

Diagnosing Tail Latency in Multi Node Object Storage

Sneha Gullapalli

Distributed Storage and Observability

Abstract

Object storage can meet its average latency target while a small but important fraction of requests experiences delays large enough to trigger application timeouts. These outliers are difficult to diagnose because a request crosses gateways, metadata services, networks, storage nodes, and background maintenance domains. This paper presents a practical method for diagnosing tail latency in multi-node object storage. The method starts with operation-specific percentile distributions, forms a slow-request cohort, segments it by workload and topology dimensions, and then compares distributed traces with a matched healthy cohort. It distinguishes service time from queueing time and requires counterevidence before assigning cause. The paper also explains how retries, erasure coding, repair traffic, compaction, and hot partitions can amplify rare component delays. The approach is intended as a diagnostic framework; no proprietary architecture or production benchmark is reported.

Keywords object storage, tail latency, distributed tracing, percentiles, observability, performance diagnosis

The Central Finding

A storage system is not fast because its mean request latency is low. It is dependable when latency remains predictable for the operations and workloads customers actually run. In a large distributed service, a rare pause on one component becomes common at the request level when each operation depends on several components. Dean and Barroso described this amplification as the tail at scale: temporary high-latency episodes that appear minor on one machine can dominate service responsiveness as fan-out grows [1].

Object storage adds its own sources of variation. A request may require authentication, metadata lookup, placement resolution, network transfer, data or parity access, checksum work, and acknowledgment. Reads can involve multiple fragments. Writes may wait for a durability threshold. Repair, rebalancing, garbage collection, compaction, or device recovery can contend with foreground traffic. The visible delay is therefore an end-to-end property, not a single disk measurement.

Why Averages Hide the Incident

Suppose 99 requests finish in 20 ms and one finishes in 2,000 ms. Their mean is approximately 40 ms, which looks acceptable even though one user experienced a two-second stall. A percentile preserves that distinction: p50 describes a typical request, while p99 marks the latency below which 99 percent of observations fall. Histograms make these distributions aggregatable, but their bucket boundaries must be chosen around meaningful service objectives; a coarse upper bucket can conceal the difference between 300 ms and several seconds [2].

Percentiles also require context. Combining GET, PUT, DELETE, and LIST operations creates a distribution that describes no single path. Mixing 4 KB objects with 4 GB objects is equally misleading. The first diagnostic task is to define a comparable cohort using operation, object-size range, tenant or workload class, region, cluster, software version, and time window. Node identity belongs in drill-down data, not necessarily in an unbounded metric label.

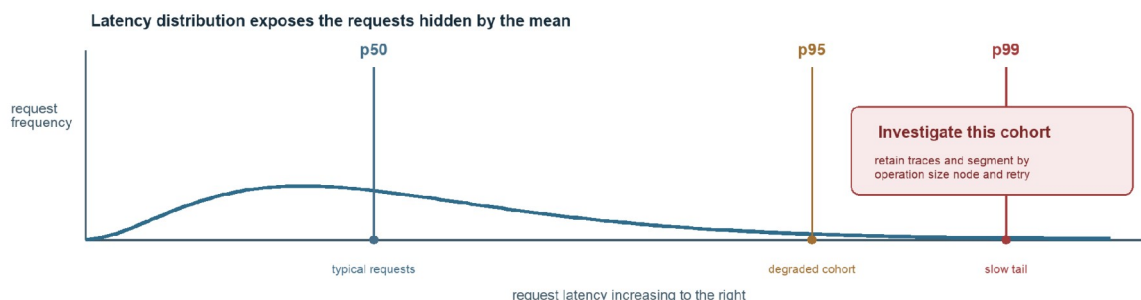


Figure 1 Percentiles separate typical requests from the slow diagnostic cohort

Request Path and Diagnostic Flow

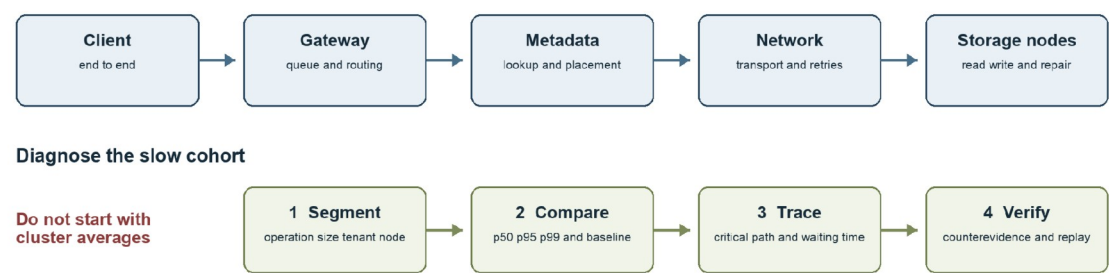


Figure 2 End to end object request path and evidence driven tail investigation

The upper path in Figure 2 identifies telemetry boundaries. Each service should record request identity, operation, status, retry count, bytes, and a span covering its own work. The lower path is the investigation sequence. It begins with the slow cohort, not a search across all cluster metrics. OpenTelemetry traces are useful because a trace records the path of one request and its child spans expose how time is distributed across services [3]. A long span identifies where time was observed, however, not automatically where the delay originated. A gateway span can grow while waiting for a downstream node, and a storage span can grow while waiting in a shared executor queue.

