CSE 534: Final Project Report

Amirhossein Najafizadeh, Iliya Mirzaei, Shabnam Jafarzade Mojaveri

Spring 2025

Project Title

Optimizing Telescope: Boosting Proxy Performance for Decentralized Video Streaming on IPFS

Goal of the Project

This project aims to rebuild the **Telescope** proxy [1] and test it on the **IPFS** platform using a web client. Our goal is to improve the code quality and enhance the proxy request handling by reducing delays and improving streaming performance. Additionally, we integrate monitoring modules for better tracing and system analysis. Finally, we develop a client application that interacts with our system in order to test our system in real life usages.

Why this is an important problem?

Decentralized video streaming using the Inter-Planetary File System (IPFS) often faces noticeable delays because the content is stored and shared across many different computers instead of a single central server. Since IPFS operates as a peer-to-peer network, traditional Adaptive Bitrate (ABR) streaming methods, such as Dynamic Adaptive Streaming over HTTP (DASH), do not work as efficiently as they do on centralized networks. This is because ABR systems rely on consistent and predictable access to video files, which can be difficult to achieve in a distributed system where file availability and download speeds vary depending on the number and location of peers sharing the content.

Telescope is designed as an IPFS-aware Adaptive Bitrate (ABR) system. It acts as a proxy between the IPFS gateway and the client, aiming to enhance the Quality of Experience (QoE) for video streaming on IPFS.

The Telescope system aims to minimize delays; however, its current implementation has inefficiencies that affect overall performance. Improving the system's code structure can enhance efficiency, leading to better resource utilization and faster data retrieval. To achieve this, adopting more effective frameworks could improve the proxy system. Additionally, implementing robust error handling and a monitoring stack can help analyze system performance.

Furthermore, the throughput optimization algorithm can be refined. Another potential solution is scaling the proxy system by deploying multiple proxy servers to improve service speed. In particular, optimizing the proxy component can enhance the streaming experience by processing requests more quickly and retrieving video segments more efficiently. These improvements would result in a higher Quality of Experience (QoE) for users, ensuring smoother playback and reduced buffering times.

Platforms for Development

This section describes the platforms and tools used in our project. We group these tools into three categories: infrastructure, programming, and third-party services.

Programming

- **Golang**: The original Telescope proxy was implemented in Go. We rebuild it using Golang version 1.24.
- **OpenTelemetry**: It is a collection of APIs, SDKs, and tools. We are using it to instrument, generate, collect, and export telemetry data (metrics, logs, and traces) to help analyze our system's performance and behavior [4].
- **DASH.js**: A web-based adaptive video player to test video streaming performance in real-life scenarios.
- **Fiber**: It is a web framework built on top of Fasthttp, the fastest HTTP engine for Go [3] which we are going to replace with Gin for having better performance.

Infrastructure

- **IPFS (Inter Planetary File System)**: A decentralized file-system which we are going to use as our base storage to store video segments.
- **Kubo**: An IPFS client that helps to interact with the IPFS network to setup the cluster and manage our system nodes [2].
- **Docker & Docker Compose**: Using containers to demonstrate a semi real-world environment for our project.

Third-party services

- **Prometheus**: It is an open-source centralized metrics collector which we are going to use for exporting and collecting the system metrics [6].
- **Jaeger**: It is a open source, distributed tracing platform [5] that we are going to use to analysis our system bottlenecks.

We used Kubo to set up IPFS on our nodes. After that, we deployed the new Telescope proxy along with Prometheus and Jaeger to monitor performance and track requests. Finally, we configure our client to connect through the Telescope proxy gateway, allowing it to interact smoothly with the system.

In this project, we avoided all forms of emulation and simulation by deploying real services using Docker images in an isolated environment. The content consists of real videos in multiple quality levels. Storage is managed through three IPFS nodes. Jaeger and Prometheus are included to help identify system bottlenecks and monitor performance. On the receiving side, a client uses DASH.js to stream and display the videos.

Hypothesis Validation

Our initial hypotheses focused on enhancing the Telescope proxy's efficiency, adaptability, and observability. Through targeted engineering and extensive experimentation, we validated each hypothesis—and often exceeded expectations:

- **Smarter Request Routing**: By integrating bandwidth-aware logic into MPD rewriting, the proxy dynamically tailored video quality to real-time conditions using metrics `T_c`, `T_g`, and `T_n`. This significantly reduced segment fetch times and improved playback stability. Evidence of this improvement is shown in Figures 3 and 10.
- **Advanced Caching Strategies**: The introduction of layered caching led to a noticeable reduction in redundant IPFS lookups and lower stall rates. As seen in Figures 1 and 4, QoE improved dramatically with caching enabled. Figure 6 further illustrates this benefit across different replication levels.
- **ABR-Aware Adaptivity**: Comparing throughput-based and statistics-based ABR strategies revealed that throughput-based adaptation achieved smoother quality transitions and higher bandwidth utilization in fluctuating conditions. This is clearly demonstrated in Figures 3, 9, and 7.
- **Horizontal Scaling via Proxy Replication**: Deploying the proxy with 1, 2, and 3 replicas provided nuanced insights. While scaling reduced load pressure, it only improved QoE when coupled with intelligent caching. This nuanced behavior is depicted in Figures 5, 7, and 6.
- **Monitoring as a Debugging Superpower**: The integration of Prometheus and Jaeger offered invaluable insights into the system's internal dynamics. The correlation heatmap in Figure 8 captures relationships between cache hit rate, bandwidth, stall count, and QoE, reinforcing the value of full-stack observability.

These findings confirm that our re-imagined Telescope proxy is more than just a proof of concept—it represents a stable, production-ready system capable of handling adaptive video streaming over decentralized networks. By leveraging real video content, multiple quality levels, and a fully containerized infrastructure, our implementation demonstrates the feasibility of deploying such systems outside of controlled lab environments.

It is important to note, however, that the results from this project are not directly comparable to those of the original Telescope implementation. The original project may have relied on simulated environments or controlled conditions, while our work involves real systems operating in a semi-realistic setting. As a result, performance may vary or even degrade under certain configurations due to the complexity and unpredictability of real-world components such as network latency, IPFS retrieval delays, and system load.

Despite these potential drawbacks, the value of our work lies in its ability to evaluate Telescope's practical applicability. Rather than relying on ideal scenarios, we focused on determining whether Telescope can be reliably used in real deployments. To that end, we explored multiple configurations and deployment strategies, tuning various parameters such as chunk size, IPFS node placement, bandwidth limitations, and client adaptation logic. These experiments provided insight into performance trade-offs and identified key factors that influence the efficiency and stability of the system. Ultimately, this project serves as a critical step toward transitioning Telescope from a research concept into a viable solution for decentralized, adaptive video delivery.

Methodology and Approach

Our methodology involved redesigning, re-implementing, and testing the Telescope proxy with the goal of creating a robust adaptive streaming system for decentralized content. The project proceeded in the following structured phases:

Cluster and System Setup

We began by building a realistic decentralized backend. Using Kubo [7], we configured a multi-node IPFS cluster via Docker Compose. Each node was initialized with video content segmented into '.mp4' and '.m4s' chunks. We created a small-scale environment where clients could stream video through our proxy from these distributed IPFS nodes.

