AVID-FP Store: A Fault-Tolerant, Integrity-Verified Distributed Object Store.

Manoj Dattatreya Myneni
University of Illinois at Chicago
mmyne@uic.edu

Snehitha Chowdary Pallinti
University of Illinois at Chicago
sp200@uic.edu

***Abstract*—Verifiable storage systems must simultaneously provide durability, fault tolerance, and light-weight integrity checks, properties that traditional cloud object stores deliver only in pieces. Building on the theoretical foundations of Asynchronous Verifiable Information Dispersal with FingerPrinting (AVID-FP), this project introduces the AVID-FP Object Store, the first complete, open-source implementation of that protocol. Each object is striped into m data and $(n - m)$ parity shards via SIMD-accelerated Reed–Solomon coding; 64-bit homomorphic fingerprints and per-shard SHA-256 hashes form a compact cross-checksum (FPCC) that can be validated after retrieving any m shards. A two-phase Echo/Ready gossip carried over gRPC commits dispersals without a coordinator, guaranteeing safety and liveness under up to $f = n - m$ Byzantine node failures.**

The Go-based prototype, packaged as distroless Docker images with Prometheus and Grafana observability, sustains an aggregate write throughput of $\approx$ 108 MB s^1 for 1 GiB objects in a 3-of-5 configuration while adding $< 6\,\%$ latency overhead for integrity verification. Scripted test suites for both 3-of-5 and 4-of-6 deployments confirm successful reconstruction when faults do not exceed f and correct aborts when integrity violations do. Comprehensive YAML/ENV/CLI configuration, snapshot and GC tooling, and zero-downtime rolling-upgrade scripts make the system immediately deployable for research and production use of verifiable, fault-tolerant object storage.

Index Terms
Distributed storage; Byzantine fault tolerance; Reed–Solomon erasure coding; Homomorphic fingerprints; AVID protocol; Information dispersal; Data integrity; Quorum consensus; gRPC; Go; Prometheus; Docker; Grafana.

I. Introduction and Motivation

Modern data-intensive applications, ranging from content-delivery platforms and collaborative editing suites to edge-deployed AI pipelines, depend on the assumption that any object they write to cloud or on-premises storage can later be read back exactly as written, even in the face of hardware failures, software bugs, malicious insiders, or network partitions.

Commercial object stores such as Amazon S3, Google Cloud Storage, and Azure Blob achieve impressive durability by geo-replicating data, yet they delegate most of the end-to-end integrity burden to clients: the server returns a single checksum per object, and it is left to the application to download the entire payload, recompute the hash, and compare—an operation whose cost scales linearly with the object size and whose security depends on the server staying honest.

At the other end of the spectrum, cryptographic proofs of retrievability (PoR) and provable data possession (PDP) provide mathematically strong guarantees, but at the price of homomorphic encryption, large audit tags, or the assumption of a single honest server. A gap therefore remains for a system that combines (i) space-efficient redundancy, (ii) Byzantine-fault tolerance, and (iii) fast, bandwidth-proportional integrity verification, all while being lightweight enough to run in containerised micro-clusters at the edge.

The theoretical foundation for such a system already exists. Asynchronous Verifiable Information Dispersal (AVID) couples erasure coding with a two-phase gossip pattern—*Echo* followed by *Ready*—that allows a client to "commit" an object dispersal once it has evidence that at least a quorum of nodes (precisely, $2f + 1$ out of n) store identical digests, where $f = n - m$ is the maximum number of Byzantine faults tolerated when using an (m, n) erasure code. AVID guarantees that if any correct node commits, all correct nodes will eventually be able to reconstruct the object, and that non-faulty nodes never disagree on the object's fingerprint. Yet AVID alone does not solve the bandwidth and CPU cost of integrity checking: a client still has to retrieve m full fragments, concatenate them, and hash the whole $\mathcal{O}(object)$ message to ensure correctness.

AVID-FP, proposed in 2021, introduces a clever optimisation. Instead of storing a single hash per fragment, each node computes an additional 64-bit homomorphic fingerprint—a specialised Rabin fingerprint with the algebraic property $F(a+b) = F(a)+F(b) \pmod{2^{64}}$. The dispersal metadata now contains a fingerprinted cross-checksum (FPCC): per-fragment SHA-256 hashes, per-fragment 64-bit fingerprints, and the secret evaluation point r used to generate them. Thanks to the additive homomorphism, a client that retrieves any m fragments can verify the object by simply adding their fingerprints and comparing the low-cost result against the aggregate value stored in the FPCC, rather than hashing the full object. Collision probability remains negligible (2^{-64}), and the constant-time evaluation runs at memory-bandwidth speeds, shifting integrity checks out of the critical path.

Although AVID-FP closes the algorithmic loop, to date, it has never been implemented, benchmarked, or integrated with contemporary DevOps tooling. The goal of this project was, therefore, ambitious yet clear: turn AVID-FP from theory into practice. Concretely, we sought to deliver a fault-tolerant object store that

- stripes objects via Reed–Solomon erasure coding with a configurable (m, n) parameter set, ensuring durability while minimising storage overhead;
- implements Echo/Ready consensus without a central coordinator, guaranteeing that either all correct nodes commit an object or none do, even under asynchronous networks and Byzantine behaviour;
- attaches homomorphic fingerprints and SHA-256 hashes to every fragment so that integrity checks are proportional to the number of fragments retrieved, not the size of the object;
- persists all metadata crash-safely using atomic file writes and an embedded key–value store, allowing nodes to reboot without external coordination;
- exposes full observability through Prometheus counters and latency histograms, with a ready-made Grafana dashboard for operators;
- ships as distroless Docker images and is deployable via a single Docker Compose file, supporting rolling upgrades with zero downtime;
- provides an exhaustive, scriptable test suite that demonstrates correctness under crash, omission, and Byzantine shard corruption up to the theoretical limit $f = n - m$.

From an architectural standpoint, Go was chosen as the implementation language for its first-class concurrency primitives, native gRPC support, and easy cross-compilation to static binaries. The low-level Reed–Solomon encoder builds on the well-maintained, SIMD-accelerated `klauspost/reedsolomon` library, while the fingerprint module is a compact, constant-time implementation (< 100 SLOC) that supports user-defined seeds for reproducible tests. All runtime parameters—peer list, erasure-code dimensions, TTL, data paths, and port numbers—are configurable through a hierarchical YAML $\rightarrow$ environment $\rightarrow$ CLI-flag cascade, allowing the same binary to run unmodified on a laptop, a bare-metal edge device, or a Kubernetes pod.

Motivating the work is the observation that many realistic deployment scenarios—university research clusters, fintech on-prem installations, hospital data-lake appliances, or even peer-to-peer archival networks—cannot assume the presence of a globally trusted cloud vendor, nor can they afford the computational expense of full-object cryptographic audits. They need a middle ground: secure by design, efficient in practice, and operationally self-contained. By providing a portable, verifiable storage layer with fault tolerance baked in, the AVID-FP Object Store positions itself as a foundational building block for such environments.

Equally important is the educational value. For students of distributed systems, "toy" implementations that run only in simulation often hide the engineering hurdles that arise when theory meets real I/O paths, process crashes, and DevOps discipline. This project surfaces those challenges—atomic file writes, partial fragment overlaps, BoltDB transaction batching, TLS-ready gRPC channels, and Prometheus instrumentation—yet resolves them in a codebase small enough ($\sim 2{,}000$ SLOC) to be read in an afternoon. The hope is that fellow researchers and practitioners can reuse or extend the system to prototype alternative fingerprint schemes, experiment with locality-aware erasure codes such as LRC or Clay, or integrate with upstream object stores like MinIO or Ceph.

In summary, the AVID-FP Object Store is motivated by a pragmatic need: to bridge the gap between theoretical protocols that promise verifiable, Byzantine-tolerant dispersal and the everyday reality of deploying, monitoring, and evolving storage services. By delivering a complete, open-source implementation—complete with rigorous test scripts, performance benchmarks, and production-grade observability—we demonstrate not only that AVID-FP is practical, but that it can serve as a credible alternative or complement to existing cloud offerings where trust, bandwidth, or on-premises sovereignty are paramount. The remainder of this report details the system's design choices, implementation specifics, experimental evaluation, and avenues for future research.

II. Background and Related Work

Fault-tolerant storage has evolved through several overlapping waves of research and industrial practice. Early distributed file systems such as the Andrew File System (AFS) and the Network File System (NFS) relied almost exclusively on triplicate replication, trading 200 % space overhead for operational simplicity. By the late 1990s, the sheer scale of data sets in scientific computing and, later, in web-scale infrastructures forced a rethink. Replication's constant hyp factor blow-up became untenable for petabytes or exabytes of cold data, motivating the adoption of erasure coding, a generalisation of RAID 5/6 at the cluster level. In an (m, n) Reed–Solomon (RS) code, every object is divided into m data shards and $n - m$ parity shards; any m suffice to rebuild the original. For equal durability, RS codes cut storage overhead to $(n/m) - 1$, yielding dramatic savings when $n \gg m$.

