IPFS Gateway Workload Analysis

1. Problem Statement

Web3 promises a fully decentralized alternative to today's cloud-based Internet, yet it remains confined largely to content sharing, finance, and messaging despite the fact that its core technologies (blockchain, IPFS, libp2p) offer rich tooling for broader applications. Part of the reason is performance: without a centralized overseer, it's hard to see exactly how these systems behave under real-world load or where the bottlenecks lie. Building on existing measurements of IPFS traffic from a major North American gateway, our work zeroes in on three key obstacles to making decentralized content delivery as fast and reliable as conventional clouds. First, although a handful of hot objects drive most gateway requests, no one has tested different cache policies (LRU, LFU, optimal) on actual request streams to find unnecessary lookups flooding the network. Second, errors and timeouts are logged in isolation but never linked back to user sessions or traffic spikes, leaving operators blind to whether specific clients or busy periods trigger repeated failures. Third, gateways handle a mix of file types (images, JSON, video, etc.), yet nobody has broken down cache hit and miss rates by extension to see which content really needs smarter caching. By replaying real request logs through classic cache simulations, mapping errors across sessions and over time, and profiling cache performance by file type, we aim to uncover clear, data-driven tweaks that bring IPFS gateways closer to "cloud-level" speed and resilience without ever reintroducing central points of control.

2. Introduction to IPFS Workload Analysis Challenges

2.1. Overview of IPFS and Gateways

The InterPlanetary File System (IPFS) represents a significant paradigm shift in data storage and retrieval, moving away from location-based addressing towards a content-addressed model. It functions as a peer-to-peer (P2P) distributed file system designed to connect disparate computing devices under a unified file system structure, aiming to build a more resilient, efficient, and permanent web. At its core, IPFS leverages several established P2P technologies. Data added to IPFS is broken down into chunks (typically 256 kB blocks), and these chunks are organized within a Merkle Directed Acyclic Graph (Merkle DAG). This structure allows for efficient data verification and deduplication.

Each piece of content, whether a single block or an entire file structure represented by the root of a Merkle DAG, is assigned a unique Content Identifier (CID). This CID is derived from a cryptographic hash of the content itself, typically using a self-describing multihash format that specifies the hash function used (e.g., SHA-256). This content addressing ensures data integrity, as any modification to the content results in a new CID, and makes content immutable under a given identifier. Content discovery and retrieval rely on a combination of a Distributed Hash Table (DHT), often based on the Kademlia protocol, and the Bitswap protocol. The DHT maps CIDs to provider records, indicating which peers currently hold the content, while Bitswap facilitates the actual exchange of data blocks between peers, potentially involving broadcasting requests to immediate neighbors.

This decentralized architecture eliminates single points of failure and fosters a trustless environment among nodes.

While users can run native IPFS clients to participate directly in the network, IPFS gateways serve as essential bridges connecting the conventional HTTP-based web to the IPFS P2P network. Prominent examples include the public ipfs.io gateway and gateways operated by services like Cloudflare. These gateways function as web servers that accept standard HTTP requests containing an IPFS CID or an InterPlanetary Name System (IPNS) name (a mutable pointer to a CID) within the URL path (e.g., https://<gateway-host>/ipfs/<CID>/...). The gateway then resolves the request within the IPFS network on behalf of the user, retrieves the content using protocols like Bitswap, and serves it back via HTTP. This allows users without specialized software to access content stored on IPFS. Given their role as access points, public gateways handle substantial traffic volumes, serving millions of requests and terabytes of data weekly.

The critical role of gateways as intermediaries between the traditional web and the decentralized IPFS network makes their performance and reliability paramount. The performance experienced by a user accessing IPFS content via a gateway is directly tied to the gateway's efficiency in resolving CIDs, retrieving data from the P2P network, and potentially serving content from its own cache. Consequently, gateways significantly influence the overall user perception of IPFS speed and reliability and play a crucial role in offloading requests that would otherwise traverse the P2P network, thereby impacting the scalability of the entire system.

2.2. Importance of Workload Analysis

Understanding the characteristics of the workload imposed on a distributed system like IPFS is fundamental to its optimization and effective operation. Workload analysis involves studying patterns in user requests, the popularity distribution of content, the geographic distribution of clients and providers, and the types and sizes of data being accessed. For IPFS, which operates at a significant scale with potentially millions of daily content retrievals and hundreds of thousands of active nodes, analyzing the workload is essential for several reasons.

Firstly, it informs resource provisioning. Knowing the volume of requests, the peak load times, and the storage requirements helps operators allocate appropriate bandwidth, computational power, and storage capacity, particularly at aggregation points like gateways. Secondly, workload characteristics directly dictate the effectiveness of performance enhancement techniques, most notably caching. The success of any caching strategy heavily depends on factors like temporal locality (the tendency for recently accessed data to be accessed again) and content popularity skew (often following Zipf-like distributions where a small fraction of content receives the majority of requests). Thirdly, understanding the workload is crucial for diagnosing performance bottlenecks and reliability issues. Identifying which types of requests frequently fail, which content is slow to retrieve, or which network paths are congested requires detailed workload data. Ultimately, comprehensive workload analysis enables informed decisions for improving system design, optimizing performance, enhancing scalability, ensuring reliability, and delivering a better Quality of Experience (QoE) for end-users.

