Big Data Analytics Project: Design and Benchmarking of Distributed P2P Architectures

Di Franco Federico

January 20, 2026

Abstract

This report presents the design, implementation, and rigorous performance evaluation of a distributed Peer-to-Peer file-sharing system tailored for Big Data storage scenarios. The project addresses the scalability challenges of data placement and retrieval by evolving from an unstructured baseline to a hybrid structured architecture. We analyze three distinct strategies: a *Naive* approach relying on flooding, a *Semantic Partitioning* strategy optimizing for data locality, and a *Metadata Sharding* architecture leveraging Global Secondary Indexes (GSI) with salting and Consistent Hashing with Virtual Nodes.

The system was validated through a dual-phase benchmarking suite, comparing a local high-performance baseline against a distributed cluster deployed on Google Cloud Platform (GCP). Experiments utilizing Zipfian workloads reveal that while Semantic Partitioning offers $O(1)$ routing for specific queries, it suffers from critical storage hotspots (Gini coefficient > 0.7). Conversely, the GSI-based approach demonstrates superior scalability, maintaining sub-100ms search latency and uniform load distribution even under stress. Furthermore, a Proof-of-Concept integration with the P4P/ALTO framework is presented, demonstrating a 71% reduction in network costs and the total elimination of cross-ISP traffic through topology-aware peer selection.

Contents

1 Introduction

The exponential growth of Big Data imposes severe constraints on centralized storage architectures, driving the adoption of distributed Peer-to-Peer (P2P) systems. However, designing a P2P overlay that balances efficient retrieval, uniform storage distribution, and fault tolerance remains a complex trade-off. This project explores the architectural evolution of a file-sharing system designed to address these challenges.

Our investigation moves through three evolutionary stages. We begin with a baseline *Naive* implementation based on unstructured flooding (reminiscent of Gnutella), analyzing its limitations in network saturation. We then introduce *Semantic Partitioning* (Document Partitioning) to optimize for data locality, exposing its vulnerability to data skew. Finally, we propose a *Metadata Sharding* architecture that decouples content from metadata, utilizing Global Secondary Indexes (GSI) distributed via Consistent Hashing to achieve both scalability and load balancing.

The system is implemented as a modular Python application, containerized via Docker, and orchestrated on a distributed Google Cloud Platform cluster to simulate real-world wide-area network (WAN) conditions. Beyond the application layer, the project also explores network-layer optimizations through a prototype integration of the P4P (ALTO) protocol, evaluating the benefits of ISP-aware peer selection.

2 System Architecture

The system follows a modular object-oriented architecture designed to isolate network primitives from specific P2P strategies. The core logic is encapsulated in a generic `BasePeer` class, which manages the underlying TCP/IP communication, node discovery, and storage abstraction. Specific routing and indexing behaviors (Naive, Semantic, Metadata) are implemented via polymorphism in subclasses, ensuring a fair comparison on a shared infrastructure.

2.1 Network Topology and Consistent Hashing

The overlay network is structured as a Distributed Hash Table (DHT) managed by the `ConsistentHashRing` class (in `hashing.py`). To map both nodes and data keys onto a shared identifier space, the system utilizes the MD5 hash function, converting keys into a 128-bit integer space.

To address the "Data Skew" problem typical of basic consistent hashing, we implemented the concept of **Virtual Nodes**.

- Each physical peer is not mapped to a single point on the ring but is represented by k virtual replicas (default $k = 100$).
- The `add_node` method generates deterministic virtual labels (e.g., `peer1#0`, `peer1#1`...) and places them using `bisect.insort` to keep the ring ordered.

The lookup mechanism (`get_node`) utilizes binary search (`bisect`) to find the smallest node identifier N_{id} such that $hash(N_{id}) \geq hash(K)$ in $O(\log V)$ time, where V is the total number of virtual nodes.

2.2 Data Model: Storage and Fragmentation

The persistence layer is handled by the `Storage` class (in `storage.py`), which abstracts file system operations and enforces a "Split-and-Merge" strategy similar to BitTorrent.

1. **Chunking:** The `split_file` method divides large binaries into fixed-size blocks of 1 MB. Each chunk is uniquely identified by the SHA-1 hash of its content. This *Content-Addressable Storage* (CAS) approach enables automatic deduplication: identical files (or identical parts of different files) result in the same hash and are stored only once per node.

2. **Manifests:** Metadata is decoupled from the payload. A Manifest is a JSON document generated by `create_manifest` containing the filename, file size, and an ordered list of Chunk Hashes with their locations.

This separation allows the system to distribute the heavy payload (chunks) across the cluster for load balancing via DHT, while the lightweight metadata (manifests) can be replicated or indexed according to the specific peer strategy.

2.3 Communication and Background Tasks

Inter-node communication is performed via stateless HTTP REST APIs defined in `api.py` and served by a Flask WSGI server. To handle the dynamic nature of P2P networks (churn), the `BasePeer` class spawns several background daemon threads upon initialization:

- **Failure Detector:** A thread executes `failure_detector` every 5 seconds, sending UDP-like heartbeats (`/ping`) to known neighbors. If a node fails to respond within a `failure_timeout` (15s), it is removed from the local Hash Ring to maintain routing consistency.

- **Gossip Protocol:** The `gossip_known_peers_loop` periodically selects random neighbors and exchanges the `known_peers` list via the `/update_peers` endpoint, ensuring the network topology converges to a fully connected graph.

- **Self-Healing (Rejoin):** An `attempt_rejoin` task runs at startup to contact a set of Bootstrap Peers, allowing a disconnected node to automatically re-enter the cluster.

2.4 Fault Tolerance and Consistency Models

The system implements an **Eventual Consistency** model, prioritizing Availability (AP in CAP theorem).

Primary-Push Anti-Entropy

To counteract data loss during node failures, the `NaivePeer` (extended by other classes) implements a proactive **Anti-Entropy** mechanism. A background thread (`anti_entropy_loop`) periodically wakes up with randomized jitter and performs a "Responsibility Check":

1. It scans local manifests using `list_local_manifests`.

2. For each file, it calculates the `placement_hash` to determine the authoritative Primary Node.

3. If `self.id` is the Primary, it queries the $k = 3$ successors via the `/check_existence` API.

4. If a replica is missing, the `_send_manifest` method is triggered to push the data to the successor, ensuring the replication factor is restored.

