are defined as follows:

$$T_{query_upper} = t_1^{\uparrow} - t_0$$

$$T_{query_lower} = t_1^{\downarrow} - t_1^{\uparrow}$$

$$T_{verify} = t_2 - t_1^{\downarrow}$$

$$T_{total} = t_3 - t_0 = T_{query_{upper}} + T_{query_{lower}} + T_{verify} \tag{15}$$

The upper-layer consensus used for this experiment was BFT/PoA, ensuring deterministic confirmation.

4.4.2 Point estimates and variability

Each measurement was repeated at least five times under identical workload and network conditions. The mean and standard deviation for each component are given in Equation (16). Let $\mu_T = \mathrm{E}[T_{total}]$ and $\sigma_T = \sqrt{Var(T_{total})}$ denote the mean and standard deviation of the end-to-end verification latency, respectively (see Table 1).

$$T_{query_{upper}} \approx 25 \pm 4 \text{ ms}, \quad T_{query_{lower}} \approx 150 \pm 18 \text{ ms},$$
$$T_{verify} \approx 10 \pm 2 \text{ ms} \tag{16}$$

The overall mean end-to-end latency was given by Equation (17) with a 95% confidence interval within ± 0.03 s.

$$\mathrm{E}[T_{total}] \approx 0.185 \text{ s} \tag{17}$$

On average, upper query processing accounted for 13.5%, lower query time for 81.1%, and verification for 5.4%, confirming that the lower-layer query dominated the total latency budget.

The lower-query time includes (i) network transfer/RTT between the upper and the target CP chain, (ii) CP-side lookup using the content index (content_id $\rightarrow$ (block_idx, entry_idx)), and (iii) Merkle-proof assembly/serialization for the requested leaf. The request payload is constant-size (cp_id, content_id, region, t_now), whereas the response includes the storage locator plus a Merkle inclusion proof whose size grows slowly with block size as O(log *Entries*(*b*)).

A single verification request queries one CP chain, so per-request latency is dominated by that CP's lookup path. At system level, throughput scales because CP chains are independent and can be served concurrently across requests. In production, CP-side indexing and web caching (ETag validation SLO-bounded TTL/purge coupling) can reduce the lower-chain contribution for hot items; RTT, proof generation time, and cache hit ratio become the primary latency levers for OTT response predictability.

Using the standard identity $\mathrm{Var}(X+Y+Z) = \mathrm{Var}(X) + \mathrm{Var}(Y) + \mathrm{Var}(Z) + 2\mathrm{Cov}(\cdot)$, we approximate the variance decomposition in (18) by neglecting the cross-component covariance under controlled measurements [26].

$$\mathrm{Var}(T_{total}) \approx \mathrm{Var}(T_{query_{upper}}) + Var(T_{query_{lower}}) + \mathrm{Var}(T_{verify}) \tag{18}$$

Substituting the standard deviations in (16) gives

$$\sigma_T \approx \sqrt{4^2 + 18^2 + 2^2} \approx 18.55 \text{ ms} \tag{19}$$

Using a normal approximation, the *p*th percentiles are derived as in (20) using the standard-normal quantiles [27]:

$$P_{95} \approx \mu_T + 1.645\sigma_T \approx 215.6 \text{ ms}, \quad P_{99} \approx \mu_T + 2.326\sigma_T \approx 228.2 \text{ ms} \tag{20}$$

These results show that the system satisfied the OTT latency requirement (<0.3 s) comfortably.

Given that the lower-chain lookup accounts for $\sim 81\%$ of T_{total}, the μ–w–TTL policy in Equation (12) is expected to increase edge/browser reuse and reduce the perceived latency.

4.4.3 Analytical sensitivity

The lower-query component can be approximated as a function of the network RTT and number of PBFT validators (21):

$$T_{query_{lower}} \alpha + \beta \cdot \mathrm{RTT} + \gamma \cdot f(n) \tag{21}$$

$$f(n) \in O(n^2)(PBFT\ message\ complexity) \tag{22}$$

Empirical analysis shows $\beta \approx 1{-}2$, and $f(n)$ exhibits a weak near-linear growth when $n = 20$, indicating that the overall T_{total} is primarily sensitive to RTT rather than validator count.

To reflect WAN conditions beyond our emulation range, we apply the same sensitivity interpretation to RTT = 20–50 ms. With $\beta \approx 1{-}2$, increasing RTT from 20 ms to 50 ms ($\Delta\mathrm{RTT} = 30$ ms) increases the lower-query component by about 30 to 60 ms, thus increasing T_{total} by the same order. Using $E[T_{total}] \approx 0.185$ s as a reference, this yields an illustrative range of ~0.215–0.245 s at $\mathrm{RTT} \approx 50$ ms, which remains within the 0.3 s OTT budget while highlighting RTT as the dominant operational lever.