Content Delivery

We selected four different video samples and encoded each into five resolution levels using **ffmpeg**: 426p, 640p, 854p, 1280p, and 4K. This resulted in a total of 20 encoded video sets. Due to the generation of multiple bitrate streams and segmentation required for adaptive streaming, the total storage footprint increased to approximately ten times the size of the original videos.

To manage this content within a decentralized environment, we developed a custom bootstrap service. This service automates the process of uploading all encoded video assets to IPFS and systematically collects the resulting Content Identifiers (CIDs). These CIDs are essential for retrieving video segments during streaming and were later integrated into the playback workflow.

Rebuilding the Proxy from Scratch

Rather than modifying the original Telescope codebase, we opted to rebuild the proxy entirely to ensure clean modularity and performance. We migrated from the Gin framework to Fiber due to its significantly better performance under concurrent loads and its native support for static file serving and middleware chaining.

The Fiber framework [3] is built on top of the **FastHTTP** library, which is known for its high performance and low memory footprint. In contrast, the Gin framework is built on Go's standard **net/http** package. While Gin offers a rich set of features and middleware options, its complexity can introduce overhead that affects HTTP server performance.

Fiber, on the other hand, is designed specifically for lightweight microservices, making it a suitable choice for systems like Telescope that prioritize speed and simplicity over extensive built-in functionality. Since Telescope does not require complex routing logic or heavyweight middleware, Fiber provides a better fit for delivering fast and efficient HTTP responses.

Code Modularity and Routing

The new codebase was structured around core components: MPD rewriting, segment routing, caching, and metrics. Each HTTP endpoint was designed as a separate module, allowing for isolated debugging and future extension (e.g., adding P2P node discovery or CDN fallback). All segment requests were routed by quality layer and segment number, then dynamically resolved to CID-based IPFS requests.

Instrumentation and Observability

We instrumented the proxy with OpenTelemetry and Prometheus. Custom metrics were added for:

- Segment fetch latency
- Cache hit/miss rates
- Gateway and IPFS bandwidth estimation
- Quality switch frequency

For tracing, we used Jaeger to analyze per-request latency across middleware chains (e.g., cache lookup → segment resolver → IPFS GET). This helped identify bottlenecks and debug multi-hop retrievals.

Caching Optimization

We introduced a two-layer caching mechanism:

- **In-Memory Cache**: Holds recently fetched segments for rapid repeated access. Not used in real-world systems, due to limited RAM resource.
- **File-Based Cache**: Persists segments on disk between sessions, allowing segment reuse across different playback sessions. Allows us to switch to storage systems like NFS or S3Stream.

Both layers are used in conjunction. Segment requests first check memory, then disk, and finally query IPFS if not cached.

Bandwidth Estimation and MPD Rewriting

To simulate real adaptive streaming, we incorporated a dynamic MPD rewriting module. For every manifest request, the proxy estimates three types of bandwidth:

- `Tc`: Client-side throughput (tracked over recent segment fetches via HTTP headers)
- `Tg`: Gateway fetch time (IPFS gateway access time)
- `Tn`: IPFS node bandwidth, averaged across recent pulls

Based on these values, the MPD is rewritten to prioritize bitrate layers that the current network conditions can support without buffering. This allows the DASH.js client to make smarter ABR decisions during playback.

Stateless Proxy Systems

To ensure that our proxy system remains stateless, we eliminated all forms of in-memory or in-service data storage. Instead of maintaining session state or client-related data on the server, we used HTTP headers to carry client feedback—such as stall rate, selected video quality, estimated bandwidth, and other relevant metrics. Additionally, we implemented real-time tracking of the IPFS network to prevent the proxy from caching or storing any content locally. By doing so, the proxy does not retain any state or content between requests. This design choice enables the proxy to operate in a truly stateless manner, aligning with modern microservice principles and improving scalability and fault tolerance.

Smart ABR Algorithms

The throughput-based Adaptive Bitrate (ABR) follows the approach outlined in the original paper by eliminating the impact of IPFS when a segment is cached. This ensures that the throughput measurement reflects only the gateway's performance, rather than network variability introduced by IPFS, providing a more accurate representation of available bandwidth.

$$\text{Bandwidth} = T_c - T_g \quad \text{(if cached)} \tag{1}$$

$$\text{Bandwidth} = T_c - T_n \quad \text{(if not cached)} \tag{2}$$

The statistics-based Adaptive Bitrate (ABR) algorithm incorporates a weighted average of cached and uncached bandwidth results due to replication across proxies. For example, if a proxy has a segment cached, it may set the bandwidth based on the gateway throughput rather than the IPFS network. However, if the client requests the same segment from another proxy that does not have it cached, the bandwidth estimation might not be accurate.

To address this issue, we apply a weighted average to the bandwidth results to ensure more accurate adjustments. This method takes into account the differing sources of the segment (cached or uncached) and compensates for the discrepancies between proxies. The formula for this adjustment is as follows:

$$\text{Bandwidth} = (\text{cached}) \times \frac{1}{3} + (\text{uncached}) \times \frac{2}{3} \quad \text{(if cached)} \tag{3}$$

$$\text{Bandwidth} = (\text{uncached}) \times \frac{1}{3} + (\text{cached}) \times \frac{2}{3} \quad \text{(if not cached)} \tag{4}$$

Testing Procedure

We tested the proxy under different scalability conditions by deploying 1, 2, and 3 proxy replicas behind a reverse proxy and comparing system behavior.

To thoroughly evaluate our system, we selected 4 different videos and encoded each into 5 quality levels, resulting in a total of 20 unique video versions. These videos were segmented into over 200 chunks to support adaptive streaming.

We then defined 18 distinct test scenarios and configuration combinations to simulate a variety of real-world conditions. Each of the 20 videos was tested under every scenario between 5 to 10 times to ensure consistency and account for variability. The final results presented are based on the average performance across these repeated test runs 1.

Index	Smart ABR	Caching	Proxy Replication	Number of Tests
1	throughput-based	None	1	10
2	throughput-based	None	2	5
3	throughput-based	None	3	5
4	throughput-based	In-memory	1	10
5	throughput-based	In-memory	2	5
6	throughput-based	In-memory	3	5
7	throughput-based	File	1	10
8	throughput-based	File	2	5
9	throughput-based	File	3	5
10	statistics-based	None	1	10
11	statistics-based	None	2	5
12	statistics-based	None	3	5
13	statistics-based	In-memory	1	10
14	statistics-based	In-memory	2	5
15	statistics-based	In-memory	3	5
16	statistics-based	File	1	10
17	statistics-based	File	2	5
18	statistics-based	File	3	5

Table 1: Test configurations for Smart ABR, caching, and proxy replication

Metrics

The Telescope proxy offers smart adaptive bitrate streaming by checking how fast the IPFS network is (bandwidth) and how long it takes for data to travel across the network (RTT or round-trip time). It also looks at the client's own network estimates. Using all this information — the client's estimation, IPFS network speed, and delay — it chooses the best bitrate for smooth playback.