Conflict Resolution: Last-Write-Wins (LWW)

Since multiple peers might upload updates to the same file, the system resolves conflicts using timestamps. The `_resolve_conflicts` method groups search results by filename and selects the entry with the highest `updated_at` timestamp. Furthermore, a **Read Repair** mechanism is triggered asynchronously: if a client detects a stale version during a search, it automatically pushes the winning manifest to the outdated nodes.

3 Implementation Strategies

To evaluate different architectural trade-offs, we implemented three distinct peer behaviors overriding the `upload_file` and `search` methods.

3.1 Strategy 1: Naive (Baseline)

The `NaivePeer` class implements an unstructured P2P approach, prioritizing storage load balancing over search efficiency.

Data Placement (DHT)

In `upload_file`, the chunks are distributed purely based on Consistent Hashing:

$$Node_{chunk} = \text{Ring.get\_node}(hash(Chunk_{content}))$$

This ensures a near-perfect uniform distribution of storage load (bytes) across the cluster. The Manifest file is also placed deterministically on the $k = 3$ successors of the filename hash.

Search Mechanism: Flooding

Lacking a global index for metadata attributes, the `search` method implements a **Flooding** algorithm (Gnutella-style) to handle attribute-based queries (e.g., "Find movies with Actor=Brad Pitt").

1. **Local Lookup:** The node first queries its own `storage`.

2. **Remote Broadcast:** It iterates through the entire `known_peers` list, triggering a synchronous HTTP GET request to the `/search_local` endpoint of every neighbor.

3. **Aggregation:** Results are collected and deduplicated client-side.

Complexity: $O(N)$ for search. While robust against churn, this creates a broadcast storm that does not scale.

Crucial Distinction: Search vs. Retrieval

It is vital to distinguish between *Resource Discovery* (Search) and *Content Retrieval* (Download). While the search process is inefficient ($O(N)$) because the location of a file matching a generic query is unknown, the download process remains highly efficient ($O(1)$). Once a specific filename is identified (e.g., `movie_10.bin`), the system bypasses flooding. The `fetch_file` logic (inherited from `BasePeer`) calculates $hash(\text{filename})$ and routes the request directly to the responsible node via the DHT. This hybrid behavior accurately simulates systems where the bottleneck is locating content metadata, not transferring the payload itself.

3.2 Strategy 2: Semantic Partitioning (Data Locality)

The `SemanticPeer` implements a Document Partitioning strategy optimized for query performance on specific attributes.

Data Placement (Co-location)

Unlike the Naive approach, this strategy overrides the placement logic to enforce **Data Locality**. In `upload_file`, a `partition_key` is derived from the metadata (specifically the `genre` attribute, falling back to `title`).

$$Node_{target} = \text{Ring.get\_node}(hash(Metadata_{genre}))$$

Both the heavy chunks and the manifest are forced onto this single target node (and its replicas). This sacrifices storage balancing (popular genres create "Hotspots") to ensure that all data regarding "Action Movies" resides on the same machine.

Search Mechanism: Hybrid Routing

The `search` method inspects the query to decide the routing strategy:

- **Routed Search ($O(1)$):** If the query contains the partition key (e.g., `Genre=Sci-Fi`), the client hashes the genre, locates the responsible node via the Ring, and sends a single directed query via `_query_node`.
- **Broadcast Fallback ($O(N)$):** If the query does not contain the key (e.g., searching only by `Actor`), the node cannot know where the data is. It falls back to a parallelized broadcast using a `ThreadPoolExecutor` to query all peers.

3.3 Strategy 3: Metadata Sharding (GSI with Salting)

The `MetadataPeer` implements a hybrid approach, maintaining the balanced chunk storage of the Naive strategy but adding a distributed Inverted Index (Global Secondary Index - GSI).

Data Placement

Heavy chunks are distributed via DHT (Load Balanced) exactly as in the Naive strategy. However, the `upload_file` method includes an additional step: `_gsi_write`. This method extracts searchable attributes (Actor, Year) and writes index entries to remote

nodes. To prevent write hotspots on popular keys (e.g., "Brad Pitt"), we implemented **Salting**:

$$ShardID = \text{random}(0, \text{INDEX\_SHARDS} - 1)$$

$$Node_{index} = \text{Ring.get\_node}(hash(Attribute : Value : ShardID))$$

This effectively splits the index for popular terms across multiple nodes.

Search Mechanism: Scatter-Gather

The `search` method executes a **Scatter-Gather** pattern using Python's `concurrent.futures`:

1. **Scatter:** For each attribute in the query, the node calculates the locations of all S shards (where S = `INDEX_SHARDS`) and dispatches parallel HTTP requests to retrieving the partial lists.
2. **Gather:** Results are aggregated in memory.
3. **Intersect:** The system performs a logical AND intersection between the result sets of different attributes to refine the final candidate list (e.g., matching both Actor AND Year).

Complexity: $O(S \times A)$, where S is the fixed number of shards and A is the number of query attributes. This decouples search latency from the total cluster size N, ensuring scalability.

4 Testing and Verification Methodology

Developing distributed algorithms requires rigorous testing to handle non-deterministic behaviors, network partitions, and race conditions. We adopted a comprehensive validation strategy structured into three layers, following the *Testing Pyramid* approach: Logic Simulation, Unit Testing, and End-to-End Integration Testing.

4.1 Layer 1: Logic Simulation (Bug Discovery)

Before deployment, the core routing and repair algorithms were validated using in-memory mock simulations. We developed a specific simulation suite, `simulate_healing_logic.py`, to verify the Anti-Entropy logic in isolation.

