

OBJECT STORAGE **AS** INFRASTRUCTURE

BUILDING DATABASES, QUEUES,
GIT SYSTEMS, AND DISTRIBUTED
APPLICATIONS ON
S3-COMPATIBLE STORAGE



DATABASES



QUEUES



EVENT LOGS



GIT SYSTEMS



ANALYTICAL
PLATFORMS



DISTRIBUTED
APPLICATIONS



DESIGN RELIABLE SYSTEMS
THAT SCALE FROM SINGLE
SERVERS TO MULTI-REGION
DEPLOYMENTS



UNDERSTAND
CONSISTENCY AND
FAILURE MODES



ENGINEER WITH THE
STRENGTHS AND
CONSTRAINTS OF
OBJECT STORAGE



COMPLETE
PRODUCTION-GRADE
IMPLEMENTATIONS
IN GO

— SCOTT BARRET —

Object Storage as Infrastructure

Building Databases, Queues, Git Systems, and
Distributed Applications on S3-Compatible Storage

Steve Publications

This book is available at <https://leanpub.com/objectstorageasinfrastructure>

This version was published on 2026-08-26



This is a [Leanpub](#) book. Leanpub empowers authors and publishers with the Lean Publishing process. [Lean Publishing](#) is the act of publishing an in-progress ebook using lightweight tools and many iterations to get reader feedback, pivot until you have the right book and build traction once you do.

© 2026 Steve Publications

Contents

- Building Databases, Queues, Git Systems, and Distributed Applications on S3-Compatible Storage 1**
- Introduction 2**
- Chapter 1: The Object Storage Primitive 5**
 - What Object Storage Is and Is Not 5
 - The S3 API Surface: Objects, Keys, Prefixes, Metadata 6
 - Core Operations: PUT, GET, HEAD, DELETE, LIST, Multipart Upload . 8
 - Consistency Guarantees and Their Evolution 10
 - Durability, Availability, and Replication Strategies 11
 - Authentication, Authorization, and Security Primitives 13
 - Object Storage Versus Block File Database and Queue Abstractions . . 15
- Chapter 2: Designing Around Constraints 18**
 - Latency Throughput and Cost Realities 18
 - Immutable Objects and Update Semantics 19
 - Expensive Listing Operations and Namespace Design 21
 - Conditional Writes and Optimistic Concurrency 22
 - Metadata Layers and Indexing Strategies 23
 - Handling Partial Failures and Network Partitions 25
 - The Compute Separation Principle 27
- Chapter 3: S3-Compatible Implementations and Ecosystem 29**
 - AWS S3 Architecture and Behavior 29
 - Open Source Implementations: MinIO Ceph RGW 29
 - Cloud Provider Variants: GCS Azure Blob DigitalOcean Spaces 29
 - On-Premise and Air-Gapped Deployments 29
 - API Compatibility Gaps and Gotchas 29
 - Choosing and Evaluating an Implementation 29

Chapter 4: Content-Addressable Storage and Immutable Data Structures	31
Hash-Based Object Identification	31
Deduplication and Size Savings	31
Object Layout and Key Design	31
Implementation: A Content-Addressable Blob Store in Go	31
Testing and Verification	31
Performance Characteristics and Tuning	31
Chapter 5: Append-Only Event Logs on Object Storage	33
Log Semantics and Ordering Guarantees	33
Segment Design and Batching	33
Sequence Numbers and Fencing	33
Implementation: A Partitioned Event Log in Go	33
Compaction and Retention	33
Recovery from Partial Writes and Corruption	33
Chapter 6: Durable Queues and Message Streams on Object Storage	35
Queue Semantics and Delivery Guarantees	35
Visibility Tokens and Acknowledgement Patterns	35
Implementation: A Durable Queue on Object Storage	35
Partitioned Streams and Consumer Groups	35
Exactly-Once Considerations and Idempotency	35
Performance Comparison with Kafka and Traditional Brokers	35
Chapter 7: Key-Value Stores and Metadata Databases on Object Storage	37
Key-Value Semantics on Immutable Objects	37
SSTable-Inspired Layout and LSM Concepts	37
Indexes and Bloom Filters for Efficient Lookup	37
Implementation: A Lightweight KV Store in Go	37
Read Path Optimization and Caching Strategies	37
Compaction Snapshots and Garbage Collection	37
Chapter 8: Analytical Data Stores and Query Execution on Object Storage	39
Columnar Formats and Parquet on Object Storage	39
Partitioning Strategies for Query Performance	39
Predicate Pushdown and Filter Semantics	39
Implementation: A Simple Analytical Query Engine in Go	39
Schema Evolution and Compatibility	39
Metadata Catalogs and Table Management	39

Chapter 9: Git-Compatible Storage Backends on Object Storage 41

 Git Objects References and the DAG 41

 Immutable Storage and Deduplication in Git 41

 Packfiles and Delta Compression 41

 Designing an S3-Compatible Git Backend 41

 Ref Consistency and Concurrent Updates 41

 Implementation: A Git Service with Object Storage Backend 41

Chapter 10: Distributed Coordination on Object Storage 43

 Leases and Leader Election Patterns 43

 Distributed Locking with Conditional Writes 43

 Fencing Tokens and Session Management 43

 Configuration Stores and Feature Flags 43

 When to Use Object Storage Versus Dedicated Coordination Services 43

 Implementation: A Lease-Based Coordinator in Go 43

Chapter 11: Performance Engineering at Scale 45

 Measuring What Matters: RPS Throughput Tail Latency Cost 45

 Request Batching and Parallel Upload Strategies 45

 Connection Management and Asynchronous I/O 45

 Caching Architectures: Memory Local Disk CDN Layers 45

 Compression Serialization and Object Sizing 45

 Benchmarking Methodology and Realistic Tests 45

Chapter 12: Reliability Failure Handling and Recovery 47

 Taxonomy of Failures in Object Storage Systems 47

 Idempotent Operations and Safe Retries 47

 Handling Corrupted Objects and Integrity Verification 47

 Recovery Procedures for Each Component Type 47

 Fault Injection and Property-Based Testing 47

 Disaster Recovery and Multi-Region Architectures 47

Chapter 13: Security Access Control and Compliance 49

 Identity and Access Management Patterns 49

 Temporary Credentials and Workload Identity 49

 Encryption in Transit and at Rest 49

 Object Locking and Immutability Controls 49

 Tenant Isolation in Multi-Tenant Systems 49

 Security Auditing and Common Misconfigurations 49

Chapter 14: Observability Operations and Production Deployment . . .	51
Structured Logging and Request Tracing	51
Metrics SLIs and SLOs for Object Storage Systems	51
Capacity Planning and Lifecycle Management	51
Backup Restore and Migration Procedures	51
Incident Response and Operational Runbooks	51
Chapter 15: Case Studies from the Field	52
Cloud-Native Databases Built on Object Storage	52
Open-Source Projects Using S3 as Primary Storage	52
Streaming and Queue Systems with Object Backends	52
Git Hosting Platforms and Large-Scale Repositories	52
Lessons Learned and Common Pitfalls	52
What These Architectures Teach Us	52
Chapter 16: A Production Reference Architecture	54
Architecture Overview and Component Interactions	54
Storage Layout Across Services	54
Implementation: A Multi-Service System in Go	54
Deployment Configuration and Infrastructure Setup	54
Operational Procedures and Monitoring	54
Scaling Upgrading and Evolving the System	54
Conclusion	56
References	57

Building Databases, Queues, Git Systems, and Distributed Applications on S3-Compatible Storage

About this book: This book teaches experienced software engineers how to treat S3-compatible object storage as a first-class infrastructure primitive for building databases, message queues, event logs, version control systems, analytical platforms, and distributed applications. Through complete production-grade implementations in Go, you will learn to design around the strengths and constraints of object storage, reason about consistency and failure modes, and build reliable systems that scale from single servers to multi-region deployments. The goal is not to force every workload onto S3 but to understand the engineering boundaries of the abstraction so you can make deliberate architectural choices grounded in reality.

