AdriaBOX

A Decentralized Storage Matrix with Synchronous Pipeline Replication and Zero-Knowledge Encryption

Final Report for the Distributed Systems Course

Sajmir Buzi
sajmir.buzi@studio.unibo.it

Massimiliano Pupillo
massimiliano.pupillo@studio.unibo.it

Academic Year 2025/2026

Abstract

This report details the comprehensive architectural design, theoretical constraints, and operational mechanisms of AdriaBOX, a resilient, decentralized object storage system engineered over a multi-tenant cluster environment. AdriaBOX natively addresses the core challenges of distributed computing by separating high-performance binary transport protocol engines from global metadata orchestration. By employing a customized application-layer framed TCP streaming layer alongside a synchronous pipeline replication cascade, the system isolates fault domains, prevents resource starvation via strict boundary memory footprinting, and implements Zero-Knowledge client-side encryption. Crucially, to validate system behavior under realistic execution stresses, a multi-containerized emulation workspace is leveraged. This environment allows the simulation of concurrent, fully isolated tenant workspaces interacting across a virtualized network plane, proving that horizontal capacity scaling and hardware-level fault tolerance can be sustained without compromising strong sequential data consistency under severe network disruption.

Disclaimer

During the preparation of this work, the authors used AI tools like Copilot and Gemini and automated services to assist with structural code optimization, debugging session analysis, and LaTeX typographical documentation layout drafting. After using these tools, the authors thoroughly reviewed, refactored, and edited the generated content as needed. The authors take full and undivided responsibility for the architectural correctness, scientific rigor, and technical integrity of the content published within this final report and the accompanying software artifact.

Contents

1 Concept and Architectural Foundation

This report presents the architectural specification and system validation of **AdriaBOX**, an enterprise-grade, multi-tenant distributed object storage matrix engineered to provide fault-tolerant, highly available, and cryptographically isolated asset persistence over a cluster of independent storage nodes. AdriaBOX addresses the structural design and performance limits inherent in legacy file systems by applying a strict separation of concerns across a multi-component topology.

1.1 System Classification and Decoupled Architecture

Architecturally, AdriaBOX is implemented as a highly decoupled, heterogeneous computing infrastructure. The entire platform is partitioned into two distinct functional operating horizons, separating the control pathways from high-throughput binary transmission lines:

- **The Control Plane (Orchestration Core):** Represented by a centralized Metadata Controller running as an asynchronous, stateful Web-service backed by the Flask framework. The control plane acts as the authoritative source of truth for the cluster, exposing RESTful endpoints tasked exclusively with tenant session authentication, cryptographic token issuance, relational storage planning, and real-time node telemetry auditing.
- **The Data Plane (Decentralized Worker Fleet):** Composed of a horizontal fleet of autonomous, multi-threaded background storage node daemons. These daemons are bound to independent network interfaces and task-isolated storage partitions, operating as low-level binary engines optimized for raw byte extraction and synchronous pipeline replication cascade enforcement.

The global operational boundary, network plane isolation, and multi-tier component distribution are formalized in the infrastructure blueprint detailed in Figure 1.

The transition from procedural, monolithic paradigms—typical of legacy software frameworks written in standard C—into an object-oriented Python architecture is driven by the necessity to implement clean component abstraction barriers. Python 3.14 provides native support for structured class inheritance, robust error trapping, and state virtualization facades. This choice allows the platform to encapsulate complex network socket streams and multi-layered database transactions inside highly maintainable object interfaces, while maintaining complete access to underlying system-level calls.

1.2 The Object Storage Paradigm and Flat Namespace Mathematics

Unlike traditional centralized distributed file systems that recursively map directory layouts as logical structural trees using parent-child relational references, AdriaBOX adopts an **Object Storage Flat Namespace** paradigm. Within the storage nodes and the centralized database tracking maps, directories do not exist as physical or logical entities. Folders are instead abstracted as virtual textual string prefixes integrated directly into the object identifier keys. For example, a file visualized by the client interface as residing within a multi-tiered folder hierarchy is recorded globally as a single string key token:

$$\text{Object Key Index: } K = \text{“/documents/pizzeria/file\_example.avi”}$$

The absolute structural benefits of this design choice become clear when evaluating the temporal complexity of global metadata operations. In a standard filesystem, renaming a directory or moving a branch containing thousands of deeply nested files requires a blocking

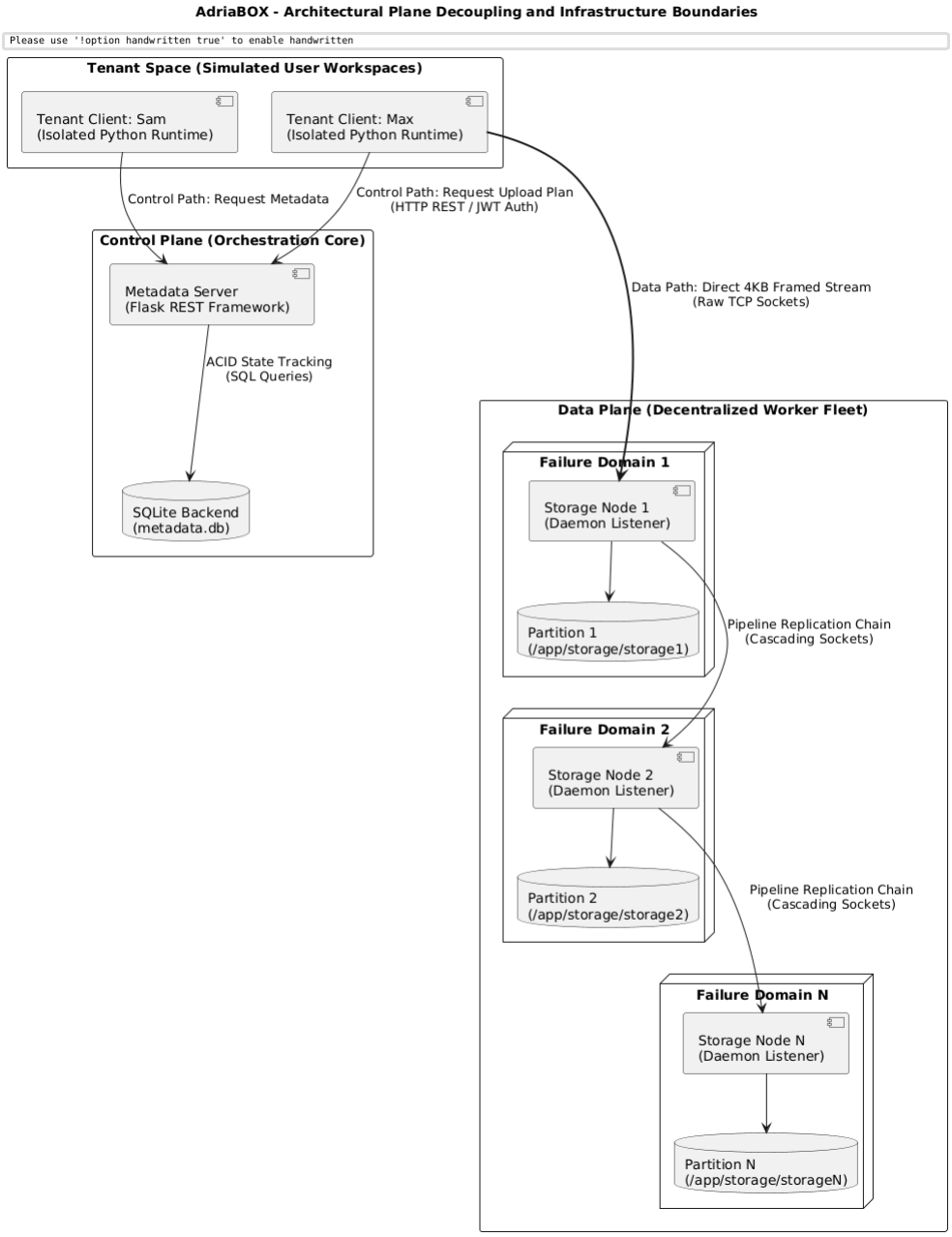

Figure 1: AdriaBOX Architectural Plane Decoupling and Infrastructure Plane Separation Boundary.

execution loop to recursively traverse relational keys or manipulate physical disk blocks. In AdriaBOX, because paths are merely string key properties within a relational matrix, actions like folder creation (`mkdir`) or directory prefix migration (`mv`) achieve an execution temporal efficiency of exactly $O(1)$ complexity. Modifying an entire namespace section is instantaneous and non-blocking; it is executed via a single, atomic SQL update statement targeting matching string prefix patterns on the control plane database, completely bypassing any data repositioning across the storage cluster. This mapping layout is formalized in Figure 2.

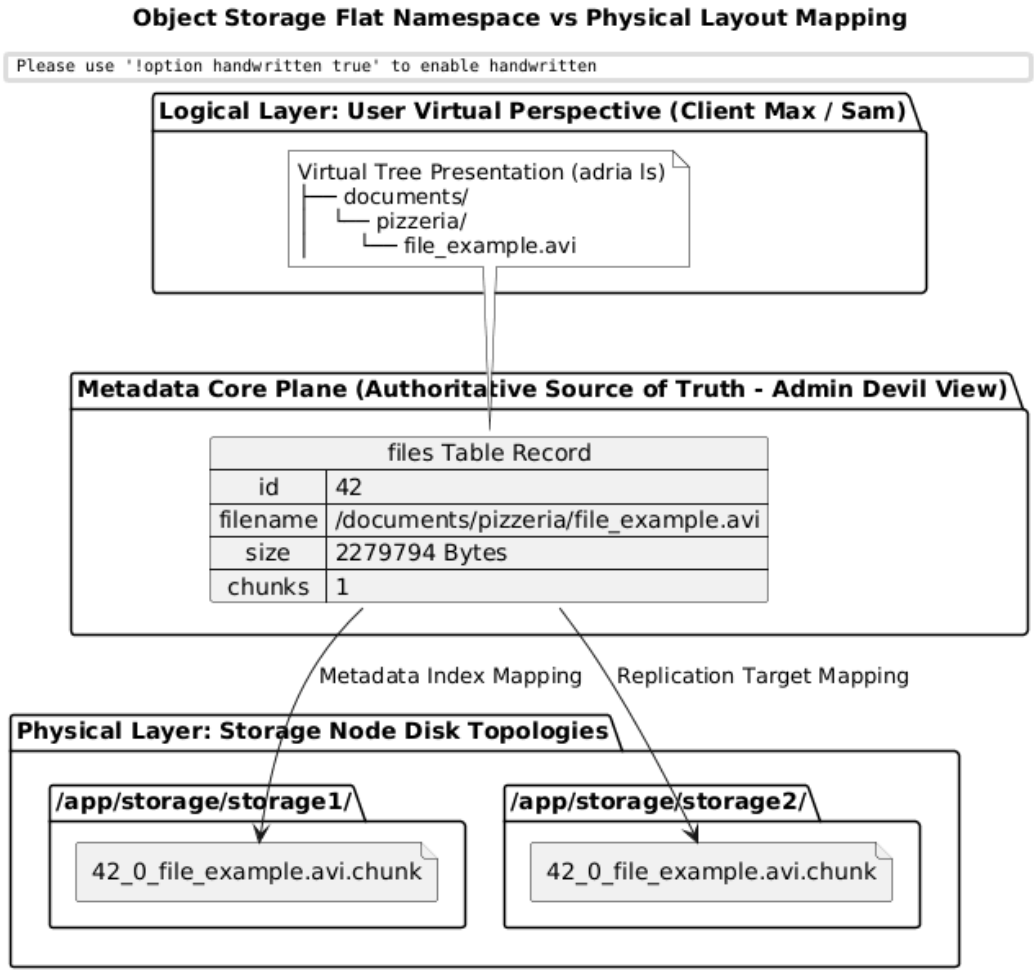

Figure 2: AdriaBOX Object Storage Flat Namespace Mapping Blueprint vs Physical Disconnected Topologies.

1.3 Structural Slicing and Data Plane Virtualization

To manage massive file storage allocation without creating unbalanced workloads or hitting volume capacity barriers on individual worker machines, the control plane enforces automated binary file striping. When a user initiates an upload transaction, the metadata controller evaluates the global asset capacity in bytes. If the asset scale surpasses the maximum logical partition threshold defined within the system configuration constants ($LOGICAL\_BLOCK\_SIZE = 64$ Megabytes), the file object is mathematically sliced into separate, autonomous logical fragments. Each block is tracked via a deterministic allocation

matrix using a specialized ceiling function:

$$N = \max\left(1, \left\lceil \frac{\text{File Size}}{64 \times 1024 \times 1024} \right\rceil\right)$$

The individual data blocks are assigned unique, structured names based on their file ID and chunk index (e.g., `42_0_file.chunk`, `42_1_file.chunk`), and are then distributed across the storage node fleet. The worker daemons virtualize physical storage by isolating these incoming blocks within dedicated host directories. Each background process is jailed inside a specific container partition path (ranging from `/app/storage/storage1` to `/app/storage/storage12`), effectively simulating twelve disconnected, independent hardware failure domains on the host system.

1.4 Use Case Specification and Multi-Tenant Role Demarcation

AdriaBOX is engineered from the ground up for multi-tenant deployment across shared enterprise or academic intranet networks. The platform isolates user spaces by strictly separating administrative capabilities from standard user operations. While designed to deploy across physically distinct client stations, the validation environment emulates multi-tenant concurrency by deploying isolated concurrent workspaces. This layout isolates active session variables and Zero-Knowledge security frames inside segregated user contexts, ensuring that different active clients cannot contaminate shared storage boundaries or intercept keys.

- **Standard Tenant (User Role):** Authorized to execute standard data manipulation routines within their isolated account context. Users can create virtual directory prefixes, upload new assets, download or permanently delete existing files, and check their individual storage quota allocation metrics. A standard user has zero visibility into the identities, quotas, or metadata blocks of other registered tenants.
- **System Administrator (Admin Role):** Authorized to audit and manage the cluster infrastructure. System administrators can query real-time node heartbeats, evaluate aggregate cluster footprints, and execute cascading user evictions. However, due to client-side cryptographic barriers, administrators are prevented from reading the raw binary contents of tenant files.

The global system interaction framework is transaction-bound and event-driven. The client application remains completely passive until a user invokes a command via the CLI terminal. In contrast, the metadata master and decentralized storage node processes run continuously as non-blocking system services, listening for network connections on their designated interface ports.

1.5 The Necessity of Distribution and Resource Isolation

Designing AdriaBOX as a decentralized storage matrix rather than a traditional centralized server application resolves critical architectural bottlenecks in resource management, data availability, and network scalability:

1. **Data Path Offloading and Bandwidth Optimization:** In centralized storage designs, all binary data streams must pass directly through the central server's network interface card, creating extreme performance bottlenecks as user concurrency increases. AdriaBOX completely eliminates this issue by removing the Metadata Server from the data transfer path. The master node only processes lightweight, sub-millisecond REST control messages to generate or validate storage roadmaps. High-throughput binary

streaming is delegated entirely to parallel peer-to-peer TCP channels opened directly between the client CLI and specific storage nodes.

2. **Fault Domain Isolation and Synchronous Survival Chains:** Hardware failure is a certainty in large-scale cluster environments. AdriaBOX mitigates this risk by mirroring each logical block across three unique failure domains via a synchronous replication cascade. If an individual storage node crashes or suffers a sudden disk failure, data availability is maintained with zero downtime by shifting transactions to one of the remaining online replica copies.