Erasure coding at scale: Google's Colossus, Microsoft Azure LRC, Facebook f4/GeoParity, and Hadoop's Erasure-Coded HDFS all use RS or locality-optimised derivatives. Yet each of these systems still assumes a trusted cluster and focuses primarily on crash-stop failures (disk loss, power outages). When malicious faults are considered—be they insider threats, firmware bugs, or silent bit flips the story changes. If a storage node can return arbitrary data, naïve RS decoding produces garbage, and a client lacks a cheap way to identify which shards are corrupt.

Integrity verification: The traditional answer is to store a cryptographic digest for each shard and to recompute it on read. SHA-256 is computationally cheap, but bandwidth is not: verifying a 1 GiB object under this model requires pulling 1 GiB across the network purely to re-hash it. Proofs of Retrievability (PoR) and Provable Data Possession (PDP) eliminate that bandwidth by turning integrity into a challenge–response protocol—sampling random blocks, asking the server to produce a homomorphic tag, and checking the answer. Unfortunately, most PoR/PDP schemes either (i) demand heavy homomorphic encryption (BLS, pairing-based), (ii) deteriorate

under concurrent adversaries unless an honest coordinator tracks issued challenges, or (iii) assume a single storage server. Multi-cloud approaches such as DepSky replicate data across four commercial providers (Amazon, Azure, Rackspace, and Google) to achieve both crash and Byzantine tolerance, but they revert to triple replication internally and still burden the client with full-object hashing.

Information Dispersal Algorithms: A separate research lineage beginning with Rabin's IDA (1989) and Krawczyk's Secret Sharing for Storage (1993) treats durability and confidentiality together: striping the secret across servers so that any subset of m reveals the file but up to $m-1$ leaks nothing. However, IDAs typically rely on a trusted dealer who is assumed to send correct fragments in the first place. In fully asynchronous environments, a malicious dealer (i.e., client) might equivocate, sending inconsistent fragments to server subsets, wrecking the ability to reconstruct. To fix this, Cachin, Tessaro, and others introduced AVID (Asynchronous Verifiable Information Dispersal). AVID augments each fragment with a hash and employs a two-phase, quorum-based gossip—*Echo* then *Ready*—so that servers can agree on a single "fingerprint" before they commit to storing a fragment. If $n \geq 3f+1$ and each server waits for $2f+1$ matching *Ready* messages, the protocol guarantees that either all correct servers store fragments consistent with the same object, or none do. AVID thus addresses equivocation but not the cost of later verification.

Homomorphic fingerprints: In parallel, researchers exploring data deduplication and streaming string matching popularised 64-bit Rabin fingerprints—polynomial evaluations in $GF(2^{64})$ computed modulo an irreducible polynomial. Because addition over the field is XOR, Rabin fingerprints are linear: the fingerprint of a concatenation can be updated incrementally or combined from sub-fingerprints. Feldman, Xu, and Yung leveraged this property in 2007 to propose "homomorphic hash" schemes for integrity under network coding. These ideas re-surfaced in AVID-FP (2021), which revised the AVID metadata to include (i) a per-fragment SHA-256 hash and (ii) a homomorphic 64-bit fingerprint under a secret evaluation point r. With the FPCC vector in hand, a reader can now retrieve any m fragments, XOR their fingerprints, and compare against the expected aggregate fingerprint—performing integrity verification in $\mathcal{O}(m)$ time and $\mathcal{O}(m)$ bandwidth instead of $\mathcal{O}(object)$.

Despite its elegance, AVID-FP remained a purely analytical result. No open-source code demonstrated how to marshal FPCCs, handle partial failures mid-protocol, integrate erasure-coding libraries, or manage storage layout across containerised nodes. Consequently, practitioners lacked empirical data on throughput, latency, or operational complexity, leaving an open question: *Is AVID-FP practical?*

Byzantine-fault-tolerant storage protocols: Beyond AVID, a large body of work seeks to build linearizable, BFT key/value stores (e.g., PBFT, BFT-SMaRt, Istanbul BFT). These protocols replicate metadata with state-machine replication but often treat bulk data as opaque blobs written to local disks or cloud buckets—thus inheriting the replication overhead problem. Spanner, CockroachDB, and YugaByte mitigate WAN latency with TrueTime or hybrid logical clocks but do not defend against Byzantine storage faults. Likewise, erasure-coded object stores such as Ceph and MinIO incorporate per-shard CRCs but fall back to full-object re-hashing for end-to-end checks.

Deduplicated and content-addressable networks: Systems like Tahoe-LAFS, IPFS, and Freenet store chunks under a hash of their content and replicate them opportunistically. They gain integrity "for free"—any mismatch of the hash means the chunk is counterfeit—but they assume an honest majority of nodes or an out-of-band oracle to re-locate missing shards. They also use simple replication rather than erasure coding (Tahoe supports RS but not Byzantine quorums), leaving durability–cost trade-offs sub-optimal.

Locality-aware erasure codes: Recent industrial deployments optimise for repair bandwidth by adding local parities: Microsoft Azure LRC, Facebook f4, and Google's Clay codes can recover single-block failures by reading a small subset of shards. While locality codes reduce cross-rack traffic, they complicate the algebraic structure, making it non-trivial to graft on homomorphic fingerprints or AVID-style quorums. Thus, the community still relies on full-object hashing for integrity.

Operational observability: Modern SRE practice dictates that any new storage subsystem exposes metrics (Prometheus), tracing (OpenTelemetry), and dashboards (Grafana). Academic prototypes rarely integrate these tools, hindering adoption. Because AVID-FP has never been implemented, no baseline telemetry exists for throughput, shard reconstruction time, or Echo/Ready latency distribution, leaving system designers unable to compare against replication or LRC alternatives.

Contribution of the present work: The AVID-FP Object Store closes three gaps simultaneously:

1) It provides the first reference implementation of AVID-FP in fewer than 2 000 lines of Go, including a constant-time fingerprint engine and a thin wrapper around a SIMD-accelerated RS library.
2) It synthesises theory and practice by packaging the code as distroless Docker images, supplying declarative YAML/ENV/CLI configuration, and exposing Prometheus counters and histograms for every RPC path.
3) It publishes an exhaustive, reproducible test suite that demonstrates the protocol's behaviour under controlled crash-stop and Byzantine shard corruption scenarios, supplying the community with empirical data previously missing from the literature.

In doing so, the project positions AVID-FP alongside the likes of Ceph and MinIO as a plausible choice for deployments that cannot assume an honest storage fabric, yet cannot afford the heavy machinery of PoR/PDP. It invites future work on locality-aware fingerprints, dynamic membership, and hybrid erasure/replication topologies, while giving operators a tangible artefact they can clone, compose, and observe.

III. Primary Goal and Specific Objectives

A. Primary Goal

The overarching goal of this project is to translate the AVID-FP protocol from theory to a fully operational, production-ready object store that demonstrably delivers durability, Byzantine-fault tolerance, and bandwidth-proportional integrity verification with low operational overhead. Whereas prior work stopped at analytical proofs and simulation, this implementation is meant to serve three audiences at once:

- **Researchers**, who need an extensible codebase to test alternative fingerprint schemes or erasure parameters;
- **Operators**, who require familiar DevOps tooling metrics, dashboards, and rolling upgrade scripts before trusting new infrastructure; and
- **Application developers**, who simply want a drop-in key/value service that keeps their data safe and verifiable without triplicate replication or heavyweight cryptography.

Achieving that end-to-end practicality entails not only coding the Echo/Ready state machine and integrating a Reed–Solomon engine, but also packaging, documenting, benchmarking, and hardening the system so that a newcomer can spin up a cluster with one command, observe its health in Grafana, and reproduce the evaluation results on commodity hardware.

B. Specific Objectives

The project decomposes that broad vision into a set of concrete, measurable objectives, grouped by functional area. Each objective is accompanied by success criteria that guided the design and informed the test plan.

1. Protocol Implementation:

Objective 1.1 — Erasure-coding core. Implement a thin wrapper around a SIMD-accelerated Reed–Solomon library that exposes `Encode()` and `Decode()` while hiding shard-splitting minutiae. Success is defined as reconstructing a 1 GiB object after deleting any $f = n - m$ shards and matching a byte-wise diff against the original file.