4.4.4 SLO-style latency assurance

Operational SLOs are defined probabilistically as:

$$P_{\mathrm{r}}[T_{total} = \tau] \geq \rho \tag{23}$$

Under the normal approximation, this relation becomes

$$\tau = \mu_T + z_p \sigma_T \Longleftrightarrow \rho \leq \Phi\left(\frac{\tau - \mu_T}{\sigma_T}\right) \tag{24}$$

where $\Phi(\cdot)$ denotes the standard normal cumulative distribution.

Substituting $\tau = 0.3$ s and the values from (19), $\rho \approx 0.999$, indicates that HBCMS meets the latency SLO for OTT requirements with a significant safety margin.

4.5 Integrity Verification and Detection

4.5.1 Definitions and metrics

For each verification request, integrity is defined as (25).

$$V_{integrity} = \begin{cases} 1, & \text{if } M_{upper} = M_{lower} \land MerkleProofValid \\ 0, & otherwise \end{cases} \tag{25}$$

The overall verification accuracy is computed as (26), while the detection time is the elapsed interval between the moment a lower-layer fault is injected (root modification, replay, or forged signature) and the instant when the upper layer first observes an *AnchorMismatch* or *ProofInvalid* event.

$$\text{Accuracy} = \frac{\#successful\ verifications}{\#total\ verifications} \tag{26}$$

4.5.2 Measurement protocol

All experiments were conducted under identical workload and network conditions (RTT $=$ 5–20 ms), using upper-layer BFT/PoA consensus and lower-layer PBFT finalization.

Each experimental configuration was repeated at least five times, with no fewer than 100 fault-injection events per run. Three fault types, equally weighted in the evaluation, were considered: (i) *root tampering*, which modifies the fingerprint after block finalization; (ii) *replay*, which reuses an outdated (root,ts) pair, thereby violating monotonicity; and (iii) *signature forgery*, which introduces an invalid HMAC or key pair.

Detection latency $T_{det} = t_{obs} - t_{inj}$ was measured between the injection timestamp t_{inj} and the first observation t_{obs}.

Periodic audit probes were scheduled every 0.5 s with random jitter U(0, 0.1 s) to minimize request-load bias.

The sample's 95% confidence interval of each sample was estimated using bootstrap resampling (1000 iterations) following the standard procedure in [28].

4.5.3 Analytical model

Let ν be the audit-probe rate, μ the anchor-arrival rate, and Δ the fixed overhead of the verification pipeline.

Assuming exponential arrival processes, the detection delay is as follows [24]:

$$T_{detect} = \Delta + W, \quad W \sim Exp(v + \mu), \tag{27}$$

Table 11 Integrity accuracy and detection time

# CPs	Accuracy (%)	Detection Time(s)
10	99.9 ± 0.1	0.62 ± 0.06
100	99.8 ± 0.2	0.87 ± 0.09
250	99.5 ± 0.3	1.10 ± 0.11
500	99.3 ± 0.3	1.34 ± 0.12
1000	99.1 ± 0.4	1.88 ± 0.15

with expected value and cumulative distribution,

$$\mathrm{E}[T_{detect}] = \Delta + \frac{1}{v+\mu}, \quad P_r[T_{detect}] = 1 - e^{-(v+\mu)(w-\Delta)}, \quad w \geq \Delta \tag{28}$$

given a reliability ρ within deadline w, the required combined rate is

$$v + \mu = \frac{-\ln(1-\rho)}{w - \Delta}, \quad (w > \Delta) \tag{29}$$

From the empirical averages in Table 11, the estimated effective rates were $\hat{v} + \widehat{\mu} \approx \{2.3, 1.5, 1.1, 0.8, 0.6\}$ s^{-1} for $N = 10$, 100, 250, 500, 1000, indicating that larger CP counts reduce the observable audit frequency because of scheduling overheads.

4.5.4 Result and discussion

With an average integrity-verification success rate of ∼99.4%, the system detected mismatches or forged anchors within =2 s even at the largest scale. Across all evaluated system sizes, the proposed three-stage verification pipeline – (i) contract validation, (ii) anchor matching, and (iii) Merkle-proof verification – exhibited stable and reliable behavior. Moreover, the detection latency increases with *N* and closely follows Equation (28), suggesting that the dominant contributor to end-to-end delay is the periodic audit schedule rather than verification computation.

5 Conclusion

This paper presented HBCMS, a web-native hierarchical blockchain-based content management system for OTT delivery that decouples contract-aware governance from high-rate metadata finalization. By placing governance and policy enforcement at an upper layer while allowing each content provider

to finalize operational metadata on its own lower chain, HBCMS aims to combine auditable integrity guarantees with practical scalability. The architecture further couples cross-layer Merkle-root anchoring with web-facing consistency mechanisms so that the verification results can be aligned with HTTP-level behaviors in real deployments.