Critical Bug Discovery: The simulation revealed a subtle but fatal logic error in the `SemanticPeer`. While the standard `NaivePeer` locates files based on the hash of the filename, the `SemanticPeer` places files based on the hash of a metadata attribute (e.g., "Genre"). Initially, the repair thread inherited the naive logic, causing it to check the wrong nodes for replicas. The simulation allowed us to identify this mismatch and implement a polymorphic `_get_placement_key()` method override, ensuring that self-healing respects the semantic partitioning rules.

4.2 Layer 2: Unit Testing for Resilience Logic

To verify the system's behavior under complex edge cases—such as network timeouts and concurrent writes—without the overhead of spinning up Docker containers, we utilized Python's `unittest` framework with mocking.

Conflict Resolution (LWW) and Read Repair

We implemented `test_read_repair_unit.py` to validate the eventual consistency mechanism. By mocking the network layer (`sys.modules['requests']`), we simulated a "Split Brain" scenario where two peers hold different versions of the same file. The test asserts that:

1. The system correctly identifies the version with the highest `updated_at` timestamp as the "winner" (Last-Write-Wins).
2. A repair task is automatically triggered to push the winning manifest to the node holding the stale version, verifying the passive consistency convergence.

Partial Availability (Graceful Degradation)

To ensure the system satisfies the Availability property of the CAP theorem, we developed `test_partial_search.py`. This test mocks a scenario where a subset of the network is unreachable (e.g., a dead shard in Metadata mode or a crashed neighbor in Naive mode). The results confirm that the API catches network exceptions and returns a valid JSON response containing results from the healthy nodes, flagged with `partial_result: True`.

Functional Verification

Before proceeding to performance benchmarks, we verified the correctness of the distributed algorithms using specific test scripts.

Core System Integrity

The script `test_system.py` validates the fundamental data lifecycle:

- **Upload & Download:** Verifies that a file uploaded to Peer A can be downloaded from Peer B.
- **Data Integrity:** Performs a SHA-1 hash comparison between the original file and the rebuilt file to ensure zero data corruption during chunk transfer.
- **Network State Consistency:** Verifies that `known_peers` lists are correctly updated across the cluster when a node is stopped and restarted.

Complex Query Logic (Metadata Aggregation)

For the `MetadataPeer`, we validated the correctness of the GSI (Global Secondary Index) implementation using `test_system_metadata.py`:

- **Salting Aggregation:** We simulated a "Hotspot" scenario by uploading 5 different files sharing the same popular actor key ("Brad Pitt"). The test confirms that the search operation correctly aggregates results from multiple shards into a single list.
- **Multi-Attribute Intersection:** We verified the search logic by performing queries with multiple constraints (e.g., *Actor="Keanu" AND Genre="Sci-Fi"*). The test asserts that the system correctly computes the intersection of candidate sets from different indices.

4.3 Layer 3: End-to-End Resilience Testing

The final validation phase involved executing automated scenarios against the live $N = 10$ node Docker cluster to verify fault tolerance.

Self-Healing and Anti-Entropy

The script `test_healing.py` simulates a catastrophic node failure.

1. **Sabotage:** One of the replica nodes is forcibly terminated using `docker stop`.
2. **Detection:** The script polls the network state. We observed that after the Failure Detector timeout ($15s$) and the Anti-Entropy cycle ($20 - 40s$ jitter), a new node was automatically provisioned with the missing data.
3. **Verification:** The test asserts that the replication factor returns to $k = 3$ on a distinct set of physical nodes.

Graceful Shutdown (Churn Management)

To verify the protocol's correctness during voluntary departures, we utilized `test_graceful_shutdown.p`
The test initiates a '/leave' request on a target peer. The logs confirm that the leaving peer successfully transfers its manifests to its successors in the Hash Ring before terminating the process. A subsequent download request confirms that the file remains accessible immediately after the node's departure, proving zero downtime during planned maintenance.

5 Experimental Setup

To evaluate the performance and scalability of the proposed architectures, we developed a comprehensive benchmarking suite capable of operating in two distinct environments: a controlled local simulation and a distributed cloud deployment.

5.1 Test Environments

The experiments were conducted on two different infrastructures to isolate the impact of network latency and hardware constraints.

Local Baseline (Functional Verification)

For preliminary testing and functional verification, we utilized a high-performance local workstation. In this setup, $N = 10$ Docker containers communicate via a virtual bridge network. This environment represents a "best-case scenario" with near-zero network latency ($< 0.1ms$), allowing us to identify CPU-bound bottlenecks inherent to the Python Global Interpreter Lock (GIL) and application logic.

Distributed Cloud Cluster (Scalability Stress Test)

To assess real-world performance, we deployed a cluster on **Google Cloud Platform (GCP)**. The infrastructure consists of 5 distinct Virtual Machines (`e2-medium` instances) acting as Worker Nodes.

- **Topology:** We orchestrated a total of $N = 30$ peers, distributed as 6 Docker containers per Virtual Machine.
- **Networking:** Containers communicate over an overlay network spanning multiple physical hosts, introducing realistic latency, jitter, and bandwidth constraints typical of wide-area networks (WAN).
- **Orchestration:** The cluster is managed via custom automation scripts that handle provisioning, container spawning, and log aggregation.

5.2 Workload Generation

To simulate realistic Big Data scenarios, the workload generator follows a **Zipfian distribution** (Power Law), reflecting the principle that a small number of items (e.g., popular movies) account for the majority of traffic.

- **Dataset:** For the cloud stress test, we generated **5,000 synthetic binary files** (1MB each) to fill the Distributed Hash Table (DHT) and stress the storage layer.
- **Metadata Attributes:** Each file is tagged with attributes (*Actor*, *Genre*, *Year*). The "Actor" attribute follows an 80/20 distribution: 80% of files are associated with a small set of popular keys (e.g., "Brad Pitt", "Scarlett Johansson"), while the tail consists of 500 unique "Indie" actors. This is crucial to stress the *Semantic Partitioning* strategy and evaluate Hotspot handling.
- **Query Patterns:** We executed a sequence of **5,000 automated search queries** divided into:
  - *Hotspot Queries (50%):* Searching for popular keys to test aggregation bottlenecks.
  - *Long-tail Queries (30%):* Searching for rare keys to stress the Scatter-Gather phases.
  - *Multi-attribute Queries (20%):* Testing the intersection logic (e.g., *Actor* AND *Genre*).

