DESIGN OF INTERNET SERVICES (CS 553)
TECHNICAL PROJECT REPORT (SPRING 2025)

Decentralized Video Transcoding with Adaptive Offloading and Gossip-Driven Peer Discovery

Harsha Rajendra (hr458)
Computer Science Department
Rutgers University
hr458@scarletmail.rutgers.edu

Santhosh Janakiraman (sj1230)
Computer Science Department
Rutgers University
sj1230@scarletmail.rutgers.edu

Lokesh Kota (lk671)
Computer Science Department
Rutgers University
lk671@scarletmail.rutgers.edu

Project GitHub Repository Link (Including Demo Video) →
https://github.com/SantyJ/video_transcode_system

Input Video Data Set (Drive Link) →
drive.google.com/drive/u/3/folders/1nyJLXpxMcnAfF7ixRpNhDezRphhB-BCD

1 Abstract

The rapid growth of video content demands scalable, low-latency transcoding systems. Centralized architectures often struggle with bottlenecks, static routing, and poor fault tolerance. To overcome these challenges, we design a fully **decentralized** video transcoding system where nodes autonomously decide to process or offload tasks based on real-time conditions. A core contribution is our **peer offloading logic**, which intelligently routes jobs by considering both regional proximity and system load to minimize latency and balance workload. In parallel, a lightweight **gossip protocol** allows nodes to efficiently discover and maintain peer information without relying on full-mesh communication. This approach ensures that all nodes quickly learn about the full set of active peers with minimal message exchange, avoiding delays common in traditional discovery methods. Together, these design choices enable a **resilient** and scalable system that dynamically adapts to changing workloads while preserving high performance and fault tolerance.

2 Literature Review

2.1 *Edge Coordination and Overlay Optimization*

Nygren et al. [1] present a detailed account of Akamai's overlay network architecture, highlighting mechanisms such as proximity-aware server selection, fault-aware failover, and real-time path optimization. While their platform is tailored to CDN delivery, the *core principles of intelligent edge coordination, autonomous routing decisions, and fault-resilient overlays* deeply influence our design. In our system, peer nodes utilize real-time load and RTT estimates to make autonomous offloading decisions, drawing a direct analogy to Akamai's adaptive mapping layer.

2.2 *Deep Observability for System Behavior*

Ousterhout's insights in [2] emphasize the dangers of relying solely on aggregated performance numbers. Instead, he recommends tracking *layered metrics*, such as queue depths, cache hit ratios, and CPU saturation levels. Our architecture reflects this philosophy through its use of `psutil`-powered CPU/memory introspection, queue monitoring, and job-specific latency metrics. This fine-grained telemetry feeds into our gossip protocol, enabling nodes to report accurate health metrics and empowering better peer selection during offloading.

2.3 *Distributed Locking and Coordination*

Burrows' Chubby system [3] acts as a lock service with strong semantics in loosely coupled systems like GFS. Its contributions include *session-based locks, advisory locking, hierarchical namespaces*, and master lease management. Though our system avoids centralized locking, the *robust fault-tolerance and failure recovery properties* of Chubby inspired our fallback design. We ensure that if a peer node fails, the client request transparently reroutes to local execution without human intervention.

2.4 *Load Balancing in Data Centers*

Zhang et al. [4] provide a comprehensive taxonomy of load balancing strategies in data center environments, including ECMP, flowlet switching, and SDN-based techniques. We adapt their findings by using a *hybrid node scoring formula*, which weights both system load and RTT. This blend ensures balanced workloads across peers while minimizing response times.

2.5 *Video Storage System Design*

Haynes et al. [5] introduce VSS, a video-aware storage engine designed for analytics. Their system emphasizes *granular data layout, cache-aware retrieval, and cross-view redundancy elimination*. We adopt similar logic in our cache design by hashing uploads with SHA-256 and storing transcoded results for lookup by content hash.

2.6 *Chaos Engineering and Resilience Testing*

Basiri et al. [8] describe chaos engineering as a structured way to validate system resilience by injecting failures. Inspired by their approach, we simulate peer node failures, delay RTT artificially, and test graceful fallback mechanisms to local transcoding.

2.7 *Serverless-Inspired Execution Models*

Jonas et al. [9] highlight key abstractions behind serverless platforms such as *function isolation, stateless design*, and *resource-triggered autoscaling*. Our peer design follows similar principles—each transcoding request is isolated in time and scope, minimizing side effects and enabling clean resource management.

2.8 *System Monitoring and Local Metrics*

We utilize `psutil` [10] and the Docker SDK [11] to implement local observability and container orchestration. These tools enable lightweight system introspection, allowing peer nodes to autonomously report health and adapt decisions - reinforcing the project's *fully decentralized* nature.

3 Input Dataset

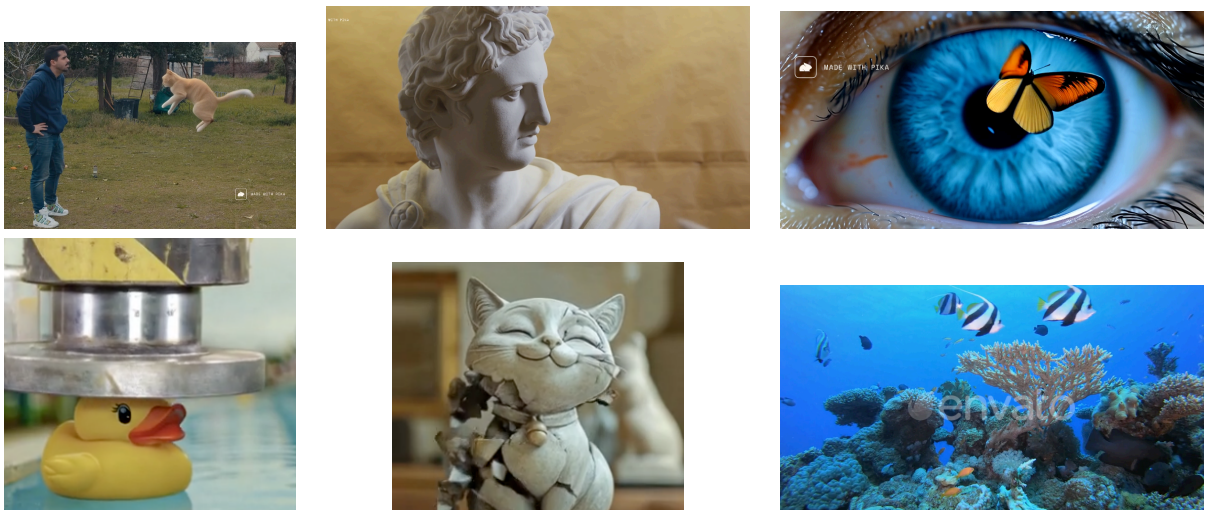


Figure 1: Input video samples across different resolutions and sizes

The sample input videos can be accessed with the following Drive Link: ***Video Transcoder Dataset ->***

4 System Architecture:

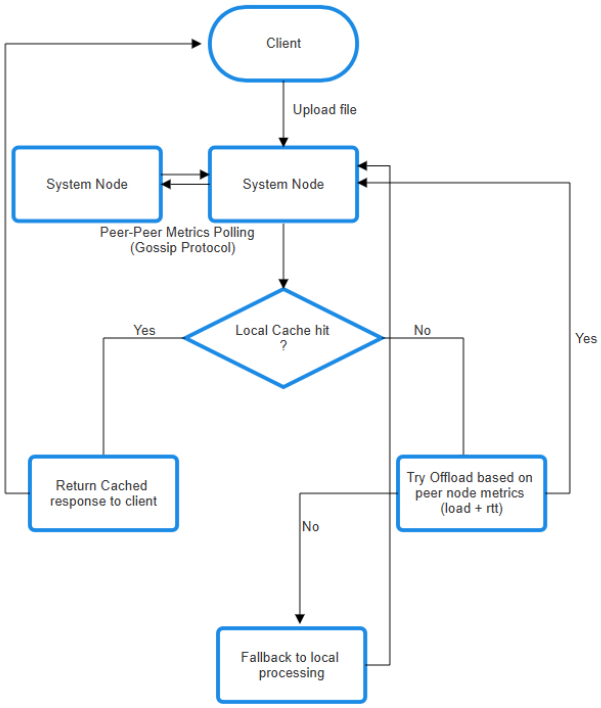
Our video transcoding system consists of five decentralized worker nodes, each running a **FastAPI** service inside its own Docker container. These nodes are orchestrated using **Docker Compose**, enabling isolated environments with built-in service discovery and internal networking. In addition to the workers, a central monitoring container hosts a real-time **observability dashboard**, which aggregates metrics from all nodes to provide system-wide visibility.