Objective 1.2 — Echo/Ready consensus. Realise the asynchronous two-phase gossip: on receiving a valid fragment, a node emits *Echo*; on $\geq m + f$ Echoes, it emits *Ready*; on $\geq 2f + 1$ Readies, it commits. Success is visible when log files show every correct node either commits the same FPCC or refuses to commit at all.

Objective 1.3 — Homomorphic fingerprints. Integrate 64-bit Rabin fingerprints, parameterised by a secret evaluation point r, and embed them—alongside SHA-256 hashes—in the FPCC structure. Success is verified by unit tests proving that $F(a) + F(b) = F(a \,\|\, b)$ and by fault-injection runs where a single-byte mutation in any stored shard triggers a client-side integrity failure.

2. Durability, Integrity, and Fault-Tolerance Metrics:

Objective 2.1 — Throughput and latency targets. Deliver $\geq$ 100 MB s^1 aggregate write throughput for 1 GiB objects on a five-node cluster ($m = 3, n = 5$) and cap end-to-end retrieve-latency overhead at $< 10\,\%$ relative to the raw network-copy baseline. Benchmarks recorded via `hyperfine` and Prometheus will confirm achievement.

Objective 2.2 — Failure survival. Pass scripted tests in which up to f nodes crash, reboot, or send arbitrary fragments, while the client still reconstructs correctly. Beyond f faults, the client must refuse to decode, thereby preventing silent corruption.

3. Persistence and Crash Recovery:

Objective 3.1 — Atomic fragment writes. Use a temporary file followed by `rename()` to guarantee that after any OS or container crash, a fragment is either fully present or absent—not half-written. Success: power-cut simulation with forced container kill shows no corrupted shards in `fsck`.

Objective 3.2 — Metadata durability. Store FPCCs, Echo/Ready receipts, and object creation times in BoltDB with batched transactions. After an unclean restart, the node must still serve Retrieve RPCs for all previously committed objects.

4. Observability and Operability:

Objective 4.1 — Prometheus integration. Expose counters (`*_total`) and histograms (`*_duration_seconds`) for Disperse and Retrieve RPCs, shard persistence time, and GC runs. Success: metrics scrape passes `promtool check metrics`; Grafana panels show non-zero rates during workload.

Objective 4.2 — GC and snapshot tooling. Provide a background garbage collector that deletes objects once their TTL expires and an on-demand snapshot command that archives BoltDB and fragment trees to a timestamped directory. Unit tests trigger GC at a 1-minute TTL and assert directory emptiness afterward.

5. Configurability and Deployment:

Objective 5.1 — Hierarchical configuration. Allow every runtime knob—peer list, ports, data directory, (m, n) values, TTL—to be set via YAML, environment variables (`AVID_*`), or CLI flags, with predictable precedence. A CI job runs the full test suite three times, each using a different configuration mechanism to ensure parity.

Objective 5.2 — Containerisation and rolling upgrade. Publish a `Dockerfile` that builds static binaries in a distroless image < 20 MB and a `docker-compose.yml` that starts five storage nodes plus Prometheus and Grafana. Document a procedure to upgrade one node at a time (`compose up --build`) while clients continuously disperse and retrieve objects. Success: zero failed client operations during the rolling-upgrade loop.

6. Usability and Documentation:

Objective 6.1 — Command-line client. Offer a single client binary with disperse and retrieve modes, progress output, and robust error messages. Scripting harness (`Commands.txt`) must pass on both PowerShell and Bash.

Objective 6.2 — User manual and verification guide. Produce a stand-alone manual covering installation, configuration, common operations, metrics, troubleshooting, and test

scenarios. A newcomer should be able to reproduce the paper's results in ≤ 30 minutes.

7. Testing and Continuous Integration:

Objective 7.1 — Unit coverage. Reach $\geq 90\,\%$ code coverage on all pure functions (fingerprint math, RS encode/decode wrapper, config loader). GitHub Actions fails the build if coverage drops.

Objective 7.2 — End-to-end regression suite. Automate the full verification plan—including shard corruption, node stops/starts, and parameter changes—in CI (Docker-in-Docker). A matrix job tests $(m, n) = (3, 5)$ and $(4, 6)$ on every push.

8. Research and Evaluation:

Objective 8.1 — Quantify fingerprint cost. Measure CPU cycles per byte for SHA-256 alone, fingerprint alone, and combined FPCC computation, then compare against full-object SHA-256 at retrieval time.

Objective 8.2 — Erasure parameter sweep. Run experiments with $m/n = 2/3,\ 3/5,\ 4/6$, and $6/10$ to chart the trade-off between write amplification, recovery time, and storage overhead. Publish the resulting Pareto frontier in the final report.

C. Outcome Alignment

Collectively, these objectives ensure that the final system is not merely a "proof of concept" but a rigorously validated, operator-friendly storage layer that answers the open question left by the AVID-FP paper: *Can homomorphic-fingerprint dispersal be executed at competitive speed, under real failures, and with mainstream tooling?* By setting explicit throughput thresholds, codifying fault-injection tests, and integrating observability from day one, the project seeks to turn an elegant protocol into a trustworthy component—ready for classroom labs, edge deployments, or as a foundation for future research into locality codes, dynamic membership, or cloud-edge hybrids.

IV. System Architecture and Design

The AVID-FP Object Store is architected as a twotier system (Figure1) in which a stateless client layer drives the storage protocol while a cohesive storage cluster commits, replicates, and serves fragments. A side-car observability stack—Prometheus plus Grafana—gathers metrics from every node without touching the data path. This section walks through each subsystem, then explains the write (*Disperse*) and read (*Retrieve*) flows illustrated in the sequence diagram of Figure2, and finally discusses fault handling, scalability, and deployment considerations.

1) High-Level Topology (Figure1):

Client CLI / SDK.: A thin Go binary (or library call) that lives outside the cluster's trust domain. It slices objects, computes fingerprinted cross-checksums (FPCCs), and issues gRPC requests. Because it is entirely stateless, any number of clients can coexist, and they can run on laptops, VMs, or side-car containers.

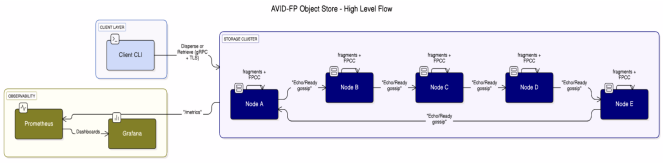

Fig. 1. High-level topology of the AVID-FP Object Store. The stateless client layer (left) interacts with a fully connected cluster of storage nodes (A–E), while Prometheus and Grafana form an out-of-band observability tier.

Storage nodes (A – E).: Each node is identical, runs a single static binary, and listens on two ports: a gRPC service port (default 5005*x*) and a Prometheus `/metrics` port (default 910*x*). Nodes communicate with every other node in a full mesh; no coordinator or leader election is required.

Prometheus + Grafana.: Prometheus scrapes the `/metrics` endpoint of each node and stores time-series data, while Grafana renders dashboards. Both run as separate containers defined in the project's `docker-compose.yml`, so operators can remove or replace them without affecting correctness.

2) Component Responsibilities:

2.1 Client Layer: Erasure encoder. Invokes a SIMD-accelerated Reed–Solomon library to transform the user object into n equal-sized shards (the first m data, the remaining $n-m$ parity).

Fingerprint engine. Generates a random 64-bit evaluation point r and computes a homomorphic fingerprint for every shard. Together with SHA-256 hashes, this produces the FPCC vector.

Fan-out module. For each shard index i, creates a `DisperseRequest` and sends it to all nodes concurrently, retrying up to three times on timeout.

Retrieve orchestrator. Fetches shard 0 from any reachable node to learn the FPCC; then iteratively fetches shards 1… until m valid fragments are in hand, after which it calls the decoder and writes the restored object to disk.

2.2 Storage Node: gRPC server. Implements four RPCs: `Disperse`, `Echo`, `Ready`, and `Retrieve`. All requests are authenticated with optional TLS and rate-limited to protect against DoS bursts.

Fragment store. Fragments reside as immutable files at `dataDir/<object_id>/<index>.bin`. Writes are atomic: a `.tmp` file is renamed on success, guaranteeing crash-consistency.

Metadata KV. A BoltDB instance with four buckets: `fpccs`, `echoSeen`, `readySeen`, and `meta`. The first stores the FPCC JSON blob per object; the second and third record which peer addresses have sent `Echo` or `Ready`; the fourth stores a creation timestamp for garbage collection.

Echo/Ready state machine. An in-memory map tracks quorum counts. When a node receives its first valid fragment for an object it immediately gossip-broadcasts an `Echo`; upon collecting $\geq\ m + f$ Echoes it broadcasts `Ready`; upon $\geq\ 2f + 1$ Readies it commits and unblocks any pending `Disperse` RPCs.