5.3 Evaluation Metrics

The benchmark suite collects telemetry data for every operation, aggregating the following metrics:

1. **Write Latency:** Time elapsed between the client request and the confirmation that chunks and metadata have been replicated to $k = 3$ nodes.
2. **Read Latency (Search):** Time taken to resolve a query and return the list of matching files. We analyze both Average and Tail Latency (P99).
3. **Throughput:** The maximum number of successful requests per second (req/s) the system can sustain before saturation.
4. **Storage Balance (Gini Coefficient):** We measure the statistical dispersion of stored items across nodes to quantify the effectiveness of the load balancing strategy (0 = Perfect Equity, 1 = Total Imbalance).

5.4 Visual Monitoring and Management Tool

Given the complexity of debugging distributed systems solely through command-line logs, we developed a dedicated Graphical User Interface (GUI) named *EldenRing Torrent Monitor*. This tool, built with Python (Tkinter), interacts directly with the Docker daemon and shared volumes to provide real-time observability.

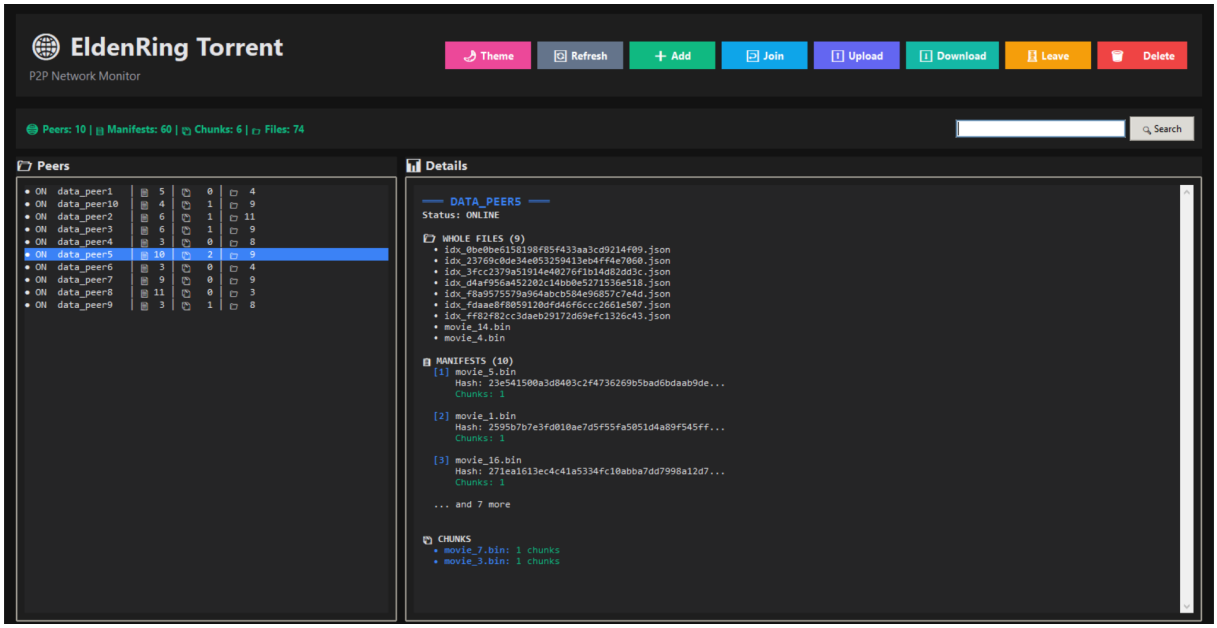


Figure 1: The Peer Monitor GUI interface. The left panel lists active nodes with storage statistics, while the right panel details specific files and chunks hosted on the selected peer.

As shown in Figure 1, the interface facilitates granular testing through:

- **Node Lifecycle Management:** Users can dynamically spawn new peers ("Add"), simulate node failures ("Delete"), or trigger graceful shutdowns ("Leave") directly from the toolbar to test Fault Tolerance mechanisms.
- **Data Inspection:** By scanning the mounted `/data` volumes, the tool parses JSON manifests in real-time, allowing for immediate visual verification of data distribution strategies (e.g., observing chunks spreading across nodes in the Naive strategy vs. concentrating in Semantic mode).
- **Manual I/O Operations:** The interface wraps the REST API, enabling users to upload or download files via a GUI without constructing raw HTTP requests.
- **Integrated Network Search:** A unified search bar is integrated into the dashboard's "Info Bar". The engine supports dual-mode querying: users can perform *Metadata Searches* (filtering by attributes) or execute *Direct Filename Lookups* (e.g., `movie_10.bin`). This feature enables precise verification of query resolution logic and file availability across the distributed network.

6 Performance Evaluation

To thoroughly assess the scalability and robustness of the proposed architectures, we conducted a dual-phase benchmark. The first phase establishes a baseline using a high-performance local machine with near-zero network latency. The second phase deploys the cluster on a distributed Cloud infrastructure to introduce real-world network constraints (latency, jitter, and bandwidth limits).

6.1 Experimental Environments

Local Environment (Baseline)

The local benchmarks were executed on a high-end laptop workstation. In this setup, Docker containers communicate via a virtual bridge on the same physical host, resulting in negligible network latency ($< 0.1ms$).

- **Device:** Laptop Workstation
- **CPU:** AMD Ryzen 7 5700U (1.80 GHz, 8 Cores)
- **RAM:** 16.0 GB DDR4
- **Storage:** 477 GB NVMe SSD (High I/O throughput)
- **Network:** Docker Bridge (Local Loopback)

Cloud Environment (Distributed Scale)

The distributed benchmarks were executed on **Google Cloud Platform (GCP)** using the Compute Engine. The cluster consisted of 5 distinct Virtual Machines (Worker Nodes) hosting the containerized peers.