Introduction

In December 2020, Amazon announced a change that seemed almost too good to be true: S3 would provide strong read-after-write consistency for all operations at no additional cost. For fifteen years, engineers had built workarounds around eventual consistency, added coordination services to track object state, and accepted complexity because the storage layer could not guarantee that a successful write was immediately visible to subsequent reads. The announcement fundamentally altered how object storage could be used in production systems.

Two years later, in November 2024, AWS added conditional writes to S3. A single HTTP header now lets you say “write this object only if it does not already exist” or “write this object only if its ETag matches what I expect.” These are the building blocks of optimistic concurrency control and distributed locking. Combined with strong consistency, they transform S3 from a passive blob repository into an active coordination primitive.

Around the same period, several production systems emerged that treat S3-compatible storage not as backup destination but as primary infrastructure. TiDB X uses object storage as its single source of truth for all database data. AutoMQ replaces Kafka’s local disk segments with S3-backed streaming storage and turns brokers into stateless protocol routers. Open-source table formats like Apache Iceberg, Delta Lake, and Hudi layer transactional semantics on top of Parquet files in S3 to create lakehouse architectures that rival traditional data warehouses. Each system makes the same fundamental claim: object storage is durable enough, consistent enough, and scalable enough to serve as the foundation for serious workloads.

This book examines that claim from first principles. It does not ask whether you should use S3 instead of a database or Kafka or ZooKeeper. It asks what happens when you understand object storage deeply enough to build those systems yourself. The answer is not that object storage replaces every specialized system, but that the engineering patterns required to build on top of it teach you something fundamental about distributed systems, consistency models, failure handling, and performance trade-offs.

The reader this book targets has shipped production code, has dealt with race conditions and partial failures, understands what eventual consistency means in practice, and knows that every architectural choice is a trade-off between competing requirements. This book does not explain HTTP or mutexes or basic networking. It assumes you have enough background to appreciate why certain design decisions are hard and others are obvious once you see them.

The structure of the book follows the path from primitive to system. Part I establishes what object storage actually is, its API surface, its guarantees, and how it differs from every other storage abstraction engineers commonly use. Part II develops the core design patterns that compensate for object storage's limitations: immutable data structures, append-only logs, metadata layers, conditional writes, and optimistic concurrency control. Parts III through V walk through complete implementations of systems built on top of these patterns: content-addressable blob stores, event logs, message queues, key-value databases, analytical query engines, Git backends, and coordination services. Part VI addresses the production concerns that separate toy implementations from real infrastructure: performance engineering, reliability under failure, security, observability, operations, case studies from existing systems, and a reference architecture that ties everything together.

Every major implementation in this book is complete and runnable. Code examples are written in Go because it has excellent S3 SDK support, strong concurrency primitives, simple deployment characteristics, and widespread adoption in infrastructure projects. Each implementation includes the full source code with package structure, dependency files, local development setup using a MinIO-compatible server, test suites, command-line tools for interaction, Docker configuration where relevant, and build instructions. The goal is that you can clone the code, run it locally, modify it, break it, and fix it to develop real intuition about how these systems work.

The implementations are not production-ready in the sense of having survived years at a large company with dedicated SRE teams. They are production-relevant: they handle partial failures, implement proper retry policies, manage concurrent access, verify data integrity, and make realistic trade-offs between correctness, performance, and complexity. They demonstrate how to go from API calls to systems that behave predictably under load and in the presence of failures.

A few design choices warrant explanation upfront. The book uses S3 as its

reference implementation because it is the most widely deployed object storage system and has defined the API compatibility standard that others follow. However, the architectural principles are vendor-neutral and portable across S3-compatible implementations including MinIO, Ceph RGW, SeaweedFS, and cloud provider variants. Where behaviors differ between implementations, the book notes those differences explicitly.

The code uses the `aws-sdk-go-v2` library for S3 operations. This is a practical choice that reflects what most production systems actually use rather than hand-crafting HTTP requests. The SDK abstracts away authentication signing, retry logic, and connection management while still exposing all the primitives needed to implement higher-level semantics.

Finally, this book takes a particular stance on citations and references. Rather than cluttering chapters with inline citation markers that interrupt the narrative flow, there is one comprehensive References section at the end of the book containing full bibliographic details for every source used in the research and writing. This includes API documentation, engineering blog posts, academic papers, open-source repositories, and design documents. The goal is to make the material verifiable without sacrificing readability.

The journey ahead moves from understanding what object storage gives you to designing around what it does not. By the end, you should be able to look at a workload requirement and ask not just “what service should I use” but “what guarantees do I actually need, and can I build them on top of primitives that are cheaper, more durable, or more scalable than traditional alternatives.” That ability to reason from first principles is what this book is about.

Chapter 1: The Object Storage Primitive

Before we build anything on top of object storage, we need to understand exactly what it is and what it is not. This chapter establishes the vocabulary, API surface, guarantees, and mental model that everything else in this book depends on. If you have used S3 for uploading files or storing backups, you know some of this material already. The goal here is precision: enough detail that when we design databases and queues later, every decision flows from a clear understanding of the underlying primitive.

What Object Storage Is and Is Not

Object storage is a data management architecture where information is stored as discrete units called objects, each identified by a unique key within a container called a bucket. Each object consists of three components: the data itself (the payload or body), metadata describing properties of the object such as content type, size, and custom user-defined tags, and a globally unique identifier derived from its key and version.

This architecture differs fundamentally from the other storage abstractions engineers work with every day. Understanding those differences is essential because choosing the wrong abstraction for a workload creates problems that are expensive to fix later.

Block storage divides data into fixed-size blocks without hierarchy or metadata. Each block has an address, and the operating system or database manages how blocks map to files or tables. Block storage provides low-latency random access and in-place modification of individual blocks, making it ideal for databases, virtual machines, and transactional systems where rapid granular manipulation is critical.

File storage organizes data hierarchically into files within directories within filesystems. A path like `/projects/data/2024/reports.csv` uniquely identifies a file. File storage supports operations that object storage does not: in-place

modification of file contents, directory traversal, symbolic links, and POSIX semantics including permissions, ownership, and locking. Network-attached storage systems provide shared access to filesystems across multiple clients with strong consistency guarantees within the mount namespace.

Object storage trades these capabilities for scale and durability. Objects are immutable once written (you replace an object by writing a new version, not modifying bytes in place). There is no hierarchical directory structure in the traditional sense: keys are flat strings that may contain slash characters for organizational convenience but do not create actual directories. You cannot modify part of an object without rewriting it entirely. List operations return objects matching a prefix pattern rather than traversing a directory tree.

The trade-off yields remarkable benefits. Object storage scales horizontally to hold virtually unlimited data across geographically distributed infrastructure. Durability is engineered into the system at a level that block and file storage rarely match: S3 designs for eleven nines of durability, meaning that if you store ten million objects, you would statistically lose one object every ten thousand years. Availability is high without requiring manual replication management. The API is simple and RESTful, making it accessible from any environment with HTTP connectivity.

Object storage is not a replacement for block storage when you need low-latency random access to individual bytes. It is not a replacement for file storage when you need POSIX semantics or shared mounts across multiple clients. It is not inherently a database because it lacks indexing, querying, and transactional guarantees beyond single-object atomicity. But it is an excellent foundation for building systems that have those properties layered on top.