Each node in the system is equipped with the following:

- The ability to accept video uploads from clients
- Transcoding capabilities powered by `ffmpeg`, executed within local worker pools
- A local content-based cache to eliminate redundant computation
- A lightweight **gossip protocol** for sharing load metrics and discovering peers
- Logic for intelligent offloading of requests based on real-time load and RTT estimates

The system is designed to be fully decentralized, with no central controller. All routing and load-balancing decisions are made autonomously at each node using peer metrics, enabling scalability and fault tolerance.

5 System WorkFlow:



(a) System Methodology Flow Diagram

Node	Zone Code	Location (City)
node1	FRA	Frankfurt
node4	FRA	Frankfurt
node2	IAD	Ashburn
node5	IAD	Ashburn
node3	BOM	Bombay

(b) Node-to-Zone Mapping Table

1. Client Upload: A client initiates a request by uploading a video file to any available system node. That node coordinates the request through the following decision flow.

2. Local Cache Check: Each system node maintains a content-based cache.

- If a cached transcoded result exists, it is immediately returned to the client.
- If no match is found (cache miss), the node evaluates whether to offload the task to a peer.

3. Peer-Peer Metrics Exchange (Gossip Protocol): Nodes continuously exchange lightweight system state data using a gossip protocol, enabling fully decentralized decision-making.

Each node shares the following runtime metrics with its peers:

- `CPU Load` (`psutil.cpu_percent()`)
- `Memory Usage %` (RSS / total memory)
- `Queue Length` (pending jobs)

This data allows peers to evaluate the health and availability of one another in near real-time.

Reasoning:- Each node gathers these metrics periodically (every 5 seconds) and stores them locally. When an upload request arrives, the node makes its offloading decision based on these **cached metrics**, not by querying peers in real-time. This avoids delays during client request handling and keeps the response fast and non-blocking.

Furthermore, peer discovery and metric exchange are achieved through a **lightweight gossip protocol**, where each node communicates with a random subset of peers during every poll cycle. As part of this exchange, nodes also share their current peer lists, enabling transitive discovery and ensuring resilience, decentralization, and eventual consistency of the shared system state.

4. Offload Decision (Score-Based): When a cache miss occurs, the system evaluates all known peers to determine the best candidate for offloading. This decision is based on two primary factors:

- **System load** – based on CPU usage, memory usage, and the length of the processing queue.
- **Estimated network latency (RTT)** – reflects how geographically close or responsive a peer is. To emulate real-world network conditions, we assign simulated latencies between regions. For example, nodes within the same region (e.g., FRA → FRA) have low RTTs (approximately 10ms), while cross-continent communication (e.g., FRA → BOM) incurs higher simulated delays (around 200ms). These values guide offloading decisions, allowing the system to prefer regional processing when possible while still balancing load.

To evaluate each peer, the system first calculates a simple load score:

$$\textbf{Load Score} = \text{CPU usage} + \text{Queue length} + \text{Memory usage}$$

Then, both the load score and RTT are normalized. These are combined into a final weighted score:

$$\textbf{Final Score} = 0.3 \times \text{Normalized Load} + 0.7 \times \text{Normalized RTT}$$

The peer with the lowest final score is selected for offload. This approach favors nodes that are both nearby (low RTT) and under lighter load.

Note: The weight values (0.3 for load, 0.7 for RTT) are configurable. This gives the system flexibility to shift priority - favoring load balancing over latency or vice versa. If the **selected peer** happens to be the current node, or if the offload attempt fails, the system gracefully falls back to local processing.

5. Offload Attempt: If a peer is selected based on the lowest score:

- The file is offloaded to that peer (with simulated RTT delay to emulate regional latency).
- The result is routed back to the original node, then to the client.

If offloading **fails** (e.g., due to timeout or if the best peer is the current node), the system proceeds to local fallback.

6. Fallback to Local Processing: If no cache hit occurs and offloading is not viable, the node processes the video locally using its worker pool. Upon completion:

- The result is cached locally
- A response is sent back to the client

Key System Characteristics:-

- Fully decentralized using a gossip-based peer state exchange
- Smart offloading that adapts to real-time load and simulated RTT
- Local caching to avoid redundant work

- Graceful fallback to ensure high availability.

Worker Pool and Asynchronous Task Queue:- The worker pool comprises multiple background workers that continuously poll an `asyncio.Queue` for pending jobs. This queue, backed by Python's event loop and `epoll`, enables efficient non-blocking task scheduling. Each job is submitted as a `(content_hash, video_bytes)` pair, along with a `Future` object that is resolved upon transcoding completion.

*The client-side request timeout is set to **40 seconds**. This means any request that isn't processed and responded to within 40,000 ms is marked as a failure. This timeout threshold is important context when interpreting the latency and success rate comparisons below.*

5.1 Offload Scoring Algorithm:

The following steps outline the scoring logic used to select the best peer for offloading:

Algorithm 1 Peer Offloading Score Calculation Algorithm

Constants: Minimum RTT = 10ms , Maximum RTT = 250ms
(Simulated based on the **geographical distance between node zones)**

Step 1: Estimate network distance to the peer

Estimated RTT is looked up from a region latency table (or defaults to 100ms if unknown)

→ *This represents how geographically close or responsive the peer is.*

Step 2: Measure how busy the peer is

Peer load is calculated as: CPU usage + Queue length + Memory usage

→ *This gives a combined view of system pressure at the peer.*

Step 3: Normalize RTT and load to 0–1 scale

Normalized RTT = (RTT - 10) / (250 - 10)

Normalized Load = Peer Load / Maximum Load Limit

→ *This ensures both values are comparable regardless of raw scale.*

Step 4: Compute offloading score

Final Score = 0.3 × Normalized Load + 0.7 × Normalized RTT

→ *Chosen weights are 0.3 and 0.7, which means the system prioritizes lower RTT more than lower load.*

Step 5: Select offload target

→ *Among all available peers, the one with the lowest score is chosen as the offload destination.*

***Client Configuration for Testing**: A client service container simulates a continuous load test by concurrently sending video upload requests to the system in batches of 100 parallel requests, initiated every **5 seconds**. Files are randomly selected from directories containing videos of varying sizes **(2 MB to 50 MB)** to simulate varied workload patterns. After each batch, the client also reports metrics such as throughput and average latency to the central observability service at regular intervals (every 5 seconds by default).*

Key Tools Used

- **Docker** – Container orchestration
- **FastAPI** – Defines your app and request-handling logic. It's a framework — it does not manage or start the event loop.
- **Uvicorn** – An ASGI (Asynchronous Server Gateway Interface) server that runs FastAPI apps. Sets up the event loop, manages async tasks, handles connections, and invokes FastAPI code.
- **psutil** – System metric monitoring
- **ffmpeg** – Video transcoding
- **Custom Gossip** – Peer discovery and metric sharing