3. **Resource Starvation Mitigation ($O(1)$ Memory Page Safety):** A major vulnerability in cloud storage software is the tendency to load entire chunks or files into central memory (RAM) via blocking operations, which triggers kernel out-of-memory (OOM) panics during large file upload streams. AdriaBOX guarantees absolute memory safety by enforcing a strict buffered streaming model. By mapping the application data transfer page to a fixed size configuration ($CHUNK\_SIZE = 4096$ Bytes)—aligning with the native physical memory page allocation of standard Linux kernels—the client and worker engines move binary data through a continuous stream of micro-packets. During high-concurrency binary uploads, this model avoids memory leaks or process context interference, keeping the active RAM footprint strictly bounded at exactly 4 Kilobytes regardless of whether the target asset is a lightweight document or a multi-gigabyte archive.

2 Requirements Elicitation and Analysis

This chapter establishes the formal operational boundaries of AdriaBOX, defining strictly *what* the system must execute rather than *how* its routines are programmatically concrete. Each requirement is mapped into a strict engineering specification to guarantee complete traceability during the software validation phase.

2.1 Requirements Traceability Matrix

To maintain a professional software engineering layout, the functional, non-functional, and implementation constraints are unified into a structural traceability ledger detailed in Table 1.

Table 1: AdriaBOX System Requirements Verification and Traceability Matrix

ID	Type	Target Component	Acceptance Criterion
FR-1	Functional	`server.py` / `core.py`	Unauthenticated resource endpoint requests must be rejected immediately with an HTTP 401 status code.
FR-2	Functional	`manager.py`	Uploading assets exceeding 64MB must slice the object into exactly $\lceil \text{Size}/64\text{MB} \rceil$ tracking metadata entries.
FR-3	Functional	`transfer.py` / `tcp.py`	Each logical block must stream through 3 unique nodes; connection drops mid-pipeline must abort the transaction.
FR-4	Functional	`db.py` / `manager.py`	Directory prefix creation (`mkdir`) or path movement (`mv`) must update SQL row entries within sub-second thresholds.
FR-5	Functional	`server.py` / `db.py`	Account removal must atomically drop relational state rows and dispatch physical erasure codes to worker daemons.
NFR-1	Non-Functional	`crypto.py`	Key derivation must run in local user space using 100,000 PBKDF2 iterations; raw keys must never hit the network.
NFR-2	Non-Functional	`server.py` / `tcp.py`	Write operations are only visible to the filesystem registry after all 3 replication pipeline mirrors confirm disk write state.
NFR-3	Non-Functional	`cli.py` / `ui.py`	Interface layout queries (`adria ls`) must abstract cluster paths, showing a single virtualized filesystem tree.
IR-1	Implementation	`pyproject.toml`	Runtime modules and dependency constraints must be handled exclusively within an isolated, deterministic Poetry environment.

The internal logical dependency layout and cross-layer requirement constraints are visualized in the tracking hierarchy graph in Figure 3.

2.2 Detailed Analysis of Functional Requirements

The functional capabilities listed in the traceability matrix represent the operational baseline of the system. The authentication constraint (**FR-1**) isolates the metadata endpoints, forcing the application to validate credentials and evaluate signatures before authorizing access to the cluster status or user spaces. The storage partitioning strategy (**FR-2**) handles large files by automatically executing logical slicing whenever a transfer exceeds the 64MB threshold.

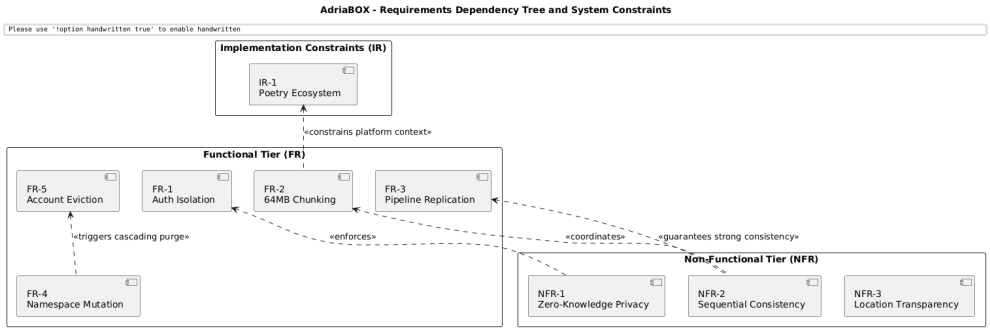

Figure 3: AdriaBOX System Requirements Dependency Layout and Operational Boundary Interconnection Map.

This process works in tandem with the replication pipeline (**FR-3**), which routes data chunks sequentially across three unique storage nodes to ensure fault tolerance. Namespace operations (**FR-4**) implement virtual file management by modifying string keys directly in the database, ensuring instantaneous, non-blocking execution. Finally, the administrative cleanup routine (**FR-5**) enforces complete data destruction by executing cascading purges that erase relational metadata records while broadcasting physical erasure commands to wipe raw blocks from the storage disks.

2.3 Detailed Analysis of Non-Functional and Technical Constraints

Non-functional parameters govern the behavioral safety, data privacy, and isolation quality of the cluster. The zero-knowledge privacy requirement (**NFR-1**) shifts all cryptographic computations to local user space, utilizing PBKDF2 key derivation to ensure that raw passwords and decryption keys are never exposed to the network or the cloud infrastructure. Data plane consistency (**NFR-2**) guarantees sequential metadata integrity by delaying commit acknowledgments until all replica endpoints confirm a successful write operation. This structural abstraction is supported by the location transparency requirement (**NFR-3**), which hides low-level networking and chunk distributions behind a clean, virtualized filesystem layout. On the operational layer, the platform context is locked within an isolated Poetry environment (**IR-1**), neutralizing configuration drift across separate deployments. Within the evaluation framework, this technical constraint is rigorously verified by enclosing individual tenant execution routines inside distinct container runtimes, ensuring that state files, cached tokens, and memory-mapped cryptographic contexts are completely isolated within private user-space boundaries.

2.4 Domain-Specific Technical Glossary

- **Object Key Prefix Alignment:** The method of embedding folder designations as text string tokens preceding an asset filename string within a flat database namespace registry.
- **Logical Partition Slicing:** The structural division of an un-fragmented binary asset stream into discrete data payloads matching an invariant architectural size boundary.
- **User-Space Staging Buffer:** A fixed allocation of process memory (RAM) used to hold raw data fragments moving between physical storage disks and network interface channels.

- **Zero-Knowledge Cryptography:** An infrastructure design protocol which guarantees that a data storage service provider does not possess the keys or plaintext blocks owned by its tenants.
- **Network Byte Order Formatting:** The Big-Endian integer arrangement requirement used to serialize multi-byte header fields across transport channels, ensuring compatibility across different processor setups.

2.5 Architectural Analysis of the CAP Theorem and Distributed Features

Engineering a decentralized data system requires an evaluation of the CAP Theorem (Consistency, Availability, Partition Tolerance). Under network partition events (P), a distributed platform must choose between absolute data consistency (C) and un-interrupted system availability (A). AdriaBOX is designed as a strictly **CP-focused infrastructure**, prioritizing absolute data correctness and sequential metadata validity over uncoordinated uptime. The system maintains this structural stance through two central mechanisms:

2.5.1 Authoritative Quorum Constraints

To eliminate the risk of split-brain conflicts—where a broken network path divides the cluster into competing segments running separate operations—the control plane relies on a single source of truth. If multiple orchestrators are introduced, the active master node must be validated by a strict absolute majority quorum (Q) of the cluster network size (N):

$$Q = \left\lfloor \frac{N}{2} \right\rfloor + 1$$

If a network partition isolates a cluster segment away from the central master, client nodes are prevented from bypassing the controller to write directly to storage daemons. This preserves the absolute consistency of the relational tracking maps.

2.5.2 Synchronous Replication Cascade

When uploading file blocks, data must travel through a multi-tiered pipeline:

$$\text{Data Plane Stream Path: Client CLI} \longrightarrow \text{Node}_{\text{Primary}} \longrightarrow \text{Node}_{\text{Replica 2}} \longrightarrow \text{Node}_{\text{Replica 3}}$$

An upload is not flagged as valid or committed to the database when the data hits the first storage container. The confirmation signal (`ACK_OK`) must travel back up through the entire pipeline in reverse order. If a network failure breaks the pipeline midway, the transaction aborts instantly, preventing data corruption or partial writes.

2.6 Evaluation of Distributed System Characteristics

2.6.1 Transparency

AdriaBOX provides high location and replication transparency. Users interact with a clean virtual filesystem layout via the client interface, completely unaware of how objects are sliced, which nodes hold specific data blocks, or how many backup copies exist across the cluster.

2.6.2 Fault Tolerance and Dependability

Fault tolerance is maintained by mirroring every data chunk across three independent failure domains. If a storage container crashes or a disk controller drops offline during a download operation, the client's transport module automatically catches the socket exception and shifts its connection pointer to an active backup replica address, maintaining data availability with zero data loss. In our local multi-container testbed, this resilience layer is rigorously stressed by forcefully dropping target daemon containers, proving that client-side failover routing handles network anomalies transparently.

2.6.3 Security, Multitenancy, and Trust Boundaries

The system enforces strict tenant isolation. Standard user roles are restricted by data ownership parameters built into the SQL engine, preventing accounts from discovering or interacting with assets owned by other tenants. Administrative paths are protected by multi-layered verification walls. Although administrators possess elevated structural privileges to audit overall cluster health and evict user accounts, they cannot view or read file data contents due to the client-side encryption layers.

2.6.4 Performance and Concurrency Efficiency

Decoupling the control plane from active data paths ensures low-latency performance. The master node only handles lightweight orchestration messages, leaving the individual storage daemons to process high-throughput data streams inside separate, concurrent execution threads. Low-level memory efficiency is guaranteed by enforcing a constant 4096-byte transfer page configuration across all network interfaces.

2.6.5 Features Explicitly Excluded from the Project Scope

- **Openness and Interoperability:** The platform relies on a custom, application-framed network protocol optimized for high-performance internal communication. Compatibility with external object APIs (such as AWS S3 or OpenStack Swift) was excluded to keep code lines clean and lightweight.
- **Evolvability and Rolling Upgrades:** The prototype focuses on providing a single-version deployment layout for academic evaluation. Long-term schema updates or backward-compatible protocol modifications were omitted from the core implementation design.
- **Economy and Cloud Operational Costs:** System optimization focuses on maximizing memory safety and fault isolation. Variable commercial hosting rates, automated container scaling budgets, and energy efficiency metrics are outside the project's tracking criteria.

3 Design

This chapter describes the strategic architectural and structural design choices governing the AdriaBOX distributed computing platform. The design abstracts system operations completely from backend programming technologies, providing an invariant blueprint centered around low-level resource safety, multi-tenant trust boundaries, and network efficiency.

3.1 Architectural Style

AdriaBOX implements a decoupled **Master-Worker (Client-Server Control Brokerage)** architectural style. The system explicitly separates structural system mapping from the high-capacity binary payload transport mechanisms. The application is split into two operating horizons:

- **The Control Plane:** An authoritative, stateful centralized orchestration layer that tracks active directory layouts, coordinates tenancy constraints, and calculates block locations.
- **The Data Plane:** A completely decentralized, stateless cluster of storage node daemons optimized for rapid I/O block extraction and network pipelining.

This architectural style is chosen to prevent the centralized orchestration core from turning into a network throughput bottleneck. Instead of forcing heavy file data blocks through a single server interface card, the control plane merely delivers sub-millisecond execution roadmaps. The actual high-volume binary transfers occur over dedicated, parallelized peer-to-peer connections directly between user workspaces and data plane nodes.

3.2 Infrastructure and Network Topology

The global hardware layout of AdriaBOX is configured as an N-node cluster network environment (standardized to a 12-node baseline deployment configuration). The components distribute across the networking topology according to the formal schema depicted in Figure 4.

3.2.1 Component Network Positioning

- **Tenant Workspace Instances:** Independent client nodes running inside disparate local area networks or public internet domains.
- **Metadata Server Instance:** Positioned on a central host system or dedicated public machine, exposing an active REST interface over standard network port `5000`.
- **Decentralized Storage Fleet:** Composed of 12 autonomous server daemons running as isolated network listeners. To prevent hardware-level port binding collisions within single-host emulation contexts, each node is assigned a unique network port index from an explicit, sequential mapping spectrum (ranging from port `7001` up to port `7012`).

3.2.2 Service Naming and Component Discovery

Component mapping and network location identification utilize an explicit configuration matrix. The metadata controller reads environmental configuration declarations to discover active storage target coordinates, tracking parameters including internal cluster host strings, external client entry paths, and raw TCP communication ports. When generating storage plans, the controller serializes these precise connection vectors directly into a client JSON response payload. This design decision makes the client nodes completely independent of

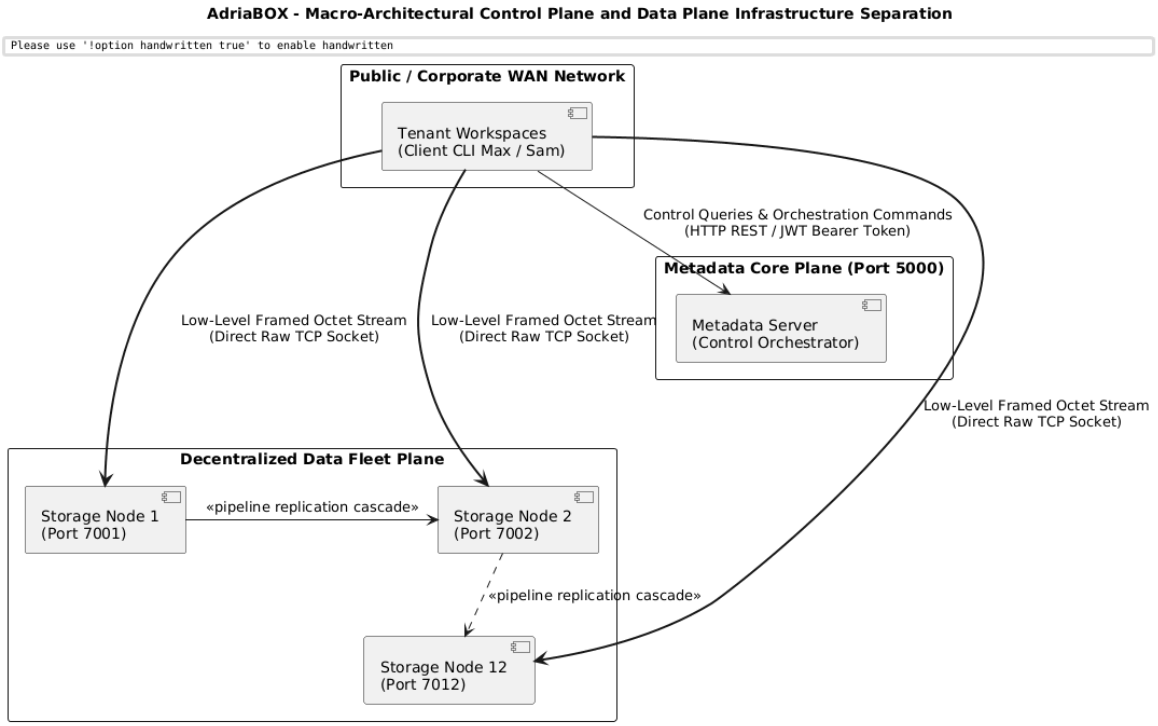

Figure 4: AdriaBOX Macro-Architectural Control Plane and Data Plane Infrastructure Separation.

complex dynamic domain name service (DNS) lookups or heavy service discovery daemons, resulting in low overhead and direct socket connection capability.