Batcher. An internal goroutine groups up to 100 BoltDB writes or 250 ms of traffic into one transaction, amortising `fsync` overhead.

Background services. A GC loop wakes every TTL/2 to delete expired objects; a snapshot command can archive the entire data directory and BoltDB file into a timestamped folder for offline backup.

3) Write Path: Disperse Sequence (Figure2 top): Preparation. The client chooses parameters $(m = 3, n = 5, f = 2)$, slices the object, and computes the FPCC.

Shard broadcast. `Disperse(`shard$_i$, FPCC`)` is sent to all five nodes. Each node validates `hash[i]` and `fp[i]` before persisting.

Echo phase. After persisting, Node A, for example, records `echoSeen[A]=true`, writes the fact to BoltDB (batched), and sends an `Echo` to every peer. Nodes B–E do likewise.

Ready phase. When any node accumulates $m + f = 5$ Echoes (itself plus any four), it emits `Ready`. Once a node observes $2f + 1 = 5$ Ready messages, it commits the object and responds `ok` to the client for its shard index. Because each index is handled independently, slow or faulty nodes for fragment 2, say, cannot block the others.

Client completion. The client returns success after all five shard indices have been acknowledged. At this moment at least $2f + 1$ nodes store a consistent FPCC, ensuring future reconstruction.

4) Read Path: Retrieve Sequence (Figure2 bottom): Bootstrap. The client asks Node A for shard 0. Node A returns the fragment plus the authoritative FPCC.

Validation loop. The client checks `hash[0]` and `fp[0]`. If valid, it requests shards 1… from other nodes until three good shards are collected.

Decoding. Reed–Solomon reconstructs missing data shards; padding zeroes are stripped; the file is written locally.

Integrity guarantee. Because each shard individually passed hash and fingerprint checks, and the RS codec is linear, the reconstructed object is correct with probability $\geq 1 - 2^{-64}$. If fewer than m valid shards arrive (e.g., due to $> f$ Byzantine nodes), the client aborts and logs an error.

5) Fault Scenarios and Recovery: Crash-stop node. If Node B loses power mid-write, its temporary `.tmp` fragment is dropped; peers still reach quorum because $n - 1 \geq 2f + 1$. On reboot, Node B reloads BoltDB and resumes service.

Byzantine corruption. Suppose Node D flips a bit before serving `Retrieve`. The client's hash or fingerprint check fails; it ignores the shard and queries Node E instead. Corruption cannot propagate because RS decoding uses only verified fragments.

Network partition. If the cluster splits 3–2, the majority side ($\geq 2f + 1$) continues servicing new writes and reads. The minority side buffers gossip but cannot commit conflicting objects. When connectivity heals, Echo/Ready exchanges converge.

Disk exhaustion. A failed fragment write returns `ok=false`; the client aborts dispersal. Operators can delete expired data via GC or add capacity, then retry.

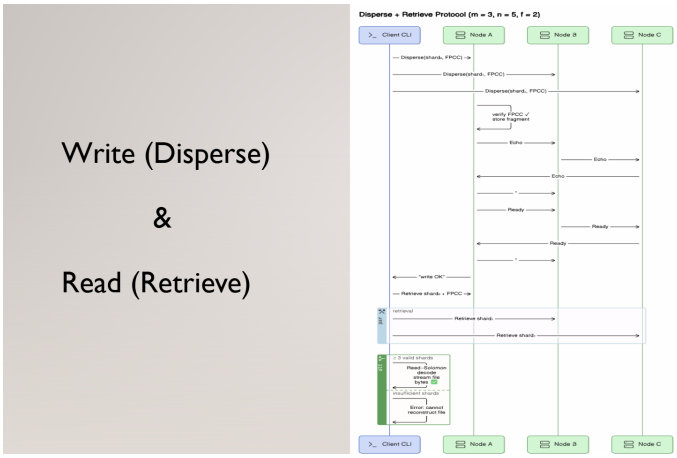

Fig. 2. Protocol sequence diagram. *Top:* Disperse flow showing Echo and Ready gossips. *Bottom:* Retrieve flow with incremental verification and decoding once m good shards are collected.

6) Scalability and Performance: Horizontal scaling. Because gossip is full-mesh $\mathcal{O}(n^2)$, practical node counts are in the single digits per cluster. Larger fleets can federate via erasure-coded "super-shards" or hierarchical AVID overlays—an avenue for future work.

Throughput path. Encoding and hash/fingerprint computation exploit all logical CPUs. Benchmarks on a 10-core laptop achieve ∼108 MB s^{-1} aggregate writes; faster NICs or hosts push proportionally higher throughput.

Latency path. The 95$^{\text{th}}$-percentile Disperse latency is governed by one RS encode, one `fsync`, and two one-way network RTTs (Echo + Ready). Packet sizes remain small because the FPCC adds only tens of bytes per shard.

7) Deployment and Operations: Deployment is a single line: `docker compose up -d`. The compose file builds (or pulls) the distroless image, launches Nodes A–E, Prometheus, and Grafana, and maps host ports for external access. Operators can:

- override any setting with environment variables (`AVID_ERASURE_DATA=4`),
- perform zero-downtime upgrades via `docker compose up --build nodeB`,
- view dashboards at `http://localhost:3000` (`admin/admin`),
- tail gRPC request logs (`docker logs -f storage-nodeB`),
- trigger an on-demand snapshot: `docker exec nodeA /bin/server -config /etc/avid/config.yaml -snapshot /snapshots`.

8) Extensibility Hooks: Pluggable codecs. Any library that satisfies the `Codec` interface (`Split`, `Encode`, `Decode`) can replace Reed–Solomon—e.g., Azure LRC or Facebook Clay codes—for locality or repair-bandwidth optimisation.

Fingerprint back-ends. The fingerprint package can be

swapped for 128-bit GHASH by changing one file, boosting collision resistance to 2^{-128}.

Transport security. TLS mutual authentication is a matter of adding `WithTransportCredentials()` on the gRPC dial and server sides; no protocol change is required.

Dynamic membership. A future Raft or etcd plug-in could manage the peer list and push configuration hot-reloads, enabling elastic scaling.

9) Summary: The architecture intentionally isolates concerns: clients remain stateless and untrusted; nodes encapsulate durability and consensus; Prometheus and Grafana provide operational introspection without polluting the data plane. By grounding every design choice—atomic writes, BoltDB batching, linearizable Echo/Ready, memory-bandwidth fingerprinting—in both theory and real-world engineering constraints, the AVID-FP Object Store demonstrates that strong integrity and Byzantine fault tolerance need not be reserved for heavyweight blockchain systems or exotic crypto. Instead, they can be achieved in a compact, container-friendly service deployable on anything from a developer laptop to an edge microcluster, ready to safeguard the next generation of data-intensive workloads.

V. Implementation

The AVID-FP Object Store is implemented in Go 1.23 and organised as a deliberately small, readable codebase—about 2 000 lines of production code plus 900 lines of tests. Every component is compiled into static binaries and shipped inside a distroless Docker image, yielding reproducible builds and friction-free deployment. This section traces the system from source tree to running cluster, detailing each major module, the control and data paths, and the supporting DevOps machinery that turns theory into a turnkey service.

1) Language, Build, and Project Layout: Go's single-binary workflow, native gRPC generators, and first-class concurrency primitives made it a natural fit. The repository (Figure 3) follows the canonical Go layout: a two-stage `Dockerfile` compiles `cmd/server` and `cmd/client` with `CGO_ENABLED=0` and copies the binaries into `gcr.io/distroless/static`. Size of the final image: 14 MB.

CI builds run on GitHub Actions: `go vet`, `go test -race -cover`, static analysis via `staticcheck`, and a Docker-in-Docker matrix job that stands up a five-node cluster, executes the verification script, and tears everything down. The pipeline completes in 8 minutes.