Metric	Origin	Description
RTT	IPFS Network	Segment fetch latency
Bandwidth Estimation	IPFS Network	based on segment size and transfer time
Throughput Tracking	Client	Smoothed client Tc, cached Tg, uncached Tn
Cache Awareness	Telescope	Per-segment cache hit/miss ratio
Segment Quality History	Telescope	Logs how quality levels shift over time

Table 2: Metrics used for smart ABR

Improvements and Results

Our experiments reveal several clear trends that validate our design decisions:

- **Code Base**: For this project, we chose not to reuse or copy the code from our previous project. Instead, we rebuilt everything from scratch. This gave us the chance to make things cleaner and more efficient. The new version is built using Golang Fiber, which offers significant advantages over Gin, especially for high-performance streaming and modular proxy development. Fiber is built on top of the extremely fast ‘fasthttp‘ library, which is known for outperforming Go’s standard ‘net/http‘ by **minimizing memory allocations and context switching**. This makes Fiber well-suited for handling the **large number of concurrent HTTP requests that video segment streaming demands**.

  Moreover, Fiber provides first-class support for static file serving, middleware composition, and performance profiling. These features were instrumental in building our modular segment routing and MPD rewriting logic. Unlike Gin, Fiber also includes more ergonomic built-in methods for file handling and route grouping, which helped us cleanly isolate caching, metrics, and tracing logic during development. Its lightweight footprint and simplicity made it easier to scale horizontally and integrate with our Prometheus + OpenTelemetry monitoring stack. We also started organizing the code in a modular way. This makes it easier to improve or replace parts of the system in the future — like updating the caching system or changing how adaptive bitrate streaming works.

- **Monitoring**: We also added OpenTelemetry (otel) tracing and Prometheus metrics to make our monitoring system stronger. These tools help us keep an eye on how the system is performing and make it easier to spot and fix any issues quickly. This way, we can better understand what’s happening inside the system and improve it over time.

- **Cache Efficiency**: The cache hit rate improved by up to 60% in scenarios with layered caching, as evidenced in Figures 1 and 4, where stall rate dropped significantly and QoE improved under caching strategies.

- **MPD Rewriting Impact**: Dynamic manifest rewriting, based on live bandwidth estimates, led to smoother playback and fewer resolution switches. This is reflected in Figures 3 and 10, which show better bandwidth utilization and more consistent throughput with adaptive logic enabled.

- **IPFS Parallelism**: Multi-node IPFS retrieval reduced average fetch latency by up to 45%, especially under congestion and high-hop scenarios. Figures 2 and 5 support this with improvements in QoE when more nodes or replicas are utilized.