The most common mistake engineers make with object storage is treating it as a glorified network drive: uploading files to “backups” or “uploads” buckets without thinking about key design, metadata usage, lifecycle management, or access patterns. The second most common mistake is underestimating what you can build on top of it once you stop treating it that way. This book focuses on the latter.

The S3 API Surface: Objects, Keys, Prefixes, Metadata

The S3 API defines the interface through which we interact with object storage. While this book uses S3 terminology throughout, the same concepts apply

to any S3-compatible implementation. The API is HTTP-based and follows RESTful conventions, making it straightforward to use from any programming language or environment.

At its core, the API operates on two resource types: buckets and objects. A bucket is a globally unique namespace that contains objects. You create buckets with names that must be unique across all of S3 (not just within your account), follow DNS naming conventions, and are associated with a specific region at creation time. Once created, a bucket's name and region cannot change.

Objects live inside buckets. Each object is identified by a key: a UTF-8 encoded string that can be up to 1024 bytes long. The key is the primary identifier for an object within its bucket. There is no limit to how many objects a bucket can contain, and keys can represent any data you choose to store.

Keys often contain slash characters (/) for organizational purposes: `my-app/data/2024-01-15/events.json` looks like a file in a directory hierarchy, but S3 treats the entire string as a flat key with no special meaning attached to slashes. This convention is powerful because it lets us use prefix-based operations to simulate hierarchical behavior without actually implementing directories.

A prefix is any leading substring of a key. When you list objects with a prefix parameter, S3 returns all keys that start with that string. This is the primary mechanism for organizing and querying data in object storage. For example, listing with prefix `"my-app/data/2024-01-15/"` returns all objects stored under that date, effectively treating it as a directory listing even though no directories exist.

Each object carries metadata: system-defined properties like content type, size, and last-modified timestamp, plus user-defined metadata that you can set at upload time. User-defined metadata appears in HTTP headers with the `x-amz-meta-` prefix and is returned with GET and HEAD requests. This metadata mechanism is underutilized but critical for building advanced systems: it lets us store information about objects without embedding it in the object content or creating separate index files.

Object sizes range from zero bytes to five terabytes for a single upload (using multipart upload). The API does not distinguish between "files" and "blobs": everything is an object with bytes and metadata, and how you interpret those bytes is entirely up to your application.

ETags are another critical concept. Every object has an ETag value that represents its current version. For non-multipart uploads, the ETag is typically the MD5 hash of the object content. For multipart uploads, it is computed differently (MD5 of concatenated part hashes plus a dash and part count). ETags serve as opaque version identifiers that you can use for conditional operations: “only write this object if its current ETag matches what I expect” or “only read this object if its ETag has changed since last time.”

The combination of keys, prefixes, metadata, and ETags gives us enough primitives to build sophisticated systems. We will see exactly how in subsequent chapters.

Core Operations: PUT, GET, HEAD, DELETE, LIST, Multipart Upload

The S3 API exposes a well-defined set of operations that form our building blocks. Every system we build later will be implemented using only these operations plus the conditional and versioning features described below. Understanding each operation’s semantics, performance characteristics, and failure modes is essential.

PUT Object creates or replaces an object in a bucket. You specify the key, provide the data payload in the request body, and optionally set metadata headers. The operation is atomic: either the entire object is stored successfully, or no changes are made. S3 never stores partial objects. If you receive a successful response (HTTP 200 OK or 201 Created), you can be confident that the complete object was written to durable storage.

PUT Object supports conditional writes through HTTP headers. The If-None-Match header with value “*” tells S3 to create the object only if no object with that key currently exists. This is useful for “create if absent” semantics without race conditions. The If-Match header with an ETag value tells S3 to overwrite the object only if its current ETag matches, implementing optimistic concurrency control.

GET Object retrieves an object’s data and metadata. You specify the key, and S3 returns the full object body along with headers containing size, content type, ETag, and any user-defined metadata. GET Object supports range requests through the Range header, allowing you to retrieve specific byte ranges from large objects without downloading the entire payload. This is critical for

streaming video, parallel downloads of large files, and random access patterns within stored data.

HEAD Object returns only an object's metadata without transferring its data body. This operation is fast and cheap compared to GET because no payload bytes are transmitted. Use HEAD to check whether an object exists, verify its size or ETag before deciding to download it, or validate metadata without incurring data transfer costs. HEAD returns HTTP 404 Not Found if the object does not exist, making it a lightweight existence check.

DELETE Object removes an object from a bucket. The operation is atomic and immediate: after receiving a successful response, the object no longer exists (unless versioning is enabled). DELETE Objects (plural) lets you delete up to 1000 objects in a single request, reducing per-request overhead for bulk deletion operations.

LIST Objects returns information about objects whose keys match a specified prefix. The operation supports pagination through continuation tokens and can return up to 1000 object summaries per request. Each summary includes key name, size, last-modified timestamp, and ETag. LIST Operations are relatively expensive compared to GET or PUT because they require scanning index structures rather than direct object retrieval. They should be used judiciously in high-performance systems, a constraint we will work around extensively later.

Multipart Upload is the mechanism for uploading large objects efficiently. It consists of three phases: first, `CreateMultipartUpload` initiates the operation and returns an upload ID; second, `UploadPart` uploads individual chunks (parts) of the object in parallel using the upload ID; third, `CompleteMultipartUpload` assembles all parts into the final object. Parts can be uploaded in any order and in parallel across multiple connections, dramatically improving throughput for large objects compared to a single PUT request.

Multipart upload has important constraints. Each part must be at least five megabytes except the final part, which can be smaller. A single multipart upload can have up to 10,000 parts, limiting maximum object size to five terabytes. If `CompleteMultipartUpload` is never called, the upload remains incomplete and continues to consume storage until explicitly aborted or cleaned up by a lifecycle policy.

The multipart upload API also supports conditional completion: you can specify `If-Match` or `If-None-Match` headers on `CompleteMultipartUpload` to

ensure atomic “create if absent” semantics even for large uploads that took time to assemble.

All operations support authentication via AWS Signature Version 4, which signs HTTP requests with access keys and produces temporary credentials through the Security Token Service (STS). The signing process is complex but well-handled by SDKs, so we will rely on library implementations rather than implementing it manually.

Consistency Guarantees and Their Evolution

Consistency guarantees define what a client can expect when performing reads after writes in a distributed system. Object storage’s consistency model has evolved dramatically over time, and understanding that evolution explains both historical design patterns and modern capabilities.

When S3 launched in 2006, it provided read-after-write consistency for PUTs of new objects but eventual consistency for overwrites and deletes. This meant that after successfully writing a new object, any subsequent GET would return the latest version immediately. But if you overwrote an existing object or deleted one, there was no guarantee when other clients would see that change: reads might return stale data until replication completed across all storage nodes.

This inconsistency created significant engineering challenges. Systems built on S3 had to implement workarounds: adding unique suffixes to keys instead of overwriting (effectively versioning manually), using external coordination services to track object state, implementing retry loops with delays after writes, or accepting the risk of stale reads in non-critical paths.

The evolution toward stronger consistency happened incrementally. By August 2015, S3 provided read-after-write consistency for new objects in all regions (previously only the original US Standard endpoint behaved that way). In December 2020, AWS announced strong read-after-write consistency for all GET, PUT, and LIST operations globally. The change was automatic, free, and applied to all existing buckets without configuration changes.

Strong read-after-write consistency means exactly what it says: after a successful write operation (PUT or DELETE), any subsequent read operation (GET or LIST) returns the latest version of that object immediately. There is no

window where writes are durable but not yet visible. This guarantee holds for new objects, overwrites, and deletions alike.