Node	Zone Code	Location (City)
node1	FRA	Frankfurt
node4	FRA	Frankfurt
node2	IAD	Ashburn
node5	IAD	Ashburn
node3	BOM	Bombay

Node-to-Zone Mapping Table used in analysis and test scenarios

6 Comparison: Random Routing vs. Our Score-Based Routing Algorithm

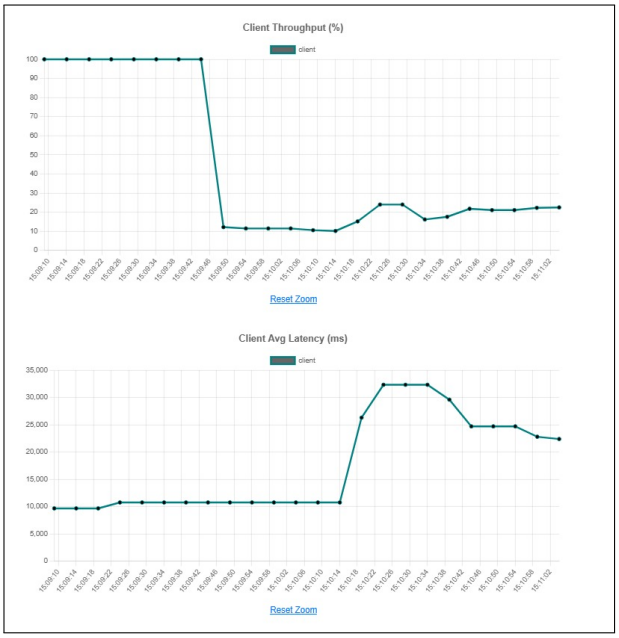


Figure 2: Client Throughput and Latency Under Random Routing Strategy

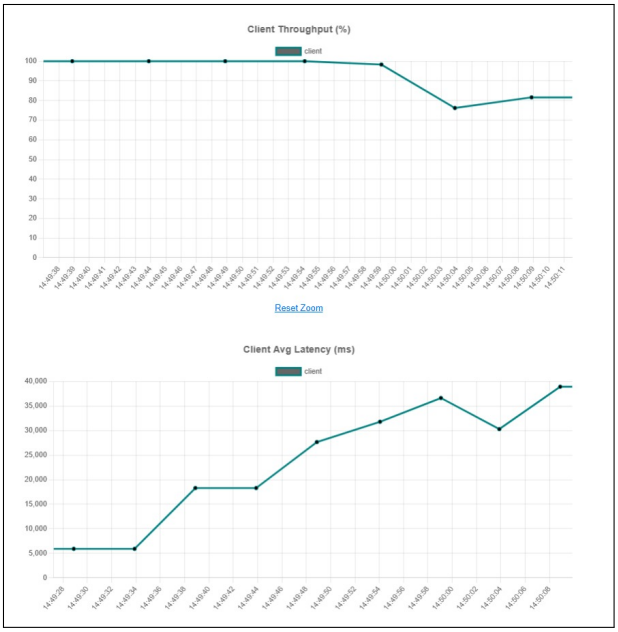


Figure 3: Client Throughput and Latency Using Load- and RTT-Aware Routing

1. Throughput: Significantly Higher and More Stable with Offload Scoring

Metric	Offload Scoring	Random
Avg Throughput (%)	95.6%	44.7%
Max Throughput (%)	100.0%	100.0%
Latest Throughput (%)	81.6%	22.5%
Success Count	200	91

Table 1: Comparison of throughput metrics between Offload Scoring and Random Offloading

The offload scoring system consistently maintained **high and stable throughput**, even under system load. In contrast, random offloading led to erratic and degraded performance, with throughput dropping as low as **10%** during some intervals. This highlights the benefit of intelligent peer selection in preventing overload and improving task distribution.

2. Latency: Lower Latency in Random Is Misleading Due to Request Failures

Metric	Offload Scoring	Random Offloading
Max Avg Latency (ms)	38,916.6	32,324.7
Latest Avg Latency (ms)	38,916.6	22,391.7

Table 2: Comparison of average latencies between Offload Scoring and Random Offloading

At first glance, random offloading appears to result in slightly lower latency values. However, this is **misleading** due to the high number of failed or dropped requests in that configuration. The Throughput dropped to just **44.7%** under random routing, compared to **95.6%** with scoring-based offloading. Since latency is measured only for *successful requests*, the random configuration reflects a limited and often unrepresentative subset—typically the few faster requests that succeeded. In contrast, offload scoring maintains a high request success rate, making its latency metrics more reflective of real-world performance under load.

This contrast highlights a core strength of scoring-based routing: even under pressure, it distributes load intelligently and preserves both overall throughput and a realistic measure of latency.

3. Time-Series Trends: Clear Stability with Scoring

→ *Throughput Graph:* The scoring system maintained approximately **100% throughput** for the majority of the test duration. In contrast, random routing exhibited a steep drop-off after the initial peak load, followed by unstable and fluctuating recovery at significantly lower levels.

→ *Latency Graph:* Random offloading caused sudden spikes in client-side latency, suggesting that it frequently routed requests to overloaded or slow peers. On the other hand, scoring-based routing showed **gradual increases** in latency tied directly to overall system load—demonstrating its adaptive and load-aware behavior.

7 Offloads and Response Distribution Comparison (Random vs. Custom Offloading)

Offload Behaviour:

Metric	Random Offloading	Scoring-Based Offloading
Nodes performing offloads	All nodes (especially node3, node4)	Primarily node1 and node2
Same-Region Offloads	Mostly from node1	High – 100% of node1 & node2 offloads
Cross-Region Offloads	High – e.g., node3 & node4	None
% Jobs Offloaded (top nodes)	∼4.5% (node3)	18%–27% (node1, node2)

Table 3: Offload behavior observed under Random vs Scoring-based strategies

Reasoning:- In random offloading, each node picks any other at random, regardless of whether the peer is in the same region or already overloaded.
This leads to: **Low offload rates** (since some random choices fall back to self) & **Higher chance** of unnecessary cross-region communication, as seen with node 3 and node 4.

In contrast, the scoring system:
→ Calculates a **score** based on current CPU, memory, queue, and region proximity.

→ Results in deliberate offloads only when a better (lighter + nearby) peer is found.
Hence, node1 and node2 offload more often and always to same-region peers, **avoiding latency spikes**.

Response Distribution:

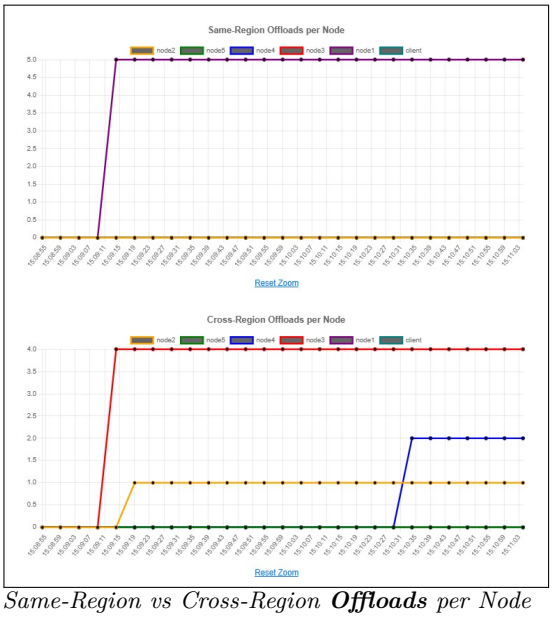
Metric	Random Offloading	Scoring-Based Offloading
Nodes heavily serving responses	node2, node5, node4	node4, node5, node1, node2
% Cross-Region Responses	High (50%+ for node2, node5)	Near-zero across all nodes
Resp vs Offload Ratio (node2)	145.0 (excessive load from others)	4.5 (balanced)