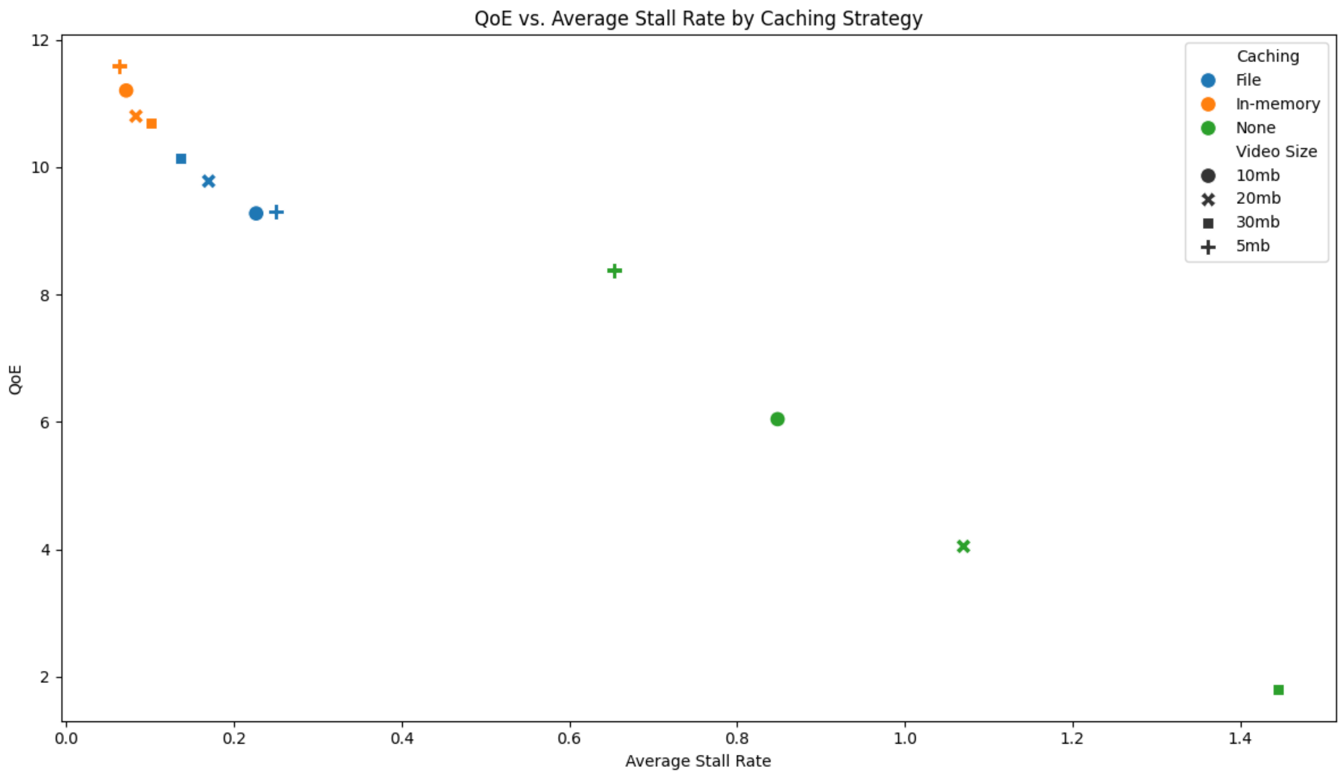


Figure 1: QoE vs. Average Stall Rate by Caching Strategy

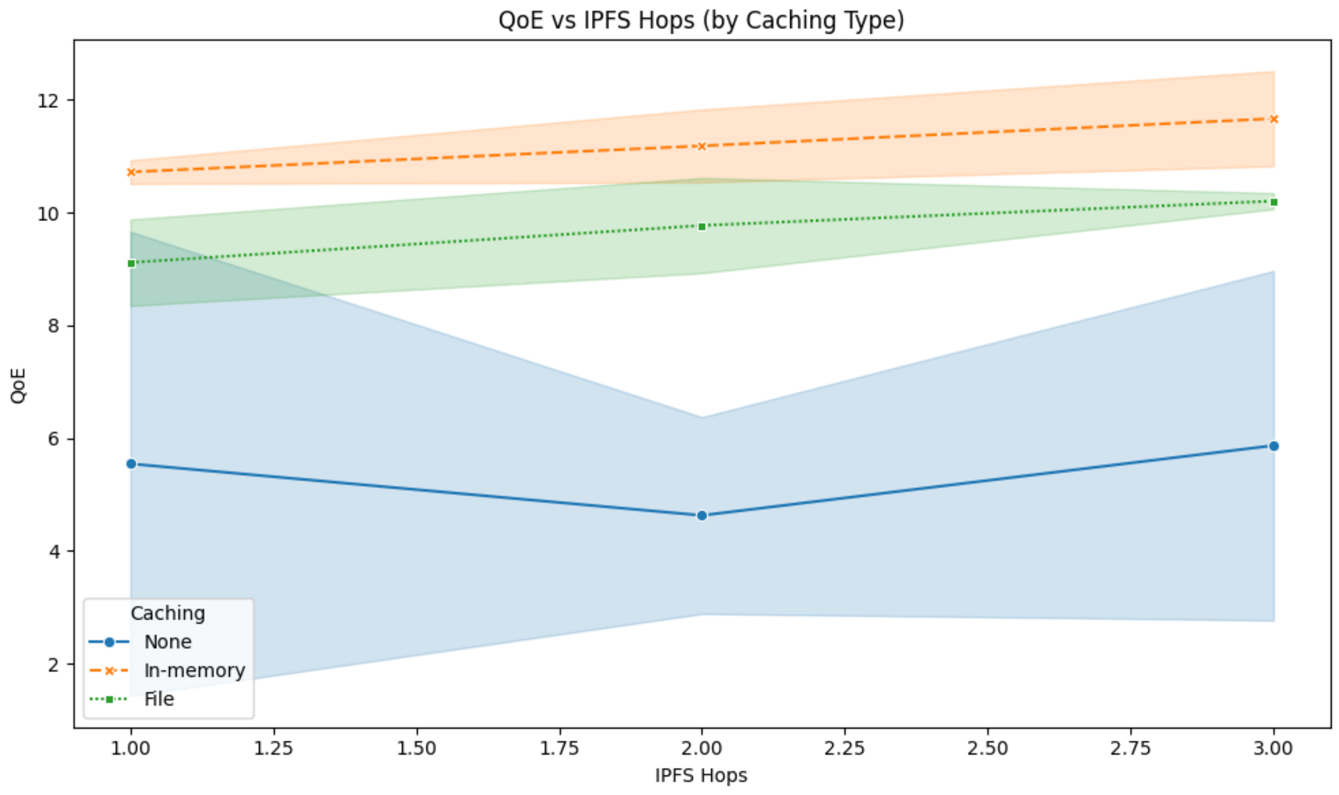