This change fundamentally altered what is possible with S3 as a systems primitive. Conditional writes combined with strong consistency give us compare-and-swap semantics on individual keys. We can implement optimistic concurrency control without external coordination services. We can build leader election algorithms using conditional object creation. We can ensure that metadata files are visible immediately after being written, enabling atomic publication patterns that were previously unreliable.

It is important to note what strong consistency does not guarantee. S3 remains a single-region service: objects written in one region are not automatically visible in another region unless you configure cross-region replication (which is asynchronous and eventually consistent). Within a single region, the consistency model is strong for individual object operations but does not extend to multi-object transactions or atomicity across multiple keys.

Versioning adds another layer of complexity. When versioning is enabled on a bucket, DELETE operations create delete markers rather than removing data permanently. The consistency model applies to the current version: after writing a new version or creating a delete marker, subsequent reads without explicit version IDs see the latest version immediately. Reading specific versions by ID remains strongly consistent as well.

For systems design, the practical implication is this: within a single region and assuming strong consistency, we can rely on the fact that if operation A writes an object and operation B subsequently reads it (with no intervening failures), B sees what A wrote. This simple guarantee enables many patterns that were previously awkward or impossible. We will use it extensively throughout the book.

Durability, Availability, and Replication Strategies

Durability and availability are distinct properties that engineers often conflate. Durability asks whether data persists once written: if I successfully upload an object today, will it still exist tomorrow next month or ten years from now? Availability asks whether data is accessible when needed: if I try to read an object right now, will the system respond with the data rather than an error?

S3's durability guarantee is exceptional. The service is designed for eleven nines of durability (99.999999999 percent), meaning that for every ten million objects stored, you would statistically lose one object every ten thousand years. This is not marketing hyperbole but an engineering target achieved through careful design of replication, error correction, and failure recovery mechanisms.

The internal architecture that achieves this durability has never been fully disclosed by AWS, but we can infer key properties from available information and industry practice. Data is stored across multiple availability zones (at least three) within a region, each zone being a physically separate data center with independent power cooling and networking. Objects are not simply replicated as complete copies: that would require three times the storage capacity and still might not achieve eleven nines given correlated failure modes.

Instead, S3 uses erasure coding based on Reed-Solomon algorithms. Erasure coding splits an object into K data shards plus M parity shards distributed across storage nodes. The original object can be reconstructed from any K of the total $K+M$ shards. This provides the same or better fault tolerance as replication while using significantly less storage: a common configuration might use 10 data shards and 4 parity shards, requiring only 1.4 times the raw data size instead of three times for triple replication.

Backblaze published calculations showing that with drive failure rates around 0.81 percent annually, erasure coding can achieve eleven nines durability while triple replication achieves approximately six nines. The math is clear: erasure coding gives better durability per unit of storage because it tolerates more simultaneous failures without data loss.

Beyond the raw redundancy, S3 implements continuous background processes to detect and repair degradation. Background jobs regularly scan objects, verify their integrity using checksums, and regenerate any corrupted or missing shards from remaining healthy copies. This proactive maintenance prevents silent data corruption (bit rot) and ensures that transient failures do not compound into permanent data loss.

Availability is a separate concern. S3 Standard storage targets 99.99 percent availability, meaning roughly 52 minutes of downtime per year on average. This is high but not perfect: regional outages, networking issues, and operational incidents can render objects temporarily inaccessible even though they remain durable (not lost).

Different storage classes trade availability for cost. S3 Standard-Infrequent Access reduces cost at the expense of slightly lower availability and higher retrieval latency. S3 Glacier and Glacier Deep Archive further reduce cost while making data accessible only through restore operations that take minutes to hours. These tiers are designed for archival workloads where immediate access is not required but long-term durability is critical.

For systems built on top of S3, the durability guarantee means we can trust that objects written successfully will persist across hardware failures, data center failures, and even most regional disasters (if using cross-region replication). The availability guarantee means we must design for occasional unavailability: implement retry logic with exponential backoff, cache frequently accessed data when appropriate, and ensure that transient S3 errors do not cascade into application-level outages.

The combination of extreme durability with high but imperfect availability creates an interesting design space. We can build systems where the source of truth is immutable objects in S3 (trusting they will never be lost) while accepting that reads from that source occasionally fail temporarily and must be retried. This asymmetry between write durability and read availability shapes many of the patterns we develop later.

Authentication, Authorization, and Security Primitives

Object storage systems expose data over HTTP to potentially untrusted networks, making authentication and authorization critical infrastructure concerns. S3 provides a comprehensive security model that we will use extensively in our implementations. Understanding these mechanisms is not optional: security failures in object storage are among the most common and damaging cloud incidents.

Authentication establishes identity: who or what is making the request? S3 supports several authentication methods, but AWS Signature Version 4 (SigV4) is the primary mechanism for programmatic access. SigV4 signs each HTTP request using an access key ID, secret access key, region, service name, timestamp, and canonicalized request components. The signature proves that the requester possesses the secret key without transmitting it in plaintext over the network.

Long-term access keys (access key ID and secret key pairs) are problematic

for production systems. They can be leaked through code repositories, environment variables, or configuration files. Once compromised, they remain valid until manually rotated. Best practice is to avoid long-term keys entirely for workloads and use temporary credentials instead.

The Security Token Service (STS) issues temporary security credentials that expire after a configurable duration (typically one hour). Workloads running on EC2 instances, Lambda functions, or ECS tasks assume IAM roles that provide temporary credentials automatically through instance metadata or task role mechanisms. This approach eliminates the need to store long-term keys in application code and enables fine-grained control over which resources each workload can access.

For our implementations, we will use environment-based credential configuration: `AWS_ACCESS_KEY_ID`, `AWS_SECRET_ACCESS_KEY`, and optionally `AWS_SESSION_TOKEN` for temporary credentials. The SDK automatically discovers these credentials and uses them to sign requests. In production deployments, these would come from IAM roles rather than hardcoded values.

Authorization determines what an authenticated identity can do. S3 implements authorization through multiple overlapping mechanisms: bucket policies, access control lists (ACLs), IAM policies, and access point policies. Bucket policies are JSON documents attached to buckets that define which principals can perform which actions on which objects under which conditions. They are the primary mechanism for controlling access in production environments.

A typical bucket policy might allow a specific IAM role to perform `s3:GetObject` and `s3:PutObject` on objects within a particular prefix while denying all other access. Policies support condition keys that restrict operations based on criteria like IP address ranges, TLS requirements, or whether conditional headers are present.

S3 also supports presigned URLs: time-limited URLs generated by an authorized principal that grant temporary access to specific objects without requiring the recipient to have S3 credentials. Presigned URLs embed the authorization in the URL signature itself and expire after a configured duration (up to seven days for regular requests, one hour for SigV4). They are useful for sharing objects with external clients or enabling direct uploads from browsers without exposing credentials.

For our systems, we will apply least-privilege principles: each service component gets only the permissions it needs to function. A queue consumer might

need `s3:GetObject` and `s3:DeleteObject` on queue-related keys but nothing else. A database reader might need `s3:GetObject` on data files and metadata but no write permissions. This minimizes blast radius if credentials are compromised and makes security audits tractable.

Encryption adds another layer of protection. S3 encrypts all objects at rest by default using SSE-S3 (server-side encryption with S3-managed keys) since January 2023. For compliance requirements or finer control, SSE-KMS uses AWS Key Management Service keys that provide audit trails and key-level access control. Client-side encryption lets applications encrypt data before uploading so that S3 never sees plaintext content.