The evaluation indicates that this hierarchical separation improves scalability compared to monolithic or single-chain approaches, while preserving verifiable access decisions. Across a wide range of content-provider scales, the system maintains a high throughput and stable end-to-end verification latency that remains compatible with common OTT delivery constraints. In addition, integrity experiments demonstrate that the verification path can reliably detect tampering and inconsistency, supporting practical auditing workflows rather than relying on manual, post hoc investigation.

A key design takeaway is that end-to-end verifiability can be treated as a pipeline that maps (i) contract validity, (ii) cross-layer anchor consistency, and (iii) cryptographic proof verification into deterministic allow/denial outcomes. Modeling anchoring as a probabilistic synchronization process makes the verification target explicit and tunable, and operators can select the anchoring cadence and cache lifetimes to meet a desired service-level objective for observing fresh anchors, rather than treating consistency as an implicit side effect. Where needed, anchoring aggregation provides an additional knob to reduce the upper-layer posting pressure while retaining per-provider verifiability, enabling the system to adapt to heterogeneous provider scales and operational budgets.

This study had some limitations. The current prototype and its experimental setup primarily isolate architectural effects; validating the same behaviors under production-grade stacks, heterogeneous WAN conditions, and stronger adversarial assumptions. Consensus configurations that were not the primary focus of the evaluation, such as open-participation settings and their confirmation and reorganization dynamics, require a deeper empirical study. Finally, privacy and incentive aspects, including confidentiality of the policy/contract scope and economic mechanisms that discourage stale or inconsistent anchoring, are not addressed in the present prototype.

While full privacy-preserving verification is left as future work, deployments can reduce exposure by minimizing on-chain contract detail (e.g., storing only enforcement scope predicates or cryptographic commitments) and keeping commercial terms encrypted off-chain referenced by identifiers. This reduces leakage risk while preserving auditability until privacy-preserving proofs are integrated.

Future work will therefore focus on hardening the system for deployment and expanding its guarantees by migrating the implementation to production blockchain frameworks, conducting geo-distributed trials, characterizing the operational impact of anchoring aggregation under different fan-out choices, strengthening governance with transparency and accountability mechanisms, and exploring privacy-preserving verification methods. Extending the service-level objective model to incorporate bursty workloads and correlated failures is also an important direction, enabling anchoring and audit policies to be co-optimized with cost, availability, and user-perceived performance.

Acknowledgement

This research project supported by Ministry of Culture, Sport and Tourism R&D Program through the Korea Creative Content Agency grant funded by the Ministry of Culture, Sport and Tourism in 2025(Project Name: Development of Copyright Technology(+Law) for OTT Contents Copyright Protection Technology Development and Application, Project Number: RS-RS-2023-00225267, Contribution Rate: 100%).

References

[1] PwC, Global Entertainment and Media Outlook 2025–2029, PwC, Jul. 24, 2025.

[2] IFPI, Global Music Report 2022: State of the Industry, Int. Fed. of the Phonographic Industry, London, U.K., 2022.

[3] W. Son, S. Kwon, S. Oh, and J.-H. Lee, "OTT service copyright management framework using ODRL within Hyperledger Fabric," Electronics, vol. 13, no. 2, p. 336, Jan. 2024.

[4] S. Gupta, A. Kumar, and R. Jain, "A blockchain-based platform architecture for multimedia data management," Multimedia Tools and Applications, vol. 79, nos. 15–16, pp. 10471–10493, 2020.

[5] Omdia, "SVOD subscriptions worldwide (forecast: 1,356 million by end-2025)," DEG Summary Page, Mar. 20, 2025.

[6] Ampere Analysis, "Almost 40% of titles in the U.S. VoD market are now available non-exclusively; Prime Video exclusivity share 41% → 24% (Aug. 2023–Jul. 2025)," Ampere Insights, Sep. 2025. [Online]. Available: https://www.ampereanalysis.com.

[7] Parrot Analytics, “How licensed content is Netflix’s secret weapon for low churn (licensed TV $\approx$ half of catalog demand, Sept. 2024),” Parrot Analytics Insight, Nov. 10, 2024. [Online]. Available: https://www.parrotanalytics.com.

[8] R. W. Kroon, “Media and entertainment metadata governance,” Journal of Digital Media Management, vol. 12, no. 2, 2024. [Also: EIDR Use-Cases, Entertainment ID Registry].

[9] W3C, “ODRL 2.2: Information Model and Vocabulary,” W3C Recommendation, Feb. 15, 2018. [Online]. Available: https://www.w3.org/TR/odrl-model/.

[10] J. E. Abang et al., “Latency performance modelling in Hyperledger Fabric,” Internet of Things and Cyber-Physical Systems, 2024.