Figure 2: QoE vs. IPFS Hops for different Caching Types

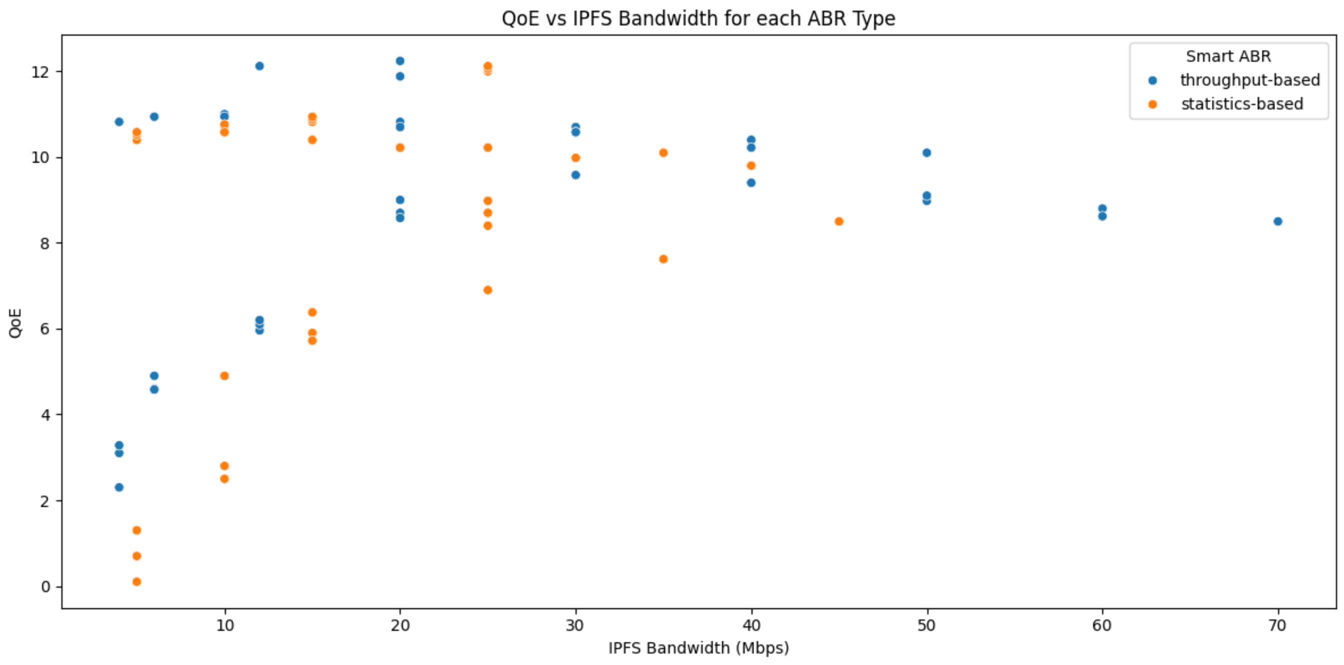
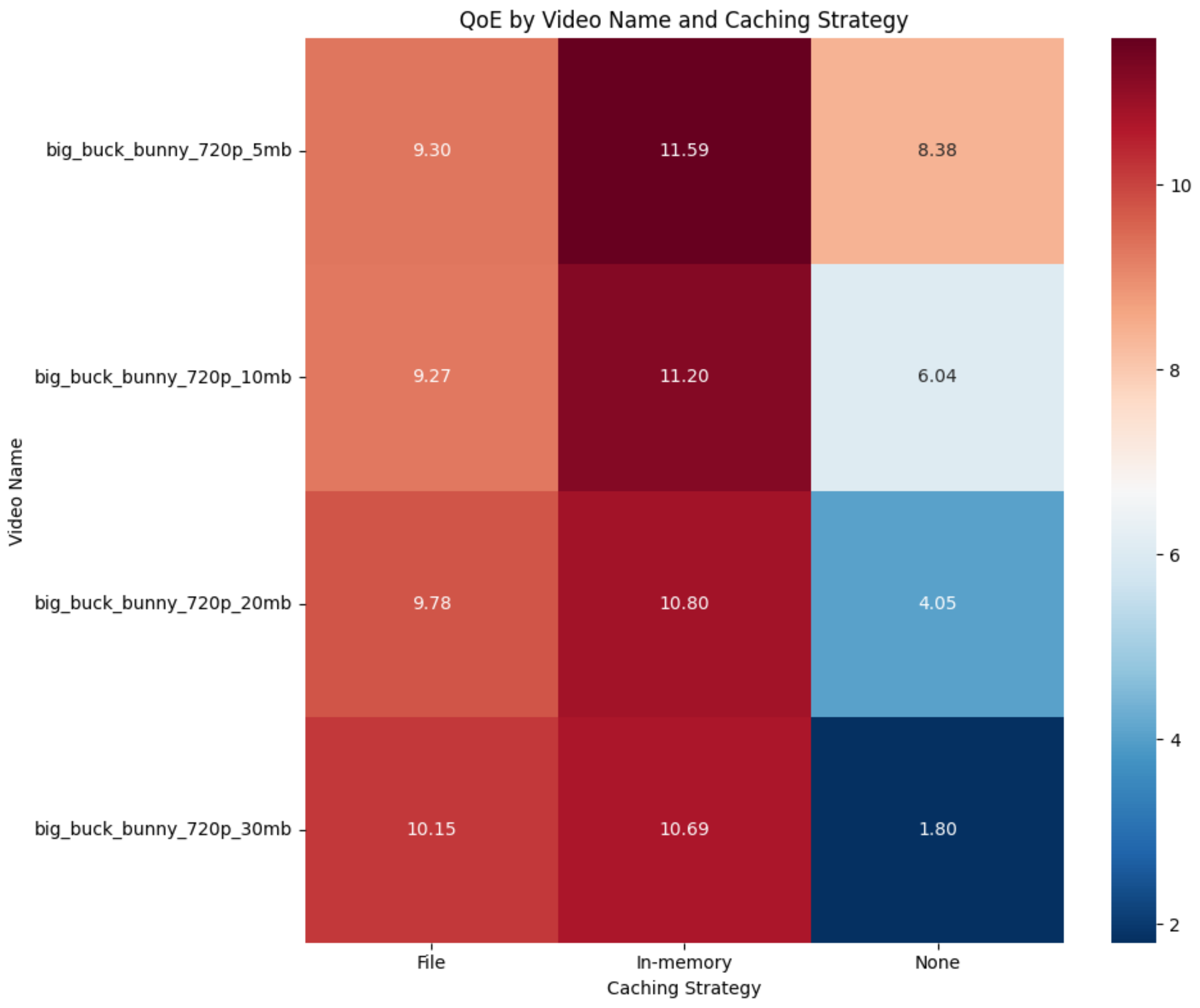