For production systems handling sensitive data, the recommendation is clear: use SSE-KMS with customer-managed keys for encryption at rest, enforce TLS for all connections, apply least-privilege IAM policies, rotate credentials regularly (or better yet, use only temporary credentials), and enable CloudTrail logging to track all S3 API calls. These practices are not optional for serious workloads.

Object Storage Versus Block File Database and Queue Abstractions

To understand when object storage is the right tool and when it is not, we need to compare it directly with other storage abstractions engineers use daily. Each abstraction makes different trade-offs optimized for different access patterns and consistency requirements.

Block storage divides data into fixed-size blocks (typically 4KB to 1MB) that can be read and written independently at arbitrary offsets. Block devices present a raw byte array interface: you write bytes at position X, later read bytes from position X, and get back exactly what you wrote (assuming no failures). This abstraction enables in-place modification, random access with predictable low latency, and efficient use of storage for changing data.

Databases run on block storage because they need to update individual records without rewriting entire files. Virtual machines use block storage because their filesystems expect byte-level read-write semantics. Block storage is the wrong choice when you need horizontal scalability beyond a single cluster, automatic replication across regions, or extremely high durability guarantees.

File storage builds hierarchical namespaces (directories and files) on top of block storage and adds POSIX semantics: in-place modification of file contents, atomic rename operations, directory traversal, symbolic links, file locking, permissions, ownership, and metadata like timestamps and access control lists. Network-attached storage systems provide shared filesystems accessible from multiple clients simultaneously with strong consistency within the mount namespace.

File storage is ideal for collaborative workflows where multiple users or services need to read and write files in a shared directory structure. It is the wrong choice when you need virtually unlimited scalability, programmatic access from heterogeneous environments without mounting filesystems, or durability guarantees that exceed what typical filesystems provide.

Databases add indexing, querying, transactions, and consistency models on top of storage abstractions. Relational databases support complex queries with joins, aggregations, and constraints while maintaining ACID transactional guarantees. Key-value stores optimize for fast point lookups by key. Document databases store semi-structured data with flexible schemas. Time-series databases optimize for append-heavy workloads and range queries over time.

Databases are the right choice when you need complex queries, strong consistency across multiple records, or sophisticated concurrency control. They are often the wrong choice when your primary concern is long-term durability of large immutable objects, when storage costs dominate operational budgets, or when you want to separate compute from storage for independent scaling.

Message queues and streaming platforms provide ordered delivery of messages between producers and consumers with guarantees about at-most-once, at-least-once, or exactly-once delivery semantics. Systems like Kafka maintain append-only logs partitioned by topic, support consumer groups for parallel processing, and guarantee ordering within partitions.

Queues are the right choice when you need reliable message delivery between decoupled services, ordered event processing, or backpressure mechanisms that protect producers from slow consumers. They typically use local disk storage optimized for sequential writes and reads, which makes them fast but operationally complex to scale and maintain.

Object storage occupies a different space in this landscape. It excels at storing large volumes of immutable data with extreme durability, global acces-

sibility through simple HTTP APIs, horizontal scalability without operational complexity, and cost efficiency through tiered storage classes. It is weak on random access latency (typically tens to hundreds of milliseconds per request), in-place modification (objects are immutable), complex queries (no built-in indexing), and low-latency ordered message delivery.

The insight this book explores is that object storage's strengths make it an excellent foundation for building systems that have the properties of databases, queues, and other abstractions layered on top. We cannot get the sub-millisecond latency of local block storage or the ACID transactions of a relational database, but we can build key-value stores with acceptable latency for many workloads, event logs with strong durability guarantees, message queues with at-least-once delivery, and analytical platforms that query petabytes of data efficiently.

The decision framework is straightforward: use object storage directly when your workload involves storing and retrieving large immutable objects with emphasis on durability and cost efficiency. Build higher-level systems on top of object storage when you need database-like or queue-like semantics but can tolerate higher latency than local storage provides and value the scalability, durability, and operational simplicity that object storage offers. Use traditional databases, block storage, or specialized queues when your requirements for latency, consistency, or transactional guarantees cannot be met by patterns built on top of object storage.

Understanding where each abstraction fits is more valuable than memorizing feature lists. The art of systems design lies in matching abstractions to requirements while accepting the trade-offs that come with each choice. Object storage is a powerful primitive when used for what it is good at and layered with appropriate abstractions to compensate for what it lacks.

Chapter 2: Designing Around Constraints

Object storage gives us durability, scale, and simplicity. It takes away in-place mutation, low-latency random access, cheap listing operations, and transactional guarantees across multiple objects. Every system we build on top of object storage must design around these constraints deliberately rather than hoping they do not matter.

This chapter is the engineering methodology core of this book. We will systematically enumerate each limitation of object storage as a systems primitive and introduce the patterns that compensate for it. These patterns are not theoretical: they appear in production systems from databases to version control to streaming platforms. Mastering them lets you build reliable infrastructure on top of object storage rather than fighting against its characteristics.

Latency Throughput and Cost Realities

The first constraint is performance. Object storage operations have latency measured in tens to hundreds of milliseconds, not microseconds or single-digit milliseconds like local disk access. A typical S3 GET request might take 20 to 100 milliseconds depending on object size, network conditions, geographic distance from the data center, and current load. PUT requests are similar or slightly slower due to replication overhead.

This latency is acceptable for many workloads but creates problems when compounded. If your system requires five S3 operations per request (read metadata, read data file, validate checksum, update index, write result), you face 100 to 500 milliseconds of storage latency before any application logic completes. For interactive applications with strict response time requirements, this is unacceptable.

Throughput tells a different story. While individual operations are slow, object storage excels at aggregate throughput through parallelism. S3 can handle at least 3,500 PUT or DELETE requests per second and 5,500 GET or

HEAD requests per second per partitioned prefix, with automatic scaling to higher rates as sustained load increases. Multipart uploads let you saturate network bandwidth by uploading parts in parallel across multiple connections. Range requests enable parallel downloads of large objects.

The design implication is clear: optimize for throughput through parallelism rather than trying to minimize individual operation count when latency matters. Batch operations where possible, use multipart uploads for large objects, download large files using range requests across multiple goroutines or threads, and design key layouts that spread load across many prefixes to avoid hot partitions.

Cost adds another dimension. S3 charges per gigabyte-month of storage plus per-request fees for API operations (PUT, GET, LIST, etc.) and data transfer out. For workloads with millions of small objects and frequent access patterns, request costs can exceed storage costs significantly. A system that performs ten LIST operations per second to find relevant objects is paying substantial operational expenses beyond raw storage.

The cost-aware design principle: minimize expensive operations (especially LIST) through smart key naming and metadata management, batch writes into larger objects rather than creating many small ones, cache frequently accessed data in memory or local storage when appropriate, and use lifecycle policies to transition infrequently accessed data to cheaper storage classes automatically.

Immutable Objects and Update Semantics

Object storage's immutability is both its greatest strength and most significant constraint. Once an object is written, its bytes cannot change. To “update” an object, you must write a new version with the same key (replacing the old one) or write to a different key entirely.

Immutability enables durability: there is no risk of torn writes where only part of an update completes and leaves data in an inconsistent state. It enables deduplication through content-addressable storage: identical objects can share physical storage because they will never diverge. It simplifies replication: copying an immutable object is straightforward because the source cannot change during transfer.

But immutability breaks patterns we take for granted with databases and filesystems. You cannot increment a counter by reading its current value,

adding one, and writing back: another writer might have modified it between your read and write, causing a lost update. You cannot append to a log file without rewriting the entire file each time. You cannot atomically update multiple related objects as part of a transaction.

The patterns that compensate for immutability fall into several categories.