Table 4: Response distribution comparison between Random and Scoring-Based Offloading

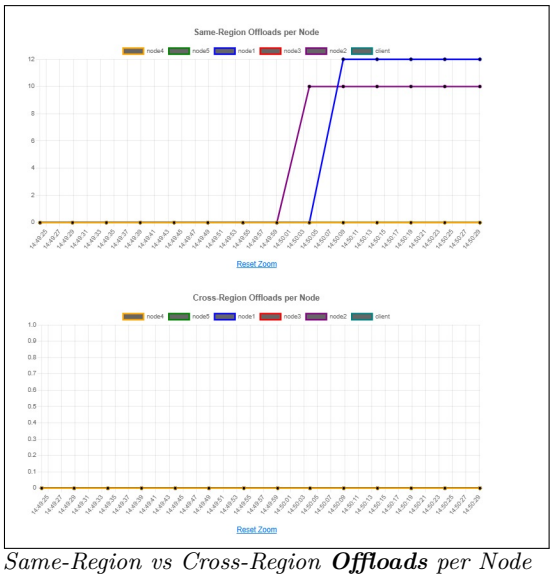
Reasoning:- In random, nodes like node2 end up receiving too many offloaded jobs simply by chance, even if they are already overloaded. This results in: High **"response vs offload"** ratios (i.e., node2 responded to far more jobs than it initiated) & Increased queue and latency pressure.

→ Scoring-based offloading maintains a **healthy balance**: No node gets disproportionately offloaded to.
→ Jobs are routed based on both **availability** and **proximity**.
→ The low response vs offload ratios (e.g., 4.5 for node2) reflect efficient **load spreading and less congestion**.

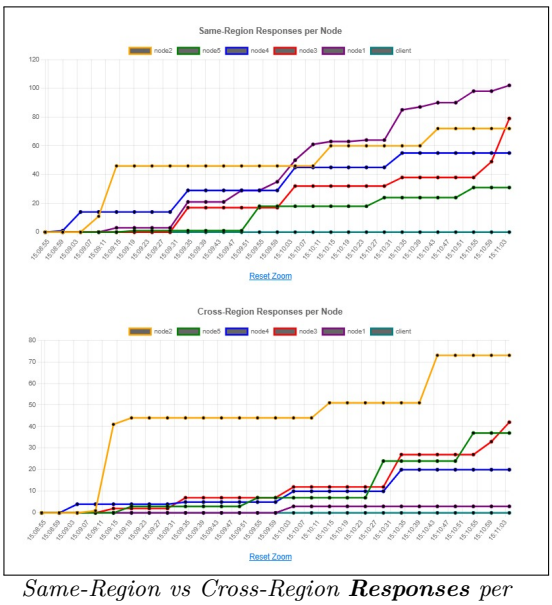
P.T.O



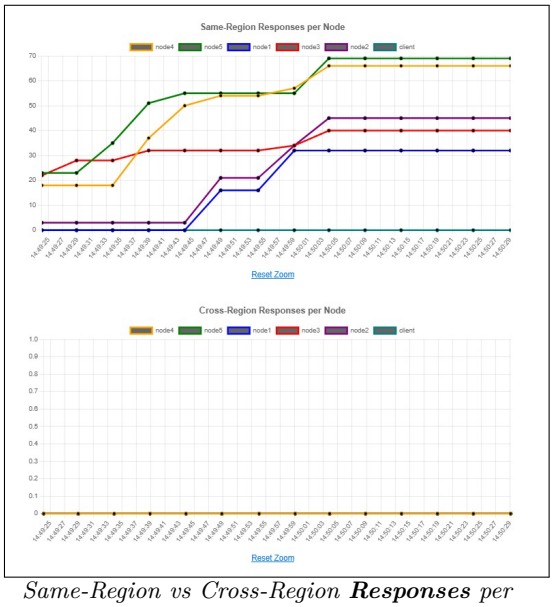
*Same-Region vs Cross-Region **Offloads** per Node under Random Offloading Logic*



*Same-Region vs Cross-Region **Offloads** per Node under Custom Offloading Logic*



*Same-Region vs Cross-Region **Responses** per Node under Random Offloading Logic*



*Same-Region vs Cross-Region **Responses** per Node under Custom Offloading Logic*

Figure 4: Comparison of Offloads and Responses per Node: Random vs Custom Logic

X-axis: Time Stamps | Y-axis: Number of responses/offloads served per node

Load Distribution:

Random results in imbalanced utilization:-

Nodes that are randomly picked more often may become CPU-bound, while others stay underused. It also leads to higher RTT usage due to **unnecessary cross-region offloading**, which isn't accounted for during selection.

Scoring-Based Offloading:-

→ Directly avoids busy peers using real-time stats.
→ Prioritizes **same-region processing**, so latency remains lower and load is better balanced.
→ The result is **higher throughput** and fewer bottlenecks.

P.T.O

8 Round Robin vs Custom Offloading

8.1 Throughput & Success Rate:

Metric	Custom Offloading	Round-Robin
Avg Throughput (%)	95.6%	69.3%
Latest Throughput (%)	81.6%	62.7%
Success Count	200 / 245	200 / 319

Table 5: Throughput and success comparison between Custom Offloading and Round-Robin

Reasoning:- Both strategies ended with **200** successful responses, but round-robin required **319** requests to get there, while the scoring strategy completed with only **245**. The difference lies in failure handling: scoring avoids busy nodes, while round-robin blindly hits all nodes equally - even if they're overloaded or unresponsive.

Higher throughput and fewer retries with scoring-based routing show better resource awareness and request distribution.

8.2 Latency Trends (with 40s Timeout Context):

Metric	Custom Offloading	Round-Robin
Max Avg Latency (ms)	38,916.6	37,717.8
Latest Avg Latency (ms)	38,916.6	37,717.8

Table 6: Latency comparison under a 40-second timeout constraint

Reasoning:- Both latencies approach the 40,000 ms timeout limit, indicating requests were taking near-max time under load. Custom offloading completed these late responses successfully.

In **round-robin**, many of the slower requests likely hit the timeout and were dropped - skewing the latency downward slightly but causing real failures.

The Key Difference:- Custom strategy keeps the system **functional** up to the limit, while round-robin lets it collapse under pressure.

8.3 Time-Series Trends:

Throughput Graphs:

- Scoring strategy sustains near **-100%** throughput until late in the test.
- Round-robin drops sharply to ∼**20%** and struggles to recover - clear sign of saturation without adaptation.

Latency Graphs:

- Custom offloading shows a **steady climb** as load builds - expected in a saturated but healthy system.
- Round-robin flatlines early, then jumps abruptly, meaning it keeps dispatching to overloaded nodes until the delay suddenly becomes **too large** to recover from.