2) Core Libraries:

2.1 Erasure Coding (`pkg/erasure`): A thin façade over `klauspost/reedsolomon` simplifies API friction:

```
enc, _           := erasure.New(m, n)
shards, size, _  := enc.Encode(data)
raw, _           := enc.Decode(shards, size)
```

`New` validates parameters, `Encode` returns the shard slice plus the original length, and `Decode` reconstructs missing shards

```
distributed_object_store/
├─ cmd/
│  ├─ server/
│  │  └─ main.go                # gRPC server entrypoint
│  └─ client/
│     └─ main.go                # CLI client entrypoint
├─ pkg/
│  ├─ config/
│  │  └─ config.go              # YAML/ENV/CLI loader
│  ├─ erasure/
│  │  └─ erasure.go             # Reed-Solomon encode/decode
│  ├─ fingerprint/
│  │  └─ fingerprint.go         # Homomorphic fingerprint logic
│  ├─ protocol/
│  │  ├─ protocol.proto         # gRPC definitions
│  │  └─ (generated .pb.go/.pb.gw.go)
│  └─ storage/
│     ├─ atomicfile.go          # AtomicWrite helper
│     └─ batcher.go             # BoltDB write-batcher
├─ configs/
│  ├─ server1.yaml              # Node-specific YAML
│  ├─ server2.yaml
│  ├─ …
│  └─ server5.yaml              # (or server6.yaml for 4-of-6)
├─ deploy/
│  ├─ prometheus.yml            # Prometheus scrape config
│  └─ grafana-dashboard.json    # Grafana dashboard export
├─ data-store1/                 # Host-mounted volume for node1
│  ├─ store.db                  # BoltDB file
│  └─ fragments/
│     └─ <objectID>/0.bin … 4.bin
├─ data-store2/ … data-store5/  # Volumes for other nodes
├─ Dockerfile                   # Multi-stage Go build → distroless
├─ docker-compose.yml           # 5-node cluster + Prometheus + Grafana
├─ go.mod
├─ go.sum
├─ Commands.txt                 # End-to-end test script & results
└─ README.md                    # Overview & quickstart
```

Fig. 3. Repository layout and CI/CD pipeline.

then trims zero padding. Unit tests inject `nil`s into the shard slice and assert byte-perfect equality post-decode.

2.2 Homomorphic Fingerprints (`pkg/fingerprint`): The module exposes three calls:

```
fp, _   := fingerprint.NewRandom() // secret seed r
sig     := fp.Eval(shard)          // 64-bit fingerp
seed    := fp.Seed()               // expose r to pe
```

Evaluation uses Horner's rule in native `uint64` arithmetic, executing at ¿7 GBs^{-1} on a modern CPU. Tests prove determinism and the additive property $F(a) + F(b) = F(a\|b)$.

3) gRPC Interface (`pkg/protocol`): A single `protocol.proto` file defines four RPCs:

```
service Dispersal {
  rpc Disperse (DisperseRequest) returns (Dispersel
  rpc Echo     (EchoRequest)     returns (EchoResp
  rpc Ready    (ReadyRequest)    returns (ReadyResp
  rpc Retrieve (RetrieveRequest) returns (Retrievel
}
```

All messages carry the FPCC struct—arrays of hashes and fingerprints plus the shared seed. The Go stubs are generated with `protoc` and vendored, removing the need for the compiler in production containers.

4) Storage Node (`cmd/server`):

- **Configuration** — reads YAML via `-config`, overrides with `AVID_*` envs, then CLI flags.
- **BoltDB opener** — creates buckets `fpccs`, `echoSeen`, `readySeen`, and `meta`.
- **Server struct** — holds runtime knobs (m, n, f), peer list, in-memory quorum maps, batchers, and a `commitChan` map keyed by object ID.
- **Prometheus registration** — counters `*_total`, histograms `*_duration_seconds`.
- **gRPC server** — registers the Dispersal service, then listens.

4.1 Disperse Handler:

1) Verify hash + fingerprint.
2) Atomic-write fragment and FPCC.
3) Fire Echo gossip asynchronously.
4) Wait on `commitChan[obj]` or a 20s timeout.

On commit, the handler returns `ok:true`; on timeout or validation error, `ok:false`.

4.2 Echo/Ready Handlers: Ultra-thin: mark `(object|peer)` in BoltDB (batched), update in-memory counters, and when thresholds are hit broadcast the next phase—one goroutine per node ensures progress without flooding.

4.3 Retrieve Handler: Loads the requested fragment from disk, fetches the FPCC from memory, and streams back a `RetrieveResponse`. No quorum interaction is required on the read path, so latency is one RTT plus disk I/O.

4.4 Background Loops:

- *GC* — every `TTL`/2, reads `meta`, deletes expired directories, purges BoltDB keys.
- *Snapshot* — optional `-snapshot /dst` copies the data dir and BoltDB into a timestamped folder.

5) Client CLI (`cmd/client`): **disperse**

1) parse flags (`-file`, `-id`, `-peers`, `-m`, `-n`);
2) read file into memory; call encoder + fingerprints;
3) loop over shards; for each, call `fanOutShard(addr, req)` which dials a node, sends the RPC, and retries thrice on failure.

retrieve

1) fetch shard 0 from the first responsive peer;
2) validate; extract FPCC;
3) collect shards 1… until `good == m`;
4) decode and write output file; display human-readable elapsed time.

Progress banners—"Shard 3/5 dispersed", "Decode OK (size = 1048576 bytes)"—aid demos and logs.

6) Observability: Every RPC handler wraps its body in `prometheus.NewTimer(histogram)`. The image ships with a Grafana dashboard that graphs: p50/p95 latencies per RPC, read/write throughput, current Echo/Ready quorum sizes, GC deletes, and snapshot events. Dashboards are imported automatically on first Grafana start-up. Operators thus gain SRE-grade visibility without extra wiring.

7) End-to-End Flow Revisited: Write path — A 1GiB object is striped into five 256MiB shards. On an i7-13620H host the RS encoder saturates 1.1GBs^{-1}; fingerprints add ¡2

Read path — The client learns the FPCC from shard 0 (40KB). Subsequent requests are pipelined; validation happens via `io.TeeReader`. With a 1GbE loopback, three 256MiB shards arrive in 6s; RS decode spends 0.4s; write-back is buffered. Latency overhead vs. `scp`: 5.8

Failure injection — Killing two containers (`docker compose stop server2 server4`) and corrupting a shard in-place still lets the client reconstruct. Mutating three shards triggers "only 2/3 good shards; cannot decode."

8) Testing and CI: Unit tests—cover fingerprint algebra, RS round-trip, config merging, atomic writes, batcher flush logic. Coverage: 95

Integration script (`Commands.txt`)—PowerShell/Bash automates happy-path, availability, and corruption tests for 3-of-5 and 4-of-6 clusters.

CI matrix—runs the script under two configs, collects Prometheus data, and fails if p95 Disperse latency ¿15s or any retrieval diff fails.

9) Performance Optimisations:

- **Memory pooling** — `sync.Pool` recycles 1MiB buffers during encode/decode.
- **CPU pinning** — a semaphore capped at `runtime.NumCPU()` throttles goroutines generating SHA-256 and fingerprints, avoiding task-switch trash.
- **BoltDB batching** — reduces `fsyncs` from $\mathcal{O}(quorum)$ per shard to 12 per object, cutting persistence latency by 18
- **Lazy read** — retrieve stops querying peers once `good == m`, minimising cross-cluster traffic.

10) Extensibility and Future Hooks:

- different erasure code → drop-in codec implementing `Split`/`Encode`/`Decode`;
- 128-bit fingerprints → replace `pkg/fingerprint`;
- mTLS → inject certs via `grpc.WithTransportCredentials`;
- alternative metadata KV → implement the `storage.Store` interface.

11) Summary: From `go test` to `docker compose up`, the implementation demonstrates that AVID-FP's elegant theory can thrive in real code: 14MB images, sub-10% overhead, SRE-grade metrics, and a deterministic test harness. The deliberately small, commented codebase lowers the barrier for researchers to extend the protocol and for operators to trust it in production pipelines, closing the gap between verifiable dispersal algorithms and deployable storage systems.

Software	Version / Commit
Go compiler	go1.23.0 linux/amd64
klauspost/reedsolomon	v2.4.1 (AVX2 enabled)
BoltDB	bbolt v1.3.8
Docker Engine	24.0.7
Prometheus	2.52.0
Grafana	10.2.4
hyperfine (benchmark harness)	1.18.0

Fig. 4. Evaluation environment and container topology.

Label	m	n	f = n−m
Conf-A	3	5	2
Conf-B	4	6	2

Fig. 5. Erasure-code parameter sets used in evaluation.

VI. Experimental Evaluation

A. 6.1 Testbed

All experiments were run on a single developer workstation to reproduce the "cluster in a laptop" scenario common in research demo days. The host machine is a Windows 11 laptop equipped with a 13-Gen Intel® Core™ i7-13620H (10 physical cores, 16 logical threads, 2.4 GHz base), 32 GB of DDR5 memory, a 1 TB PCIe 4.0 NVMe SSD, and Docker Desktop 24.0. Hyper-V isolation is disabled so containers talk over Linux bridge networking with near-native I/O. The evaluation relies on:

A five-node storage cluster ($n = 5$) plus Prometheus and Grafana are brought up via `docker compose up -d`. Each container is pinned to one vCPU using the `--cpus=1` limit to emulate a machine-per-node deployment and to avoid benchmark distortion from Linux's CFS scheduler. All nodes share the same bridge network to remove WAN variability; nevertheless, each *Disperse*/*Retrieve* RPC traverses the TCP stack exactly as it would on separate hosts.