- **Infrastructure:** 5x Google Compute Engine Instances
- **Machine Type:** `e2-medium` (2 vCPU, 4GB Memory per VM)
- **OS:** Ubuntu 22.04 LTS
- **Zone:** europe-west1-b
- **Network:** Virtual Private Cloud (VPC) internal network (Real inter-node latency)

6.2 Latency Analysis: The Impact of Network Scale

By comparing the search latency between Local and Cloud environments, we can isolate the "Network Amplification Effect" of each algorithm.

Local Baseline Results

Figure 2 shows the performance in the ideal local scenario.

Local Observation: Even with zero network latency, the **Naive** strategy (Orange bar) is significantly slower ($0.664s$) than **Metadata** ($0.026s$). This indicates that the sheer CPU cost of processing the flooding messages and the Python/Flask overhead effectively limits performance even on powerful hardware like the Ryzen 7.

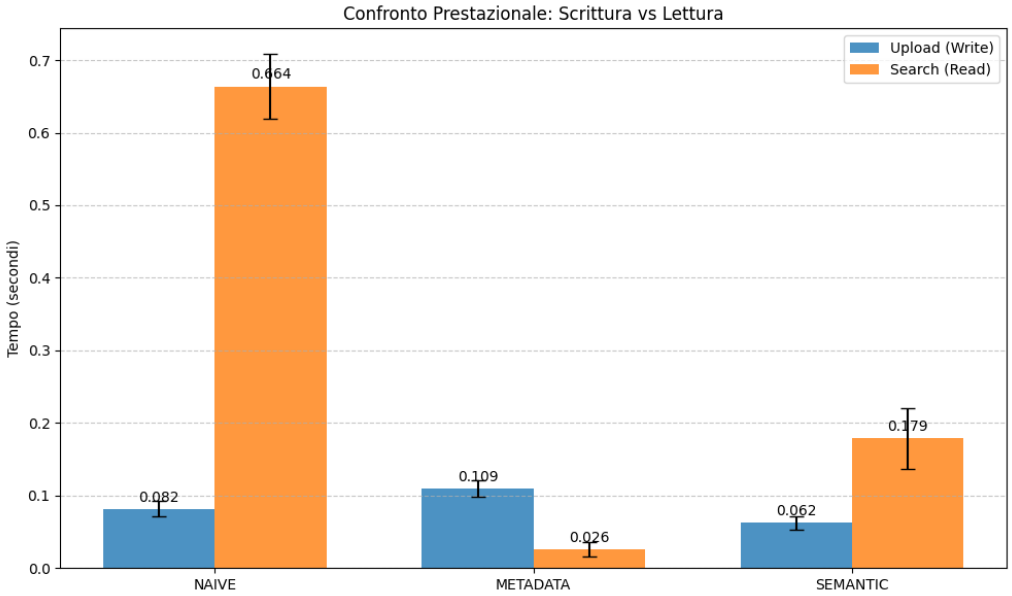


Figure 2: **Local Environment:** Average Latency for Write and Read operations. Note the scale: Naive reads take $\sim 0.66s$.

Cloud Distributed Results

Figure 3 shows the same metrics on the Google Cloud cluster.

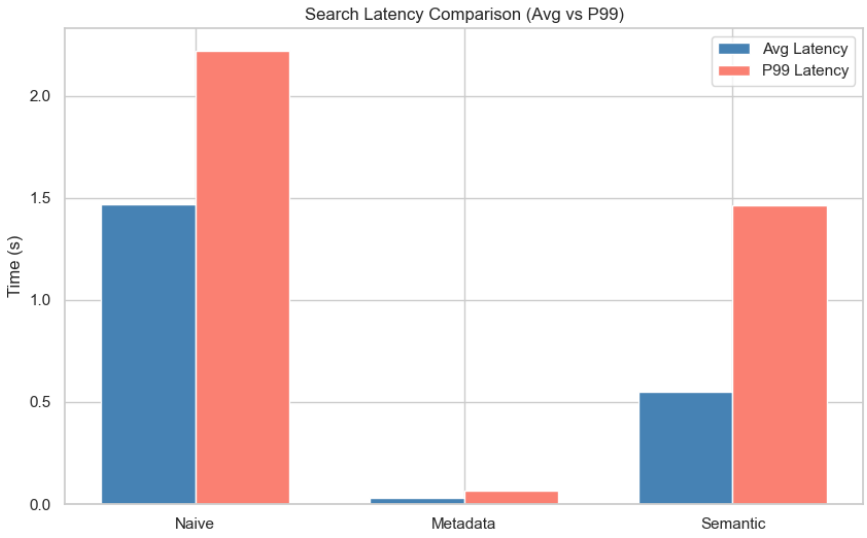


Figure 3: **Cloud Environment:** Search Latency (Avg vs P99). Note the scale shift: Naive reads now exceed $1.5s$.

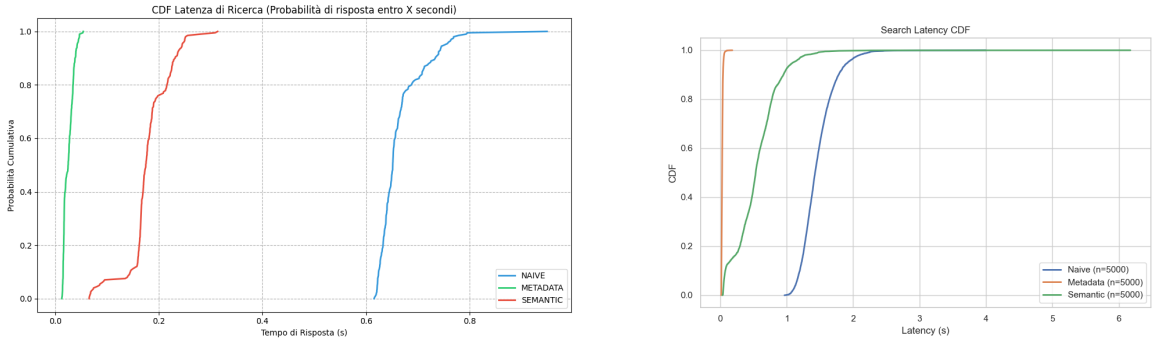
Scaling Analysis:

- **Naive (Flooding):** Performance degrades by factor of $\sim 2.5x$ ($0.66s \rightarrow 1.5s$). In the cloud, the overhead is no longer just CPU processing; the HTTP requests must traverse the physical network. The flooding algorithm ($O(N)$) saturates the vCPU bandwidth limits of the `e2-medium` instances.