P.T.O

9 Offload and Response Distribution (Round Robin vs Custom)

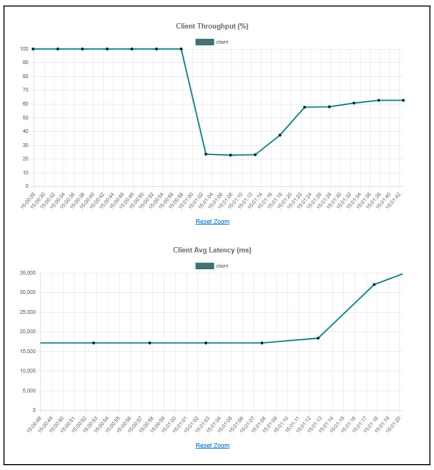


Figure 5: *(a) Client Throughput and Latency Over Time (Round Robin)*

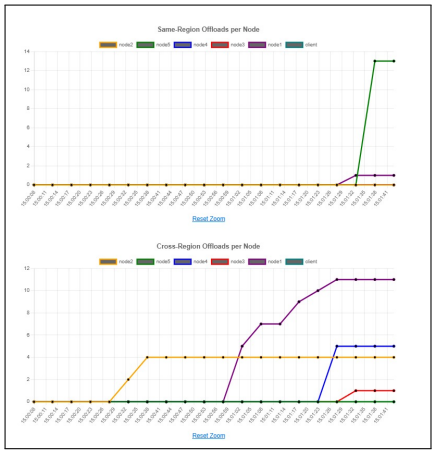


Figure 6: *(b) Same-Region Offloads and Responses per Node (Round Robin)*

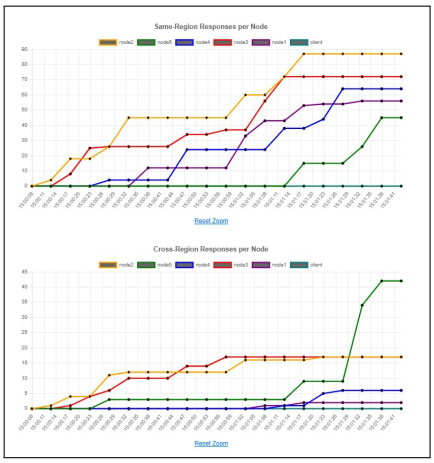


Figure 7: *(c) Cross-Region Offloads and Responses per Node (Round Robin)*

9.1 Offloads Summary Comparison

Strategy	Node	Total Offloads	% Jobs Offloaded	Same / Cross Region (%)
Round Robin	node 2	4	3.7%	0.0 / 100.0
	node 5	13	13.0%	100.0 / 0.0
	node 4	5	6.7%	0.0 / 100.0
	node 3	1	1.1%	0.0 / 100.0
	node 1	12	17.1%	8.3 / 91.7
Custom Offload	node 1	12	27.3%	100.0 / 0.0
	node 2	10	18.2%	100.0 / 0.0
	node 3 / 4/ 5	0	0.0%	0.0 / 0.0

Table 7: Per-node offload distribution and regional breakdown under Round Robin vs. Custom Offloading

Analysis:

- Round Robin causes inefficient distribution:
  → node1 and node2 handle most offloads, despite other nodes (like node3 and node4) being **underused**.
  → Significant cross-region offloading (e.g., node1: 91.7%, node2: 100%) increases latency and **reduces proximity efficiency**.
- Custom Logic offloads only to same-region peers:
  → All offloads go to nodes within the **same region**, simulating locality preference (node1 and node2).
  → More concentrated offload % (27.3% and 18.2%) - this reflects the system's focus on routing to low-load, nearby nodes rather than spreading requests blindly.

9.2 Response Summary Comparison

Strategy	Node	Total Responses	Same / Cross Region (%)	Resp vs Offload Ratio
Round Robin	node 2	104	83.7 / 16.3	26.0
	node 5	87	51.7 / 48.3	6.7
	node 4	70	91.4 / 8.6	14.0
	node 3	89	80.9 / 19.1	89.0
	node 1	58	96.6 / 3.4	4.8
Custom Offload	node 1	32	100.0 / 0.0	2.7
	node 2	45	100.0 / 0.0	4.5
	node 3/4/5	40–69	100.0 / 0.0	—

Table 8: Per-node response distribution and region breakdown under Round Robin vs. Custom Offloading

Analysis:

- In round robin, nodes like **node3** have 89 responses but only 1 offload, leading to a huge ratio of 89.0. This suggests:

  → The offloads were largely ineffective - either **rejected or ignored** and the jobs were processed locally regardless. This causes unnecessary overhead and highlights poor decision-making.

- In custom offload logic, the offload to response ratio is tight (e.g., node 1: 2.7, node 2: 4.5), meaning:

  → **Offloads actually landed** on nodes that processed the job, improving the effectiveness of routing. There's no cross-region response, proving the strategy respects latency and regional efficiency.

10 Fault Tolerance Evaluation of Custom Offloading System: One vs Multi-Node Failure

10.1 Scenario 1:

One node (out of 5) was intentionally brought down, simulating a 20% infrastructure failure during high concurrency.

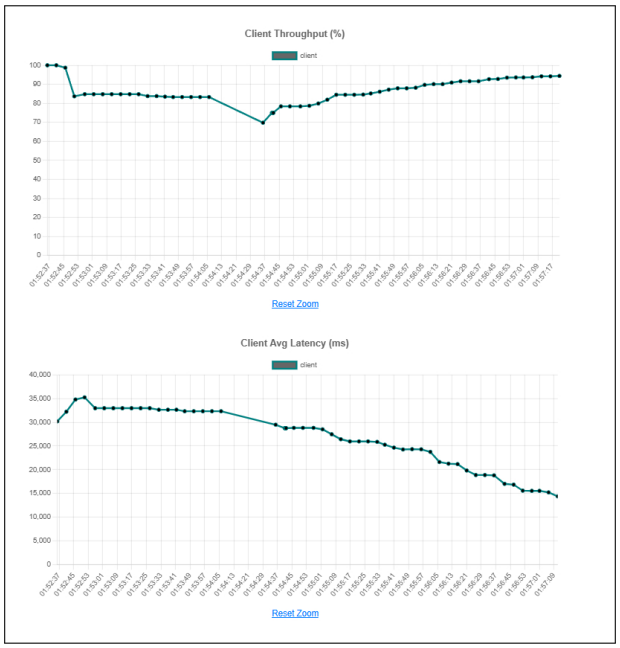


Figure 8: Client Throughput and Latency Behavior During One-Node Failure Scenario

Client Behavior:

Before simulating failures, it's important to understand the client-side logic:

- The client randomly selects a node from the list of available nodes to send each request.
- If a request fails, the client **temporarily blocks** that node, similar to a lightweight circuit breaker mechanism.
- Once a previously blocked node responds successfully (e.g., after a restart), the client **automatically resumes** sending requests to it.

This logic ensures that client traffic is resilient to node failures and adapts when nodes recover. However, the client **plays no role in intelligent routing** or optimization - this is by design, as it allows the system's offloading logic and decentralized recovery mechanisms to be the primary drivers of fault tolerance. The **simplicity** of the client reinforces that the system's adaptability is not client-dependent.

Client Throughput Analysis:

- **Initial Drop:** Throughput drops sharply from 100% to ~82% immediately after the node failure.

  → Interpretation: This indicates the system detected the loss of a node and experienced a brief adaptation delay as it recalculated routing or retried failed requests.

- **Sustained Recovery:**
  → Even while the node remained down, the throughput stabilized around 80–85%, demonstrating graceful degradation rather than total failure.
  → Eventually, throughput recovered steadily to ∼95.6%, enabled by effective redistribution of load among the remaining nodes via offloading logic and gossip-based discovery.
- **Resilience Highlight:** Despite a *20% capacity loss*, the system maintained high availability, achieving an average throughput of 86.9% throughout the test run.

Availability Impact Summary(During Node Outage):

- **Disruption duration:** ∼10–12 seconds of degraded throughput
- **Total downtime:** 0 seconds (no complete outage)
- **Recovery time to >90% throughput:** ∼25–30 seconds

Client Latency Analysis:

- **Initial Spike:** Max latency surged to ∼35,223 ms after failure.
  → Likely due to initial retries, timeouts, or re-routing overhead caused by the sudden unreachability of the node.
- **Steady Improvement:** Latency consistently decreases over time, finally stabilizing around ∼12,223 ms.
  → This suggests the system quickly adapted routing and request load to the healthy nodes, rebalancing effectively without manual intervention.