[11] IFPI, “Global recorded music revenues grew 4.8% in 2024; streaming exceeded 69% of total; paid streaming 51.2%,” IFPI Press Release, Mar. 19, 2025.

[12] Y. Jung, Y. Kim, and Y. Oh, Key Results of the Korea Media Panel Survey 2023, KISDI Report 23-23, Korea Information Society Development Institute (KISDI), Seoul, Republic of Korea, 2023.

[13] Q. Chen, Z. Zhang, and H. Li, “A scalable multi-chain framework for cross-chain interoperability,” IEEE Access, vol. 12, pp. 24187–24201, Feb. 2024.

[14] D. Liu, M. Fang, and L. Zhou, “Sharding-enhanced consensus mechanisms for decentralized networks,” IEEE Transactions on Parallel and Distributed Systems, vol. 36, no. 2, pp. 335–347, Feb. 2025.

[15] J. Park and T. Nguyen, “Cross-chain governance: A review of interoperability and trust management,” ACM Distributed Ledger Technologies: Research and Practice, vol. 5, no. 1, pp. 1–18, Jan. 2025.

[16] M. Nottingham and R. Fielding, HTTP Caching (RFC 9111), Internet Engineering Task Force (IETF), Jun. 2022. [Online]. Available: https://www.rfc-editor.org/rfc/rfc9111.

[17] A. S. Pathan and R. Buyya, “A taxonomy and survey of content delivery networks,” International Journal of Network Management, vol. 15, no. 5, pp. 283–297, Sep.–Oct. 2005.

[18] J. Laoutaris, H. Che, and C. Diot, “Inter-datacenter caching: Exploiting storage diversity in CDN architectures,” IEEE/ACM Transactions on Networking, vol. 23, no. 4, pp. 689–703, Aug. 2015.

[19] D. Huang and L. Xu, “Dynamic cache consistency for licensed OTT content,” IEEE Access, vol. 12, pp. 114560–114572, 2024.

[20] R. T. Fielding, Architectural Styles and the Design of Network-based Software Architectures, Ph.D. dissertation, Univ. of California, Irvine, USA, 2000.

[21] Google SRE Team, The Site Reliability Workbook: Practical Ways to Implement SRE, O'Reilly Media, Sebastopol, CA, USA, 2018.

[22] A. Tosic and D. Dragomirova, "Web observability metrics and their effect on perceived performance," Journal of Web Engineering, vol. 21, no. 3, pp. 445–468, 2023.

[23] W3C, "Encrypted Media Extensions (EME) and Media Source Extensions (MSE)," W3C Recommendation, Jan. 2023. [Online]. Available: https://www.w3.org/TR/encrypted-media/.

[24] J. F. C. Kingman, *Poisson Processes*. Oxford University Press, 1993.

[25] D. C. Montgomery, E. A. Peck, and G. G. Vining, *Introduction to Linear Regression Analysis*, 5th ed. Wiley, 2012.

[26] A. Papoulis and S. U. Pillai, *Probability, Random Variables, and Stochastic Processes*, 4th ed. McGraw-Hill, 2002.

[27] G. Casella and R. L. Berger, *Statistical Inference*, 2nd ed. Duxbury, 2002.

[28] B. Efron and R. J. Tibshirani, *An Introduction to the Bootstrap*. Chapman & Hall/CRC, 1993.

Biographies

Suhwan Bae received his bachelor's degree in information and communication engineering from Dongyang Mirae University in 2016 and his master's degree in convergence software from Soongsil University in 2018. He is currently pursuing his Ph.D. degree in computer science and engineering at Soongsil University. He served as a Lecturer at Korea Polytechnic University, Induk University, and Soongsil University, where he taught courses

in computer science and information and communication technologies. His research interests include blockchain, information and communication systems, computer networks, artificial intelligence, and network security.

Jin-Sook Bong received her M.Sc. degree in computer science and engineering from Soongsil University, Republic of Korea, in February 2005. She earned her Ph.D. in the same field from the same university in August 2019. From September 2019 to August 2023, she was a Lecturer at Hyupsung University. Between March 2021 and August 2023, she also served as an Invited Professor at the Spartan Software Education Institute, Soongsil University. Since September 2023, she has been serving as an Assistant Professor at the Baird College of General Education, Soongsil University. Her research interests include blockchain, computer networks, AI education, and general education.

Uijin Jang received her Ph.D. in computer science and engineering from Soongsil University, Republic of Korea, in 2010. Since 2018, she has been working at the Spartan SW Education Center at Soongsil University, Republic of Korea. Her research interests include networks, digital forensics, and DRM.

Yongtae Shin received his Ph.D. in computer science from the University of Iowa in 1994. Since 1995, he has been serving as a Professor in the School of Computer Science and Engineering at Soongsil University, Republic of Korea. His research interests include computer networks, distributed computing, Internet protocols, and e-commerce technology.