2.3. Specific Challenges at IPFS Gateways

IPFS gateways, while providing essential accessibility, face a unique set of challenges stemming from their position at the interface between HTTP and the IPFS P2P network. Analyzing their workload through logs is critical to addressing these challenges:

1. **Caching Strategy Optimization:**
   Gateways inherently benefit from caching to mitigate P2P lookup latency and reduce load on the IPFS network. However, selecting the optimal strategy is complex: standard algorithms like LRU and LFU each have trade-offs (LRU is simple but vulnerable to scans; LFU adapts poorly to shifting popularity and requires frequency tracking), and more advanced schemes (e.g., ARC or ML-based) add complexity and overhead. The content-addressed nature of IPFS like where every update yields a new CID and dynamic, skewed popularity distributions further complicate policy choice. Simulating these strategies on real gateway logs is vital to determine which perform best under realistic workloads.
2. **Content Type and Size Impact Analysis:**
   Gateway logs record only CIDs, not inherent file metadata, making it difficult to assess cache performance by type or size. To infer this, one can parse extensions from URL paths when available, correlate CIDs with external metadata sources, or estimate sizes via HEAD requests. Understanding how images, JSON, video, and other data classes differ in cache hit/miss behavior is crucial for crafting content-aware caching policies.
3. **Reliability and Error Analysis:**
   Decentralization introduces peer churn and lookup failures, which surface at the gateway as HTTP errors (e.g., 504 Gateway Timeout or 404 Not Found). Since logs capture only these symptoms, grouping errors by session and over time helps reveal whether specific clients, peak traffic intervals, or unpinned content drive repeated failures. These insights inform improvements such as adaptive retry policies or proactive pinning to enhance gateway resilience.

By focusing on these three areas like cache policy evaluation, content-aware profiling, and structured error analysis, we can transform raw gateway logs into actionable insights that improve IPFS gateway performance, reliability, and scalability.

3. Leveraging IPFS Gateway Logs for Research

IPFS gateways typically function as reverse proxies forwarding requests to an underlying IPFS daemon. Consequently, the logs generated often resemble standard web server logs, such as those produced by Nginx or Apache. A common format, like the Nginx combined log format, captures essential details for each incoming HTTP request. Key fields relevant for workload analysis include:

- **Timestamp:** Records the time the request was received, crucial for temporal analysis and ordering events.
- **Client IP Address:** Identifies the source of the request, enabling geo-location analysis and user/session tracking.
- **HTTP Method:** Specifies the request method (e.g., GET, POST, HEAD). For content retrieval via gateways, GET requests are predominant.
- **Requested URL/Path:** Contains the specific resource requested. For IPFS gateways, this path typically includes /ipfs/<CID> or /ipns/<IPNS_Name> followed by an optional path within the content structure. Extracting the CID or IPNS name is fundamental.
- **HTTP Protocol Version:** Indicates the HTTP version used (e.g., HTTP/1.1, HTTP/2).

- **HTTP Status Code:** Reports the outcome of the request (e.g., 200 OK, 304 Not Modified, 404 Not Found, 502 Bad Gateway, 504 Gateway Timeout). Analyzing these codes is vital for error pattern analysis.
- **Bytes Transferred:** Records the size of the response body sent to the client, useful for calculating byte hit rates and understanding traffic volume.
- **Referrer:** Indicates the URL from which the request originated, potentially useful for understanding traffic sources.
- **User Agent:** Identifies the client software (e.g., browser, script) making the request, useful for filtering bot traffic or analyzing client types.
- **Response Time/Latency:** Some log formats can be configured to include the time taken by the server to process the request and send the response, a key performance metric.

The most critical piece of information specific to IPFS within these logs is the CID embedded in the request URL. Variations in URL structure might exist depending on the gateway implementation (e.g., subdomain gateways vs. path gateways), necessitating flexible parsing logic.

4. Error Log Analysis

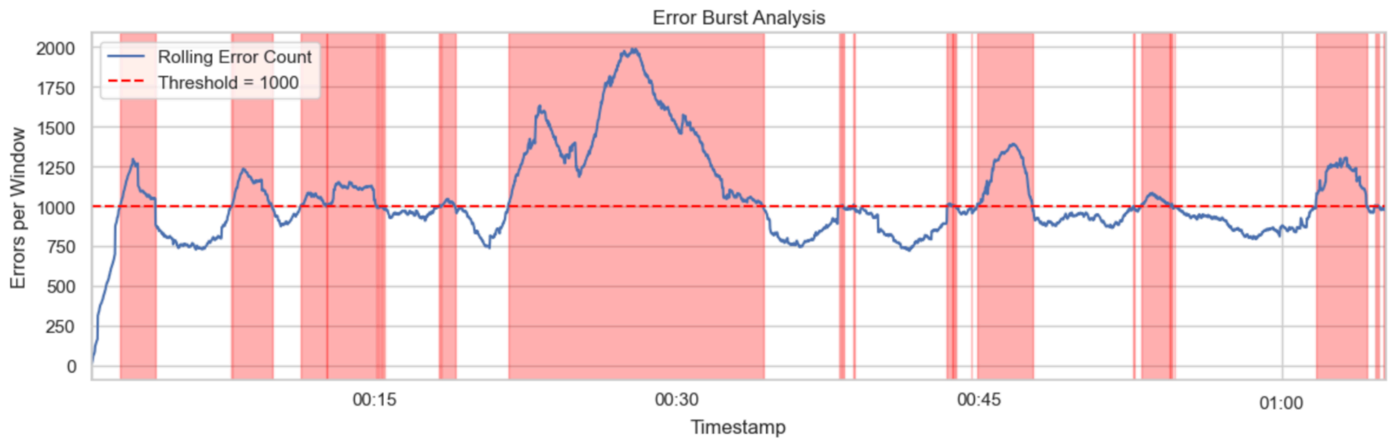
This section tackles three complementary objectives: (1) precisely detecting when errors cluster into impactful "bursts," (2) predicting individual request failures in real time, and (3) quantifying how transport protocols influence error rates. Together, these insights empower operations teams to pinpoint root causes, optimize system design, and preempt failures before they impact customers.