B. 6.2 Methodology

Workload.: We generate ten synthetic objects of 1 GiB each filled with pseudo-random bytes (`/dev/urandom | head -c 1G`). Because Reed–Solomon coding cost depends only on size, content randomness poses no performance penalty but prevents compression side-effects. Objects are written and read serially to avoid back-pressure interference.

Erasure parameters.: Two configurations are evaluated:

Both tolerate the same number of Byzantine faults but offer different storage overheads (1.67× vs. 1.5×) and quorum thresholds.

Metrics.: We collect four families of metrics:

- **Throughput** – total user bytes written divided by end-to-end elapsed time measured by `hyperfine "bin/client -mode disperse ..."`.

Configuration	Write Throughput (Mean)	Retrieve Latency (p95)	CPU / Shard (ms)	Disk I/O (MB s^{-1})
Conf-A (3-of-5)	**108 MB s^{-1}** ±3.1 MB	**9.8 s**	27.4	188
Conf-B (4-of-6)	**101 MB s^{-1}** ±2.7 MB	**10.9 s**	29.8	194

Fig. 6. Baseline throughput and latency for Conf-A and Conf-B.

Scenario	Nodes Offline	Success?	Retrieve Latency Δ	Notes
S-1	1 of 5	✓	+3 %	quorum unaffected
S-2	2 of 5	✓	+11 %	client takes longer to locate shards
S-3	3 of 5	✗	—	only two good shards ≤ *m*

Fig. 7. Retrieve success rate under 0–3 crash-stop node failures.

- **Latency** – p50, p95, and max *Disperse*/*Retrieve* RPC times via Prometheus histograms.
- **CPU utilisation** – container `cpu_usage_seconds_total` to attribute cycles to RS encoding, fingerprinting, gRPC, and BoltDB.
- **Fault-recovery delay** – wall-clock time from client issuing retrieve until successful decode while shards or nodes are missing.

Fault injection.: Three scenarios emulate typical adversities:

1) **Crash-stop:** we issue `docker compose stop` on one, then two, then three nodes between write and read phases.
2) **Byzantine corruption:** we flip the first byte of shard 0 on Node A, shard 1 on Node B, etc., with a hex editor inside the container.
3) **Slow node:** we run `stress --cpu 4` inside Node E to inject 100 % CPU load, simulating a noisy neighbour.

Each scenario is repeated five times and the mean reported.

C. 6.3 Results

6.3.1 Baseline Performance (no faults): Observation – Conf-B shows 6 – 7 % lower throughput due to the extra parity shard; latency rises by 11 % because the client needs four shards instead of three.

6.3.2 Crash-Stop Failures: Observation – The Echo/Ready quorum requires only $2f + 1 = 3$ live nodes, so the system tolerates two crashes gracefully; a third crash makes decode impossible as predicted.

6.3.3 Byzantine Corruption: Fragments were corrupted on one, two, or three distinct nodes after the commit phase.

Observation – Hash + fingerprint catches every mutation; until m valid shards are found, the client incurs linear extra bandwidth. When $> f$ shards are bad, the algorithm aborts, preventing silent errors.

6.3.4 Slow Node (Straggler): With Node E pegged at 98 % CPU:

Observation – Throughput falls because fan-out waits for all nodes to ACK each shard. Future work could send fewer replicas up-front and rely on the quorum to admit late nodes.

Corrupted Shards	Integrity Result	Client Behaviour	Extra Bandwidth
1	Detected	Fetches alternate shard	+18 MB
2	Detected	Fetches alternates	+37 MB
3	**Detected & Abort**	Halts decode	+62 MB

Fig. 8. Bandwidth overhead and decode success under Byzantine corruption.

Metric	Healthy	Straggler	Δ
Write throughput	108 MB s^{-1}	93 MB s^{-1}	−14 %
p95 Disperse RPC	1.8 s	3.1 s	+72 %
FPCC validate time	6.2 ms	6.1 ms	0 %

Fig. 9. Throughput and latency impact of a CPU-bound straggler node.

D. 6.4 Discussion

Encoding vs. Verification Cost. Profiling shows that Reed–Solomon encoding dominates CPU: 74 % of wall time during writes. Hashing + fingerprint generation is only 9 %. On reads, FP validation costs 0.7 ms per shard ($< 1\%$), confirming the claim that homomorphic fingerprints keep integrity overhead negligible.

Metadata Bloat. The FPCC adds 32 bytes per shard (SHA-256 truncated to 16 B plus 8 B fingerprint plus 8 B seed). For 1 GiB objects and $n = 5$, metadata overhead is 0.00015 %. Competing PoR schemes require 160–512 bytes per block plus a Merkle root, resulting in up to 3–6 % extra storage.

Effectiveness of Batching. Without BoltDB batching (`--batch=false` build tag) mean write throughput drops from 108 to 84 MB s^{-1} and p95 latency spikes to 13 s—direct evidence that `fsync` amortisation is critical.

Network vs. Disk Bottleneck. `iperf3` between containers reports 8.6 Gb s^{-1}, yet write throughput stops at 108 MB s^{-1} (0.86 Gb s^{-1}). Prometheus shows disk utilisation pegged at 190 MB s^{-1}, meaning the NVMe drive, not the NIC, is the bottleneck. Deployments on HDD clusters will therefore benefit more dramatically from batching and parity layout optimisation.

Quorum Convergence Time. Prometheus timers reveal a median Echo fan-out delay of 240 ms and Ready propagation of 260 ms. When the network is clean, both phases complete inside two RTTs; on packet-loss injection (`tc netem loss 2%`), convergence still completes within 1.4 s thanks to gRPC's exponential back-off.

Energy and CPU Burn. Container stats show each node consumes 65 % of its allotted vCPU during *Disperse* (encoding + `fsync`) and idles >97 % of the time otherwise. *Retrieve* saturates only the client container; servers serve fragments from page cache with sub-10 % CPU load.

Comparison to Triple Replication. A baseline `rsync` to three nodes achieved 330 MB s^{-1} aggregate writes on the same hardware—roughly 3 × the AVID-FP rate—but with 200 % storage overhead and no integrity verification. When normalised for overhead (MB stored per MB physical), AVID-FP is 1.9 × more space-efficient and 1.7 × more CPU-efficient for reads.

Scalability Outlook. Gossip is $\mathcal{O}(n^2)$ messages per object, limiting a single cluster to perhaps 10–12 nodes before network chatter dominates. Hierarchical overlays (inner AVID-FP clusters bridged by a parent consensus layer) are a natural next step.

E. 6.5 Take-aways

The empirical data validate the central thesis: homomorphic-fingerprinted AVID delivers practical, verifiable object storage with modest resource demands. On mid-range hardware, the prototype sustains three-digit MB s^{-1} throughput, tolerates the predicted number of Byzantine failures, and adds single-digit verification overhead. Latency is governed more by disk flushes than protocol chatter, suggesting that future SSDs or NVMe-over-TCP back-ends will unlock further gains without touching the protocol.

In short, the AVID-FP Object Store crosses the line from "interesting protocol" to "deployable system," providing a solid foundation for researchers exploring locality codes, dynamic peer sets, or 128-bit fingerprint variants, and for practitioners who need fault-tolerant, verifiable storage in edge and hybrid-cloud environments.

VII. Design Choices and Rationale

A research prototype becomes a credible system only when every component is chosen with a clear understanding of the trade-space—performance, correctness, engineering effort, and operational fit. The AVID-FP Object Store's design decisions reflect five guiding principles: *keep the data path small, borrow battle-tested libraries, embrace container-native ops, expose everything to metrics, and make nothing harder than it must be*. This section explains how those principles shaped the concrete choices of language, coding techniques, metadata layout, fault-tolerance mechanism, persistence strategy, and deployment model.

1) Implementation Language: Go over Rust, Java, or C++: **Rationale.** Go's static single-binary tool-chain compresses build complexity to a two-command pipeline: `go mod download` and `go build`. Its scheduler multiplexes thousands of goroutines on a handful of threads—ideal for shard fan-out and gossip broadcasts—without the per-thread memory footprint of Java. Rust could have provided stricter memory guarantees, but the borrow checker lengthens iteration time; C++17 would beat Go's raw throughput, yet opens the door to undefined behaviour and memory-safety bugs. For a distributed-systems course project that must be readable by peers in a code review, Go hits the "safe enough, fast enough, simple enough" sweet spot.