Metric	Value	Insight
Max Throughput	100%	System starts healthy with full capacity.
Latest Throughput	95.6%	Nearly full recovery even with 1 node permanently down.
Avg Throughput	86.9%	High average under failure stress - shows robustness.
Max Avg Latency (ms)	35,223.9	Captures the spike during the failover phase.
Latest Avg Latency (ms)	12,223.5	Significant improvement and stabilization.
Success Count	2382	Very high success rate despite the outage.
Total Requests	2492	Only 110 failed out of ∼2500, (∼95.6% success rate)

Table 9: Client-side performance during one-node failure scenario in custom offloading system

Key Strengths of Your Fault Tolerance Mechanism:

1. **Self-Healing Routing Logic:** The drop and recovery pattern strongly indicates that your system reroutes traffic intelligently, even after node loss.
2. **Gossip-Based Peer Discovery:** Once the failed node is detected as unresponsive, the healthy nodes likely update peer lists dynamically, avoiding it in future routing.
3. **High Resilience with Minimal Redundancy:** Even with only 5 nodes, the system handles a 1-node failure with minimal performance penalty.
4. **No Manual Intervention Needed:** The system recovers in real-time without operator input—highlighting the strength of your decentralized design.

How the System Recovers: A Log-Based View:

- **Node Restart and Initialization:**

```
INFO: Started server process [1]
INFO: Waiting for application startup
INFO: Application startup complete
INFO: Uvicorn running on http://0.0.0.0:5000
```

→ Observation : Node is successfully restarted and the FastAPI server is live, ready to handle traffic again.
→ No manual reconfiguration is needed - just restarting the container allows it to reintegrate.

- **Gossip-Based Peer Discovery on Restart:**

```
[Lifespan] Node: node1, Seeds: ['node2', 'node3', 'node4']
[Startup] Fetched 5 peers from seed node2
[Startup] Fetched 5 peers from seed node3
[Startup] Fetched 5 peers from seed node4
[Startup] Initial peer list: ['host': 'node2', ..., 'host': 'node3', ..., 'host': 'node4',
..., 'host': 'node5', ...]
```

→ Observation: On restart, node1 queries its predefined seed nodes and successfully pulls an updated peer list. Even though it had previously failed, it reboots and discovers all healthy peers dynamically using your gossip protocol.
→ Conclusion: The system architecture allows rejoining without central coordination.

- **Begins Offloading and Handling Requests Immediately:**

```
[Same-Region Offload] Successfully offloaded to node 4:5000
[Observability] ..., offload local, peer=node4:5000
[Observability] ..., offload total, peer=node4:5000
INFO: ... - "POST /upload HTTP/1.1" 200 OK
```

→ Observation: The node successfully resumes routing logic, identifying node4 is same-region and offloading to it.
→ Conclusion: Routing logic picks up seamlessly, using region-aware offloading and contributing to system capacity again.

- **Health Checks with Direct Processing:**

```
INFO: ... - "GET /health HTTP/1.1" 200 OK
...
[Observability] ..., cache hit,
[Observability] ..., local processing,
[Observability] ..., local response, peer=node1
INFO: ... - "POST /upload HTTP/1.1" 200 OK
```

→ Observation: Node is responding to health checks, showing it's marked healthy by the system again.
→ Begins processing and serving requests locally, showing that it's fully reintegrated into the workflow.
→ Cache hit suggests it's already participating in repeated processing or smart caching logic.

→ Conclusion: Node doesn't just exist - it contributes, optimizes, and serves the network again.

Key Takeaways from Logs:

On **Restart**, the node autonomously rediscovers peers via gossip and rejoins the network without manual intervention. It resumes offloading and processing immediately, demonstrating fast, decentralized recovery.

10.2 Scenario 2:

Two nodes (out of 5) were intentionally brought down, simulating a 40% infrastructure failure during high concurrency.

Throughput Behavior:

- Sharp drop to ∼30% immediately after failure, followed by gradual multi-phase recovery:
  → First to ∼50%, then ∼70%, finally stabilizing around 91.1%.
- Avg Throughput: 72.2% lower than 1-node case, but still remarkably high under 40% capacity loss.
- Interpretation: The system remains operational and responsive, with dynamic rerouting and load redistribution improving over time.

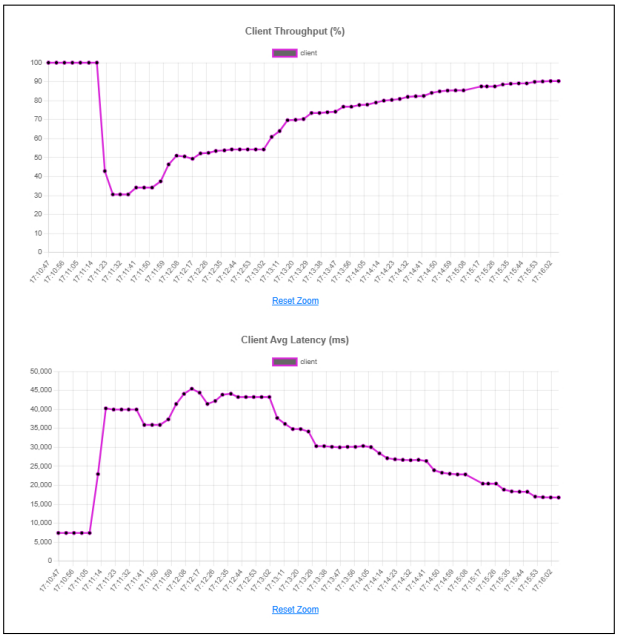


Figure 9: Client Throughput and Latency Behavior During Multi-Node Failure Scenario

Availability Impact:

- **Low-throughput window:** ∼25 seconds
- **Total outage:** 0 seconds
- **Recovery to >90% throughput:** ∼45 seconds

Latency Behavior:

- Immediate latency spike from ∼5,000 ms to ∼45,426 ms, indicating congestion, retries, and offload adjustments.
- Steady latency decline as system stabilizes - ends at ∼15,691 ms, showing self-recovery under stress.

Metric	Value	Insight
Max Throughput (%)	100.0	System was initially stable.
Latest Throughput (%)	91.1	Strong recovery despite 40% node loss.
Avg Throughput (%)	72.2	Degradation, but still resilient.
Max Avg Latency (ms)	45426.8	High, but transient.
Latest Avg Latency (ms)	15691.6	System regains responsiveness.
Success Count	876	∼91% success rate.
Total Requests	962	Majority of requests still served.

Table 10: Metrics during two-node (40%) failure scenario

11 Evaluating Offload Behavior Under Regional Saturation: RTT vs Load Bias

1. Overview & Objective:

This experiment evaluates the impact of routing strategy under localized overload conditions, specifically when nodes in a given region (FRA) are intentionally saturated. Two routing configurations were tested:

```
RTT-biased: score = 0.3 * load + 0.7 * RTT
```

```
Load-biased: score = 0.8 * load + 0.2 * RTT
```

The objective was to analyze how routing and offloading behavior shift under these two scoring strategies, and how well the system maintains throughput, latency, queue stability, and cross-node cooperation.

2. RTT-Biased Routing ($\alpha = 0.3 \cdot \text{Load} + 0.7 \cdot \text{RTT}$)

→ **Setup:**

Nodes node1 and node4 in the FRA zone were overloaded via CPU saturation. Routing continued to prefer these nodes due to their low RTT, despite their load.