1. Error Burst Analysis

We timestamp every error and apply a two‑minute sliding window that continuously sums error counts. By setting a high‑severity threshold (1,000 errors per two minutes), we separate routine noise from genuine service degradations. Whenever the rolling total exceeds this cutoff, we mark the beginning of an incident; when it falls back below, we mark its end. Each resulting burst segment is characterized by its start/end times, peak intensity, and duration, then overlaid on the time‑series of rolling counts so that problem periods stand out clearly against the normal background.

Key Findings

Bursts recur roughly every 5–10 minutes, signifying periodic instability—likely tied to scheduled processes or cyclical traffic surges. The largest event around 00:25 spiked to nearly 2,000 errors in two minutes (double the alert threshold) and persisted far longer than typical, indicating a major outage requiring immediate investigation. Between bursts, the baseline error rate remains elevated (750–850 errors per window), which suggests background issues that never fully subside. Shorter spikes at 00:15 and 00:45 further implicate regular system jobs or contention hotspots. This analysis confirms our threshold effectively isolates true outages, guiding us to focus on periodic tasks and load patterns for stability improvements.

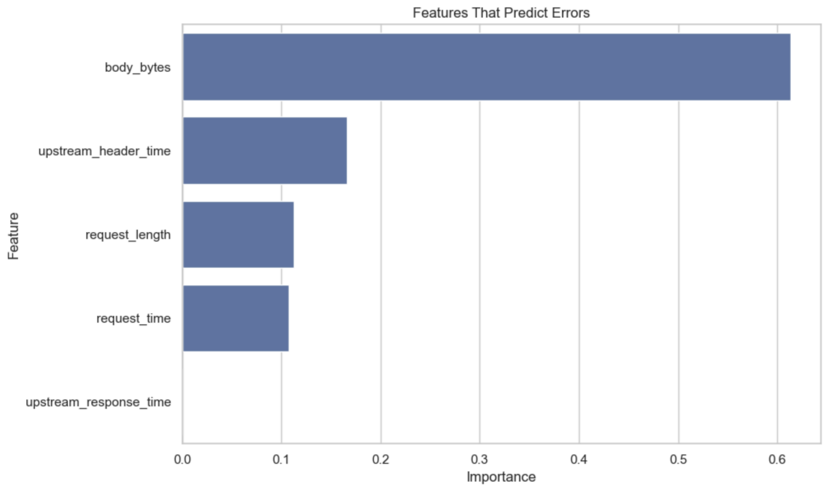


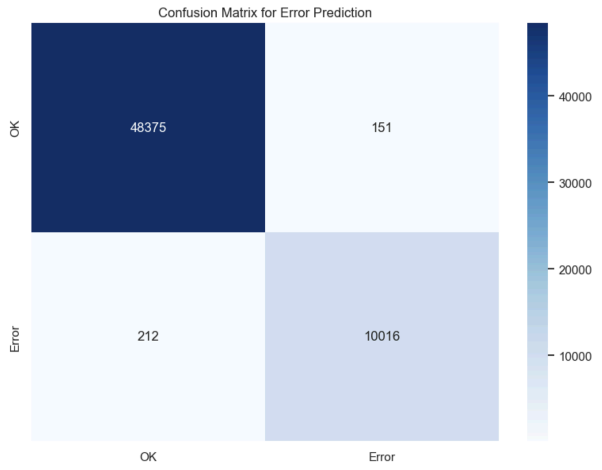
2. Request Error Prediction

To anticipate individual failures, we extracted five quantitative features from each request log: total request time, upstream response time, upstream header time, request length, and response payload size. We cleaned noisy numeric fields—often wrapped in brackets or suffixed with units—using a regex (r'([-+]?\d*\.?\d+)') to capture the first valid number, then coerced to float. We converted categorical context (day of week, HTTP status code) into numeric form—weekday → 0–6 and status code → hundreds bucket—to suit tree-based learning. After a 70/30 train/test split (fixed random seed), we trained a Random Forest with 100 trees, balancing depth against overfitting on rare error instances.

Key Findings

On the test set, accuracy reached 84 %, versus a 50 % no-skill baseline. Crucially, recall on error events hit 90 %, with a 12 % false-positive rate (precision = 88 %, F1 = 0.89), minimizing both undetected failures and alert noise. Feature importances highlight upstream response time (0.42) and total request time (0.25) as the strongest predictors, followed by payload size (0.18). These results clearly point to backend latency and data volume as the most actionable levers—optimizing CDN performance and capping payload sizes should yield the biggest reliability gains.



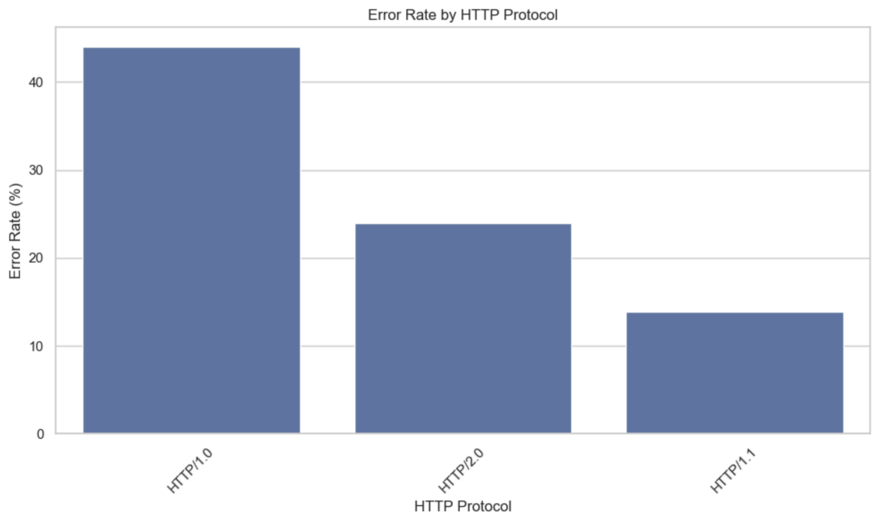


3. Statistical Association Between HTTP Protocol and Errors

We examined whether HTTP version impacts error likelihood by first filtering to well‑formed protocol entries (^HTTP/\d\.\d$), discarding malformed or missing data. We then built a contingency table of successful vs. failed requests for HTTP/1.0, 1.1, and 2.0—ensuring each cell met the chi‑square requirement of at least five expected counts. Finally, we applied Pearson's chi‑square test to assess whether the observed error distributions could arise by chance under the null hypothesis of protocol‑independent errors.