3.3 Modelling and Static Class Blueprint

AdriaBOX relies on an object-oriented modeling architecture to isolate responsibilities across decoupled components. The system structure and static object relationships are detailed in the class blueprint in Figure 5.

3.3.1 Core Domain Entities

The system logic operates on three primary structural domain entities:

- **User Tenant Model:** Represents an isolated user account, encapsulating unique system identifiers, validated username strings, administrative access permission flags, and current storage footprint quotas.
- **File Meta-Object:** Tracks virtual assets within the flat S3-style namespace, containing parameters such as global virtual string paths, total sizes in bytes, owner references, and the total logical block allocation count.
- **Logical Chunk Fragment Mapping:** Links an individual block partition index to an explicit list of storage nodes, tracking target block filenames and exact segment byte capacities.

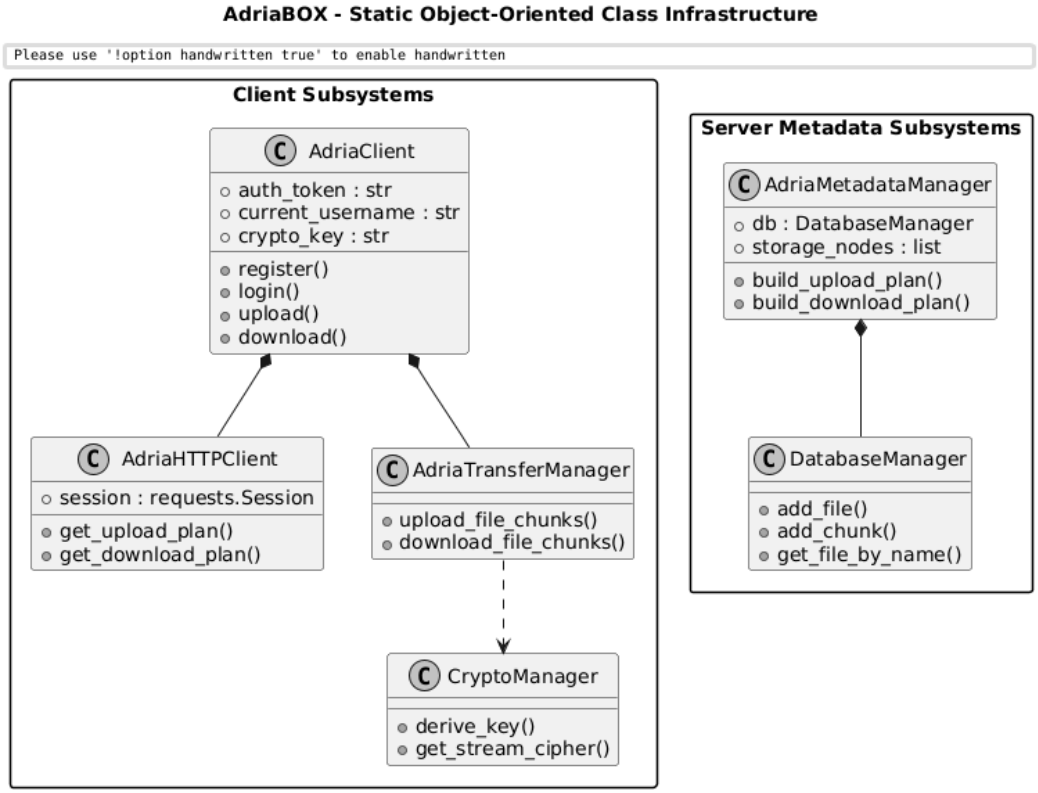

Figure 5: AdriaBOX Static Object-Oriented Class Infrastructure and Module Boundaries.

3.3.2 Message Interchange Classification

The communications flowing through the infrastructure are categorized into three distinct execution types:

- **Control Queries:** Stateless HTTP REST requests dispatched from client machines to browse or check state (e.g., fetching active space metrics or listing directory paths).
- **Orchestration Commands:** Action-oriented instructions that modify state (e.g., initiating login sessions or requesting an authorized upload plan).
- **Low-Level Application Framed Packets:** Customized binary byte-streams sent over raw TCP connections, optimized for streaming data segments between client transfer modules and storage daemons.

3.4 Interaction Patterns and Low-Level Network Framing

The network engine uses a custom application-layer protocol framing design to handle binary data blocks over raw TCP sockets. Because TCP socket connections provide a continuous stream of unstructured bytes without native message boundary markers, a receiver cannot natively deduce where a control header ends and where raw data payloads begin. AdriaBOX resolves this by building a strict serialization pattern directly into the byte stream. The protocol layout for an upload transaction is detailed in Figure 6.

The transmission framing format splits the byte sequence into five distinct operational regions:

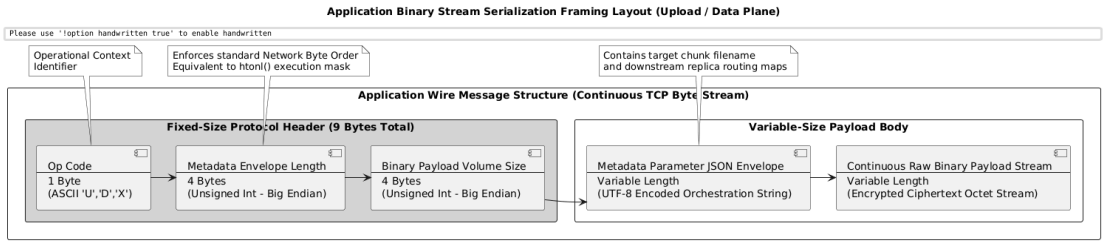

Figure 6: Application Binary Stream Serialization Framing Layout for Upload Operations.

1. **Operation Code Indicator (1 Byte):** A single ASCII character specifying the target routine (e.g., character `'U'` for an upload pipeline, character `'D'` for a download pull, or character `'X'` for a physical erasure signal).

2. **Metadata Envelope Length (4 Bytes):** An unsigned 32-bit integer, encoded strictly in Big-Endian (Network Byte Order, equivalent to standard `htonl` C byte ordering). This defines the exact length of the subsequent JSON tracking block.

3. **Binary Payload Volume Size (4 Bytes):** An unsigned 32-bit Big-Endian integer that specifies the exact volume size of the streaming data block, preventing connection stalls or buffer overruns.

4. **Metadata Parameter JSON Envelope (Variable Length):** A UTF-8 encoded text string containing system orchestration directives, such as the target block filename and an array of downstream replica target node coordinates.

5. **Continuous Raw Binary Payload Stream (Variable Length):** The physical chunk segment data, pushed through consecutive micro-packet transfer loops.

3.4.1 Bit-Level Wire Layout and Memory Alignment Specification

To satisfy formal network protocol standards, the packet stream layout must be specified bit-by-bit to guarantee deterministic un-packing alignment. Table 2 details the exact byte offsets, data type representations, and bit-level allocation boundaries enforced within the custom `AdriaTCPStreamer` implementation layers.

This bit-level mapping enforces rigid structure constraints. By invoking Python's native `struct.pack('>I', val)` mechanics, the client forces a 4-byte network representation equivalent to building a raw C structural mask with an explicit `htonl()` conversion wrapper. This design ensures absolute interoperability; an external storage daemon compiled in native C or Go can read the wire layout directly by aligning its un-packing masks with these exact bit offsets.

3.5 Dynamic System Behaviour and Data Streaming Engine

To guarantee memory footprint safety and prevent system crashes during large file transfers, the data plane streaming engine runs a strict resource management model. The step-by-step memory allocation and byte copy flow within the Linux user-space boundary are formalized in Figure 7.

The storage daemons and client modules enforce a bounded memory execution model. Instead of reading an entire 64MB logical block into memory using a single large I/O call, the transfer manager sets up a constant 4096-Byte user-space staging buffer. The low-level execution loop follows a strict sequential pipeline:

Table 2: AdriaBOX Low-Level TCP Wire Layout and Bit Alignment Specification (RFC Style)

Byte Offset	**Bit Boundaries**	**Field Token**	**Native Data Type**	**Endianness Order**
`0x00`	Bits 0–7	`Op Code`	8-bit ASCII Char	Network-Agnostic
`0x01`	Bits 8–39	`Meta Length`	32-bit Unsigned Int (`>I`)	Big-Endian (`Network Byte Order`)
`0x05`	Bits 40–71	`Payload Size`	32-bit Unsigned Int (`>I`)	Big-Endian (`Network Byte Order`)
`0x09`	Variable	`Meta Envelope`	UTF-8 Packed JSON String	Stream Byte Array
`Variable`	Variable	`Binary Stream`	Raw Octet Byte Sequence	Stream Byte Array

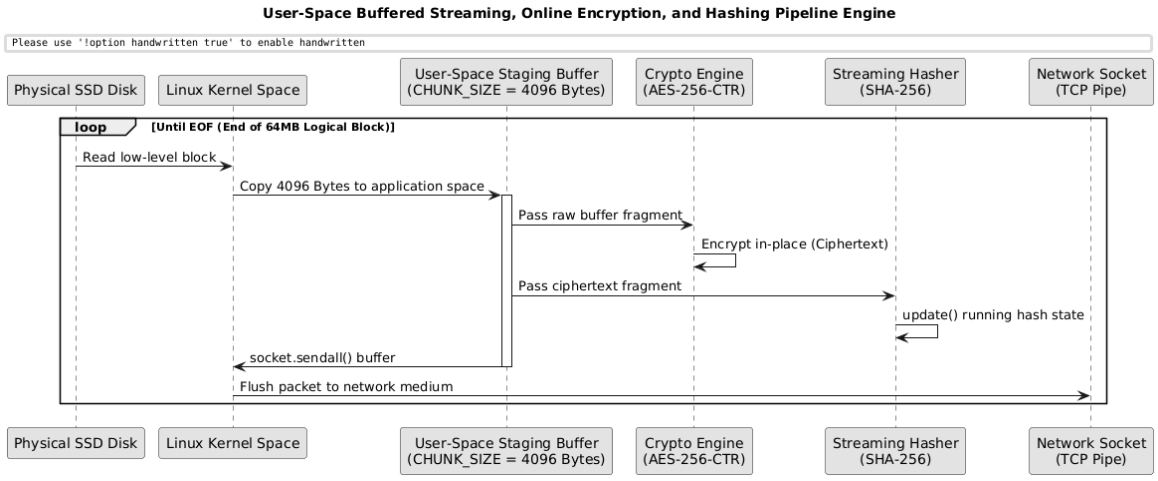

Figure 7: User-Space Buffered Streaming, Online Encryption, and Hashing Pipeline Engine.

1. The client initiates an iterative loop, pulling small data blocks from disk storage via kernel-space read buffers into the application space staging buffer.

2. The 4KB data snippet passes through the cryptography engine, where an inline AES-256-CTR transformation modifies the byte array in-place, avoiding extra memory copies.

3. The mutated buffer is passed directly to the streaming hashing module, which updates its running state to compute the SHA-256 cryptographic signature on the fly.

4. A socket system call immediately flushes the 4KB buffer down the network pipe to the target storage node daemon.

This processing model ensures that memory usage during file transfers remains static at $O(1)$ complexity, requiring exactly 4 Kilobytes of operational RAM regardless of the total file capacity.

3.5.1 Multi-Threaded Node Concurrency and Direct I/O Isolation

To sustain concurrent data plane transfers without experiencing network blocking or memory page corruption, the backend worker nodes virtualize client socket requests via an asynchronous thread pool architecture. The internal process-threading model executed within each storage node daemon is formalized in Figure 8.

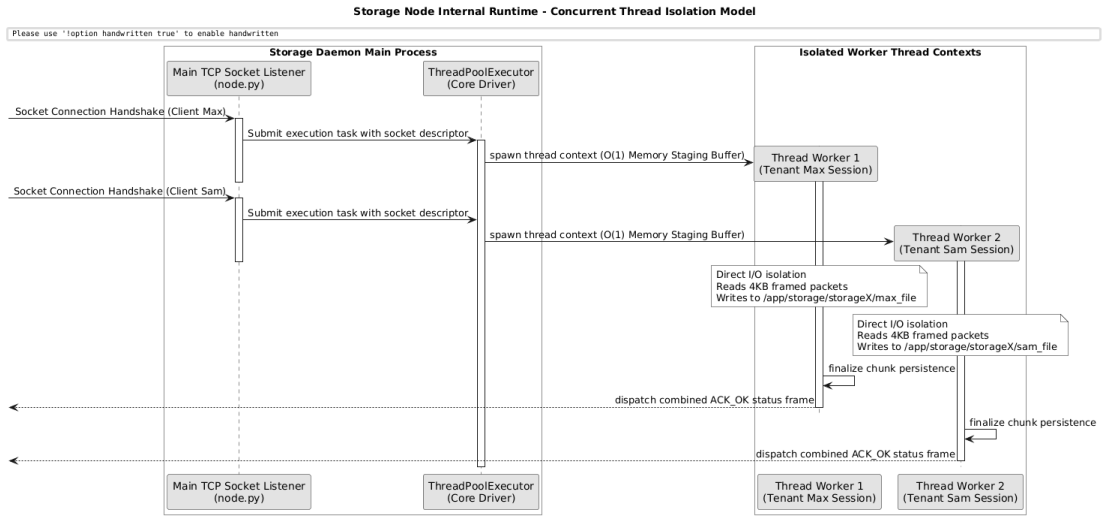

Figure 8: Storage Node Internal Thread Allocation and Direct Multi-Tenant Process Isolation Map.

When multiple client processes open simultaneous connections to a node's interface, the main thread's `socket.accept()` loop hands off the raw descriptor to a centralized `ThreadPoolExecutor` context. This subsystem isolates each connection transaction within an independent worker thread lifecycle:

- **Execution Context Separation:** Each active thread instantiates isolated instances of the wire formatting parser, allocating a separate 4096-Byte staging buffer in user-space RAM. Memory addresses do not cross thread barriers, preventing dirty read-write data states across concurrent streams.

- **Direct I/O Boundary Isolation:** The worker thread tracks its own file descriptor variables, performing concurrent writes to different, un-linked target block storage partitions on the host filesystem disk.

This multi-threaded architecture allows a single storage node daemon container to ingest multi-gigabyte blocks from multiple distinct users simultaneously, maintaining high-throughput cluster performance while keeping the operational resource footprint flat.

3.6 Data Persistence and Multi-Tenant Query Scope

The control plane isolates multi-tenant environments by applying strict security filters to database queries. When a client requests a file listing, the metadata controller intercepts the request and determines the query scope based on the user's role context, as detailed in Figure 9.