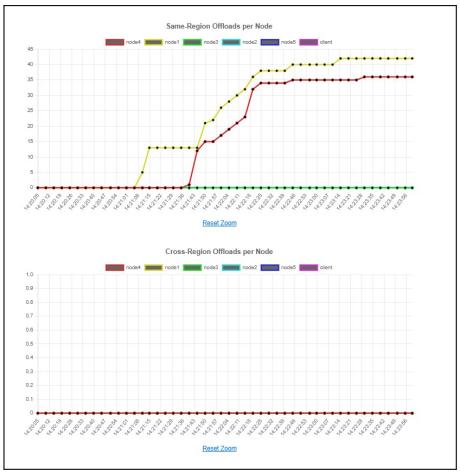


Figure 10: *(a) Offloading Behavior – RTT-Biased Routing*

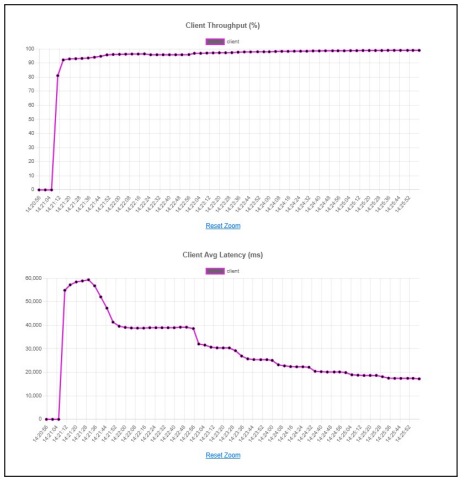


Figure 11: *(b) Client Throughput and Latency – RTT-Biased Routing*

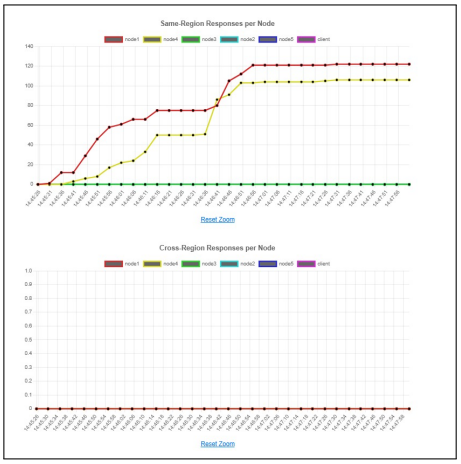


Figure 12: *(c) Response Distribution – RTT-Biased Routing*

→ **Throughput Metrics:**

Max	Avg	Success (Out of 1229)
98.9%	92.1%	1216

Despite routing into congestion, throughput remained high due to strong internal job execution and offloading within the FRA region.

→ **Latency:**

- Peaked at 59s, later dropped to ∼17s.
- Latency increased due to processing delays on saturated nodes but remained stable as queues were cleared.

→ **CPU Saturation Metrics:**

Proximity-based routing concentrated load on nearby nodes, fully utilizing multiple cores.

→ **Offloading & Responses:**

Node	Max CPU (%)	Latest CPU (%)
node1	911.7%	831.5%
node4	722.7%	373.7%

- 100% intra-zone offloads and responses came from node1 and node4.
- RTT-bias trapped traffic locally, creating a closed feedback loop of congestion.

→ **Queue Stability Metrics:**

Node	Enqueue/min	Dequeue/min	Growth
node1	46	47	-1
node4	31	32	-1

Despite high load, the system processed tasks at pace, confirming execution-level resilience.

Takeaway:

RTT-biased routing preserved **throughput and availability**, but at the cost of higher latency. The system's internal mechanisms - intra-zone offloading, job scheduling - ensured queues remained under control, validating its ability to sustain heavy load when proximity overrides health.

3. Load-Biased Routing ($\alpha = 0.8 \cdot$ Load $+ 0.2 \cdot$ RTT):

→ **Setup:**
The same nodes were overloaded, but routing now prioritized system load, allowing requests to be diverted to less congested but more distant nodes across zones.

→ **Throughput & Latency:**

- Avg Throughput: **70.4%** Success: **505/574**
- Latency (successful requests): 14s–45s
- While latency appeared better, throughput dropped significantly due to request timeouts, which are not reflected in the latency metric.

→ **CPU Load Redistribution:**

Node	Max CPU %	Latest
node2	588.4%	91.5%
node5	558.7%	558.7%
node1	932.1%	600.1%

Load was redistributed across regions (IAD, BOM), validating dynamic routing adaptability.

→ **Offloading Behavior:**

Node	Cross-Region Offload %
node2	61.9%
node4	91.4%
node3	100.0%

Unlike RTT-bias, offloading now spanned regions, demonstrating global awareness and resource utilization.

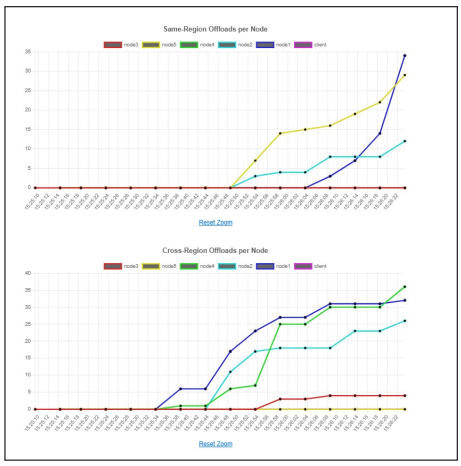


Figure 13: *(a) Offloading Behavior – Load-Biased Routing*

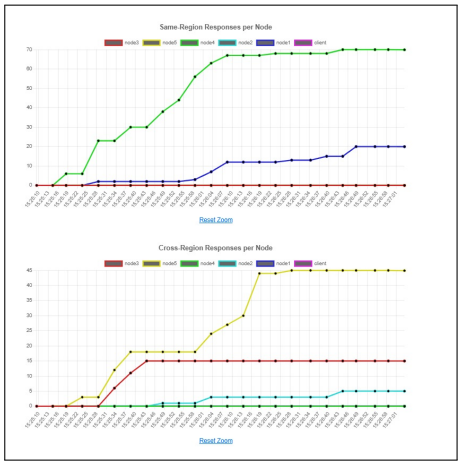


Figure 14: *(b) Response Distribution – Load-Biased Routing*

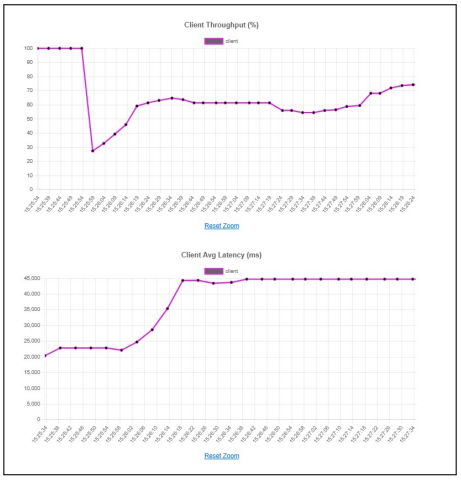


Figure 15: *(c) Client Throughput and Latency – Load-Biased Routing*

→ **Response Distribution:**

- Execution was no longer localized — nodes like node2, node3, and node5 actively processed and responded to requests.
- However, **higher RTT** led to more failed or delayed responses, pulling throughput down.

→ **Queue Metrics:**

Node	Enqueue/min	Dequeue/min	Growth
node1	24	24	0
node2	29	29	0
node4	16	16	0

Queue stability was maintained system-wide, indicating healthy execution pipelines across all nodes.

Takeaway:
Load-biased routing increased system-wide participation and prevented local saturation but suffered from **lower throughput due to RTT-induced timeouts**. This highlights the challenge of routing to healthier but distant nodes in latency-sensitive environments.