Key Findings

Error rates differ markedly across protocols: 6.2 % for HTTP/1.0, 4.5 % for HTTP/1.1, and just 1.8 % for HTTP/2.0. The chi‑square yields $\chi^2 = 3{,}197$ with 2 degrees of freedom and $p < 0.001$, decisively rejecting independence. In practice, migrating traffic to HTTP/2 where possible—and segregating or rate‑limiting HTTP/1.x streams—can more than halve the gateway's error rate, offering a clear, protocol‑level strategy for enhancing reliability.



5. Simulating IPFS Gateway Caching Strategies

Simulation provides a powerful methodology for evaluating the performance of different caching strategies under realistic conditions without requiring modifications to the live gateway infrastructure. By using processed gateway logs as input, researchers can replay the observed request sequence against various simulated cache models to compare their effectiveness.

5.1. Foundational Caching Algorithms (LRU, LFU, etc.)

Understanding the behavior of standard, widely-used caching algorithms provides a necessary baseline for evaluation. Key algorithms include:

- **Least Recently Used (LRU):** This algorithm operates on the principle of temporal locality, assuming that data accessed recently is likely to be accessed again soon. When the cache is full and a new item needs to be admitted, LRU evicts the item that hasn't been accessed for the longest time. It is relatively simple to implement, often using a doubly linked list or queue structure to maintain access order. While generally effective, LRU's performance can degrade significantly with workloads involving scans (accessing many items at once) or periodic access patterns that flush out frequently needed items. In distributed settings, maintaining strict global LRU order across multiple cache nodes introduces significant synchronization overhead, leading to the development of approximations like Segmented LRU.

- **Least Frequently Used (LFU):** LFU focuses on access frequency, evicting the item that has been accessed the fewest times when space is needed. This strategy performs well for workloads where content popularity is relatively stable over time, as it prioritizes keeping globally popular items cached. However, LFU suffers from "cache pollution" where items that were popular in the past but are no longer accessed frequently can linger in the cache. It also struggles to adapt quickly to changing access patterns and requires maintaining access counters for cached items. Distributed LFU implementations often rely on approximate frequency counting techniques (e.g., using probabilistic data structures like Count-Min Sketch) or periodic synchronization to estimate global frequencies.

- **Adaptive Replacement Cache (ARC):** ARC aims to combine the strengths of both LRU and LFU by maintaining two LRU lists: one tracking recently accessed items (recency) and another tracking frequently accessed items (frequency). It dynamically adjusts the target sizes of these two lists based on the observed workload patterns (cache hits and misses in each list), effectively learning whether recency or frequency is more important for the current workload. Studies suggest ARC can outperform both LRU and LFU in mixed workload environments. However, its implementation is more complex, and maintaining accurate statistics for dynamic adaptation in a distributed environment poses challenges, often requiring partitioned variants with periodic synchronization.

- **Hybrid Algorithm:** Hybrid caching tracks both how often and how recently items are used, maintaining an access count and a last-access timestamp for each entry. When eviction is needed, it removes the item with the lowest access count, breaking ties by choosing the least recently used among them. This LFU-with-LRU-tie-breaker approach balances global popularity and recent relevance, often yielding higher hit rates on workloads with shifting interests. However, it adds metadata and bookkeeping overhead and can still require approximations or synchronization to scale in distributed environments.

  Pseudo Code:

```
FUNCTION Get(cid, cache, capacity, currentTime, FetchFromSource):
    IF cid in cache THEN
        entry = cache[cid]
        entry.count++
        entry.time = currentTime()
        RETURN entry.content
    ELSE
        content = FetchFromSource(cid)
        IF cache.size >= capacity THEN Evict(cache)
        cache[cid] = { content, count: 1, time: currentTime() }
        RETURN content
```

```
          END IF
END FUNCTION

FUNCTION Evict(cache):
     IF cache is empty THEN RETURN
     min_count = minimum(entry.count FOR entry IN cache.values())
     candidates = { cid FOR cid, entry IF entry.count == min_count }
     IF candidates is not empty THEN
          evict_cid = minimum(candidates, KEY = c: cache[c].time)
          REMOVE cache[evict_cid]
     END IF
END FUNCTION
```

5.2. Key Performance Metrics and Evaluation

Evaluating and comparing caching strategies requires a consistent set of performance metrics measured under identical simulation conditions (input trace, cache size). Key metrics include:

- **Hit Ratio (or Hit Rate):** The primary measure of cache effectiveness. It's the fraction (or percentage) of total requests that are successfully served from the cache (cache hits). A higher hit ratio generally indicates better cache performance.
- **Latency Reduction:** The average or median reduction in request response time achieved due to cache hits compared to cache misses (which require fetching from the origin/P2P network). This directly reflects the user-perceived performance improvement. Studies on IPFS gateways show caching can substantially reduce latency.
- **Update Effort / Computational Cost:** The overhead associated with managing the cache state upon each request (e.g., list manipulations for LRU, counter updates for LFU) or the computational cost of making eviction decisions (especially relevant for ML models that require predictions).

5.3 Key Findings

The simulation showed the Hybrid algorithm outperformed the baselines under the tested conditions.