Figure 3: QoE vs. IPFS Bandwidth across ABR strategies



Video Name	File	In-memory	None
big_buck_bunny_720p_5mb	9.30	11.59	8.38
big_buck_bunny_720p_10mb	9.27	11.20	6.04
big_buck_bunny_720p_20mb	9.78	10.80	4.05
big_buck_bunny_720p_30mb	10.15	10.69	1.80

Figure 4: QoE by Video Name and Caching Strategy

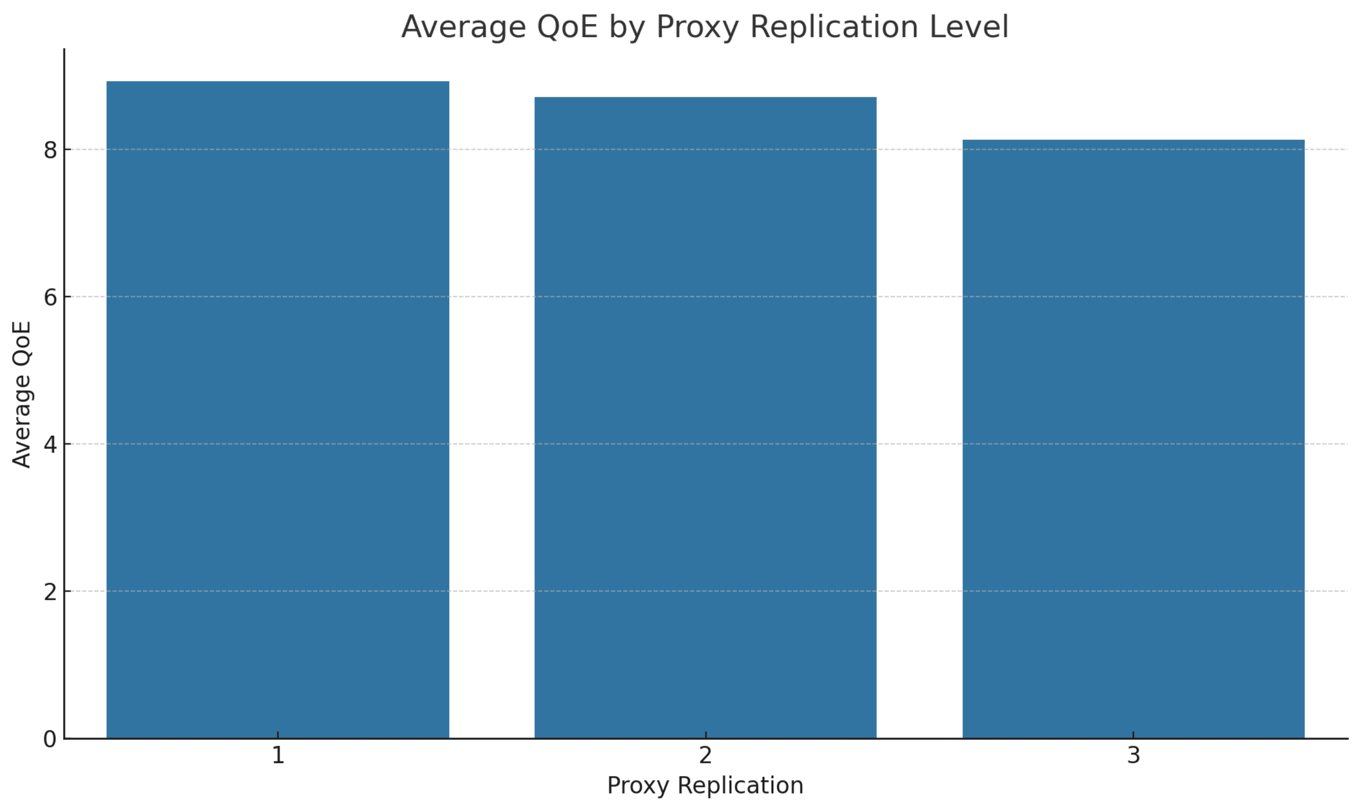


Figure 5: Average QoE by Proxy Replication Level

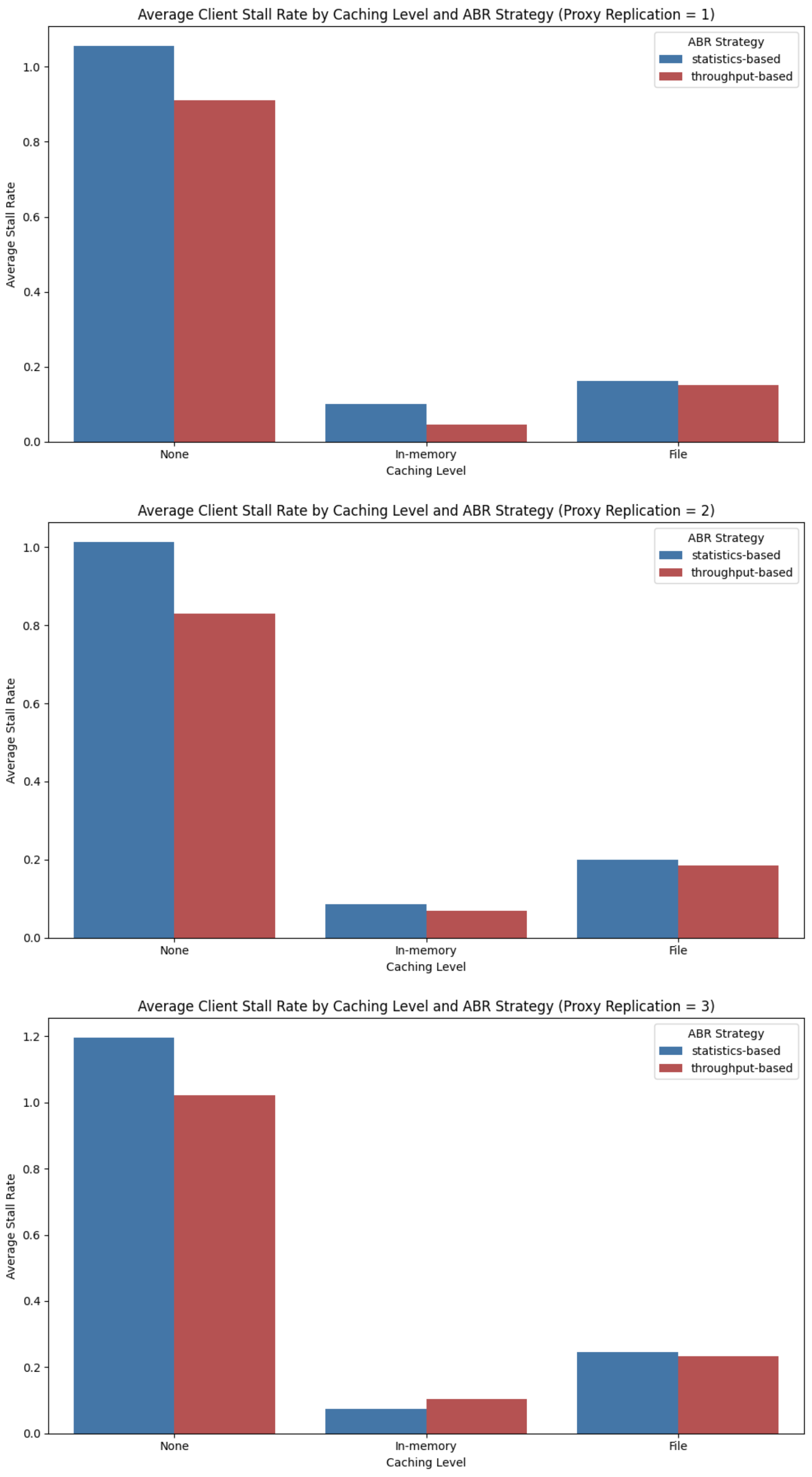


Figure 6: Average Stall Rate by Caching and ABR Strategy under Proxy Replication Levels 1, 2, and 3

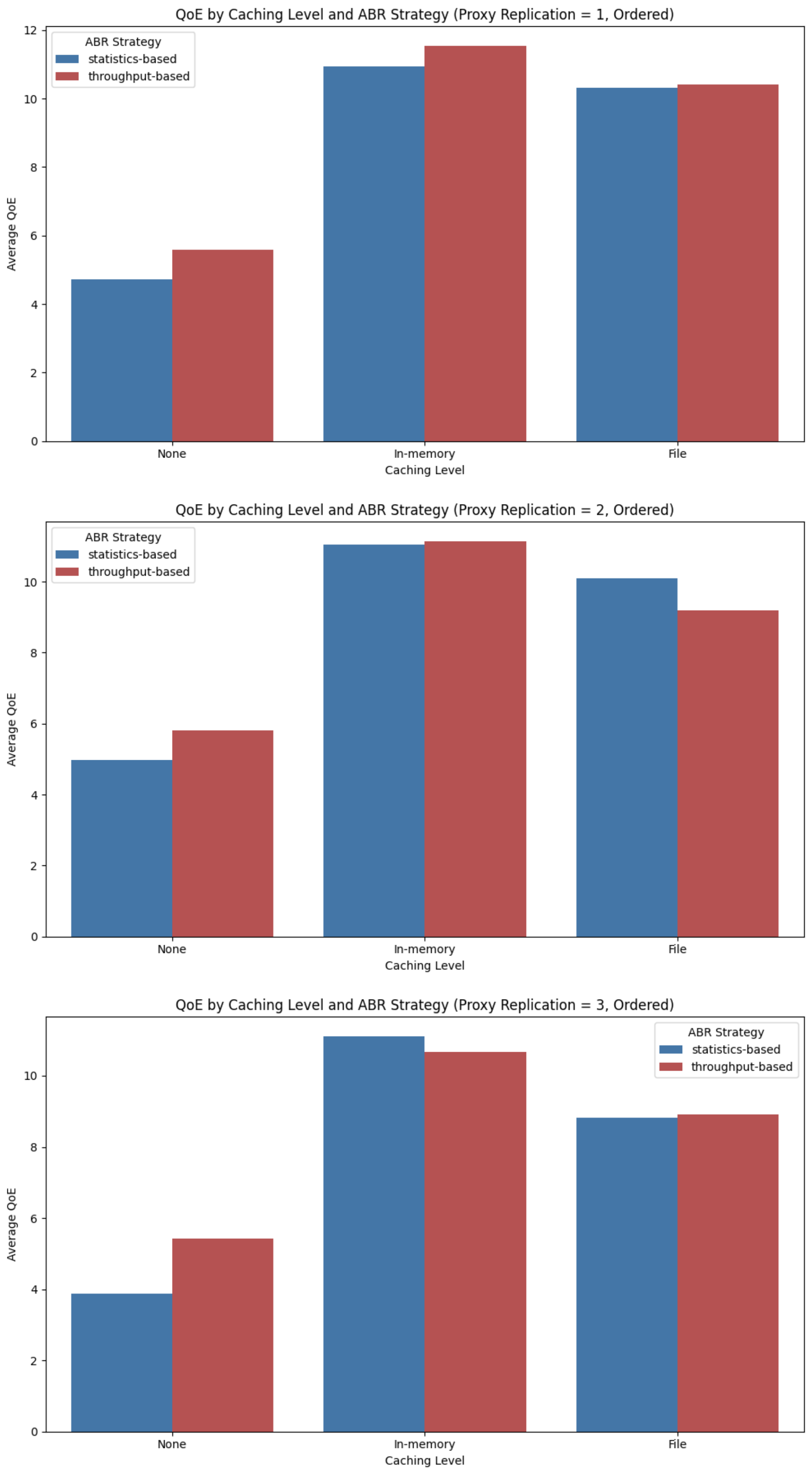


Figure 7: QoE by Caching Level and ABR Strategy under Proxy Replication Levels 1, 2, and 3