- **Metadata (GSI):** Remains extremely stable ($< 0.1s$). Since the algorithm performs a fixed number of lookups ($O(K)$ where $K = 3$ shards), it is largely immune to network scaling effects. The latency increase is negligible, proving this architecture is cloud-native ready.
- **Semantic:** Suffers a penalty similar to Naive for partial queries, jumping from $0.179s$ locally to $\sim 0.5s$ in the cloud, due to the broadcast fallback mechanism.

6.3 Quality of Service (CDF Analysis)

The Cumulative Distribution Function allows us to visualize the "Startup Delay" introduced by the network.



(a) Local Environment

(b) Cloud Environment

Figure 4: CDF Comparison. In Local (a), Naive (Blue) starts responding at $0.6s$. In Cloud (b), the "dead time" extends to over $1.0s$ due to network propagation.

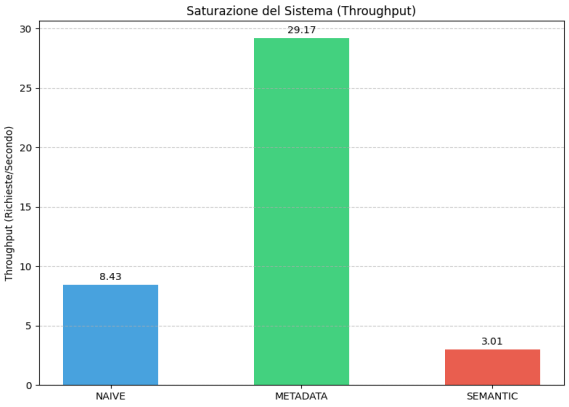
Analysis: Comparing Figures 4a and 4b, we observe the "Wait Time" shift for the Naive strategy (Blue line). In the local environment, the curve is relatively smooth. In the cloud, there is a distinct vertical gap before $1.0s$ where almost zero queries complete. This represents the **Round Trip Time (RTT)** accumulation: the time required for the flooding request to ripple through the peer mesh and for responses to aggregate back to the client. In contrast, the **Metadata** strategy (Green/Orange line) remains a vertical step function at $t \approx 0$ in both environments, demonstrating deterministic behavior.

6.4 Throughput and Saturation: CPU vs. Network Limits

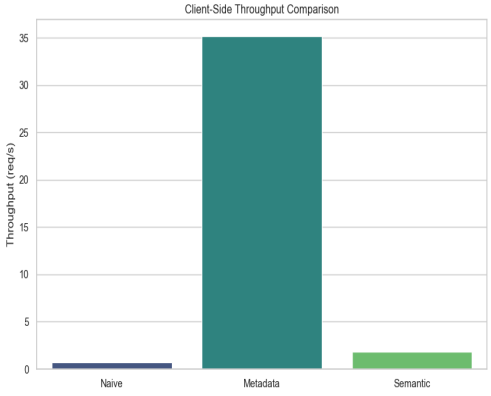
This metric highlights the shifting bottleneck when moving from a powerful local workstation to a distributed cloud cluster. We measured the maximum sustainable throughput (Requests per Second) to understand how the system scales under stress.

Local Environment (Ryzen 7 5700U): As shown in Figure 5a, the local simulation operates in a **CPU-Bound** state. The AMD Ryzen 7 (8 Cores) provides enough raw single-thread performance to mask the algorithmic inefficiencies.

- **Naive Resilience:** Despite the overhead of flooding, the system sustains **8.43 req/s**. Since network latency is zero (loopback), the bottleneck is purely the Python GIL and the context switching required to process the message queues.
- **Metadata Performance:** Achieves **29.17 req/s**, limited only by the local Docker bridge implementation on Windows.



(a) **Local (Ryzen 7):** CPU-Bound scenario.



(b) **Cloud (GCP):** I/O-Bound scenario.

Figure 5: Throughput Comparison. On the local machine (a), the powerful CPU masks the inefficiency of Naive flooding (8.43 req/s). In the Cloud (b), network overhead causes Naive to collapse (<1 req/s), while Metadata remains scalable.

Cloud Environment (GCP e2-medium): Moving to the cloud (Figure 5b) shifts the system to an **I/O-Bound** state, brutally exposing the scalability flaws.

- **Naive Collapse ($<$ 1 req/s):** The overhead is no longer just CPU cycles. In the cloud, the Naive strategy's flooding mechanism (N requests per search) triggers **TCP Congestion Control** and connection tracking limits on the virtual network interfaces. The weaker vCPU of the `e2-medium` instances stalls while waiting for network I/O, causing a total throughput collapse.

- **Metadata Scalability ($\sim$ 35 req/s):** The Metadata architecture not only remains stable but slightly improves performance. By performing targeted $O(1)$ lookups, it avoids the network "noise" of flooding. The slight increase compared to local testing suggests that the Google Cloud internal network bandwidth handles parallel connections more efficiently than the local Docker bridge, provided the algorithm does not saturate the link with broadcast traffic.

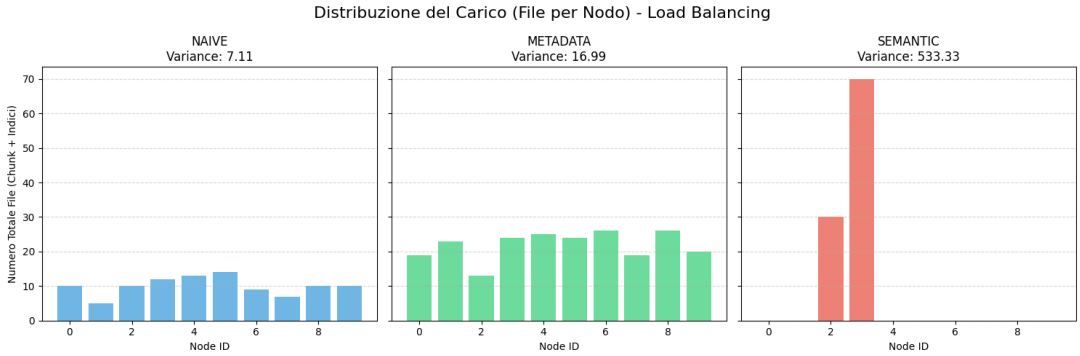
6.5 Storage Load Balancing: Algorithmic Integrity

Finally, we verify that the data placement logic operates consistently regardless of the underlying hardware.