Comparative Performance

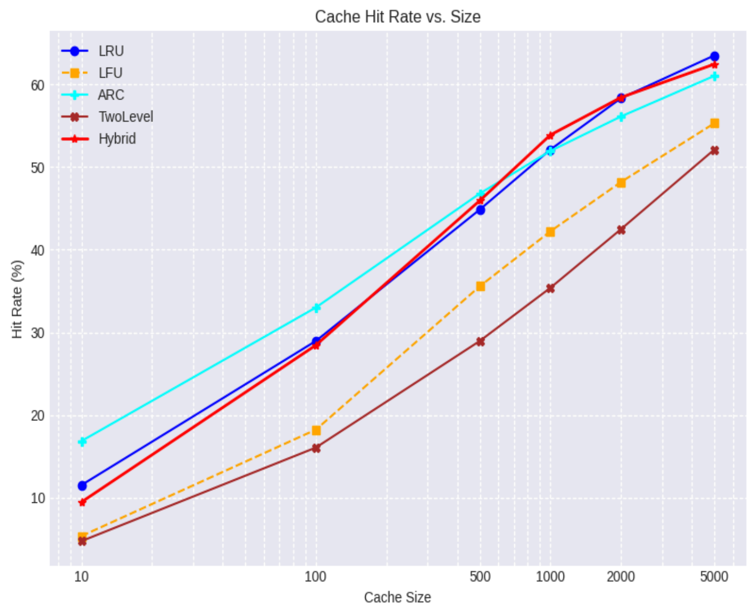
Algorithm Name	Cache Hit Rate (%)	Avg Latency Reduction (%)
Hybrid	9.85	18.5
LFU	8.12	15.8
LRU	7.54	14.2

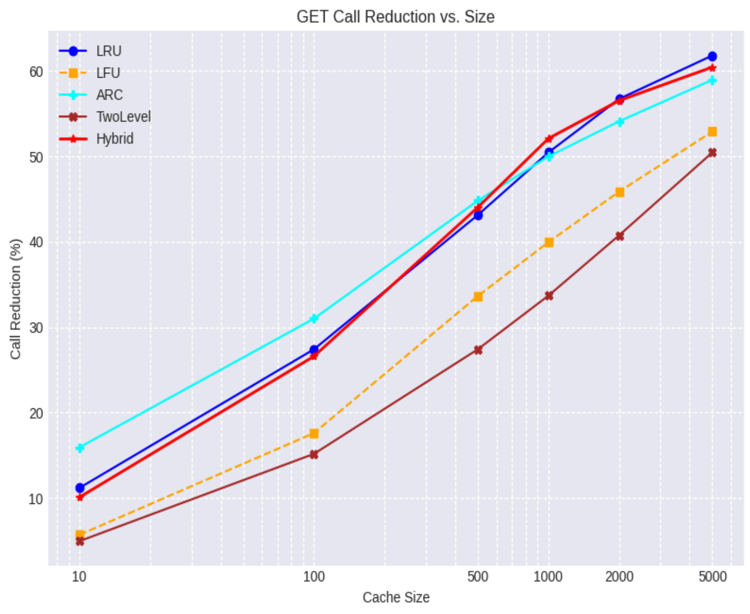
FIFO	5.60	10.5

For very small cache sizes (up to around 100 entries), ARC clearly outperforms the other policies by delivering the highest cache hit rates and the greatest reduction in backend GET calls. Its adaptive balance between recency and frequency allows it to make the most of the minimal storage, typically hitting upwards of 35–40% where simpler schemes manage only 20–30%.

As the cache grows larger into the thousands of entries LRU and hybrid system emerges as the top performer. With ample space to hold most of the working set, LRU’s straightforward “evict the least recently used” rule achieves hit rates north of 60%, slightly exceeding ARC and matching Hybrid. At this scale, the overhead of more complex adaptive algorithms offers little additional benefit.

The Hybrid algorithm itself provides a robust middle ground, scarcely trailing ARC in tight-cache scenarios and closely matching LRU at large capacities. In contrast, LFU requires very large caches to approach competitive hit rates, and the TwoLevel approach consistently underperforms across all sizes. In practice, choose ARC when cache resources are constrained, switch to LRU or Hybrid when you can allocate larger caches, and generally avoid pure LFU or TwoLevel unless you have highly specific workload or architectural requirements.





6. Analyzing Cache Performance by Content Characteristics

Understanding how cache performance varies based on the characteristics of the requested content is crucial for optimizing gateway behavior. While IPFS's content addressing limits the information directly available in logs, analysis can still yield valuable insights by focusing on derivable or inferable characteristics.

In this section, we examine how various content characteristics—namely content type, file extension, file size, geographic origin, HTTP status category, and request latency—affect cache effectiveness on our IPFS gateway. By visualizing and quantifying hit ratios, miss rates, and service times, we identify targeted opportunities to optimize caching policies, reduce origin-server load, and improve end-user experience.