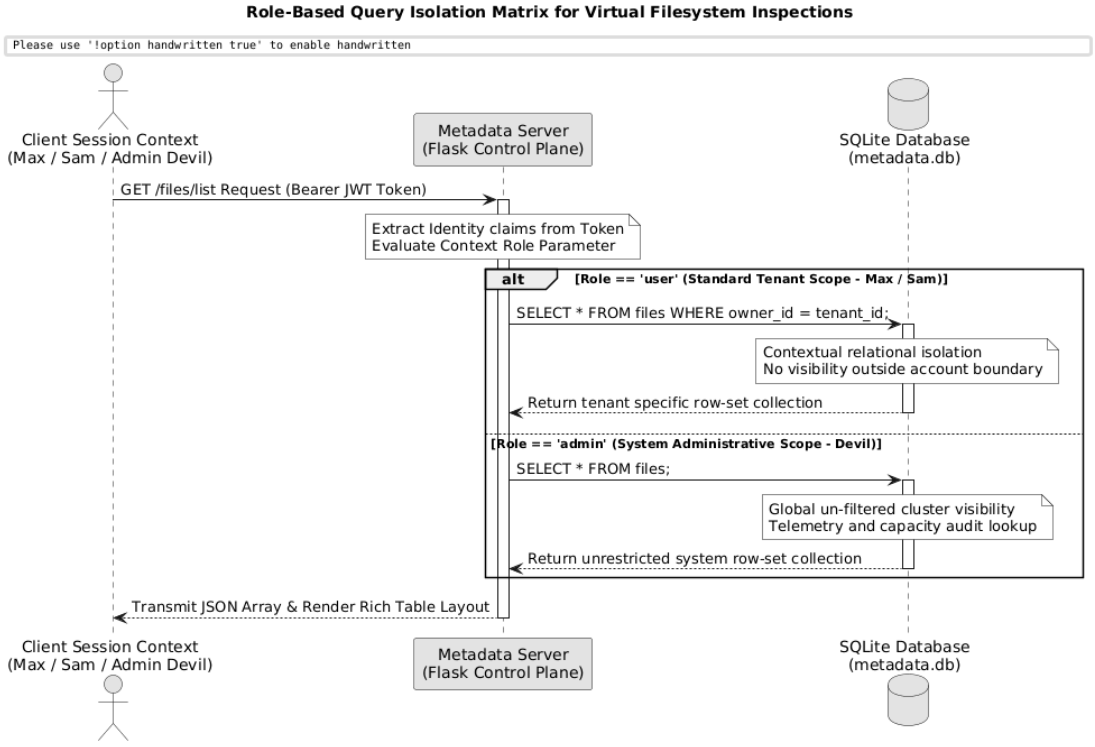

Figure 9: Role-Based Query Isolation Matrix for Virtual Filesystem Inspections.

The metadata database relies on a strictly relational SQL schema to maintain system state:

- `users` **Table:** Stores unique tenant identifiers, system usernames, secure authentication hashes, active system roles, and creation timestamps.
- `files` **Table:** Records global file paths, file sizes, block counts, creation dates, and owner ID references to enforce multi-tenant separation.
- `chunks` **Table:** Maps individual file segment blocks to their primary storage node locations, tracking block filenames and chunk sizes.

Query scoping handles permissions via role-based access logic:

- **Standard User Scope:** Queries are bound to the tenant's context via explicit SQL constraints (e.g., appending a restrictive `WHERE owner_id = user_id` clause). This ensures users can only view or interact with their own assets.
- **Administrative Scope:** Queries run without tenant-level filters, giving administrators broad visibility into system metrics, aggregate disk usage, and multi-tenant user lists.

3.7 Fault Tolerance and Resilient Pipeline Replication

AdriaBOX provides data redundancy and cluster availability using a synchronous pipeline replication strategy. When writing file chunks to the data plane, data flows down a coordinated node cascade to guarantee safety across failure domains, as shown in Figure 10.

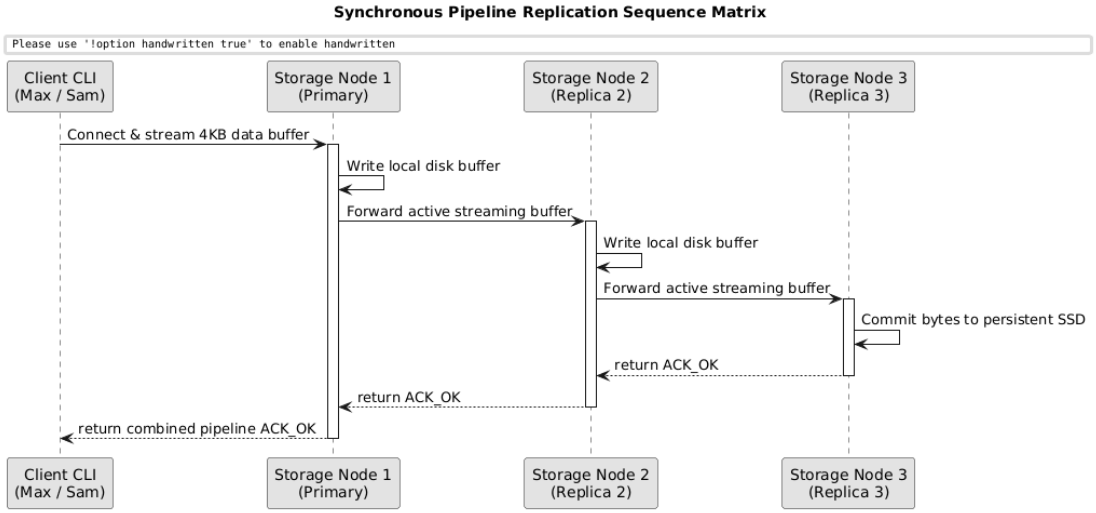

Figure 10: Synchronous Pipeline Replication Sequence and Cascading Verification Verification Flow.

3.7.1 Synchronous Redundancy Chain

The metadata master uses a Chained Round-Robin scheduling algorithm to calculate three unique node addresses for each block. Data flows down the chain sequentially:

$$\text{Replication Cascade: Client} \longrightarrow \text{Node}_{\text{Primary}} \longrightarrow \text{Node}_{\text{Replica 2}} \longrightarrow \text{Node}_{\text{Replica 3}}$$

The data streaming and write confirmations flow in opposing directions:

1. The client streams the 4KB data buffers directly to the primary storage node.
2. As the primary node receives the data, it copies the buffers to its local disk while concurrently forwarding the stream to the second node in the pipeline.
3. The second node copies the bytes locally and forwards them to the final replica node.
4. Write confirmations flow back up the chain in reverse order ($\text{Node}_3 \rightarrow \text{Node}_2 \rightarrow \text{Node}_1 \rightarrow$ Client). An upload is only flagged as successful when the primary node receives a valid confirmation from the downstream replicas, ensuring the block is safely written across three separate failure domains.

3.7.2 Client-Side Failover Architecture

To remain online during node crashes or network partitions, the client transfer manager uses an automated client-side failover loop, as detailed in Figure 11.

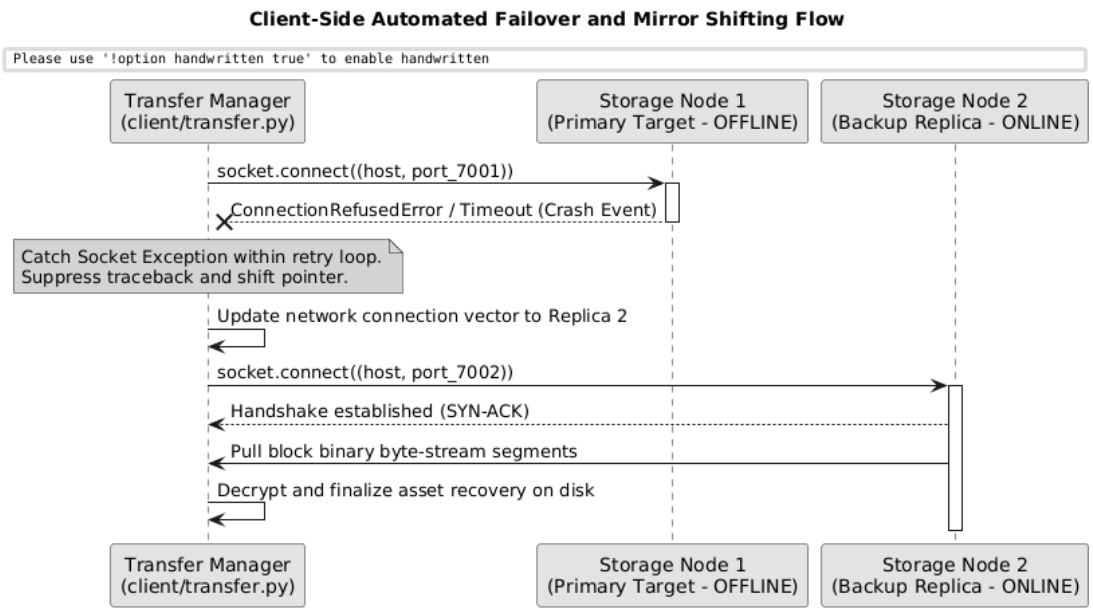

Figure 11: Client-Side Automated Failover and Mirror Shifting Flow during Node Failure.

When a download transaction begins, the client fetches an orchestration plan containing the primary location and all replica addresses for each chunk. If the primary node is offline or times out, the transfer manager catches the exception, logs a warning to the interface, and automatically shifts its connection pointer to the next backup replica address in the plan. This failover loop runs entirely in user-space, maintaining system availability without adding extra query overhead to the central metadata database.

3.8 Availability and CAP Strategy

The system is explicitly designed as a **CP-focused infrastructure** (Consistency and Partition Tolerance) under the CAP Theorem framework. It prioritizes absolute data correctness and metadata validity over uncoordinated uptime. If a network partition isolates a cluster segment, the system handles state conflicts through two strict design rules:

- **Central Control Authority:** All metadata modifications must clear through the authoritative central controller. If a client is partitioned away from the master node, it cannot write data directly to the storage nodes, preventing split-brain or divergent metadata scenarios.
- **Atomic Transaction Abort:** If a network partition interrupts a replication pipeline midway through a file transfer, the transaction aborts. The primary node returns an error to the client, preventing partial writes or uncoordinated block states.

3.9 Security and Access Isolation

Security boundaries are enforced using strict authorization layers and client-side encryption routines:

3.9.1 Session Authentication and Token Isolation

Users register and authenticate credentials through the central control interface. Upon a successful login, the server issues a stateful JSON Web Token (JWT) signed with a secure server key. The client CLI caches this token locally and appends it to the authorization headers of all subsequent control plane requests, keeping the system's control APIs secure.

3.9.2 Zero-Knowledge Cryptographic Isolation

Data privacy is protected using a Zero-Knowledge encryption model. When a user authenticates, the client CLI reads the raw password and uses the PBKDF2 function with 100,000 hashing iterations to derive a secure 256-bit AES key. This key is isolated within the local client workspace and is never transmitted over the network or exposed to the master node or storage cluster. Data blocks are encrypted on the fly using the AES-256-CTR stream cipher before leaving the client machine, ensuring that data stored on the cluster remains secure and unreadable to unauthorized parties or system administrators.

3.10 Dual-Layer Cryptographic Architecture and Zero-Knowledge Paradigm

To satisfy the stringent privacy and isolation metrics defined in Section 2, AdriaBOX implements a dual-layer cryptographic framework. This architecture clean cuts the boundaries between transit-layer wire protection and persistent asset confidentiality, enforcing a strict **Zero-Knowledge paradigm** where the cloud infrastructure provider possesses zero visibility into the tenant's plaintext binary payload.

3.10.1 Control Plane Protection: Transport-Layer Security (HTTPS)

All administrative, authentication, and metadata orchestration pathways operating over the Control Plane (directed from the Client CLI to the Flask Metadata Server) utilize standard **Transport-Layer Security (TLS/HTTPS)** in a production environment. This mechanism encapsulates standard HTTP REST JSON structures inside an asymmetric/symmetric handshake envelope (Layer 5 of the OSI stack). While HTTPS successfully mitigates middleman eavesdropping, packet tampering, and session hijacking during network transit, it terminates at the edge of the Control Plane: the Metadata Server decrypts the incoming TLS packet to evaluate and execute the business logic. Consequently, transport-layer encryption alone is mathematically insufficient to guarantee complete data privacy inside a multi-tenant storage cluster.

3.10.2 Data Plane Isolation: Client-Side Zero-Knowledge Encryption

To ensure that files remain completely unreadable even to malicious system administrators or compromised storage hardware, the Data Plane enforces an end-to-end symmetric encryption model executing exclusively within the local host user-space boundary.

1. **Deterministic Key Derivation (PBKDF2HMAC):** When a user triggers an execution session via the CLI, the local runtime intercepts the plain password. To prevent cryptographic key collision attacks—where distinct users utilizing identical weak passwords yield equivalent keys—the system feeds the password into the PBKDF2HMAC algorithm. The unique system `username` is systematically injected as a deterministic cryptographic *salt*, stretched over exactly 100,000 computation iterations using the SHA-256 hashing primitive. The resulting 256-bit (32-byte) key is safely isolated inside the client memory page structures.

2. **In-Place Stream Ciphering (AES-256-CTR):** Prior to logical block striping and socket broadcasting, data flows through the local cryptographic engine. The system instantiates an Advanced Encryption Standard (AES) block cipher operating in Counter (CTR) mode. CTR mode virtualizes the block cipher into a continuous stream cipher by computing an invariant keystream bit-by-bit, eliminating the architectural requirement for memory padding overhead. To defeat cryptographic patterns, a unique 16-byte cryptographic Nonce (Initialization Vector) is calculated for each individual chunk by running a SHA-256 digest over the target chunk filename.

The structural interaction boundaries detailing exactly where the Transport-Layer Security (HTTPS) and the local Zero-Knowledge Engine (`crypto.py`) operate across the network plane are formalized in Figure 12.

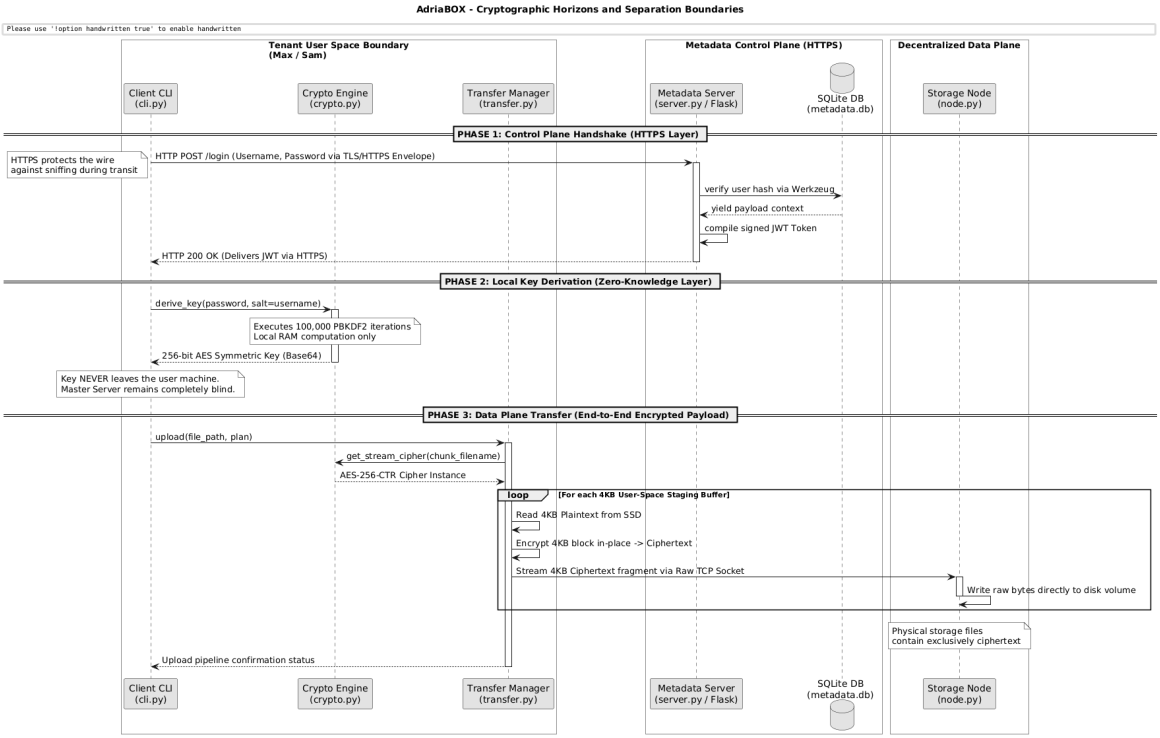

Figure 12: AdriaBOX Dual-Layer Cryptographic Boundary and Processing Horizons Map.