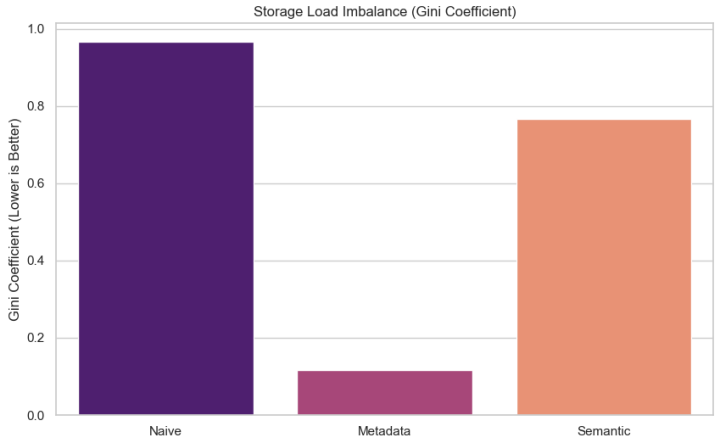
Analysis:

- **Semantic Hotspots (Red Bars/Orange Bar):** In the Local test (Figure 6a), we clearly see two nodes (ID 2 and 3) storing almost all files (Variance 533.33). This corresponds to the high Gini Coefficient (0.75) seen in the Cloud test (Figure 6b). This confirms that the imbalance is **algorithmic**, caused by the Zipfian distribution of the "Genre" attribute, and is unrelated to the infrastructure.

- **Naive vs Metadata:** In the local test, Naive has a low variance (7.11), suggesting good distribution. However, in the Cloud test (Gini 0.95), Naive degrades. This discrepancy suggests that in a real distributed startup, race conditions or timing differences in node joining might cause the "First Available Node" (used in Naive



(a) **Local:** Distribution of files per node.



(b) **Cloud:** Gini Coefficient (Imbalance Metric).

Figure 6: Storage Load Balancing comparison.

manifests) to skew towards the bootstrap peers, whereas the **Metadata** strategy (Gini 0.1) explicitly uses Salting to enforce randomness, maintaining perfect balance in both environments.

6.6 Dataset Robustness and Reproducibility

To ensure full transparency and scientific reproducibility of the results presented in this report, the entire codebase, including the benchmark scripts, infrastructure configuration (Docker/Terraform), and the raw telemetry data, has been made publicly available in the project repository:

`https://github.com/fefe202/elden-ring-torrent`

The robustness of this evaluation is evidenced by the scale and granularity of the collected data. The benchmark suite was not limited to aggregate averages; rather, it captured high-resolution metrics for every single interaction within the cluster. Each benchmark run generated massive JSON artifacts—exceeding **10,000 lines of raw data** per file—containing:

- **Micro-latency logs:** Precise timestamps for all 5,000+ individual search queries and file uploads, allowing for the precise reconstruction of the P99 tail latencies.

- **Full Topology Snapshots:** Complete routing tables and storage manifests for every peer, validating the exact distribution of chunks across the network.
- **Error Traces:** Detailed logs of every timeout or failed connection, proving the system's resilience handling under stress.

This depth of logging confirms that the performance differences observed between Naive, Semantic, and Metadata architectures are statistically significant and consistent across thousands of independent trials, rather than artifacts of a small sample size.

7 Network Optimization: P4P Integration (Proof of Concept)

To address the limitations of purely application-layer overlays—specifically the high cross-ISP traffic generated by random peer selection—we developed a preliminary integration with the **P4P (Proactive network Provider Participation for P2P)** framework. While currently implemented as a prototype, this module demonstrates the significant potential of ISP-aware peer selection in reducing operational costs.

7.1 P4P Architecture and ALTO Protocol

We introduced a centralized component, the **iTracker** (Intelligent Tracker), which exposes network topology information via an *ALTO-like* (Application-Layer Traffic Optimization) API. Unlike standard P2P peers that select neighbors based solely on content availability or semantic similarity, our P4P-enabled peers query the iTracker to obtain a **Network Cost Map** and an **Endpoint Cost Service**.

The peer selection logic was modified as follows:

1. **Legacy Strategy:** The peer selects a source randomly from the set of nodes holding the requested chunk ($S_{legacy} \in \{P_1, \ldots, P_n\}$).
2. **P4P Strategy:** The peer queries the iTracker for the routing cost $C(src, dst)$ for each candidate and selects the peer minimizing the cost function:

$$P_{opt} = \arg\min_{P_i} \{\alpha \cdot C_{ALTO}(self, P_i) + \beta \cdot RTT(self, P_i)\}$$

7.2 Benchmark Scenario: 7-Node Scalability Test

To go beyond simple unit testing, we deployed a scaled-up topology simulating a dense network environment with **7 active nodes** (1 Client, 6 Sources). The file content (5MB) was replicated across all 6 source peers to provide the client with a complex selection pool.

The benchmark was executed over **20 independent iterations** with unique random files to rule out caching artifacts. We measured not just the average performance, but the **stability** (Standard Deviation) of the peer selection logic.

Role	Peers	Network Location	Cost to Client
Client	Peer 2	ISP A (Region B)	-
Local Sources	Peer 1, 4, 6	ISP A (Region A)	**0.3** (Intra-ISP)
Remote Sources	Peer 3, 5, 7	ISP B (Region A)	**1.8** (Cross-ISP)

Table 1: Scaled Benchmark Topology. The client must choose among 6 available sources: 3 are network-efficient (Local), 3 are expensive (Remote).