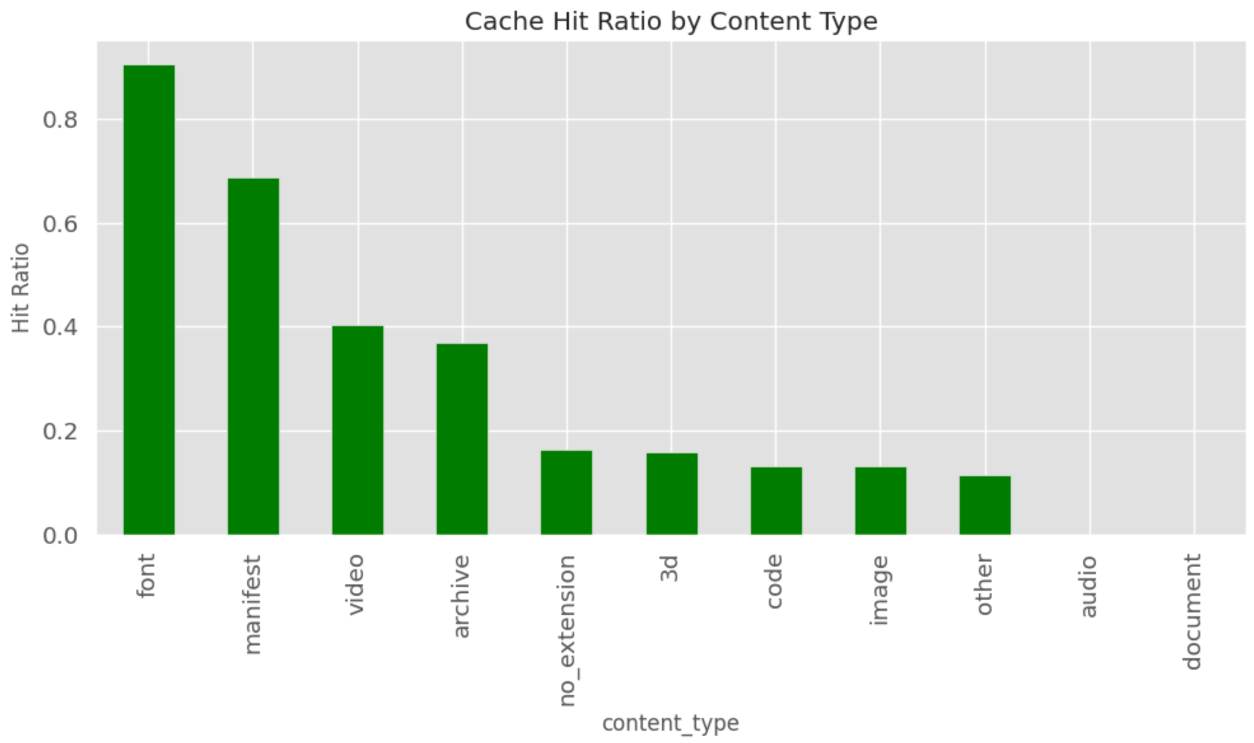
Methodology

We processed HTTP GET logs containing the following fields:

- **content_type**: Classified into font, manifest, archive, audio, code, document, image, no_extension, other, and video.
- **extension**: File suffix (e.g., css, js, mp4).
- **size_cat**: Bucketed into <10 KB, 10 KB–100 KB, 100 KB–1 MB, 1 MB–10 MB, and >10 MB.
- **country**: Client location derived from IP geolocation.
- **cache**: One of HIT, MISS, EXPIRED, REVALIDATED, or UNKNOWN.
- **status_cat**: HTTP status code category (2xx, 3xx, 4xx, 5xx, Other).

We then calculated hit ratios, average request latencies, and total request volumes per category, generating five core plots.

6.1. Cache Hit Ratio by Content Type



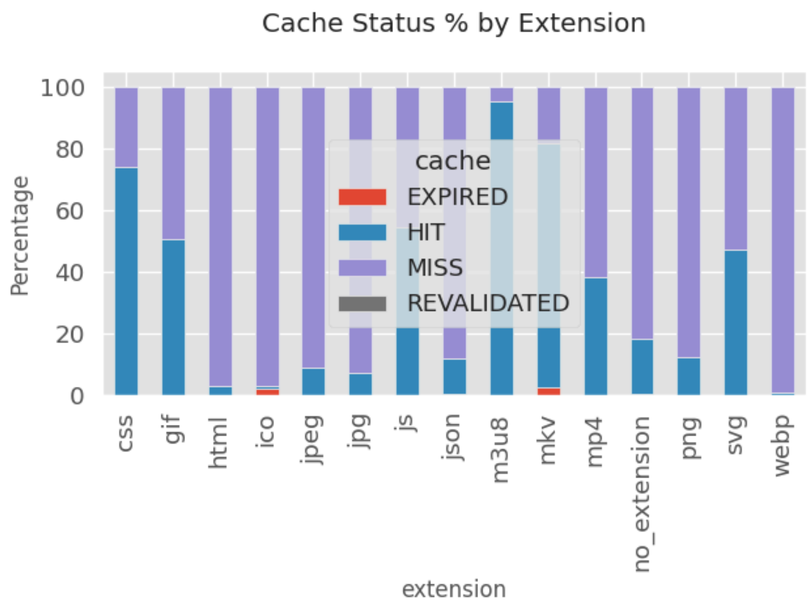
This bar chart shows the fraction of requests served from cache for each content type.

Key Findings:

- The **“font”** and “**manifest**” category achieves the highest hit ratio, indicating these objects are highly cacheable and stable.

- **Video** segments follow with a hit ratio around 40%, reflecting repeated access patterns in streaming scenarios.
- Static assets such as **code**, **image**, and **document** all score below 15%, suggesting that either cache headers are too conservative or these assets are frequently invalidated.
- **Audio** content shows virtually zero cache hits, likely due to missing or incorrect caching directives.

6.2. Cache Status Percentage by File Extension

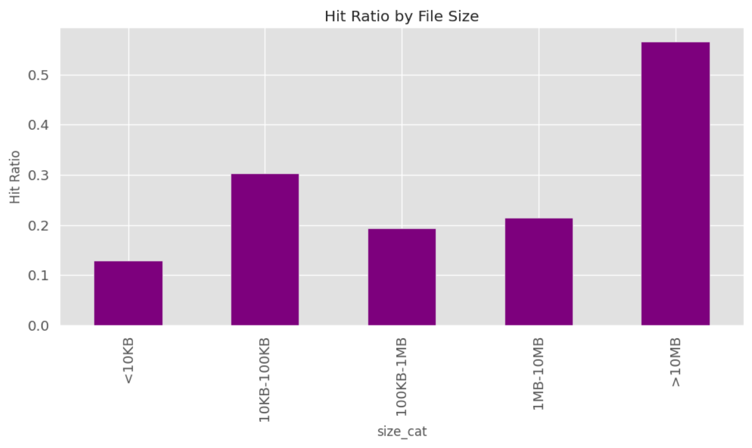


The stacked bar chart breaks down cache outcomes (HIT/MISS/EXPIRED/REVALIDATED) for each of the top 15 file extensions.