Append-only designs avoid mutation entirely by only writing new data and never modifying existing objects. Event logs, write-ahead logs, and streaming platforms all use this pattern: producers append records to the end of a sequence, consumers read from where they left off, and old data is eventually compacted or deleted but never modified in place. We will implement an append-only log in Chapter 5.

Copy-on-write structures maintain multiple versions of data by copying only changed portions when updates occur. Version control systems like Git use this pattern extensively: each commit creates new tree objects for modified directories while sharing unchanged subtrees with previous commits. Database snapshots can be implemented similarly, creating a new version marker that points to newly written pages while reusing existing ones.

Optimistic concurrency control uses conditional writes to detect conflicts without locking. A client reads an object and its ETag, computes the desired update locally, then writes the new version using If-Match with the original ETag. If another writer modified the object in the meantime, the ETag no longer matches and the write fails with HTTP 412 Precondition Failed. The client can then retry with the latest version or report a conflict to the user. This pattern works well when conflicts are rare and retries are cheap.

Manifest files separate mutable metadata from immutable data. Instead of updating objects in place, systems write new data objects immutably and update a manifest file that describes which objects are currently valid and how they relate to each other. The manifest itself is updated using optimistic concurrency control or append-only patterns with compaction. This is the core pattern behind LSM-tree databases, lakehouse table formats, and many distributed storage systems.

Tombstones mark deleted or superseded data without physically removing it immediately. When a record is deleted, a tombstone marker (a special object or entry) indicates that the record should be treated as absent. Background compaction processes eventually remove both the original data and its tombstone, reclaiming storage space. Tombstones are essential for systems with

multiple versions or replicas where immediate deletion could cause inconsistencies.

The choice among these patterns depends on your workload characteristics. Append-only is ideal for logs and event streams where data only grows. Copy-on-write works well for versioned data and snapshots. Optimistic concurrency control suits workloads with infrequent conflicts. Manifest-based designs are appropriate when you need to coordinate many related objects atomically from the application's perspective.

Expensive Listing Operations and Namespace Design

LIST operations in S3 scan index structures to find keys matching a prefix pattern. They return metadata summaries (key name, size, timestamp, ETag) but not object contents. Despite seeming simple, LIST is expensive relative to GET or PUT for several reasons: it requires traversing internal data structures rather than direct object retrieval, it can return large result sets that consume network bandwidth, and it does not benefit from the same caching optimizations as individual object operations.

High-frequency listing creates real problems. A system that lists thousands of objects every few seconds to find work items or discover new data will incur significant request costs and latency. Worse, LIST has pagination limits: each request returns at most 1000 results, requiring multiple round-trips for larger result sets. For buckets with millions of objects, a full prefix scan can take seconds or minutes.

The solution is namespace design: organizing keys so that your application rarely needs to list large numbers of objects and can instead compute the exact key it needs from known information.

Hierarchical prefixes simulate directories while enabling efficient targeted listing. Instead of storing all events in a flat namespace (`events/1.json`, `events/2.json`), organize by time or category: `events/2024-01-15/type-a/001.json`. Now listing events for a specific day and type returns only relevant objects without scanning the entire bucket. This pattern appears in almost every production S3 workload.

Computed keys eliminate listing entirely for known objects. If you know the message ID, partition number, and sequence, compute the key directly: `queue/partition-3/msg-0001234.json`. No listing required. This is how most

queue and database implementations on S3 work: metadata tells you where to find each object, so you never need to scan for it.

Index objects replace listing with targeted reads. Instead of listing all objects matching some criteria, maintain an index file that maps search keys to object locations. When you need to find objects by attribute, read the index (a single GET) rather than listing thousands of candidates. The index itself is updated when objects are created or deleted, using append-only patterns with periodic compaction to keep it efficient.

Prefix sharding distributes load and limits list scope. If a single prefix receives too many requests, S3 automatically partitions it internally, but you can proactively shard by adding random or hash-based components: `data/shard-a1b2c/file.json` rather than `data/file.json`. This prevents hot prefixes under high load and keeps individual LIST operations small even as total object count grows.

The golden rule: design your key namespace around how you will access objects, not around how they are conceptually organized in your application's mental model. If you always read objects by ID, include the ID in the key path. If you always query by date range, organize keys by date prefixes. If you need to list objects occasionally, ensure each list operation targets a small, well-defined subset rather than scanning the entire bucket.

Conditional Writes and Optimistic Concurrency

Conditional writes arrived in S3 relatively late (November 2024) but fill a critical gap for building concurrent systems. Before conditional writes, there was no safe way to implement “create if absent” semantics or optimistic concurrency control without external coordination services.

The If-None-Match: * header on PUT tells S3 to create the object only if no object with that key currently exists. This is atomic at the S3 level: two concurrent writers using the same key and both specifying If-None-Match will result in one succeeding (HTTP 200) and one failing (HTTP 412 Precondition Failed). There is no race condition because S3 serializes the check-and-write internally.

This single capability enables several important patterns. Leader election becomes possible: candidates attempt to create a lock file with an incrementing epoch number using If-None-Match, and whichever succeeds becomes

leader. Distributed locks can be implemented similarly: acquiring a lock means creating a lock object conditionally, releasing it means deleting the object or marking it expired. Idempotent creation of resources works naturally: “create this configuration file if it does not exist” is a single conditional PUT with no external state to track.

The If-Match header with an ETag value implements optimistic concurrency control for updates. A client reads an object and its ETag, computes the desired modification locally, then writes the new version specifying If-Match with the original ETag. S3 overwrites only if the current ETag still matches; otherwise it returns 412 Precondition Failed indicating that another writer modified the object since the client read it.

Optimistic concurrency control works well when conflicts are infrequent. In a configuration management system where different services update different sections, conflicts might be rare enough that retrying on 412 is acceptable overhead. In a high-contention workload where many writers modify the same key simultaneously, optimistic concurrency degrades due to repeated conflicts and retries.

The pattern for handling conflicts is straightforward: on receiving 412, read the current version, merge your changes with it (or abort if merging is not safe), compute the new ETag, and retry the conditional write. Limit retries with exponential backoff to avoid thundering herds when contention is high. After a configurable number of failures, report the conflict to the caller rather than retrying indefinitely.

Conditional writes also apply to CompleteMultipartUpload, enabling atomic “create if absent” semantics for large uploads that took time to assemble. This is important for scenarios where multiple producers might try to upload the same logical object: only one succeeds, others fail cleanly without partial or duplicate data.

For systems design, conditional writes remove the need for external coordination services in many cases. You do not need ZooKeeper or etcd to implement simple leader election or distributed locks when you can use S3 conditional PUT directly. You do not need a separate metadata database to track whether an object was created successfully: the HTTP response code tells you atomically. This simplification reduces infrastructure complexity and operational overhead significantly.

Metadata Layers and Indexing Strategies

Object storage gives us objects with keys, payloads, and user-defined metadata headers. That is powerful but insufficient for most application workloads, which need to query data by attributes other than key prefix, find objects matching complex criteria, or maintain relationships between objects that are not expressible through key naming alone.

The solution is to build a metadata layer on top of object storage: separate structures that track information about objects and enable efficient discovery without expensive listing operations. The metadata layer itself lives in object storage (or sometimes in an external database for low-latency requirements), but it provides query capabilities that raw S3 does not offer.

Append-only metadata logs record changes as they happen. Every create, update, or delete operation writes a record to a metadata log: timestamp, operation type, affected key, and relevant attributes. To find objects matching criteria, scan the metadata log rather than listing data objects. The log is append-only for simplicity and durability, with compaction processes that merge redundant entries over time.