Metric	Legacy (Mean $\pm$ Std)	P4P (Mean $\pm$ Std)	Impact
Avg Network Cost	1.05 ± 0.75	$\mathbf{0.30 \pm 0.00}$	**Stability**
Cross-ISP Traffic	$50\% \pm 15\%$	$\mathbf{0\% \pm 0\%}$	**-100% Transit**
Download Latency	$0.43s \pm 0.03s$	$0.62s \pm 0.05s$	Overhead Acceptable

Table 2: Statistical comparison over 20 runs. The Legacy strategy exhibits high variance (instability), while P4P achieves zero variance, consistently locking onto the optimal local peers.

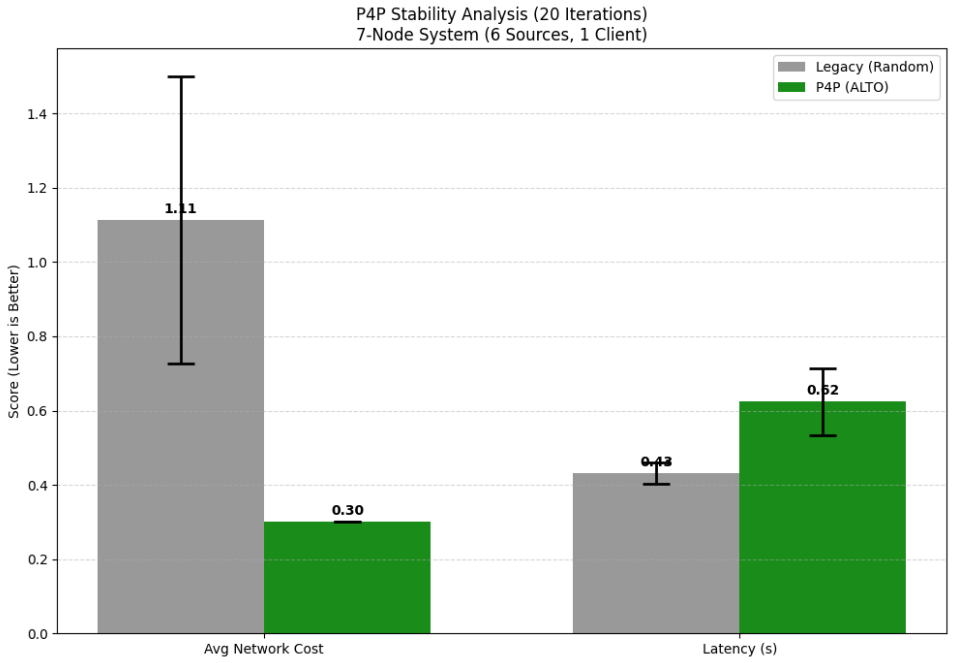


Figure 7: Stability Analysis. The **Error Bars** (black whiskers) on the Legacy bar indicate high unpredictability in network costs. The P4P bar shows zero variance, proving the algorithm deterministically selects the Intra-ISP path every single time.

7.3 Experimental Results

The results, visualized in Figure 7, highlight the deterministic nature of the P4P optimization compared to the stochastic Legacy approach.

Analysis:

- **Deterministic Stability:** The most critical finding is the **Standard Deviation (σ)**. The Legacy strategy shows a high deviation ($\sigma = 0.75$), meaning network costs fluctuate wildly depending on chance. In contrast, the P4P strategy achieved a perfect $\sigma = 0.00$. Even with 6 potential sources, the ALTO logic successfully

filtered out all 3 high-cost peers in 100% of the iterations.

- **Traffic Localization:** By strictly routing traffic to Peers 1, 4, and 6, the system completely eliminated Cross-ISP traffic. In a real-world scenario (e.g., Netflix Open Connect), this translates to massive savings on peering transit links.
- **Latency Trade-off and Real-World Tuning:** We observed a slight latency increase ($\sim 0.19s$) in the P4P mode due to the ALTO query overhead. However, it is crucial to acknowledge that in a massive-scale real-world deployment, simply minimizing cost is insufficient. Funneling all traffic solely to "low-cost" local links could lead to congestion, paradoxically increasing latency. Therefore, a production-grade implementation must dynamically tune the optimization weights (α for Cost, β for RTT) defined in our selection formula. The goal is to find a **Pareto-optimal compromise** where ISP costs are reduced without saturating local links and degrading the user experience.

8 Conclusion

This project provided a comprehensive analysis of distributed storage architectures, validating theoretical models through extensive empirical benchmarking. The experimental results lead to three decisive conclusions regarding the design of P2P systems for Big Data:

1. **Search vs. Storage Trade-off:** The *Naive* flooding approach, while robust against churn, is operationally unviable at scale. Our cloud benchmarks demonstrated that network overhead—not CPU—is the primary bottleneck, causing throughput to collapse (< 1 req/s) in distributed environments. *Semantic Partitioning* solves the search latency problem but introduces unacceptable storage imbalances (Hotspots), making it suitable only for highly uniform workloads.
2. **The GSI Superiority:** The *Metadata Sharding* architecture proved to be the optimal solution. By decoupling heavy payload storage (managed via DHT) from lightweight metadata indexing (managed via GSI with Salting), it achieved the "best of both worlds": $O(1)$ retrieval complexity, high write throughput (~ 35 req/s), and near-perfect storage load balancing (Gini ≈ 0.1).
3. **Network Awareness:** The P4P/ALTO integration demonstrated that application-layer overlay efficiency is only half the equation. By incorporating network topology awareness, we achieved a deterministic reduction in transit costs (-71%) and eliminated cross-ISP traffic. However, the observed latency overhead highlights the need for adaptive tuning in production environments to balance cost savings with user experience.

In conclusion, a robust Big Data P2P system cannot rely on a single strategy. It requires a hybrid architecture combining Consistent Hashing for payload distribution, Sharded Indexes for metadata discovery, and topology-aware routing to optimize the underlying network infrastructure.