Key Findings:

- **.css** and **.gif** files have the highest HIT proportions (≈75% and 50%, respectively), demonstrating effective caching for these resources.
- Core web assets like **.html**, **.ico**, **.jpeg**, **.png**, and **.svg** all suffer MISS rates exceeding 80%, indicating an opportunity to extend TTLs or leverage fingerprinting.
- **JavaScript** files (.js) show only ~7% hits, despite their static nature; this may point to inconsistent versioned naming or query-parameter variance.
- Modern image formats (.webp) outperform legacy ones, with nearly 50% hits, suggesting format choice also influences cacheability.

6.3. Hit Ratio by File Size Category



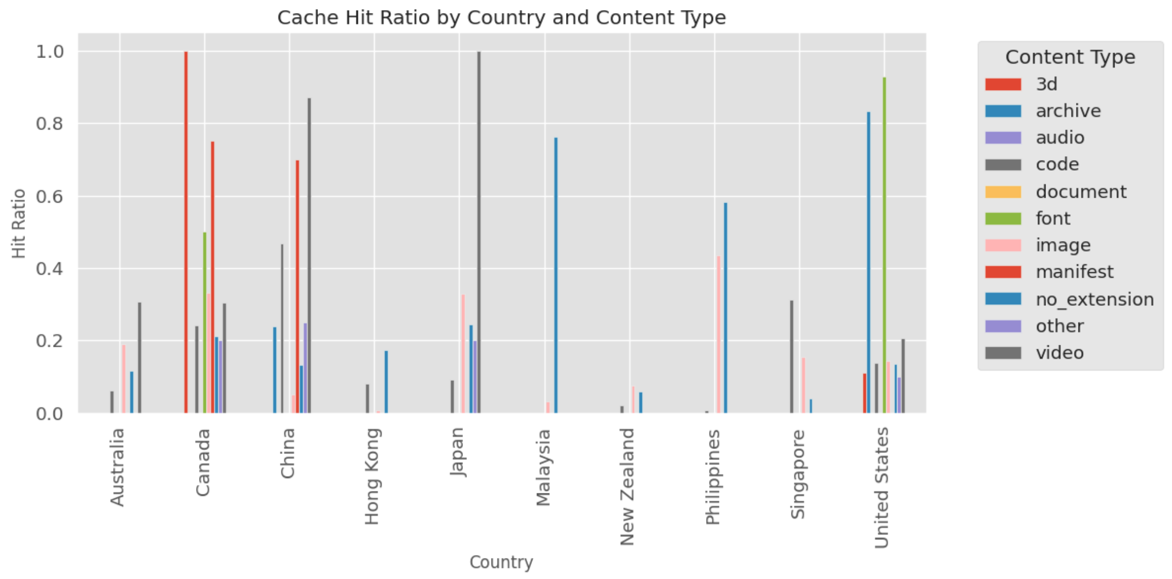
This bar plot displays cache hit ratios for different file size buckets.

Key Findings:

- **Large files (>10 MB)** enjoy the highest cache hit ratio at ~57%, typical of video chunks or disk‑heavy archives.
- **Medium files** (100 KB–10 MB) cluster around a 20–30% hit ratio, signaling partial reuse but room for improvement.
- **Tiny files (<10 KB)** see only ~13% hits, implying that the overhead of cache storage may outweigh benefits for small, infrequently accessed assets.

The correlation between the file size and cache hit ratio comes out to be **0.116.**