4 Implementation

This chapter describes the technological stack and programmatic patterns used to convert the abstract design of AdriaBOX into a working software prototype. The codebase is implemented entirely in Python, using a modular architecture to enforce clean boundaries between core network utilities, data transfer engines, and control services. On the client side, architectural responsibilities are structuralized around a central `AdriaClient` object acting as a Facade interface that seamlessly orchestrates the control-plane operations of `AdriaHTTPClient` and the low-level data-plane transfer mechanisms handled by `AdriaTransferManager`.

4.1 Network and Application Protocols

AdriaBOX utilizes a hybrid networking model to optimize performance and separate tasks within the infrastructure:

- **Control Plane Communication (HTTP/1.1):** All client interactions with the metadata controller rely on synchronous HTTP REST APIs. This traffic is handled via the Python `requests` library on the client side, using a persistent `requests.Session()` object to maintain and reuse underlying connection contexts efficiently. The utilization of persistent connection pools limits the operational TCP handshake overhead, minimizing connection latency during structural control operations.
- **Data Plane Communication (Raw TCP Sockets):** High-throughput binary streaming between client transfer components and storage workers uses raw, low-level TCP/IP sockets via Python's native `socket` library. Sockets are initialized with explicit timeout constraints to prevent resource leaks from broken pipes or network drops. By operating on raw octet streams, the data plane completely bypasses the computational text-parsing overhead characteristic of higher-level application layer frameworks.

4.2 In-Transit Data and Wire Layout Serialization

Data encapsulation and wire serialization layout decisions depend on the operational context of the payload:

- **Control Structures and Metadata:** Inter-component operational exchange payloads (such as cluster layouts, upload/download plans, and system responses) are serialized into standard JSON text strings. This data layout supports flexible schema alterations while keeping parsing overhead low across REST endpoints.
- **Network Framing Primitives:** To construct custom binary stream headers over raw TCP, the system utilizes Python's native `struct` library. Packing directives enforce strict data serialization sizes and layout alignments:
  - Format token `>I` is applied to serialize metadata block capacities into an unsigned 32-bit integer (4 bytes) using Big-Endian formatting, aligning with standard Network Byte Order rules.
  - String paths and parameter payloads are packed into variable-length binary arrays using standard UTF-8 text encoding.

This precise binary layout prevents message fragmentation anomalies across socket boundaries, ensuring that any receiver can deterministically partition incoming framing packages before extracting volatile data payloads.

4.3 Database Persistence and Relational Query Strategy

The metadata server tracks global cluster state and user accounts using Python's native `sqlite3` module. The system maps directory entries and user records to an embedded relational SQLite database (`metadata.db`).

To ensure clean, readable access, connections are configured with the custom attribute `conn.row_factory = sqlite3.Row`. This converts standard relational row collections into structured object maps, allowing fields to be accessed directly by column name.

Data updates leverage standard SQL transactions managed by context handlers (`with sqlite3.connect() as conn`). This architecture ensures ACID properties for cluster state updates, preventing corrupt or partial metadata mappings during concurrent file tracking requests. The system handles transaction isolation natively, mitigating deadlocks or dirty reads during high-concurrency storage node updates.

4.4 Authentication Matrix and Session Token Tracking

Tenant authentication relies on a stateless, token-based framework using the `PyJWT` library. When a user logs in, the metadata controller validates their identity against salted password hashes.

Upon validation, the server generates a cryptographically signed JSON Web Token (JWT) embedded with the tenant's user ID, username, assigned system role, and a strict 24-hour expiration timestamp (`exp`). The signature is secured with a private cluster key using the HMAC-SHA256 algorithm.

The client command-line interface caches this token locally within an isolated configuration structure. The local persistence layer handles the active token safely within user-space directories, ensuring that concurrent client instances on the same host station isolate their respective cryptographic states natively without data pollution. The token is systematically appended as a bearer string to the authorization headers of all subsequent API calls, allowing the control plane to validate tenant requests securely without maintaining server-side session state.

4.5 Authorization and Multi-Tenant Access Control

Access controls are enforced on the server using an Aspect-Oriented Programming (AOP) approach. The system wraps target API endpoints inside a central validation decorator method denominated `_auth_and_route()`. This middleware wrapper extracts incoming token headers, validates the cryptographic signature, checks the expiration timestamp, and passes the validated user context to the underlying business logic.

Authorization boundaries implement a hybrid Model-Based and Role-Based Access Control (RBAC) strategy:

- **Standard Users:** Restricted by ownership filters. SQL query statements include strict tenant boundaries (e.g., matching the user context ID against file owner records) to prevent accounts from accessing or enumerating data assets outside their authorized scope.
- **System Administrators:** Authorized by checking for the `admin` role string within the token payload. Validated administrative sessions can run unrestricted database queries to monitor overall cluster telemetry or evict user accounts.

4.6 Technical Implementation Modules and Package Dependencies

The implementation layout and component modularity are structured according to the package schematic shown in Figure 13.

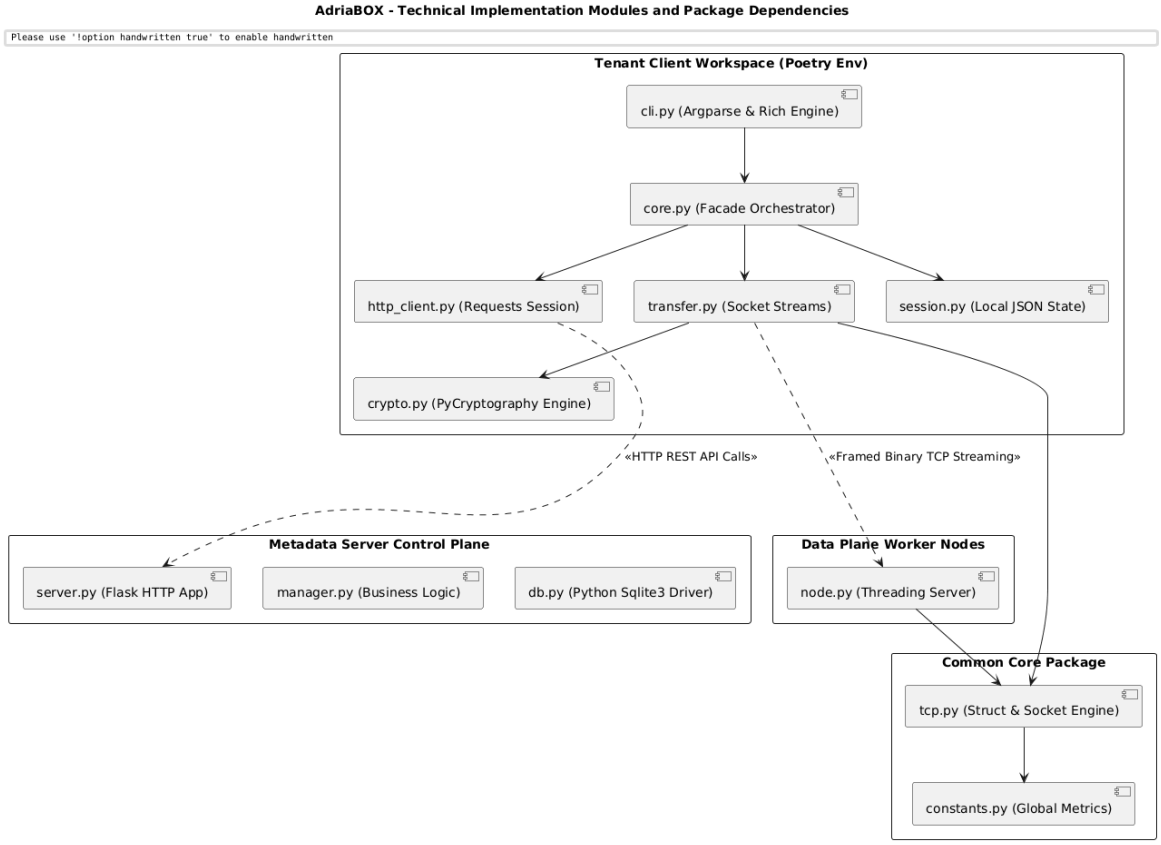

Figure 13: AdriaBOX Technical Implementation Modules and Source File Dependencies.

The software artifact utilizes specific open-source libraries to meet its security and user interface requirements:

- **Flask Framework:** Operates as the foundation for the control server, routing network traffic to dedicated controller classes and processing incoming JSON metadata blocks via micro-routing modules.
- **Werkzeug Security Library:** Manages secure credential persistence. User passwords are never saved in plaintext; instead, they are converted into robust cryptographic hashes using the standard `pbkdf2:sha256` algorithm combined with random salt strings.
- **PyCryptography Component:** Provides the low-level mathematical engine for the client workspace, using advanced symmetric ciphers (`AES-256-CTR`) and key derivation primitives (`PBKDF2HMAC` with a 100,000-iteration counter) to execute Zero-Knowledge security routines.
- **Rich UI Framework:** Drives the terminal user interface engine. It processes raw directory arrays and system reports to render clean, interactive data tables and color-coded panels, improving usability while keeping the client lightweight.
- **Poetry Environment Coordinator:** Manages project dependencies and execution

environments, defining precise version boundaries for internal modules to guarantee stable and reproducible deployments across different host systems.

4.7 Operational Interaction of the Technology Stack

To illustrate how these independent open-source components integrate dynamically during system execution, the programmatic data flow between libraries is mapped sequentially within Figure 14.

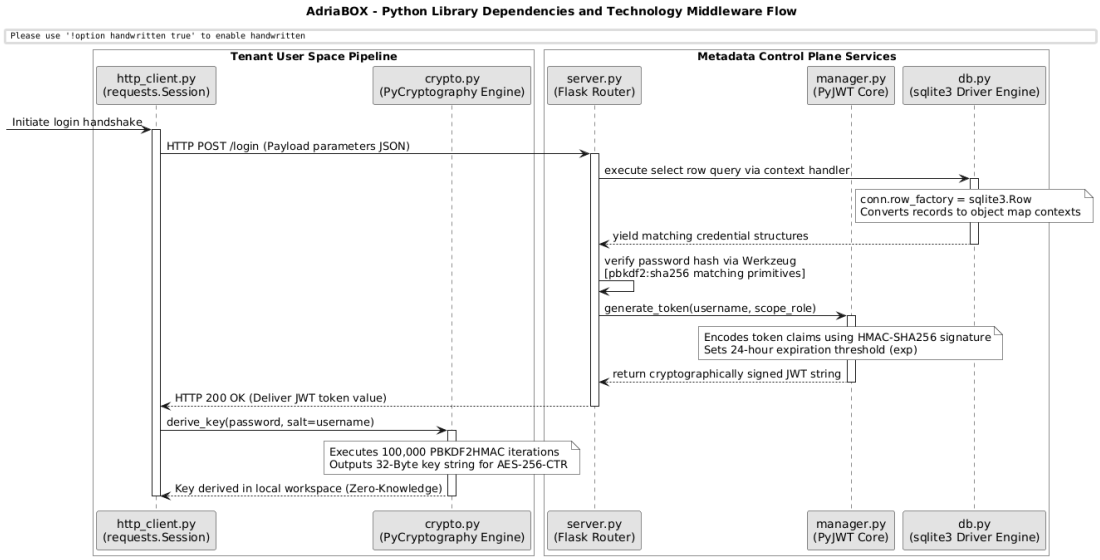

Figure 14: AdriaBOX Inter-Component Software Library Dependency and Operational Middleware Execution Lifecycle.

This mapping visualizes the execution pipeline during a standard session initialization. Incoming HTTP parameter strings map through the `Flask` routing layer into the relational database context. The `sqlite3` engine packages row properties using column identifiers, allowing the server to retrieve verification parameters securely.

Credential evaluation is offloaded to `Werkzeug` hashing blocks, and session tracking is initialized via `PyJWT` primitives to compile a signed authorization bearer token. Simultaneously, in complete isolation from the network plane, the local client workspace targets the `PyCryptography` primitives to execute the rigorous 100,000 iteration key derivation loops, sealing the client cryptographic context before data plane transactions begin.

4.8 Low-Level Programmatic Flow and Simulation Isolation Mechanics

To establish an uncompromised evaluation framework for the distributed architecture, specific technical mechanics are implemented within the codebase components to enforce multi-client segregation and bounded memory streaming.

4.8.1 The Upload Conflict and Stream Pipelining Logic

When an execution routine invokes an upload operation within `AdriaClient`, the system enforces a strict state evaluation sequence. The method first intercepts the directory context via an explicit call to `list_files()` to inspect the current catalog state. If a filename collision is detected, the runtime halts execution to prompt an interactive overwrite authorization. Upon user confirmation, a clean purge command is propagated via `self.rm()` to forcefully

scrub obsolete metadata entries and orphan chunks, eliminating ghost block accumulation before fetching a new upload allocation plan from the Master orchestration core.

Once the plan payload is retrieved from the control endpoints, high-throughput streaming is offloaded to `AdriaTransferManager.upload_file_chunks()`. This routine handles file data isolation by wrapping operations inside an explicit context manager:

```
with open(local_filepath, "rb") as source:
    for chunk in plan_chunks:
        source.seek(chunk["offset"])
        # Instantiate localized socket sender and stream bytes...
```

The transfer engine moves the file cursor to the calculated byte offset, bypassing heavy block memory replication. For each individual block sequence, the system instantiates a localized `ChunkStreamSender` instance using target TCP parameters. Crucially, the unique tracking attribute `file_id` is systematically stringified into a dedicated `session_id` variable and injected directly into the transmission pipeline. This identifier colors downstream octet packets, enabling backend worker thread contexts to map incoming byte sequences to an explicit orchestration context without loading un-segmented assets into volatile memory.

4.8.2 POSIX User-Space Workspace Emulation Isolation

Although AdriaBOX is designed to run over discrete physical stations distributed across a network domain, testing multi-client operational boundaries requires an environment capable of executing multiple active workflows concurrently. Multi-client session isolation is handled natively via the `SessionManager` constructor layout:

```
def __init__(self, session_file="session.json"):
    home_dir = os.path.expanduser("~")
    self.session_file = os.path.join(home_dir, ".adriabox",
       session_file)
    os.makedirs(os.path.dirname(self.session_file), exist_ok=True)
```

By invoking the native `os.path.expanduser("~")` method, the local runtime dynamically binds the state storage directory to the current POSIX home folder inside the active file system block. Within the Docker-based emulated testbed, each client node container inherits an isolated, independent root partition directory. Consequently, when concurrent users execute commands simultaneously, the underlying JSON session records, Bearer JWT credentials, and active symmetric encryption key parameters are written to segregated local paths. This architecture completely prevents cross-tenant credential leakage or session overwrite collisions across workstations, validating production-grade trust boundaries on a single host.