A Four Step Diagnostic Method

1 Segment the slow cohort

Select requests above the relevant percentile or service objective and compare their dimensions with requests from the same interval that completed normally. Look for concentration by operation, object size, shard or partition, gateway, storage node, device class, tenant, and retry count. A cluster-wide problem should affect many independent paths. Concentration on one placement group or node suggests a narrower resource or topology issue.

2 Separate waiting from work

For each layer, record queue delay separately from active service time. CPU saturation, connection-pool exhaustion, thread-pool limits, device queues, and network buffers can all create waiting without increasing the time spent performing the underlying operation. Queue depth and utilization should be aligned with the same time window as the slow traces. High utilization alone is not proof; the relevant question is whether the slow cohort waited behind the constrained resource.

3 Compare traces against a matched baseline

Choose healthy requests with the same operation, size, placement, and approximate arrival time. Compare span structure, retries, remote calls, and gaps between spans. Trace context must propagate across gateway and internal calls; otherwise, temporal correlation can be mistaken for causal order. Sampling policy also matters. Uniform sampling may discard rare outliers, so tail-based sampling should retain traces that exceed a duration threshold or contain errors, while protecting the collector from overload.

4 Test the explanation

A defensible diagnosis predicts additional observations. If repair traffic caused p99 read latency, affected nodes should show overlap between repair I/O and queued foreground reads, while comparable nodes without repair should remain healthier. If a hot partition caused the tail, request concentration and queueing should follow the partition when ownership changes. If evidence does not follow the proposed mechanism, the explanation should be downgraded.

Common Tail Patterns

Observed pattern	Likely mechanisms	Discriminating evidence
One node dominates outliers	device degradation, repair, local queue	node cohort, device wait, repair overlap
All nodes degrade together	shared network, control plane, synchronized work	cross-node timing and dependency spans
Large objects only	transfer, checksum, fragmentation, bandwidth	latency versus bytes and per-stage throughput

Observed pattern	Likely mechanisms	Discriminating evidence
Retries hide errors	packet loss, overloaded target, stale placement	attempt count, first-attempt status, reroute path

Table 1 Tail patterns and evidence that separates competing causes

Mitigation Without Creating More Tail

Diagnosis and mitigation must be separated. Hedged reads can reduce latency by issuing a second request after a delay, but they also add load and can worsen contention if triggered too aggressively. Load shedding protects the system when queues exceed a safe bound, but requires workload priorities. Background repair and rebalancing should use explicit budgets tied to foreground latency rather than fixed maximum throughput. The correct response depends on the proven mechanism.

Erasure-coded storage deserves special care because a read can depend on a subset of fragments and slow fragments can determine completion time. Research on object-storage latency models shows that caching, chunk count, disk operations, and queue discipline all shape the response distribution [4]. Redundancy can help when an independent alternate fragment is available, but correlated congestion or a shared network bottleneck limits its value. Any mitigation should be evaluated at p95, p99, timeout rate, resource cost, and fairness across tenants.

Validation Plan

A useful test matrix varies object size, operation mix, concurrency, node count, placement skew, repair load, and injected delay. Each run should preserve raw histograms and representative traces, not only summary averages. Fault injection should include one slow device, packet loss, metadata delay, executor saturation, and a background-repair burst. The framework succeeds when it localizes the constrained layer, distinguishes cause from propagated waiting, and produces the same conclusion when the incident is replayed. It should also report insufficient evidence when trace context or cohort dimensions are missing.

Conclusion

Tail latency is a distribution and dependency problem. Effective diagnosis starts by defining the affected request cohort, preserving percentiles, and following the request path with correlated evidence. Queue time must be separated from active work, and every proposed cause must predict observations outside the original symptom. This method turns an intermittent customer complaint into a reproducible engineering investigation while avoiding the false certainty produced by averages and unsegmented dashboards.

Declaration of AI Assisted Writing

OpenAI Codex [6] was used during preparation of this manuscript to assist with development of the initial draft, organization of the technical discussion, creation of the original diagrams, language editing, reference discovery, and document formatting. The author supplied the subject matter, professional context, technical direction, and review criteria. The author subsequently reviewed and revised the manuscript, verified the cited sources and technical statements, and accepts full responsibility for the accuracy, originality, interpretation, and final content of the work.

References

- [1] J. Dean and L. A. Barroso, "The Tail at Scale," Communications of the ACM, vol. 56, no. 2, pp. 74–80, Feb. 2013, doi: 10.1145/2408776.2408794.
- [2] Prometheus Authors, "Understanding Metric Types," Prometheus Documentation. [Online]. Available: https://prometheus.io/docs/tutorials/understanding_metric_types/.
- [3] OpenTelemetry Authors, "Observability Primer," OpenTelemetry Documentation. [Online]. Available: <https://opentelemetry.io/docs/concepts/observability-primer/>.
- [4] Y. Su, D. Feng, Y. Hua, and Z. Shi, "Understanding the Latency Distribution of Cloud Object Storage Systems," Journal of Parallel and Distributed Computing, vol. 128, pp. 71–83, June 2019, doi: 10.1016/j.jpdc.2019.01.008.
- [5] C. Hu et al., "Come Hell or Still Water: Alleviating Tail Latency in Cloud Block Store," in Proc. 23rd USENIX Symposium on Networked Systems Design and Implementation (NSDI 26), 2026. [Online]. Available: <https://www.usenix.org/conference/nsdi26/presentation/hu-chaolei>.
- [6] OpenAI, "Codex," OpenAI Developers. [Online]. Available: <https://developers.openai.com/learn/codex>.