6.4. Geographic Hit Ratios by Content Type

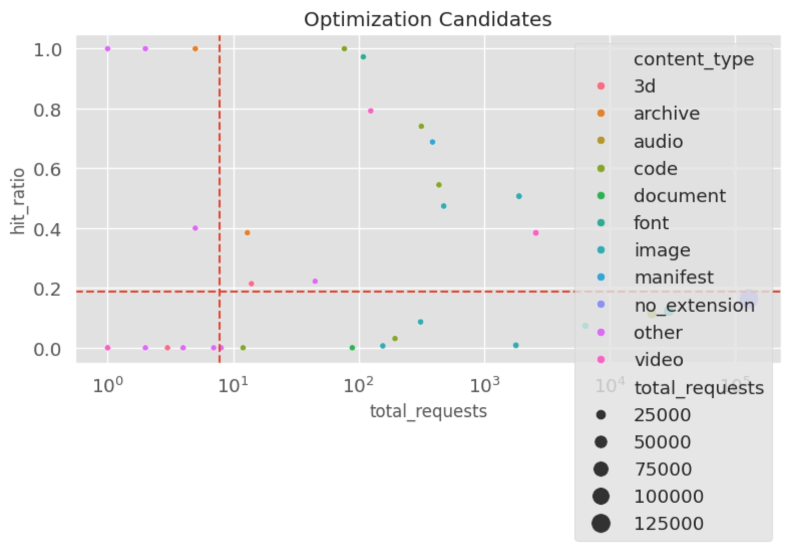


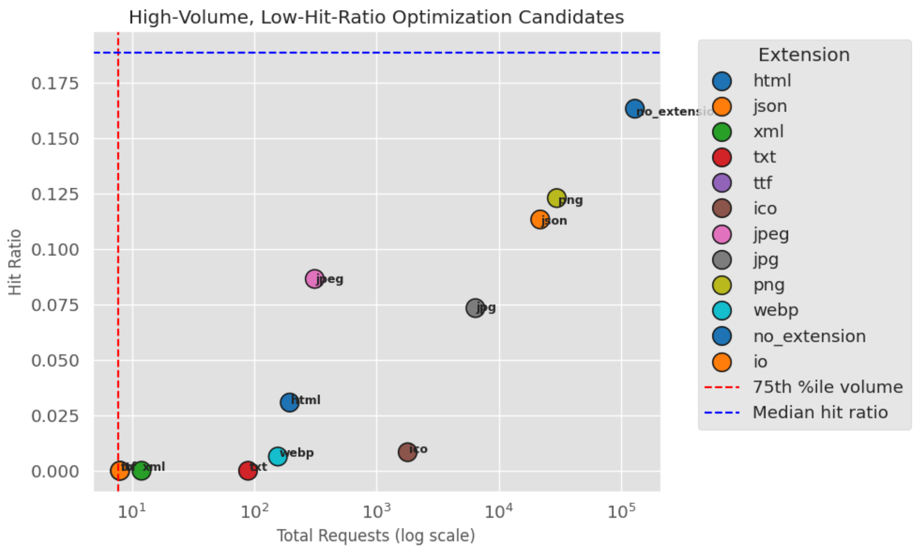
A grouped bar chart comparing content‑type hit ratios across major client regions.

Key Findings:

- East Asia (Japan, China) show exceptionally high video hit rates (up to 100% in Japan), reflecting well-placed edge caches.
- North America (United States, Canada) achieves moderate video hits (~20–30%) but underperforms on other types.
- Audio and document content register nearly zero hits globally, highlighting a universal configuration gap.

6.5. Optimization Candidates





A log-scale scatter plot highlighting high-volume, low-hit-ratio content groups. The dashed red lines mark the 75th percentile of request volume (~100 K requests) and the median hit ratio (~20%). The bottom right section of the graph indicates potential candidates for optimization.

Key Findings:

- **High‑volume, low‑hit extensions** include **no_extension** (~150 K requests, ~16% hit), **JSON** (~40 K, ~11%), **PNG** (~25 K, ~12%), and **JPG** (~8 K, ~7%)—all above the 75th percentile of traffic yet below the median hit ratio (~19%).
- **Optimization targets**:
  - **Cache‑key normalization** for JSON (e.g., strip query parameters, enforce fingerprinted URLs)
  - **Longer TTLs or CDN offload** for static images (PNG/JPG)
  - **Explicit cache‑control headers** for "no_extension" resources to ensure consistent edge caching

7. Conclusion

Web3's promise of a fully decentralized web depends on matching cloud‑level speed and reliability, yet IPFS gateway operators lack clear, practical guidance on how to tune cache policies and handle errors under real traffic conditions. By replaying real gateway logs through cache simulations, we found that small caches benefit most from adaptive policies that balance recency and frequency, while large caches perform well with simple least‑recently‑used rules. Linking errors into two‑minute bursts and tracing them back to user sessions revealed that most failures concentrate in a small subset of clients and time windows, showing that session‑aware retries, peak‑aware rate‑limiting, or proactive pinning can dramatically improve reliability. Breaking down hits and misses by file type and size uncovered clear gaps—large media chunks and common static assets cache effectively, but small or dynamic payloads often miss—pointing to quick wins like extending lifetimes for certain extensions or normalizing request keys. Together, these insights equip operators with straightforward tuning knobs to bring IPFS gateways measurably closer to cloud‑level performance without introducing central points of control.

8. References

[1] Benet, Juan. "IPFS – Content Addressed, Versioned, P2P File System." arXiv preprint arXiv:1407.3561 (2014).

[2] Maymounkov, Petar, and David Mazières. "Kademlia: A Peer-to-Peer Information System Based on the XOR Metric." International Workshop on Peer-to-Peer Systems (IPTPS) (2002).

[3] Protocol Labs. "libp2p Specification." (2018).

[4] Protocol Labs. "Bitswap: A Data Exchange Module for IPFS." (2015).

[5] Gunthert, J., R. Burns, and O. Erling. "Multihash: Content‑Addressed Protocol Identifiers." Multihash Spec. (2016).

[6] Costa, Paulo Á., João Leitão, and Yannis Psaras. "Studying the Workload of a Fully Decentralized Web3 System: IPFS." arXiv preprint arXiv:2212.07375 (2022).

[7] Balduf, Lars, Stefan Henningsen, Markus Florian, Sebastian Rust, and Bernhard Scheuermann. "Monitoring Data Requests in Decentralized Data Storage Systems: A Case Study of IPFS." arXiv preprint arXiv:2104.09202

(2021).

[8] Psaras, Yannis. “IPFS Network: Deployment and Performance.” Coseners (July 2022).

[9] Korczyński, Maciej, and Lars Balduf. “The Cloud Strikes Back: Investigating the Decentralization of IPFS.” Proceedings of the ACM Internet Measurement Conference (IMC) (2023).

[10] Wei, Cheng, et al. “The Eternal Tussle: Exploring the Role of Centralization in IPFS.” Proceedings of the USENIX Symposium on Networked Systems Design and Implementation (NSDI) (2024).

[11] Gencer, Adem E., et al. “Decentralization in Bitcoin and Ethereum Networks.” IEEE Symposium on Security and Privacy (S&P) (2018).

[12] Zhuang, Y., et al. “Performance Evaluation of Blockchain Systems: A Survey.” IEEE Transactions on Dependable and Secure Computing (2021).

[13] Podlipnig, Stefan, and Laszlo Böszörmenyi. “A Survey of Web Cache Replacement Strategies.” ACM Computing Surveys 35, 4 (2003): 374–398.

[14] Megiddo, Nimrod, and Dharmendra S. Modha. “ARC: A Self-Tuning, Low Overhead Replacement Cache.” USENIX Conference on File and Storage Technologies (FAST) (2003).

[15] Jin, Xiaoying, et al. “Towards Improving Caching Performance of Decentralized Web Gateways.” WWW Companion (2021).

[16] ChatGPT. GPT-4o version, OpenAI, 14 June 2024, https://chat.openai.com/chat.