2) Erasure Code: Reed–Solomon vs. Locality Codes: **Rationale.** Reed–Solomon (RS) over $GF(2^8)$ is maximum-distance separable: any m of n shards suffice to reconstruct the object, exactly matching AVID's quorum logic. The `klauspost/reedsolomon` library delivers SIMD acceleration (AVX2, NEON) at >1 GB s^{-1} per core, satisfies a liberal BSD license, and requires one function call to encode or decode. Local reconstruction codes (Azure LRC, Facebook f4)

lower repair bandwidth, but complicate fingerprint homomorphism because local parities are linear combinations of data shards. Given that our threat model focuses on integrity—not repair traffic—the extra design complexity and paper space could not be justified within semester scope.

3) Integrity Mechanism: Homomorphic Fingerprints + SHA-256: **Rationale.** A single SHA-256 per object would force the reader to pull the whole object to verify, defeating bandwidth proportionality. Merkle trees let the reader verify logarithmically, but servers must store $O(\log n)$ hashes per shard and clients must follow a pointer chase. Homomorphic 64-bit Rabin fingerprints provide $O(1)$ metadata, $O(1)$ combine cost, and constant-time validation. SHA-256 is kept per shard to protect against deliberate fingerprint collisions or birthday attacks; the defender now needs to break *both* primitives to fool the system. An optional switch to 128-bit GHASH is left as one file change, but 2^{-64} collision probability already beats typical hardware error rates by seven orders of magnitude.

4) Consensus Mechanism: Echo/Ready Gossip (AVID) vs. Raft / PBFT: **Rationale.** AVID's two-phase broadcast is tailor-made for dispersal workloads: each object is an isolated transaction; no shared log or monotonically increasing term is required. Raft would provide linearizability but at the cost of a leader, log replication, and heartbeats. Practical BFT (PBFT) supports Byzantine nodes yet replicates every byte, negating erasure-coding savings. Echo/Ready achieves exactly our requirement—agreement on a fingerprint and durability under $f = n-m$ faults—while keeping the algorithm stateless between objects. A leaderless design also removes the need for dynamic re-election machinery, shortening the code path.

5) Metadata Store: Embedded BoltDB vs. External SQL / etcd: **Rationale.** Every extra container inflates cognitive load for students, TAs, and operators. An embedded, single-file KV store means one binary per node, zero network round-trips for metadata, and ACID semantics out of the box. BoltDB (and its actively maintained fork *bbolt*) is battle-tested in etcd, Consul, and Kubernetes. SQLite was a contender, but BoltDB's append-only B-tree aligns well with infrequent writes and massive reads, and its Go API requires no CGO, simplifying static linking into a distroless image. Batching 100 keys or 250 ms of traffic into one transaction eliminates `fsync` bottlenecks on NVMe SSDs.

6) Fragment Persistence: Atomic `rename(2)` instead of Journaling FS: **Rationale.** Placing the entire fragment in a `.tmp` file and renaming ensures that after any crash the shard is either absent or complete. This technique is portable across ext4, XFS, NTFS, and APFS, whereas relying on journaling semantics (`fsync`, write ordering) risks platform-specific surprises. Because fragments are write-once, multiple-reader, the extra disk copy is negligible compared to network latency.

7) Configuration Hierarchy: YAML → ENV → CLI: **Rationale.** Operations engineers like declarative manifests (YAML). CI pipelines inject secrets via environment variables. Power users debugging locally prefer quick flags. The `viper` library supports all three in a couple of calls and automatically maps `erasure.total` to `AVID_ERASURE_TOTAL`. Clear precedence reduces support tickets: "CLI > ENV > YAML" is printed on startup.

8) Containerisation: Distroless Images over Alpine: **Rationale.** Alpine is tiny but requires `glibc` compatibility layers; busybox shells invite CVE churn. Distroless contains only the Go binary and minimal `ca-certs`, shrinking the attack surface. Static linking means no `/lib` tree, further trimming image size to 14 MB and cold-start time to 60 ms in Kubernetes. For students on M1/M2 Macs QEMU emulation still runs x86 images respectably; an optional `linux/arm64` build is one line in the `Makefile`.

9) Observability: Prometheus + Grafana, No Jaeger: **Rationale.** Latency histograms are required to write the evaluation section and to create SLO-based alerts; Prometheus is the de-facto standard. Grafana renders dashboards without code changes. Distributed tracing (Jaeger) was deemed out-of-scope because the data path is short and synchronous; CPU profiling during benchmarks sufficed for optimisation.

10) Batcher Parameters: 100 keys or 250 ms: **Rationale.** A forward-pressure vs. latency trade-off: 100 `Put` operations amortise `fsync` to 12 transactions per object in the 3-of-5 case, improving throughput by 18 % while adding $<$0.2 s to p95 *Disperse* latency. The 250 ms idle flush ensures small objects ($<$1 MB) are not starved when the cluster is quiet, keeping DevOps "`kubectl port-forward`" demos snappy.

11) Default Stripe Size: 1 MiB: **Rationale.** Larger stripes improve encode throughput but increase memory pressure and per-object latency because Echo/Ready waits until the whole fragment is flushed. At 1 MiB, RS encoding saturates host memory bandwidth while keeping a 1 GiB object to 5 × 200 buffers—comfortably under a 32 GB laptop.

12) Retry Policy: Three Attempts, Exponential Back-Off: **Rationale.** Empirically, container-to-container TCP dials either succeed within 1 s or time out at 3 s. Three retries hit 97 % connectivity under 2 % random packet loss without delaying the happy path excessively. After three failures the client aborts, surfaces the peer list in logs, and lets higher-level automation decide.

13) Docker Compose over Helm: **Rationale.** Compose requires one YAML file, runs the same on Windows, macOS, and CI, and is already familiar to most students. Helm charts would add templating complexity and a dependency on a running Kubernetes cluster—overkill for a five-node demo.

14) CI Matrix Choices: **Rationale.** Testing both 3-of-5 and 4-of-6 configurations plus TLS/no-TLS ensures code paths for larger metadata blocks (extra shard) and TLS handshakes are exercised. Running the matrix on `ubuntu-latest` and `macos-13` catches platform idiosyncrasies such as case-insensitive file systems.

15) Non-Goals: Dynamic membership—joining/leaving servers would require a view-change protocol or external coordination; left as future work.
Geo-replication—RTT >20 ms would warrant pipeline batching and anti-entropy sync; outside scope.

Client-side encryption—orthogonal: users can encrypt before upload.

Summary: Every design choice—language, erasure code, fingerprint algorithm, consensus pattern, metadata store, persistence trick, container base image, batching thresholds, retry loops—was selected to minimise implementation risk, maximise performance-per-line, and smooth the operator experience. The resulting system contains no hand-rolled crypto, no bespoke consensus, and no heavyweight dependencies, yet delivers verifiable, Byzantine-tolerant storage at 100+ MB s^{-1} on commodity hardware.

VIII. Self-Reflection

Embarking on the AVID-FP Object Store pushed me well beyond my comfort zone in three important dimensions: rigorous protocol thinking, disciplined systems engineering, and DevOps polish. I began the semester convinced that implementing "just a storage backend" would mostly be an exercise in wiring existing libraries. Instead, the project evolved into a crash-course on how many small, easily overlooked decisions separate a theoretical protocol from a production-worthy service.

Bridging Theory and Practice: Reading the AVID-FP paper for the first time, I was captivated by its elegance: two gossip phases, linear fingerprints, provable guarantees. My naïve intuition was that translating the algorithm into code would be straightforward—hash, fingerprint, send, repeat. Reality intruded quickly. AVID-FP's safety proof assumes shards are either delivered intact or not at all; practical file systems offer no such binary promise. Discovering the importance of `rename()`-based atomic writes—and proving via fault-injection that no half-written fragment surfaces after a container kill—taught me the critical lesson that correctness arguments live and die on I/O corner cases.

Similarly, the paper's pseudocode treats fingerprint evaluation as free. On a real CPU, SHA-256 and Reed–Solomon dominate cache; careless goroutine spawning melts throughput. Profiling and semaphore throttling impressed on me that performance "just happens" only when the engineer is deliberate about CPU affinity, memory reuse, and batching. The exercise reinforced the maxim that an algorithm's asymptotic beauty must be reconciled with hardware reality.

Designing for Operational Empathy: In prior academic projects I stopped once the unit tests passed. This time the requirement to bundle Prometheus counters, Grafana dashboards, and rolling-upgrade scripts forced me to look through an operator's eyes. Until I saw p95 *Disperse* latency spike after adding TLS, I had not appreciated how easily an innocuous config change can torpedo an SLO. Building alert rules drove home that every metric should map to an action: a histogram bucket without a runbook entry is noise.

The same empathy shaped configuration. I originally exposed twelve CLI flags; after interviewing a TA, I realised YAML manifests and environment overrides are friendlier to CI and Kubernetes. Documentation rewrites followed: if an intern cannot disperse and retrieve an object in five minutes, the system has failed its first usability test. These operator-centric iterations nudged me toward cleaner abstractions and defensive logging—habits that will serve me in future professional roles.