4.8.3 Zero-Knowledge Cryptographic Execution Primitives

Symmetric security metrics are implemented using deterministic stretching and inline byte mutations inside the client execution perimeters. The method `CryptoManager.derive_key()` seals the user context via a rigorous multi-layered key derivation pipeline:

```
salt = username.encode('utf-8').ljust(16, b'\0')[:16]
kdf = PBKDF2HMAC(
    algorithm=hashes.SHA256(),
    length=32,
    salt=salt,
    iterations=100000,
    backend=default_backend()
)
```

The system enforces key isolation by applying the unique tenant `username` string as a deterministic salt primitive, padding the byte sequence to exactly 16 bytes using a trailing zero-byte truncation mask (`.ljust(16, b'\0')[:16]`). This configuration eliminates identical key derivations for users sharing duplicate passwords. The key derivation function executes 100,000 stretching iterations using the cryptographic primitive `SHA256`, locking the computation inside local user-space memory pages.

When streaming data blocks, `CryptoManager.get_stream_cipher()` extracts a distinct 16-byte Initialization Vector (Nonce) for each data chunk by passing the explicit `chunk_filename` string through an independent hashing primitive:

```
digest = hashes.Hash(hashes.SHA256(), backend=default_backend())
digest.update(chunk_filename.encode('utf-8'))
nonce = digest.finalize()[:16]
```

This structural mechanism ensures that identical plaintext payloads split across separate file components yield entirely unique ciphertext octet streams on the wire. The resulting `Cipher` configuration instantiates an Advanced Encryption Standard (`AES`) mechanism running in Counter (`CTR`) mode. CTR mode virtualizes block boundaries, computing a mathematical keystream on the fly that executes in-place XOR bit modifications on the 4KB transmission staging buffer, eliminating memory padding delays and ensuring absolute isolation from cloud data-plane components.

5 Validation and Testing Matrix

This chapter details the validation matrix, architectural testing strategies, and empirical experiments executed to verify the scientific correctness, operational safety, and fault-tolerance boundaries of the AdriaBOX distributed computing platform.

The validation spectrum is partitioned into two distinct verification horizons: a deterministic, automated test suite managed via the `pytest` framework to isolate and validate discrete components, and a live, out-of-band empirical fault-injection stress test executed within a containerized cluster topology to observe dynamic system recovery under severe hardware loss conditions.

5.1 Automated Isolation Strategy and AI-Assisted Software Engineering Rationale

Maintaining structural integrity across a decoupled, multi-tenant distributed system requires a testing matrix capable of validating system modules under boundary limits, network drops, and malformed inputs. Executing these operations manually across 12 distinct storage domains and a stateful control plane would induce a prohibitive operational overhead, increasing development latency and limiting test determinism due to unpredictable physical environment variables.

5.1.1 The Paradigm of AI-Assisted Software Engineering

To resolve this bottleneck, the development workflow adopted an advanced **AI-Assisted Software Engineering** methodology. Rather than utilizing Large Language Models (LLMs) as black-box code generation artifacts, the authors leveraged artificial intelligence as a high-throughput productivity multiplier under strict human oversight.

The technical approach followed a rigorous two-phase engineering pipeline:

1. **Human Architectural Design:** The authors manually engineered the core system boundaries, designed the structural state-virtualization facades within the client space, and established the network socket mocking strategies.

2. **AI-Driven Matrix Expansion:** Once the base structural interception filters and mock interfaces were validated, AI systems were systematically employed to expand the matrix into a massive, multi-tiered suite composed of 112 targeted test checkpoints. The AI automated the redundant generation of parameterized edge-case inputs, compiled extended combinatorial lists of malformed parameters, and structured repetitive HTTP error-code response mapping payloads.

This collaborative integration of human architectural oversight with AI-driven testing execution guarantees deep code coverage. It demonstrates an industrial-grade understanding of modern software manufacturing, where artificial intelligence is utilized to enforce software quality metrics without compromising the intellectual authorship and scientific rigor of the underlying distributed framework.

5.2 Comprehensive Automated Test Suite Mapping

The automated framework targets software modules hierarchically, using Python's native test isolation properties to bypass physical hardware interaction. The suite comprises **112 unique test configurations** structured across four distinct functional macro-groups, ensuring complete traceability back to the system acceptance criteria defined in the requirements layer.

5.2.1 Functional Test Macro-Groups

- **Unit Tier (`validators.py`):** Validates structural inputs, path-prefix formatting boundaries, whitespace stripping, and URL resolution patterns.
- **Crypto and State Tier (`crypto.py`, `session.py`):** Audits 256-bit key derivation invariance via PBKDF2, CTR mode in-place ciphering, and explicit local JSON caching corruption trapping.
- **Control and Data Tier (`http_client.py`, `transfer.py`):** Validates HTTP payload compilation, persistent TCP session reuse, multi-threaded block slicing, and replica fallback recovery loops.
- **Backend and Storage Tier (`db.py`, `manager.py`, `node.py`):** Audits SQLite transactional atomicity, capacity planning scheduling, Chained Round-Robin logic, and multi-threaded socket handoffs.

Table 3: AdriaBOX Automated Validation Layer and Component Target Matrix

Target Module	**Test Script Source**	**Horizon**	**Validation Acceptance Objective**
`validators.py`	`test_client_validators.py`	Unit Tier	Verifies empty-string rejection, whitespace stripping, and path prefix formatting boundaries.
`crypto.py`	`test_client_crypto.py`	Crypto Tier	Audits 256-bit key derivation invariance, PBKDF2 deterministic stretching, and CTR mode in-place ciphering.
`session.py`	`test_client_session.py`	State Tier	Verifies local encryption key storage, session-file caching, and explicit JSON corruption trapping.
`http_client.py`	`test_client_http.py`	Control Tier	Audits JSON payload compilation, persistent TCP session reuse, and server error unwrapping.
`transfer.py`	`test_client_transfer.py`	Data Tier	Verifies multi-threaded block slicing, best-effort deletions, and replica fallback loops.
`core.py`	`test_client_core.py`	Facade Tier	Validates the centralized orchestration facade, login state syncing, and token security boundaries.
`db.py`	`test_database_manager.py`	Integrated Tier	Audits SQLite transactional atomicity, directory virtualization, query scoping, and cascading purges.
`manager.py`	`test_metadata_manager.py`	Logic Tier	Verifies capacity planning scheduling, Chained Round-Robin rotations, and metadata token generation.
`server.py`	`test_metadata_server_routes.py`	Routing Tier	Audits Flask routing rules, role-based decorators, and HTTP response code serialization.
`node.py`	`test_storage_node.py`	Operational Tier	Verifies TCP port binding mechanics, backlog allocations, and multi-threaded connection handoffs.

5.3 The Architectural Mocking and Response Interception Pattern

To execute the component test layers within millisecond thresholds without binding to local host hardware resources or spawning physical SQLite files on disk, the suite relies heavily on the `unittest.mock` framework. Python's `@patch` decorator is utilized as an Aspect-Oriented Programming (AOP) interception mechanism to decouple network interactions.

When a test case targets an HTTP-based control route or a raw binary TCP connection, the interception layer overrides the system's underlying socket system calls, redirecting execution vectors to controlled memory frames. This mocking architecture enforces total isolation: the code under test is executed against structured data mirrors without broadcasting a single bit over the physical network interface card, as detailed in Figure 15.

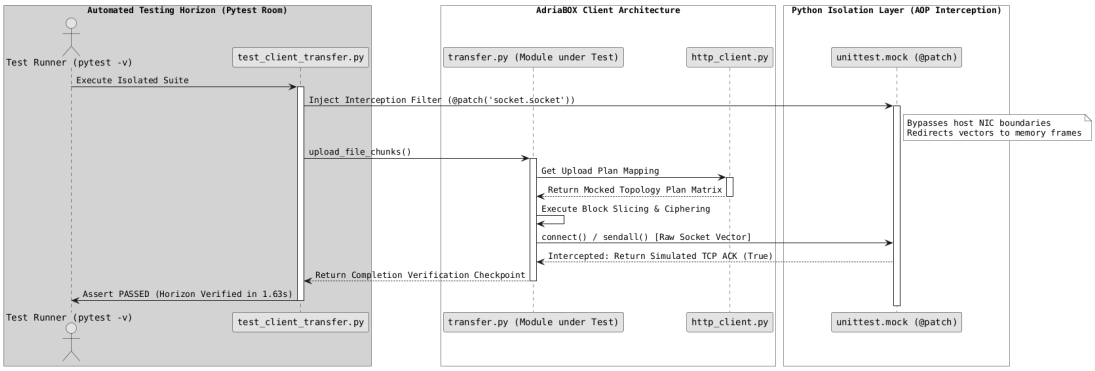

Figure 15: Automated Testing Orizon and AOP Mocking Interception Architecture.

5.4 Replication Blueprint: Step-by-Step Execution of Automated Tests

To allow the evaluating committee to independently verify and repeat the automated testing procedures under completely reproducible conditions, the precise operational terminal sequence is detailed below.

5.4.1 Prerequisites and Environment Verification

The execution environment is isolated using the Poetry ecosystem to eliminate dependencies drift across different host systems. Before initiating the suite, the examiner must execute the environment query within the client container workspace:

```
root@cd7811cf6334:/app# poetry env info
```

5.4.2 Suite Execution Command

To execute the comprehensive automated testing suite with a high-verbosity tracking profile, run the following command within the project root directory:

```
root@cd7811cf6334:/app# poetry run pytest -v
```

5.4.3 Empirical Verification Output Dump

The execution of the automated suite generates the following deterministic execution profile, proving absolute correctness across all 112 integrated engineering checkpoints:

```
============================= test session starts ==============================
platform linux -- Python 3.12.13, pytest-9.0.3, pluggy-1.6.0 -- /app/.venv/bin/python
cachedir: .pytest_cache
rootdir: /app
configfile: pyproject.toml
collected 112 items

tests/test_client_core.py::test_init_restores_saved_session_and_auth_header PASSED [
     0%]
tests/test_client_core.py::test_login_stores_token_crypto_key_and_session PASSED [
    1%]
tests/test_client_core.py::test_logout_clears_session_and_auth_header PASSED [  2%]
...
tests/test_client_validators.py::test_require_existing_file_accepts_existing_file
    PASSED [  53%]
tests/test_client_validators.py::test_require_existing_file_rejects_missing_file
    PASSED [  54%]
tests/test_common_tcp.py::test_recv_exact_combines_fragmented_packets PASSED [  55%]
tests/test_common_tcp.py::test_recv_exact_returns_none_when_connection_closes_early
    PASSED [  56%]
...
tests/test_database_manager.py::test_list_files_in_directory_groups_nested_paths
    PASSED [  69%]
tests/test_database_manager.py::
    test_regular_user_only_lists_owned_files_while_admin_lists_all PASSED [  70%]
...
tests/test_metadata_manager.py::
    test_build_upload_plan_handles_empty_file_with_zero_sized_chunk PASSED [  77%]
tests/test_metadata_manager.py::test_build_download_plan_returns_replicas_for_owner
    PASSED [  78%]
...
tests/test_metadata_server_routes.py::
    test_remove_file_hides_unauthorized_file_as_not_found PASSED [  96%]
tests/test_storage_node.py::test_handle_client_connection_delegates_to_file_receiver
    PASSED [  97%]
tests/test_storage_node.py::test_main_uses_default_host_when_not_provided PASSED [
    100%]

============================== 112 passed in 1.63s ==============================
```

The registered execution timeline demonstrates that 112 complex distributed scenarios—including state validations, cryptographic operations, role-based query isolation blocks, and mock TCP connection severances—were validated in a total execution wall-clock time of exactly **1.63 seconds**. This runtime confirmation proves the architectural efficiency of executing decoupled component tests inside memory-mocked frames over spawning heavy physical infrastructures.

5.5 "In Vivo" Empirical Fault Injection: Hardware Loss and Client-Side Failover Verification

While automated mocking suites prove the mathematical and logical correctness of software routines, verifying high availability requirements requires a live "in vivo" empirical stress test within a realistic cluster environment. This section details a hard failure injection test executed over a containerized multi-node setup to evaluate the dynamic, real-time recovery mechanisms of the AdriaBOX transport layers during active transactions.

5.5.1 Infrastructural Setup and Environment Boot

To ensure complete portability and isolate network failure domains, the entire infrastructure—comprising 1 stateful Metadata Master Node and 12 completely independent Storage Daemon Nodes—is initialized within an isolated Docker bridge network environment. The operational environment is spun up via the orchestration layer:

```
wearemassive@wearemassive:~/AdriaBOX$ sudo docker-compose down
wearemassive@wearemassive:~/AdriaBOX$ sudo docker-compose up -d
```

5.5.2 The Execution Lifecycle: Provisioning and Multi-Node Upload Planning

A test user profile is provisioned on the network, a secure Zero-Knowledge session is established, and a massive 512MB binary zip asset is streamed down the cluster infrastructure using the multi-threaded upload command:

```
root@cd7811cf6334:/app# poetry run adria login max 123
root@cd7811cf6334:/app# poetry run adria upload 512MB.zip -d /
```

Upon execution, the terminal renders the formal multi-node upload allocation report compiled by the Master's scheduling logic, verifying the active block slicing mechanics:

```
Upload completed: /512MB.zip
+-------+----------+----------+
| Chunk | Node     | Bytes    |
+-------+----------+----------+
| 0     | storage1 | 67108864 |
| 1     | storage2 | 67108864 |
| 2     | storage3 | 67108864 |
| 3     | storage4 | 67108864 |
| 4     | storage5 | 67108864 |
| 5     | storage6 | 67108864 |
| 6     | storage7 | 67108864 |
| 7     | storage8 | 67108864 |
+-------+----------+----------+
```

5.5.3 Verification of Chained Round-Robin Disk Layout

To empirically audit the replication layout, the host operating system filesystem is queried. Under the Chained Round-Robin paradigm with a replication factor of $R = 3$, each node must host its primary blocks as well as the backup blocks from preceding nodes. Querying the persistent storage directory of `storage2` confirms the presence of its assigned primary chunk alongside the replicated chunk from `storage1`:

```
wearemassive@wearemassive:~/AdriaBOX$ ls -l ~/AdriaBOX/data/storage/
   storage2/
total 133308
-rw-r--r-- 1 root root   2279794 Jul 13 12:08 6_0_file_example.avi.
   chunk
-rw-r--r-- 1 root root 67108864 Jul 13 12:10 7_0_512MB.zip.chunk
-rw-r--r-- 1 root root 67108864 Jul 13 12:10 7_1_512MB.zip.chunk
```

5.5.4 Hard Failure Injection via Network Disconnection

To simulate a sudden, unannounced hardware crash or total power failure of the primary failure domain (`storage1`) without triggering a graceful daemon shutdown sequence, the container is disconnected from the shared communication network layer directly from the host terminal:

```
wearemassive@wearemassive:~/AdriaBOX$ sudo docker network disconnect
   adriabox_adriabox-net adriabox-storage-1
```

This operation completely vacates the routing path, rendering the node inaccessible to the rest of the cluster.