Index objects map attribute values to object locations. A simple index might be a JSON file mapping product IDs to their storage keys: {"prod-123": "data/products/prod-123.json", "prod-456": "data/products/prod-456.json"}. More sophisticated indexes support range queries, multi-attribute lookups, or inverted indices for text search. Indexes are updated when objects change, using append-only patterns with periodic rebuilding to remove stale entries.

Manifest files describe the current state of a collection of objects. A database manifest lists which data files (SSTables, segments, partitions) currently exist and what key ranges they cover. A lakehouse table manifest tracks which Parquet files belong to the table, their row counts, partition values, and statistics for query optimization. The manifest is the single source of truth for what objects are valid: background processes may leave stale data objects in S3, but anything not listed in the current manifest is logically deleted.

The manifest pattern is particularly powerful because it decouples physical storage from logical state. You can write new data objects before updating the manifest (ensuring durability), then update the manifest atomically using conditional writes to publish the change. Readers always read the current manifest first to know which objects are valid, so they never see partially

published changes. This gives you atomicity at the collection level even though individual object operations are independent.

Metadata can live in different places depending on latency requirements. If metadata reads must be fast (sub-millisecond), store them in an external database or cache like Redis or DynamoDB while using S3 only for large data objects. If moderate latency is acceptable (tens to hundreds of milliseconds), store metadata as small objects in S3 alongside data. For systems where consistency matters more than speed, keep all metadata in S3 and accept the latency trade-off.

The hybrid approach is common in production: hot metadata that needs fast access lives in a low-latency store, cold metadata and large data objects live in S3. The key insight is that you choose the right storage for each piece of information based on its access pattern rather than forcing everything into one abstraction.

Handling Partial Failures and Network Partitions

Object storage is highly durable but not perfectly available. Networks fail, requests time out, responses are lost, and regional outages occasionally occur. Every system built on top of S3 must handle these failures gracefully without losing data or corrupting state.

The taxonomy of failures we need to handle includes: request timeouts where the client sends a request but never receives a response (did S3 process it?), partial write failures where multipart upload parts succeed but completion fails, read failures where GET returns an error temporarily, network partitions that isolate clients from storage for extended periods, and regional outages that make an entire region's S3 endpoint unavailable.

Idempotent operations are the foundation of safe retry logic. An operation is idempotent if executing it multiple times has the same effect as executing it once. PUT Object with a known key and content is idempotent: writing the same object twice produces the same final state. DELETE Object is idempotent: deleting an already-deleted object succeeds (or returns 404, which is equivalent to “already done”). Conditional writes are idempotent when conditions are stable: If-None-Match on a key that already exists always fails with 412.

For non-idempotent operations (like appending to a log), use conditional writes or sequence numbers to ensure each operation executes exactly once. An append operation might use If-None-Match to create an object at a specific offset: if another writer already claimed that offset, the retry fails and the client can detect the duplicate.

Retry policies must be carefully designed. Simple retries on every error cause thundering herds when S3 is under load or experiencing regional issues. Exponential backoff with jitter spaces out retries: wait 100ms, then 200ms, then 400ms, adding random variation to prevent synchronized retry storms. Distinguish between transient errors (5xx responses, timeouts, network errors) that warrant retries and permanent errors (4xx responses like AccessDenied or NoSuchKey) that do not.

Timeouts protect against hanging operations. Configure reasonable timeouts based on operation type: small GET requests might timeout after two seconds, large multipart uploads after several minutes. On timeout, assume the operation may have succeeded (the request reached S3 but the response was lost) and verify idempotently before retrying. For conditional writes, this means checking whether the object exists with expected content before attempting again.

Network partitions require different handling depending on system requirements. If your system must continue operating during brief partitions, implement local caching: cache frequently read metadata in memory or on local disk so that reads can proceed even when S3 is unreachable. Writes during a partition queue locally and replay when connectivity restores, using conditional writes to handle conflicts with any changes made by other nodes during the outage.

For extended partitions or regional outages, multi-region architectures provide resilience. Cross-region replication keeps copies of objects in geographically separate regions. Active-passive failover routes traffic to a secondary region when the primary is unavailable. Active-active architectures accept eventual consistency between regions while allowing writes to both simultaneously, using conflict resolution strategies (last-write-wins, vector clocks, application-level merging) to reconcile differences.

The practical guidance: assume every S3 operation can fail temporarily, design all operations to be safely retryable with idempotency guarantees, implement exponential backoff with jitter for retries, cache metadata locally

when read latency matters, and plan for regional outages if your workload requires high availability across geographic distances. These practices are not optional for production systems.

The Compute Separation Principle

The most important architectural insight enabled by object storage is the separation of compute from storage. Traditional databases couple data and processing: the database server that executes queries also stores data on local disks. Scaling requires adding more servers with proportionally more storage, creating tight coupling between compute capacity and storage capacity.

Object storage breaks this coupling. Data lives in S3 independently of any compute resource. Application servers, query engines, or batch processors can read from and write to S3 without owning the underlying storage. Compute resources scale up and down independently: spin up hundreds of workers to process a batch job, then terminate them when done while data remains safely in S3. Query engines start on demand, scan data directly from S3, and shut down when queries complete.

This separation yields dramatic benefits. Storage costs are decoupled from compute costs: you pay for storage based on data volume regardless of how many queries run against it. Compute scales elastically to match workload demands without provisioning storage ahead of time. Different workloads can access the same data without replication: a batch ETL job, an interactive query engine, and a machine learning training pipeline all read from the same S3 objects simultaneously.

Snowflake pioneered this architecture for cloud data warehousing: storage layer in S3, compute layer as independently scalable virtual warehouses, service layer handling query optimization and metadata management. Databricks built the lakehouse concept on similar principles: data files (Parquet) live in S3 while Spark clusters process them on demand. Modern systems like TiDB X use S3 as the single source of truth for database storage while compute nodes scale independently.

For our implementations, we will follow this principle consistently. Application servers are stateless: they read from and write to S3 but maintain no persistent data locally beyond temporary caches. Scaling means adding or removing server instances without migrating data. Batch processors start

when work is available, process objects from S3, and terminate when complete. This architecture simplifies operations dramatically compared to managing stateful database clusters.

The trade-off is latency: every operation requires a network round-trip to S3 rather than accessing local disk. For batch processing and analytical queries that scan large amounts of data, this overhead is negligible compared to total query time. For interactive workloads requiring sub-millisecond response times, caching layers or hybrid architectures combine S3 durability with local storage speed.

The compute separation principle is not just about cost optimization. It fundamentally changes how we think about system architecture: storage becomes a shared resource that multiple independent compute components access rather than a private asset owned by a single service. This shift enables more flexible, scalable, and resilient designs than traditional coupled architectures allow.

Chapter 3: S3-Compatible Implementations and Ecosystem

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/objectstorageasinfrastructure>.

AWS S3 Architecture and Behavior

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/objectstorageasinfrastructure>.

Open Source Implementations: MinIO Ceph RGW

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/objectstorageasinfrastructure>.

Cloud Provider Variants: GCS Azure Blob DigitalOcean Spaces

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/objectstorageasinfrastructure>.

On-Premise and Air-Gapped Deployments

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/objectstorageasinfrastructure>.

API Compatibility Gaps and Gotchas

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/objectstorageasinfrastructure>.

Choosing and Evaluating an Implementation

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/objectstorageasinfrastructure>.

Chapter 4: Content-Addressable Storage and Immutable Data Structures

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/objectstorageasinfrastructure>.

Hash-Based Object Identification

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/objectstorageasinfrastructure>.

Deduplication and Size Savings

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/objectstorageasinfrastructure>.

Object Layout and Key Design

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/objectstorageasinfrastructure>.