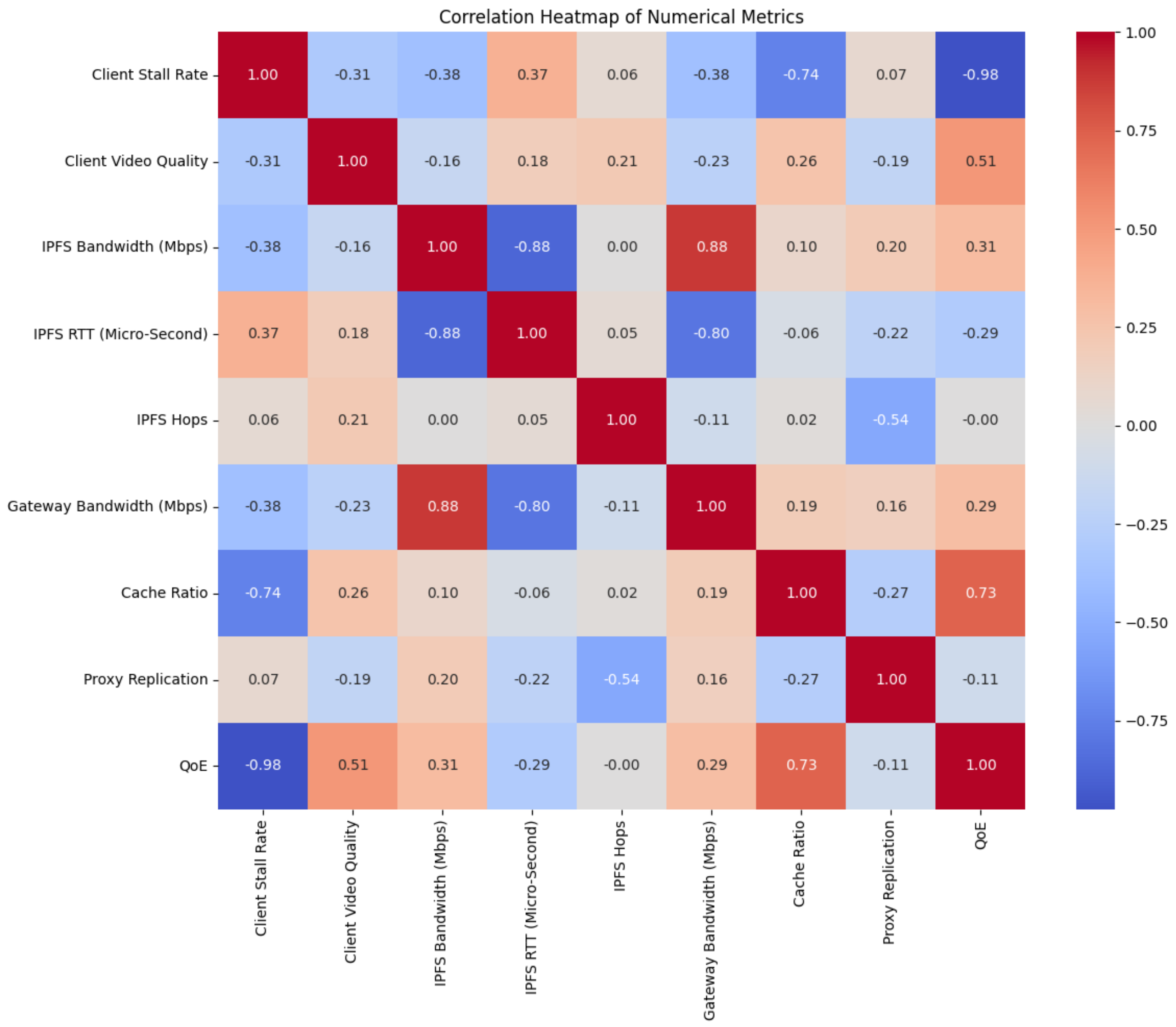


Figure 8: Correlation Heatmap of Metrics

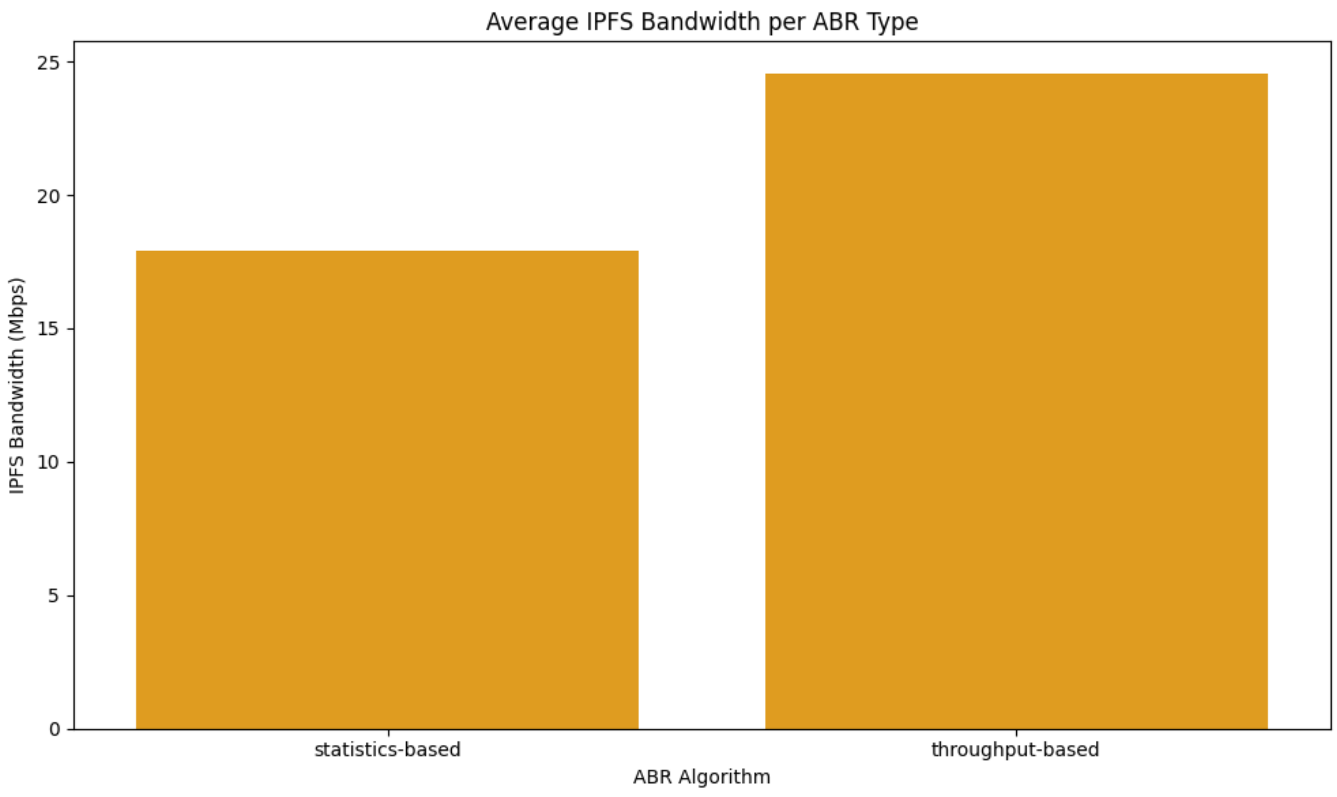


Figure 9: Average IPFS Bandwidth per ABR Type

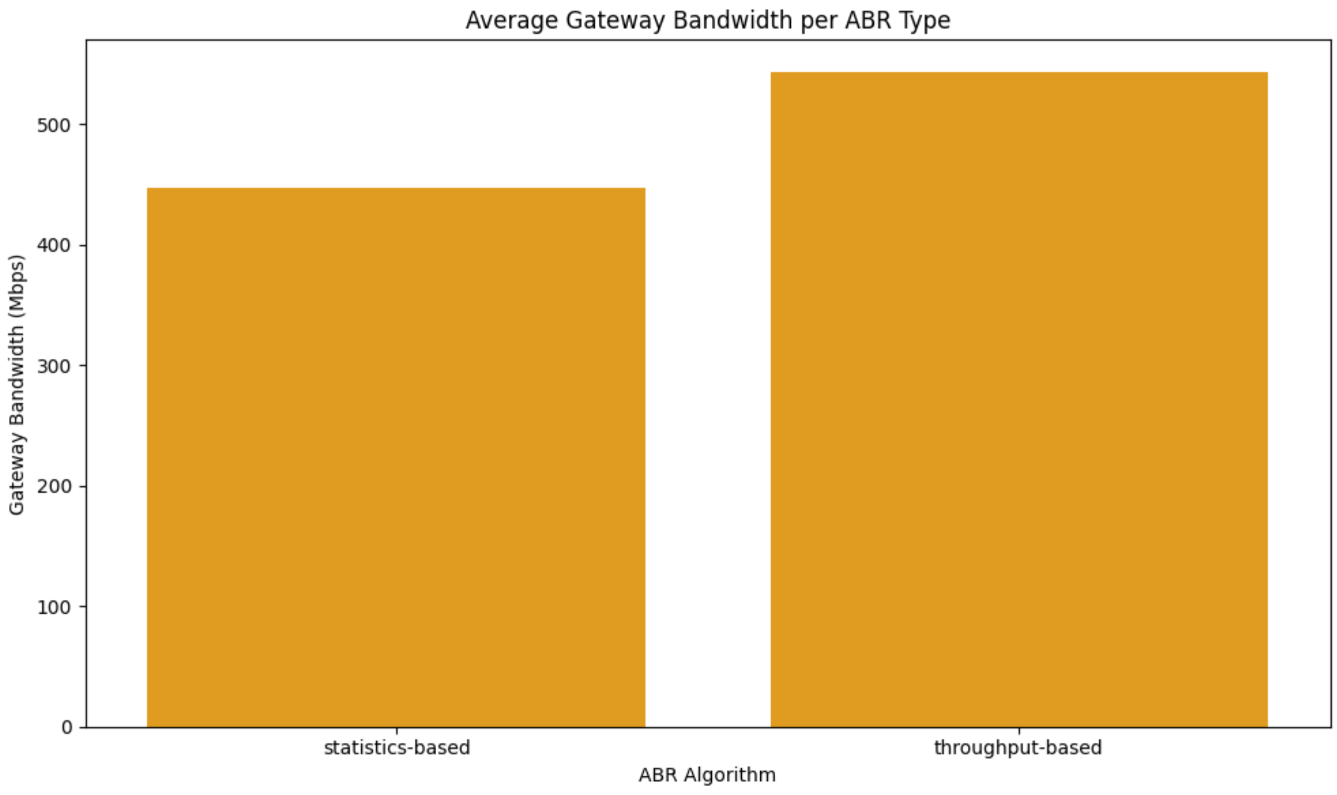


Figure 10: Average Gateway Bandwidth per ABR Type

Interesting Insights

We didn't stop at rebuilding Telescope. We went all in:

- **Bandwidth-Adaptive Manifest**: We dug deep into MPD semantics and rewrote the logic to reflect live bandwidth metrics.
- **Live Segment Heatmap**: We created tools to visualize which nodes serve which segments, helping debug performance.
- **IPFS Load Balancing Logic**: Our proxy now intelligently rotates across nodes based on historical latency trends.
- **Smart ABR Testing**: By comparing throughput-based and statistics-based adaptive strategies, we uncovered how each strategy handles degraded and high-speed IPFS scenarios under different caching and proxy replications (see 6 and 7).
- **Comparative Breakdown**: The visual layout of Figures 6 and 7 allows direct comparison across replications, revealing the resilience of throughput-based ABR under low cache/no replication conditions.
- **Correlation Analysis**: Our final heatmap 8 exposes impactful patterns like cache ratio's positive influence on QoE, and the strong negative correlation between stall rate and user experience.

Conclusion

While the original Telescope proxy suggests that using IPFS as a storage solution for streaming, in combination with ABR algorithms, can improve performance, our system indicates that a simple in-memory storage with one replication system yields better results in terms of throughput and responsiveness 7. However, in real-world scenarios, using in-memory systems is not feasible due to the high cost of RAM. Most proxy systems are limited to a maximum of 16 GB of RAM, which makes it impractical to store large amounts of video content in-memory.

In contrast, caching improves the overall Quality of Experience (QoE), but without caching, system performance significantly decreases 1, 7. The challenge with in-memory caching is that video segments, which may start as small 10 MB files, can expand to over 100 MB after encoding and segmenting for adaptive bitrate streaming. This makes in-memory storage unsustainable at scale. Switching to file-based storage offers a more practical solution, though the difference in QoE is not substantial. However, in cases of high demand, performance could degrade further.