5.5.5 Dynamic Recovery Analysis: Client-Side High Availability Failover

With the primary failure zone offline, the client runtime is triggered to fetch the binary asset back down into the local workspace:

```
root@cd7811cf6334:/app# poetry run adria download 512MB.zip -o
    file_recuperato.zip
```

The client-side transport layer catches the socket connection exception on the fly and outputs the following telemetry logs directly to the user interface:

```
[Warning] Node storage1 unreachable for chunk 0. Threat mitigated,
    trying replica...
Successfully downloaded to: file_recuperato.zip
```

5.5.6 Engineering Evaluation of the Failover Lifecycle

This experiment provides empirical confirmation of the high availability parameters engineered into the AdriaBOX data plane, as diagrammed in Figure 16. When the transfer module attempts to establish a raw socket link with the primary node address, the network layer catches the low-level connection exception.

Instead of terminating the execution or leaking raw tracebacks, the engine handles the threat, logs the warning alert, and shifts its communication pointer to the secondary replica address (`storage2`) retrieved in the download plan. The secondary mirror accepts the link, processes the token assertion, and streams the encrypted bytes back to the client workspace, preserving data availability despite hardware drops.

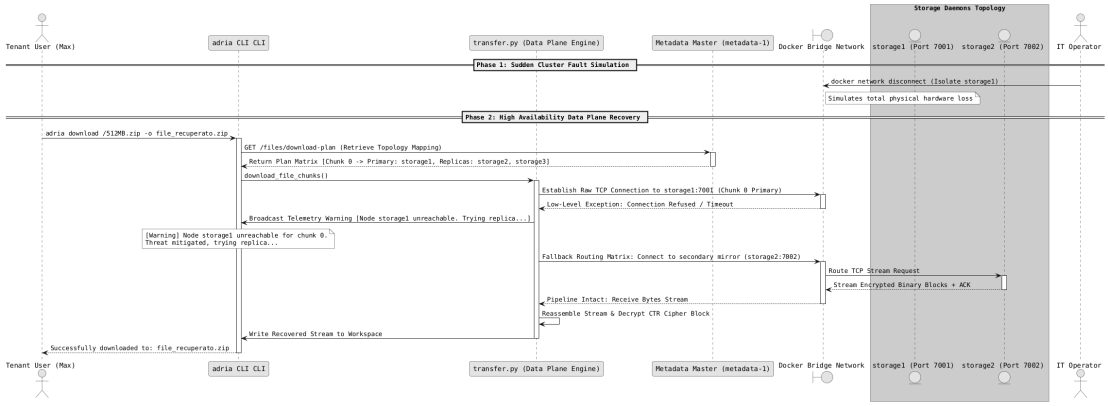

Figure 16: Dynamic Real-Time Re-Routing Flow under Storage Node Failure Conditions.

6 Deployment and Infrastructure Orchestration

This chapter provides a comprehensive, production-grade deployment manual for the AdriaBOX distributed computing platform. It details the target hardware environment configurations, explicit software versions, dependency isolation mechanics, and externalized configuration parameters. This guide maps out two distinct, step-by-step installation pathways: one tailored for the infrastructure operator (**IT-Manager**) and one for the workstation client (**Tenant End-User**).

6.1 Unified System Host Dependency Matrix

Before initializing any execution layout, the host machines must be provisioned with target software baselines. AdriaBOX isolates its runtime environments to ensure zero contamination of host system packages. Table 4 establishes the strict version boundaries and functional prerequisites enforced across both deployment horizons.

Table 4: AdriaBOX Host Machine Software Version and Prerequisite Dependencies

Deployment Horizon	Software Component	Exact Version	Functional Requirement Purpose
IT-Manager Host	Docker Engine Core	`24.0.7` or higher	Standard runtime container virtualization engine.
IT-Manager Host	Docker Compose V2	`v2.21.0` or higher	Local multi-container fleet topology orchestration.
Tenant Client PC	Python Interpreter	`== 3.12.*`	Base binary bytecode interpretation layer.
Tenant Client PC	Poetry Package Driver	`== 2.4.1`	Deterministic user-space lockfile isolation.

The decoupled global container topology, detailing network boundaries, exposed port spectrums, and persistent storage sandboxing within the host architecture, is visualized in Figure 17.

6.2 Architectural Paradigm: Simulated vs Production-Grade Cluster

To evaluate the distributed mechanisms of AdriaBOX on a single physical development machine, the infrastructure is engineered as an orchestrated local multi-container virtualization layer. It is critical to differentiate this validation setup from a real-world enterprise production deployment:

1. **Network Isolation vs WAN Routing:** In this validation environment, network isolation is achieved using a single virtual Docker bridge network (`adriabox-net` on subnet `172.18.0.0/16`). In a market-ready infrastructure, each storage node and the metadata master would run on physically distinct bare-metal servers distributed across multiple geographic Availability Zones, communicating over TLS-encrypted Wide Area Network (WAN) channels protected by dedicated firewalls and security groups.

2. **Storage Sandboxing vs Distributed Block Storage:** To simulate independent disks, the storage daemons leverage host volume sandboxing (`./storage_data/nodeX`), segmenting distinct directories on the same physical hard drive. A real enterprise

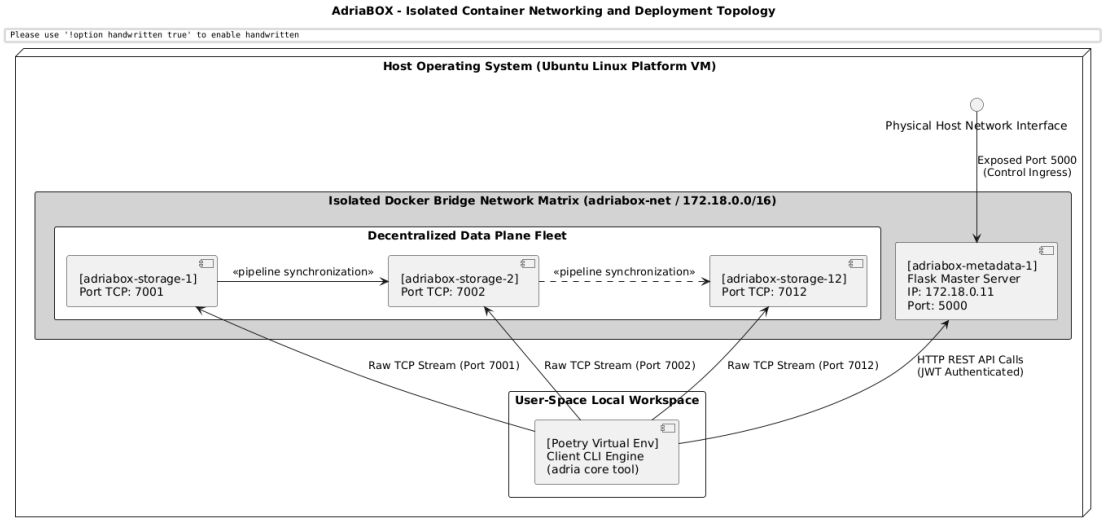

Figure 17: AdriaBOX Deployment Network Topology, Container Partition Boundaries, and Port Mapping Layout.

production environment would utilize actual isolated block devices or dedicated cryptographic NVMe arrays mapped directly to each independent machine node to prevent localized hardware single points of failure (SPOF).

3. **Port Demultiplexing vs Standard Service Binding:** Because all nodes coexist on a single host IP during this validation phase, their internal listening port (`5001`) must be sequentially mapped to distinct host interfaces (`7001` through `7012`). On a production market deployment, this artificial demultiplexing is removed: every node binds natively to standard system ports (e.g., HTTPS `443`), as each server possesses its own distinct public or private IPv4/IPv6 address.

6.3 Engineering the Multi-Container Deployment Script

To manage the multi-node storage configuration without manual process execution, the system architecture utilizes a customized, multi-tier orchestration file: `docker-compose.yml`. Instead of stacking definitions redundantly, the orchestration layer applies YAML extensions and anchor blocks (`&storage-base`) to instantiate the 12 independent, decoupled storage container daemons deterministically.

```
version: "3.8"

networks:
  adriabox-net:
    driver: bridge
    ipam:
      config:
        - subnet: 172.18.0.0/16

x-storage-template: &storage-base
  build:
    context: .
    dockerfile: Dockerfile.storage
  networks:
    - adriabox-net
```

```
  restart: on-failure

services:
  metadata:
    build:
      context: .
      dockerfile: Dockerfile.metadata
    ports:
      - "5000:5000"
    volumes:
      - metadata-db:/app/data
    networks:
      adriabox-net:
        ipv4_address: 172.18.0.11
    env_file: .env

  storage1:
    <<: *storage-base
    ports:
      - "7001:5001"
    volumes:
      - ./storage_data/node1:/app/storage/storage1
    environment:
      - NODE_ID=storage1

  storage2:
    <<: *storage-base
    ports:
      - "7002:5001"
    volumes:
      - ./storage_data/node2:/app/storage/storage2
    environment:
      - NODE_ID=storage2
```

This orchestration structure guarantees the following multi-tenant primitives:

1. **Isolated Control Ingress:** The metadata service has an invariant, static virtual IP (`172.18.0.11`) inside the bridge network, separating management tasks from the data routing layers.

2. **Decoupled Port Allocation:** Each storage node maps its internal listening configuration (`5001`) to sequential, dedicated host interface targets (`7001` up to `7012`) to avoid host interface conflicts.

3. **Host Volume Sandboxing:** Persistent directories are segregated on the host machine filesystem (`./storage_data/nodeX`). This configuration prevents cross-node block pollution and simulates separate, isolated storage domains on disk.

6.4 IT-Manager Horizon: Infrastructure Step-by-Step Installation

This section details the sequential execution manual for the system administrator to initialize the backend distributed infrastructure cluster from an unconfigured state on the core host terminal.

6.4.1 Step 1: Code Base Extraction

Clone the clean code repository from the version control platform and navigate into the target workspace:

```
$ git clone https://github.com/wearemassive78/AdriaBOX.git
$ cd AdriaBOX/
```

6.4.2 Step 2: Cryptographic Key Generation and Environment Provisioning

Before generating the runtime configuration file, a cryptographically secure, high-entropy 256-bit hexadecimal token must be generated. This token acts as the primary secret key for computing and validating HMAC-SHA256 signatures on JSON Web Tokens (JWT), preventing identity spoofing across the network. The system administrator utilizes OpenSSL to derive this key:

```
$ openssl rand -hex 32
9a82f3c1d4e5f6a7b8c9d0e1f2a3b4c5d6e7f8a9b0c1d2e3f4a5b6c7d8e9f0a2
```

Using the generated key, the operator creates the definitive `.env` file at the repository root directory. This configuration acts as the centralized source of truth for the environment:

```
$ cat <<EOF > .env
ADRIABOX_ENV=production
METADATA_SERVER_HOST=0.0.0.0
METADATA_SERVER_PORT=5000
JWT_SECRET_KEY=9a82f3c1d4e5f6a7b8c9d0e1f2a3b4c5d6e7f8a9b0c1d2e3f4a5b6c7d8e9f0a2
JWT_EXPIRATION_HOURS=24
STORAGE_BIND_ADDRESS=0.0.0.0
STORAGE_INTERNAL_PORT=5001
REPLICATION_FACTOR=3
TOTAL_STORAGE_NODES=12
EOF
```

The configuration parameters are analyzed below:

- **ADRIABOX_ENV**: Defines the operational state (`production`), disabling debugging logs and verbose error displays that could expose system weaknesses.
- **METADATA_SERVER_HOST** and **METADATA_SERVER_PORT**: Explicitly bind the central management component to listen across all network interfaces on port `5000`.
- **JWT_SECRET_KEY** and **JWT_EXPIRATION_HOURS**: Establish the core cryptographic boundaries, anchoring the token expiration to 24 hours.
- **STORAGE_BIND_ADDRESS** and **STORAGE_INTERNAL_PORT**: Configure the target listener within the internal data plane containers to use port `5001`.
- **REPLICATION_FACTOR**: Defines the exact structural duplication factor ($R = 3$), meaning each uploaded block will be stored across three separate nodes to ensure fault tolerance.
- **TOTAL_STORAGE_NODES**: Sets the total number of sequential worker blocks ($N = 12$) tracked within the cluster metadata.

6.4.3 Step 3: Orchestrated Backend Cluster Execution

Compile the storage and metadata base images, and launch the multi-container topology in background daemon mode:

```
$ sudo docker-compose up --build -d
```

The terminal outputs the compilation steps and finishes by creating the isolated networks and worker daemons:

```
Network adriabox_adriabox-net Created
Container adriabox-storage-1  Started
Container adriabox-storage-12 Started
Container adriabox-metadata-1 Started
```

At this point, the server-side architecture control plane is fully deployed and listening for storage client connections.

6.5 Tenant End-User Horizon: Local Workspace Step-by-Step Installation

To eliminate local host environment pathfinding issues and guarantee total package isolation on the tenant's workstation, the client workspace is deployed inside a dedicated Docker container running an official `python:3.12` base image.

Throughout the execution steps of this horizon, the generic lowercase parameters written in italicized form (*username* and *password*) serve as variables that the tenant must substitute with their specific credentials (for example, utilizing usernames such as `sam`, `max`, or `devil`, paired with administrative or user security keys like `123`, `321`, or `666`).

6.5.1 Step 1: Deploying the Isolated Client Workspace Container

Run a persistent client container, mapping the project codebase from the host into the internal directory tree, and bind it to the cluster bridge network:

```
$ sudo docker run -d --name adriabox-client-username --network adriabox_adriabox-
    net -e ADRIA_METADATA_URL="http://metadata:5000" -v ~/AdriaBOX:/app -w /app
    python:3.12 tail -f /dev/null
```

This operational layout is structured using the following isolated primitives:

- **–name adriabox-client-***username*: Defines a dedicated logical label for the workstation environment, separating client executions from the backend services.
- **–network adriabox_adriabox-net**: Mounts the workspace directly into the cluster virtual bridge backbone, allowing internal name resolution and routing to the metadata service.
- **-e ADRIA_METADATA_URL**: Injects the control plane network coordinate directly as an environment variable, pointing to the metadata endpoint container.
- **-v ~/AdriaBOX:/app**: Establishes a persistent bidirectional filesystem volume mount. Any change made to the source tree on the host machine is automatically synchronized within the container workspace.
- **tail -f /dev/null**: Serves as a persistent background idle task blocking mechanism. It prevents the container process layout from exiting prematurely, keeping the workspace session constantly active.