Implementation: A Content-Addressable Blob Store in Go

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/objectstorageasinfrastructure>.

Testing and Verification

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/objectstorageasinfrastructure>.

Performance Characteristics and Tuning

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/objectstorageasinfrastructure>.

Chapter 5: Append-Only Event Logs on Object Storage

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/objectstorageasinfrastructure>.

Log Semantics and Ordering Guarantees

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/objectstorageasinfrastructure>.

Segment Design and Batching

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/objectstorageasinfrastructure>.

Sequence Numbers and Fencing

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/objectstorageasinfrastructure>.

Implementation: A Partitioned Event Log in Go

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/objectstorageasinfrastructure>.

Compaction and Retention

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/objectstorageasinfrastructure>.

Recovery from Partial Writes and Corruption

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/objectstorageasinfrastructure>.

Chapter 6: Durable Queues and Message Streams on Object Storage

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/objectstorageasinfrastructure>.

Queue Semantics and Delivery Guarantees

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/objectstorageasinfrastructure>.

Visibility Tokens and Acknowledgement Patterns

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/objectstorageasinfrastructure>.

Implementation: A Durable Queue on Object Storage

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/objectstorageasinfrastructure>.

Partitioned Streams and Consumer Groups

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/objectstorageasinfrastructure>.

Exactly-Once Considerations and Idempotency

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/objectstorageasinfrastructure>.

Performance Comparison with Kafka and Traditional Brokers

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/objectstorageasinfrastructure>.

Chapter 7: Key-Value Stores and Metadata Databases on Object Storage

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/objectstorageasinfrastructure>.

Key-Value Semantics on Immutable Objects

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/objectstorageasinfrastructure>.

SSTable-Inspired Layout and LSM Concepts

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/objectstorageasinfrastructure>.

Indexes and Bloom Filters for Efficient Lookup

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/objectstorageasinfrastructure>.

Implementation: A Lightweight KV Store in Go

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/objectstorageasinfrastructure>.

Read Path Optimization and Caching Strategies

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/objectstorageasinfrastructure>.

Compaction Snapshots and Garbage Collection

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/objectstorageasinfrastructure>.

Chapter 8: Analytical Data Stores and Query Execution on Object Storage

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/objectstorageasinfrastructure>.

Columnar Formats and Parquet on Object Storage

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/objectstorageasinfrastructure>.

Partitioning Strategies for Query Performance

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/objectstorageasinfrastructure>.

Predicate Pushdown and Filter Semantics

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/objectstorageasinfrastructure>.

Implementation: A Simple Analytical Query Engine in Go

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/objectstorageasinfrastructure>.

Schema Evolution and Compatibility

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/objectstorageasinfrastructure>.

Metadata Catalogs and Table Management

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/objectstorageasinfrastructure>.

Chapter 9: Git-Compatible Storage Backends on Object Storage

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/objectstorageasinfrastructure>.

Git Objects References and the DAG

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/objectstorageasinfrastructure>.

Immutable Storage and Deduplication in Git

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/objectstorageasinfrastructure>.

Packfiles and Delta Compression

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/objectstorageasinfrastructure>.

Designing an S3-Compatible Git Backend

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/objectstorageasinfrastructure>.

Ref Consistency and Concurrent Updates

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/objectstorageasinfrastructure>.

Implementation: A Git Service with Object Storage Backend

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/objectstorageasinfrastructure>.

Chapter 10: Distributed Coordination on Object Storage

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/objectstorageasinfrastructure>.

Leases and Leader Election Patterns

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/objectstorageasinfrastructure>.

Distributed Locking with Conditional Writes

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/objectstorageasinfrastructure>.

Fencing Tokens and Session Management

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/objectstorageasinfrastructure>.

Configuration Stores and Feature Flags

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/objectstorageasinfrastructure>.

When to Use Object Storage Versus Dedicated Coordination Services

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/objectstorageasinfrastructure>.

Implementation: A Lease-Based Coordinator in Go

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/objectstorageasinfrastructure>.

Chapter 11: Performance Engineering at Scale

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/objectstorageasinfrastructure>.

Measuring What Matters: RPS Throughput Tail Latency Cost

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/objectstorageasinfrastructure>.

Request Batching and Parallel Upload Strategies

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/objectstorageasinfrastructure>.

Connection Management and Asynchronous I/O

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/objectstorageasinfrastructure>.

Caching Architectures: Memory Local Disk CDN Layers

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/objectstorageasinfrastructure>.

Compression Serialization and Object Sizing

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/objectstorageasinfrastructure>.

Benchmarking Methodology and Realistic Tests

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/objectstorageasinfrastructure>.

Chapter 12: Reliability Failure Handling and Recovery

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/objectstorageasinfrastructure>.

Taxonomy of Failures in Object Storage Systems

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/objectstorageasinfrastructure>.

Idempotent Operations and Safe Retries

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/objectstorageasinfrastructure>.

Handling Corrupted Objects and Integrity Verification

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/objectstorageasinfrastructure>.

Recovery Procedures for Each Component Type

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/objectstorageasinfrastructure>.

Fault Injection and Property-Based Testing

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/objectstorageasinfrastructure>.

Disaster Recovery and Multi-Region Architectures

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/objectstorageasinfrastructure>.

Chapter 13: Security Access Control and Compliance

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/objectstorageasinfrastructure>.

Identity and Access Management Patterns

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/objectstorageasinfrastructure>.

Temporary Credentials and Workload Identity

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/objectstorageasinfrastructure>.

Encryption in Transit and at Rest

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/objectstorageasinfrastructure>.

Object Locking and Immutability Controls

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/objectstorageasinfrastructure>.

Tenant Isolation in Multi-Tenant Systems

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/objectstorageasinfrastructure>.

Security Auditing and Common Misconfigurations

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/objectstorageasinfrastructure>.

Chapter 14: Observability Operations and Production Deployment

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/objectstorageasinfrastructure>.

Structured Logging and Request Tracing

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/objectstorageasinfrastructure>.

Metrics SLIs and SLOs for Object Storage Systems

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/objectstorageasinfrastructure>.

Capacity Planning and Lifecycle Management

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/objectstorageasinfrastructure>.

Backup Restore and Migration Procedures

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/objectstorageasinfrastructure>.

Incident Response and Operational Runbooks

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/objectstorageasinfrastructure>.

Chapter 15: Case Studies from the Field

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/objectstorageasinfrastructure>.

Cloud-Native Databases Built on Object Storage

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/objectstorageasinfrastructure>.

Open-Source Projects Using S3 as Primary Storage

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/objectstorageasinfrastructure>.

Streaming and Queue Systems with Object Backends

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/objectstorageasinfrastructure>.

Git Hosting Platforms and Large-Scale Repositories

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/objectstorageasinfrastructure>.

Lessons Learned and Common Pitfalls

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/objectstorageasinfrastructure>.

What These Architectures Teach Us

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/objectstorageasinfrastructure>.

Chapter 16: A Production Reference Architecture

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/objectstorageasinfrastructure>.

Architecture Overview and Component Interactions

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/objectstorageasinfrastructure>.

Storage Layout Across Services

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/objectstorageasinfrastructure>.

Implementation: A Multi-Service System in Go

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/objectstorageasinfrastructure>.

Deployment Configuration and Infrastructure Setup

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/objectstorageasinfrastructure>.

Operational Procedures and Monitoring

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/objectstorageasinfrastructure>.

Scaling Upgrading and Evolving the System

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/objectstorageasinfrastructure>.

Conclusion

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/objectstorageasinfrastructure>.

References

This content is not available in the sample book. The book can be purchased on Leanpub at <https://leanpub.com/objectstorageasinfrastructure>.