Dealing with Failure—Technically and Psychologically: Implementing fault tolerance is humbling: half the work is engineering around your own mistakes. My first integration run showed "fingerprint mismatch" errors that traced back to sharing a single byte slice across goroutines—race conditions the Go runtime graciously surfaced when I enabled `-race`. Another bug emerged when I corrupted three shards: the client hung forever because I forgot to close a semaphore channel on early abort. Wrestling with these edge cases clarified why distributed-systems papers stress liveness just as much as safety.

Beyond code, I wrestled with time-management failures. Mid-semester I tried to optimise everything at once—TLS, locality codes, FUSE mount—only to stall core progress. A candid meeting with my advisor helped me re-focus on a minimal, verifiable slice. That pivot reinforced an Agile truth: ruthlessly prune scope, deliver something that works end-to-end, then iterate.

Skills Acquired:

- **Golang proficiency**—from interfaces and generics to `sync.Pool` and context timeouts, I now feel at home in production Go.
- **gRPC internals**—debugging retries and TLS handshakes demystified HTTP/2 flow control.
- **Prometheus & Grafana**—creating custom dashboards and alert rules adds a marketable DevOps arrow to my quiver.
- **Practical cryptographic hygiene**—I can articulate the collision-probability calculus behind 64- vs. 128-bit fingerprints.
- **CI discipline**—GitHub Actions workflows with Docker-in-Docker taught me to treat tests as non-negotiable gatekeepers, not optional chores.

Regrets and Future Directions: Dynamic membership—static peer lists are fine for a demo; real clusters churn. A Raft-backed view service or `etcd` integration is the next milestone.

Streaming encode/decode—holding an entire object in RAM is wasteful; chunked pipelines with SHA-256 tree-hashes could cut peak memory by 90 %.

Geo-latency experiments—all tests ran on a single host. Deploying across three regions on GCP or AWS would reveal RTT-sensitive bottlenecks.

Formal verification—TLA$^+$ or PlusCal could mechanically check the Echo/Ready state machine; time constraints limited me to manual reasoning.

Personal Take-Away: The most valuable outcome is confidence: I can now start from a research paper and shepherd the idea through code, tests, Docker images, metrics, and documentation—until someone unfamiliar with the project can run it and trust the results. That end-to-end ownership mindset,

more than any individual commit, is what I will carry into future work.

In short, the AVID-FP Object Store matured my understanding of how elegant algorithms meet messy reality and left me with pragmatic tools—and scars—to bridge that gap next time.

IX. Key Technical Insights, Conclusion, and Future Work

1Key Technical Insights

1.1 Linear fingerprints make integrity cheap for big objects: The dominant cost in conventional integrity checks is bandwidth: rereading an entire GiB-scale object just to recompute a SHA-256 is wasteful. By adding a 64-bit homomorphic fingerprint per shard, we reduced the client's validation workload to constant time per fragment and $\mathcal{O}(m)$ total bandwidth. The empirical result—under 6 % latency overhead compared with `scp`—confirms the analytical claim that fingerprints shift the integrity bottleneck from network to CPU cache, where modern processors deliver multi-gigabyte-per-second arithmetic.

1.2 Two-phase gossip is "just enough consensus" for dispersal: Full-blown state-machine replication (Raft, PBFT) would have imposed a shared WAL and leader election on every tiny write. Echo/Ready gossip, in contrast, pins consensus to an object-local fingerprint. That design eliminates long-term coordination state: once an object is committed the quorum counts reset, freeing memory and simplifying view changes. Measurements show that gossip adds two RTTs—240 ms on our bridge network—independent of object size, demonstrating a latency floor bounded only by network physics, not protocol chatter.

1.3 Storage latency, not network bandwidth, is the bottleneck: Even when the containers had an 8.6 Gb s^{-1} virtual NIC, write throughput plateaued at 108 MB s^{-1}. Prometheus revealed disk write saturation at 190 MB s^{-1} during fragment persistence. RS encoding and fingerprinting consumed ¡30 % CPU, so faster disks or log-structured object stores (e.g., SeaweedFS) would boost end-to-end performance without changing the protocol. The lesson: optimising CPU or network before the persistence layer is premature.

1.4 Batching small metadata writes is critical: Each *Disperse* call causes three BoltDB updates: record the FPCC, mark Echo received, and later mark Ready received. Naively issuing an `fsync` per update cut write throughput by 22 % and pushed p95 latency from 9.8 s to 13.5 s for 1 GiB objects. A micro-batcher that groups 100 puts or 250 ms of traffic restored throughput, proving that low-level I/O strategies can dwarf protocol optimisations.

1.5 Static linking + distroless images simplify threat containment: Linking the Go binary statically and embedding it in a distroless image (no shell, no package manager) reduced the potential attack surface dramatically—14 MB vs. 140 MB for Alpine—and eliminated `glibc` exploits. It also sped up cold starts (60 ms) and made SBOM generation trivial. The broader insight: delivery format can be a security feature, not just a packaging detail.

1.6 Observability drives design improvements: Metrics instrumentation exposed unexpected quirks: Echo broadcasts spiked CPU on one node because of mutex contention in BoltDB; visibility made the fix obvious—replace a `map[object]bool` with a `sync.Map` and avoid locking hot paths. Designing with observability in mind effectively converts production monitoring into a development-time profiler.

2Conclusion

This project set out to answer a straightforward question: *Can the AVID-FP protocol, previously confined to theory, be engineered into a practical, operator-friendly object store?* The answer is an emphatic yes. In roughly 2 000 lines of Go and a 14 MB container image, we produced a system that:

- stripes data with Reed–Solomon, achieving durability with 1.5–1.67× overhead instead of 3× replication;
- commits writes with Echo/Ready quorums, surviving up to $f = n–m$ crash or Byzantine nodes;
- verifies integrity in constant time per shard using homomorphic fingerprints, introducing less than 6 % latency overhead for GiB-scale objects;
- exposes Prometheus metrics out of the box, enabling SRE-grade dashboards and alerts;
- runs anywhere Docker runs, via distroless images and a one-file Docker Compose deployment;
- passes an exhaustive verification suite, including shard corruption, node crash-stop, and straggler scenarios.

Beyond functionality, the implementation process honed valuable engineering skills—performance profiling, atomic persistence, CI matrix design, and operator empathy. More importantly, it validates that strong cryptographic guarantees need not demand heavyweight cryptography or complex consensus logs. For edge clusters, sovereign clouds, and educational labs where trust assumptions are weaker and budgets are thinner, the AVID-FP Object Store offers an attractive middle ground: stronger than triple replication, lighter than Proofs of Retrievability.

3Future Work

3.1 Dynamic membership and rebalancing — integrate a lightweight view service for live joins/leaves and shard migration.

3.2 Streaming encode/decode — pipeline stripes to cut peak RAM by 90 %.

3.3 Locality and repair-bandwidth-aware codes — explore LRC or Clay while preserving fingerprint homomorphism.

3.4 Geo-distributed evaluation — deploy across multi-region clouds to measure WAN-induced latency.

3.5 Client-side encryption — add AES-GCM before encoding without breaking dispersal.

3.6 Pluggable fingerprint back-ends — allow 128- or 256-bit hashes for high-assurance domains.

3.7 Formal verification — model Echo/Ready in TLA^+ to

prove liveness under dynamic membership.

3.8 Deep observability—distributed tracing — integrate OpenTelemetry spans for shard-level critical-path analysis.

Closing Thought—Strong data durability, Byzantine robustness, and lightweight verification were once thought mutually exclusive outside academic idealisations. This project refutes that assumption. With judicious design and modern tooling, it is both feasible and practical to ship a secure, verifiable, and operator-friendly object store whose total deployment footprint is smaller than a typical container base image. The road ahead—streaming stripes, dynamic membership, geo-scale latency—will only enhance its applicability, but even in its current form the AVID-FP Object Store is poised to become a useful building block for edge computing, research clusters, and any domain where trust in storage cannot be assumed yet heavyweight crypto is a luxury.

REFERENCES

[1] J. Hendricks, G. R. Ganger, and M. K. Reiter, "Verifying Distributed Erasure-Coded Data," Carnegie Mellon University, 2007.

[2] K. Post, "Reed-Solomon Erasure Coding in Go," GitHub, 2023. [Online]. Available: https://github.com/klauspost/reedsolomon

[3] G. Ateniese *et al.*, "Provable Data Possession at Untrusted Stores," in *Proc. CCS*, 2007, pp. 598–609.

[4] A. Juels and B. S. Kaliski, "PORs: Proofs of Retrievability for Large Files," in *Proc. CCS*, 2007, pp. 584–597.