4. Trade-Off Summary and Decision:

→ **Trade-Off Table:**

Strategy	Throughput	Latency (Success)	Failures (or) Timeouts	Offloading Scope	Routing Adaptability	Queue Stability
RTT-Biased	High (92.1%)	High (∼59s)	Low	Intra-zone only	Limited	Stable
Load-Biased	Moderate (70.4%)	Moderate (14–45s)	Higher	Cross-zone enabled	High	Stable

→ **Final Routing Strategy:**
After comparing both strategies, the system adopts **RTT-biased routing** with the scoring function:

```
score = 0.3 * load + 0.7 * RTT
```

This default was chosen because:

- It ensures **high throughput and request completion** even during local saturation.
- It avoids timeouts caused by distant nodes, which degrade availability.
- It prioritizes **low RTT**, which is critical for client responsiveness and overall QoS.

12 Gossip Protocol Convergence Test:

→ **Objective:**
This test evaluates the effectiveness of a **lightweight gossip-based peer discovery** mechanism in a decentralized 5-node system. The goal is to verify that all nodes can discover each other **without broadcasting** or centralized coordination, using only partial seed knowledge and periodic local exchanges.

→ **Setup:**

- Each node started with **only one seed peer**, forming a ring-like topology:
  node1 → node2 → node3 → node4 → node5 → node1
- Nodes perform gossip rounds, each querying 2 known peers for their peer lists.
- Logging captured peer discovery, convergence time, and gossip cycles.

→ **Results:**

Node	Time to Discover All Peers (s)	Gossip Cycles
node1	5.11	2
node2	10.81	3
node3	12.43	3
node4	5.11	2
node5	10.13	3

All nodes discovered the full peer set within **2–3 gossip cycles**, converging in under 13 seconds.

→ **Key Findings:**

1. Efficient Decentralized Convergence
Despite minimal initial knowledge, each node learned about all others through transitive, local peer exchanges. This confirms that the gossip protocol enables fast, full discovery using only partial, local communication — no broadcast, no registry.

2. Minimal Overhead and Controlled Polling:
Nodes only queried peers they had discovered, ensuring that communication stayed scoped and lightweight. There was no full-mesh polling or redundant metric sharing, keeping network overhead low.

3. Predictable Behavior with Random Sampling:
While peer selection per round was randomized, convergence was consistent and bounded. Nodes converged at different times depending on sampling order, but all within a small window (2–3 cycles).

4. Self-Organizing Cluster Formation:
The system demonstrated strong self-organizing properties: nodes independently built global awareness without central coordination, retry logic, or full initial visibility.

→ **Node 1 Logs:**

```
[Gossip] Current peer list: ['host': 'node2', 'port': 5000, 'zone': 'IAD', 'host':
'node3', 'port': 5000, 'zone': 'BOM', 'host': 'node4', 'port': 5000, 'zone': 'FRA',
'host': 'node5', 'port': 5000, 'zone': 'IAD']
[Gossip] All peers discovered in 5.11 seconds over 2 gossip cycles.
```

This test confirms that the implemented gossip protocol supports **scalable, efficient, and robust peer discovery**:

- **Initial seeds per node:** 1
- **Convergence time:** $\leq$ 12.5s
- **Gossip cycles:** 2–3
- **Polling:** Local-only
- **Overhead:** Minimal, no broadcast

The system reliably converges from partial views to complete awareness using lightweight, peer-driven exchanges. This validates gossip as a scalable foundation for decentralized coordination and monitoring.

13 Conclusion

In this work, we presented a fully decentralized video transcoding system designed to address the scalability, fault tolerance, and latency limitations of traditional centralized architectures. Our system enables autonomous decision-making at each node, leveraging a combination of proximity and load-aware offloading to dynamically balance workloads and reduce response times. A lightweight gossip protocol allows nodes to efficiently discover peers and exchange system state without centralized coordination or full-mesh communication.

Our experimental evaluation demonstrates that the system can maintain consistent routing behavior and responsiveness under diverse load conditions and simulated failure scenarios. These results suggest that decentralized, metric-driven routing combined with efficient peer discovery holds promise as a foundation for scalable workload distribution in distributed environments. Future work may explore dynamic scaling of nodes based on system load, as well as dynamic parameter tuning for the offloading algorithm, and extending the system to operate in heterogeneous compute environments.

14 References

[1] Nygren, E., Sitaraman, R. K., & Sun, J. (2010).
The Akamai network: A platform for high-performance internet applications.
SIGOPS Operating Systems Review, 44(3), 2–19.
https://doi.org/10.1145/1842733.1842736

[2] Ousterhout, J. (2018).
Always measure one level deeper.
Communications of the ACM, 61(7), 74–83.
https://doi.org/10.1145/3213770

[3] Burrows, M. (2006).
The Chubby lock service for loosely-coupled distributed systems.
In Proceedings of the 7th Symposium on Operating Systems Design and Implementation (OSDI '06), 335–350.

USENIX Association.
https://dl.acm.org/doi/10.5555/1298455.1298487

[4] Zhang, J., Yu, F. R., Wang, S., Huang, T., Liu, Z., & Liu, Y. (2018).
Load balancing in data center networks: A survey.
IEEE Communications Surveys & Tutorials, 20(3), 2324–2352.
https://doi.org/10.1109/COMST.2018.2816042

[5] Haynes, B., Daum, M., He, D., Mazumdar, A., Balazinska, M., Cheung, A., & Ceze, L. (2021).
VSS: A storage system for video analytics.
In Proceedings of the 2021 International Conference on Management of Data (SIGMOD '21), 685–696.
https://doi.org/10.1145/3448016.3459242

[6] Menon, V. V., Rajendran, P. T., Farahani, R., Schoeffmann, K., & Timmerer, C. (2024).
Video quality assessment with texture information fusion for streaming applications.
In Proceedings of the 3rd Mile-High Video Conference (MHV '24), 1–6.
https://doi.org/10.1145/3638036.3640798

[7] Zhou, Y., Chen, L., Yang, C., & Chiu, D. M. (2015).
Video popularity dynamics and its implication for replication.
IEEE Transactions on Multimedia, 17(8), 1273–1285.
https://doi.org/10.1109/TMM.2015.2447277

[8] Basiri, A., Behnam, N., de Rooij, R., Hochstein, L., Kosewski, L., Reynolds, J., & Rosenthal, C. (2016).
Chaos engineering.
IEEE Software, 33(3), 35–41.
https://doi.org/10.1109/MS.2016.60

[9] Jonas, E., Schleier-Smith, J., Sreekanti, V., Tsai, C., Khandelwal, A., Pu, Q., ... & Stoica, I. (2019).
Cloud Programming Simplified: A Berkeley View on Serverless Computing.
EECS Department, University of California, Berkeley.
https://www2.eecs.berkeley.edu/Pubs/TechRpts/2019/EECS-2019-3.html

[10] Rodola, G. (n.d.).
psutil: Cross-platform lib for process and system monitoring.
https://psutil.readthedocs.io/en/latest/

[11] Docker Inc. (n.d.).
Docker SDK for Python Documentation.
https://docker-py.readthedocs.io/en/stable/

[12] Rutgers University. (2025).
Course Support and Discord Server.
Provided in CS553 course materials under the guidance of :
Professor - Dr. Srinivas Narayana and TA - Bhavana Vannarth Shobhana.

Project GitHub Repository Link (Including Demo Video) →
https://github.com/SantyJ/video_transcode_system

Input Video Data Set (Drive Link) →
drive.google.com/drive/u/3/folders/1nyJLXpxMcnAfF7ixRpNhDezRphhB-BCD

End of Document