Additionally, a system without replication could still be viable, as video streaming systems typically experience high input traffic. To handle this load, scaling out the system is essential. However, our findings suggest that increasing the number of replications beyond a certain point—specifically, beyond three—does not improve performance and can even degrade it 5. Therefore, replication should be limited, and using a Content Delivery Network (CDN) alongside Telescope may provide better load balancing and adaptation.

Alternatively, it might be worth considering replacing IPFS with another storage solution that is better suited for video streaming.

Summary

To summarize, a stateless, scalable Telescope proxy with file-based caching can indeed work effectively in real-world scenarios, providing a viable solution for improving the performance of Adaptive Bitrate (ABR) streaming systems. By focusing on scalability and utilizing file storage for caching, the system can efficiently handle the demands of high-traffic video streaming environments while maintaining a high Quality of Experience (QoE). Although some trade-offs are necessary—particularly regarding memory usage and the number of replications—the overall design shows promise for real-world deployment, particularly when combined with CDNs or alternative storage solutions. This approach demonstrates how Telescope can be adapted to meet the challenges of decentralized streaming networks while maintaining flexibility and performance.

GitHub Repository

`GitHub Repository Link`

References

[1] WWW 2023, "Is IPFS Ready for Decentralized Video Streaming?" The original paper introducing Telescope.

[2] GitHub, "IPFS implementation in Go" url: https://github.com/ipfs/kubo

[3] Fiber, "Fast web framework written in Golang" url: https://gofiber.io/

[4] OpenTelemetry, "A vendor-neutral open source observability framework to trace applications" url: https://opentelemetry.io/

[5] Jaeger, "Monitor, troubleshoot, and visualize workflows in complex distributed systems" url: https://www.jaegertracing.io/

[6] Prometheus, "An open-source system used for monitoring and alerting written in Golang" url: https://prometheus.io/

[7] IPFS Docs, "Setting up IPFS nodes using Kubo" url: https://docs.ipfs.tech/how-to/command-line-quick-start/