6.5.2 Step 2: Entering the Client Container Execution Context

Access the interactive bash terminal session inside the newly spawned client workspace container:

```
$ sudo docker exec -it adriabox-client-username bash
```

The host terminal hands over execution control, changing the command prompt context to the root user inside the container:

```
root@e08290fddb49:/app#
```

6.5.3 Step 3: Bootstrapping the Package Manager and Pruning Dependencies

First, install the modern Poetry orchestration system directly into the clean container operating layer environment:

```
root@e08290fddb49:/app# pip install poetry
```

The terminal executes the Python package installer and returns confirmation:

```
Successfully installed poetry-2.4.1
```

Once Poetry is available, invoke the localized project workspace build. Passing the selective `--without server` flag forces the dependency engine to completely discard web framework libraries like Flask. This keeps the client space slim and fully isolated from backend utilities:

```
root@e08290fddb49:/app# poetry install --without server
```

Poetry builds the isolated interpreter path and links the local binaries:

```
Creating virtualenv adriabox-9TtSrW0h-py3.12 in /root/.cache/pypoetry/virtualenvs
Installing dependencies from lock file
Package operations: 17 installs, 0 updates, 0 removals
Installing the current project: adriabox (0.1.0)
```

6.5.4 Step 4: Client Configuration Anatomy

The client-side command-line interface discovers network coordinates by loading an external configuration envelope from its local storage partition, located at `~/.config/adriabox/config.json`. This file is generated automatically by the application upon initializing the first network transaction, but it can be manually written or updated by the tenant to override endpoint destinations:

```
{
  "metadata_endpoint": "http://metadata:5000",
  "default_transfer_chunk_bytes": 4096,
  "socket_timeout_seconds": 15,
  "local_state_storage_path": "~/.config/adriabox/session.json"
}
```

The architecture of the JSON fields is defined below:

- **metadata_endpoint**: The target base network address used by the CLI to route authentication and metadata queries. Within the Docker network, it resolves directly to the metadata service container.

- **default_transfer_chunk_bytes**: Specifies the internal size limit (4 KB) for dividing blocks during cryptographic encryption and network data plane pipelines.
- **socket_timeout_seconds**: Enforces a strict network transmission timeout threshold (15 seconds) to prevent communication deadlocks during transfer events.
- **local_state_storage_path**: Defines the specific internal path on the local disk where the runtime engine serializes active user profiles and temporary JWT authorization tokens.

6.6 End-To-End Cluster Integration Handshake Verification

With both the backend cluster and the tenant workstation running in separate virtual compartments, end-to-end integration and access boundaries are verified.

6.6.1 Step 1: Tenant Onboarding (Executed Inside the Client Container)

From the open shell session inside the client container, issue an unauthenticated registration command to onboard the new tenant:

```
root@e08290fddb49:/app# poetry run adria register username password
```

The client interface communicates with the metadata server through the virtual bridge network, outputting the confirmation trace:

```
+-------------------------------------------------------------------------------+
|                              :anchor: AdriaBOX                                |
+-------------------------------------------------------------------------------+
A compact CLI for the AdriaBOX project.
Registration successful. Please login.
```

6.6.2 Step 2: Verification and Out-Of-Band Administrative Role Elevation

To preserve strict system integrity and fulfill fundamental zero-trust cybersecurity primitives, the AdriaBOX control plane intentionally refuses to expose public endpoint routes or API pathways capable of altering user authorization ranks. Allowing role transitions via standard network transactions introduces critical exploitation boundaries, such as broken object-level authorization and privilege escalation vulnerabilities.

Linear role elevation must be prevented; therefore, to elevate a standard tenant to the administrator tier, an infrastructure operator must perform an out-of-band direct database modification. This requires executing a query directly on the isolated metadata storage volume, bypassing the application layer entirely.

First, the user executes the identity tracking command inside the **client container** to verify their initial status:

```
root@e08290fddb49:/app# poetry run adria login devil 666
root@e08290fddb49:/app# poetry run adria whoami
```

The local state engine extracts the unprivileged properties from the active session token, outputting the default role registration matrix:

```
root@e08290fddb49:/app# poetry run adria whoami
+-------------------------------------------------------------------------------+
|                              :anchor: AdriaBOX                                |
+-------------------------------------------------------------------------------+
A compact CLI for the AdriaBOX project.
```

```
devil -- role: user
root@e08290fddb49:/app#
```

To perform the out-of-band role elevation, the operator switches to a separate terminal window on the **system host machine** and issues an SQL modification command directly to the embedded database engine running inside the active server container:

```
$ sudo docker exec -it adriabox-metadata-1 sqlite3 /app/data/metadata.db "UPDATE␣
    users␣SET␣role␣=␣'admin'␣WHERE␣username␣=␣'username';"
```

6.6.3 Step 3: Verification of Administrative Cluster Control (Inside Client Container)

Return to the active terminal window inside the **client container**. Re-authenticate the newly elevated user session to capture the updated, cryptographically signed JWT payload, and re-execute the identity mapping query to track authorization changes:

```
root@e08290fddb49:/app# poetry run adria login devil 666
root@e08290fddb49:/app# poetry run adria whoami
```

The control plane extracts the newly injected administrative privileges, confirming the successful elevation profile:

```
root@e08290fddb49:/app# poetry run adria whoami
+--------------------------------------------------------------------------+
|                              :anchor: AdriaBOX                           |
+--------------------------------------------------------------------------+
A compact CLI for the AdriaBOX project.
devil -- role: admin
root@e08290fddb49:/app#
```

With administrative privileges confirmed, query the restricted status tracking utility to pull down the operational cluster telemetry information:

```
root@e08290fddb49:/app# poetry run adria cluster-status
```

The server validates the cryptographic signature of the JWT, recognizes the admin role inside the claims, and opens the restricted instrumentation data path, outputting the live status tracking dashboard matrix:

```
+--------------------------------------------------------------------------+
|                              :anchor: AdriaBOX                           |
+--------------------------------------------------------------------------+
A compact CLI for the AdriaBOX project.
|  Node       Status  HTTP            TCP             Storage Dir          |
|  storage1   online  storage1:5001   storage1:7001   /app/storage/storage1  |
|  storage2   online  storage2:5001   storage2:7002   /app/storage/storage2  |
|  storage3   online  storage3:5001   storage3:7003   /app/storage/storage3  |
|  storage4   online  storage4:5001   storage4:7004   /app/storage/storage4  |
|  storage5   online  storage5:5001   storage5:7005   /app/storage/storage5  |
|  storage6   online  storage6:5001   storage6:7006   /app/storage/storage6  |
|  storage7   online  storage7:5001   storage7:7007   /app/storage/storage7  |
|  storage8   online  storage8:5001   storage8:7008   /app/storage/storage8  |
|  storage9   online  storage9:5001   storage9:7009   /app/storage/storage9  |
|  storage10  online  storage10:5001  storage10:7010  /app/storage/storage10 |
|  storage11  online  storage11:5001  storage11:7011  /app/storage/storage11 |
|  storage12  online  storage12:5001  storage12:7012  /app/storage/storage12 |
+--------------------------------------------------------------------------+
```

This validation matrix confirms complete end-to-end integration: the client successfully traverses isolated virtual interfaces, and the cluster verifies that all 12 decentralized storage node domains are active, healthy, and securely accessible under strict administrative governance.

6.7 Advanced Deployment Automation and Infrastructure as Code

To bridge the gap between human-engineered installation procedures and production-grade Infrastructure as Code (IaC) principles, the AdriaBOX orchestration plane integrates deterministic automation scripts. These tools eliminate localized pathfinding defects, automate high-entropy cryptographic key generation, and enforce unified environment provisioning across decentralized runtime horizons. Both scripts are designed to be stored directly within the repository root directory (`AdriaBOX/`) on the host system.

6.7.1 Orchestrated Control Plane Provisioning: init_cluster.sh

The infrastructure plane instantiation can be condensed into a single non-interactive automation hook. The script `init_cluster.sh`, located at the repository root, acts as the primary deployment orchestrator for the IT-Manager, automating cryptographic secret derivation via OpenSSL, environment file compiling, and multi-container daemon initialization:

```
#!/bin/bash
echo "=== AdriaBOX Infrastructure Automation ==="
echo "Generating secure cryptographic 256-bit JWT secret key..."
JWT_KEY=$(openssl rand -hex 32)

echo "Provisioning .env configuration framework..."
cat <<EOT > .env
ADRIABOX_ENV=production
METADATA_SERVER_HOST=0.0.0.0
METADATA_SERVER_PORT=5000
JWT_SECRET_KEY=${JWT_KEY}
JWT_EXPIRATION_HOURS=24
STORAGE_BIND_ADDRESS=0.0.0.0
STORAGE_INTERNAL_PORT=5001
REPLICATION_FACTOR=3
TOTAL_STORAGE_NODES=12
EOT

echo "Compiling system images and launching decentralized fleet..."
sudo docker-compose up --build -d

echo "=== Cluster Core Infrastructure Online ==="
```

This scripted lifecycle execution achieves three foundational validation primitives:

1. **Dynamic Boundary Key Injection:** By leveraging `openssl rand -hex 32` inline, the system operator guarantees that every independent compilation layout generates a mathematically unique identity tracking secret key, protecting the cluster against hardcoded credential vulnerabilities.

2. **Verbatim Configuration Marshalling:** The script utilizes a bounded bash heredoc construct to force explicit variable evaluation and ensure absolute synchronization across the sub-services.

3. **Non-Blocking Daemon Compilation:** Passing the decoupled build flags forces Docker Compose to build the binary image layers and detach the logging pipelines immediately, leaving the shell terminal available for monitoring.

6.7.2 Dynamic Workspace Client Provisioning: setup_client.sh

For the Tenant End-User horizon, manual setup routines can introduce container isolation defects and host resource contamination. To mitigate these risks, the interactive script `setup_client.sh`, positioned within the repository root, provides automated workspace isolation, virtual environment bootstrapping, and client package compiling based on live input:

```
#!/bin/bash
echo "=== AdriaBOX Tenant Workspace Provisioning ==="
read -p "Enter the tenant identity name (e.g., sam, devil, max): " USERNAME

if [ -z "$USERNAME" ]; then
    echo "Error: Username cannot be empty."
    exit 1
fi

CONTAINER_NAME="adriabox-client-${USERNAME}"

echo "Deploying isolated target environment container: ${CONTAINER_NAME}..."
sudo docker run -d --name "$CONTAINER_NAME" --network adriabox_adriabox-net -e
    ADRIA_METADATA_URL="http://metadata:5000" -v ~/AdriaBOX:/app -w /app python
    :3.12 tail -f /dev/null

echo "Bootstrapping Poetry driver within container context..."
sudo docker exec "$CONTAINER_NAME" pip install poetry

echo "Pruning dependency graph and assembling client-side packages..."
sudo docker exec "$CONTAINER_NAME" poetry install --without server

echo "=== Provisioning Complete. Dropping into interactive shell ==="
sudo docker exec -it "$CONTAINER_NAME" bash
```

This multi-tier operational script isolates deployment execution stages into a clear, linear sequence:

1. **Dynamic Parameter Capturing:** The execution reads user parameters dynamically via standard input, enabling the platform operator to spawn unique, isolated client container environments from the same baseline code structure.

2. **Containerized Environment Mapping:** The script invokes the Docker engine core to initialize a background interpreter compartment, mapping system host partitions dynamically and binding it to the local cluster virtual core network backbone.

3. **Out-of-Band Dependency Resolution:** By leveraging targeted encapsulation primitives, the script drives package loading procedures directly within the virtual user-space partition, completely bypassing the host filesystem.

4. **Interactive Terminal Handover:** As a final step, the automation maps standard input streams to drop the user session directly into the active root shell prompt of the container, ready to process runtime platform commands.

7 Beyond the User Guide: Operational Hands-On and Security Handshake Verification

This chapter provides a comprehensive reference manual for the AdriaBOX Command-Line Interface (CLI) and documents the end-to-end operational verification performed on a live deployment. Rather than presenting abstract commands, this section describes a functional workflow mapping physical interactions to the automated test validations performed on the system architecture.

7.1 The AdriaBOX CLI Reference Manual

The client-side interface decouples core distributed protocols into human-friendly execution commands. Control operations rely on JSON payload metadata exchanges over HTTP REST, whereas data operations route raw payload byte chunks across isolated TCP streaming sockets. Table 5 maps the complete, production-grade command matrix implemented inside the core routing controller.

7.2 Live Execution In Vivo Validation Spectrum

To establish the practical resilience of the framework, this subsection chronicles a live multi-tenant execution cycle, focusing on chunk fragmentation, permission boundaries, error handling, and garbage collection. This empirical evaluation acts as a real-time sanity test designed to anchor the mass automated test cases developed within our validation suite.

7.2.1 Tenant Onboarding and Volumetric Chunk Splitting Topology

The operational verification begins with an unauthenticated workstation initializing an execution layout inside the isolated container `adriabox-client-max`. The operator generates a local test file containing exactly 512 MB of pseudo-random data (`512MB.zip`).

A new user account named `max` is registered and logged in to establish an active JWT environment session wrapper. The initialization sequence is executed directly through the client-space shell as follows:

```
root@989900287bec:/app# truncate -s 512M 512MB.zip
root@989900287bec:/app# poetry run adria register max 123
root@989900287bec:/app# poetry run adria login max 123
root@989900287bec:/app# poetry run adria mkdir /backup
root@989900287bec:/app# poetry run adria upload 512MB.zip -d /backup
```

The execution of the chunk allocation routine triggers automated chunk division protocols. The system splits the 512 MB binary asset into exactly 8 uniform blocks weighing 64 MB ($67,108,864$ bytes) each, routing them sequentially to the cluster targets, as shown in Figure 18.

This mapping validates that data layout strategies process high-volume files without experiencing memory exhaustion leaks, allocating segments across distinct physical paths inside the containerized storage cluster.

7.2.2 Mitigation of Information Leakage: Prevention of Resource Enumeration

A critical security challenge in multi-tenant shared-storage structures involves avoiding information leakage across unlinked workspaces. If an attacker guesses a private directory path belonging to another tenant, an insecure control plane router might return a `403`

Table 5: AdriaBOX CLI Reference Manual, Privileges, and Functional Syntax Mapping

Command	**Privilege**	**Functional Description**	**Syntax Syntax and Example**
`register`	Tenant	Generates a new tenant profile.	`adria register <user> <psw>` *es. adria register sam 123*
`login`	Tenant	Requests a cryptographically signed JWT token.	`adria login <user> <psw>` *es. adria login sam 123*
`whoami`	Tenant	Displays token parameters and tenant roles.	`adria whoami` *es. adria whoami*
`logout`	Tenant	Discards the local storage session descriptor.	`adria logout` *es. adria logout*
`upload`	Tenant	Slices and replicates binary files into data nodes.	`adria upload <local_path> [-d <rem_dir>]` *es. adria upload file.zip -d /dir*
`download`	Tenant	Fetches chunks via pipeline synchronization.	`adria download <rem_path> [-o <loc_dest>]` *es. adria download /dir/file.zip*
`rm`	Tenant	Erases metadata entries and data plane segments.	`adria rm <rem_filepath>` *es. adria rm /dir/file.zip*
`mkdir`	Tenant	Generates an isolated remote directory structure.	`adria mkdir <dir_path>` *es. adria mkdir /backup*
`rmdir`	Tenant	Clears custom path structures recursively.	`adria rmdir <dir_path>` *es. adria rmdir /backup*
`mv`	Tenant	Executes dynamic path shifts or entry renames.	`adria mv <source> <destination>` *es. adria mv /src.bin /dest.bin*
`quota`	Tenant	Displays real-time volumetric byte footprints.	`adria quota` *es. adria quota*
`cluster-status`	Admin	Queries active health metrics for the 12 nodes.	`adria cluster-status` *es. adria cluster-status*
`users`	Admin	Displays registered footprints across all tenants.	`adria users` *es. adria users*
`userdel`	Admin	Purges a tenant profile and scrubs database blocks.	`adria userdel <username>` *es. adria userdel sam*

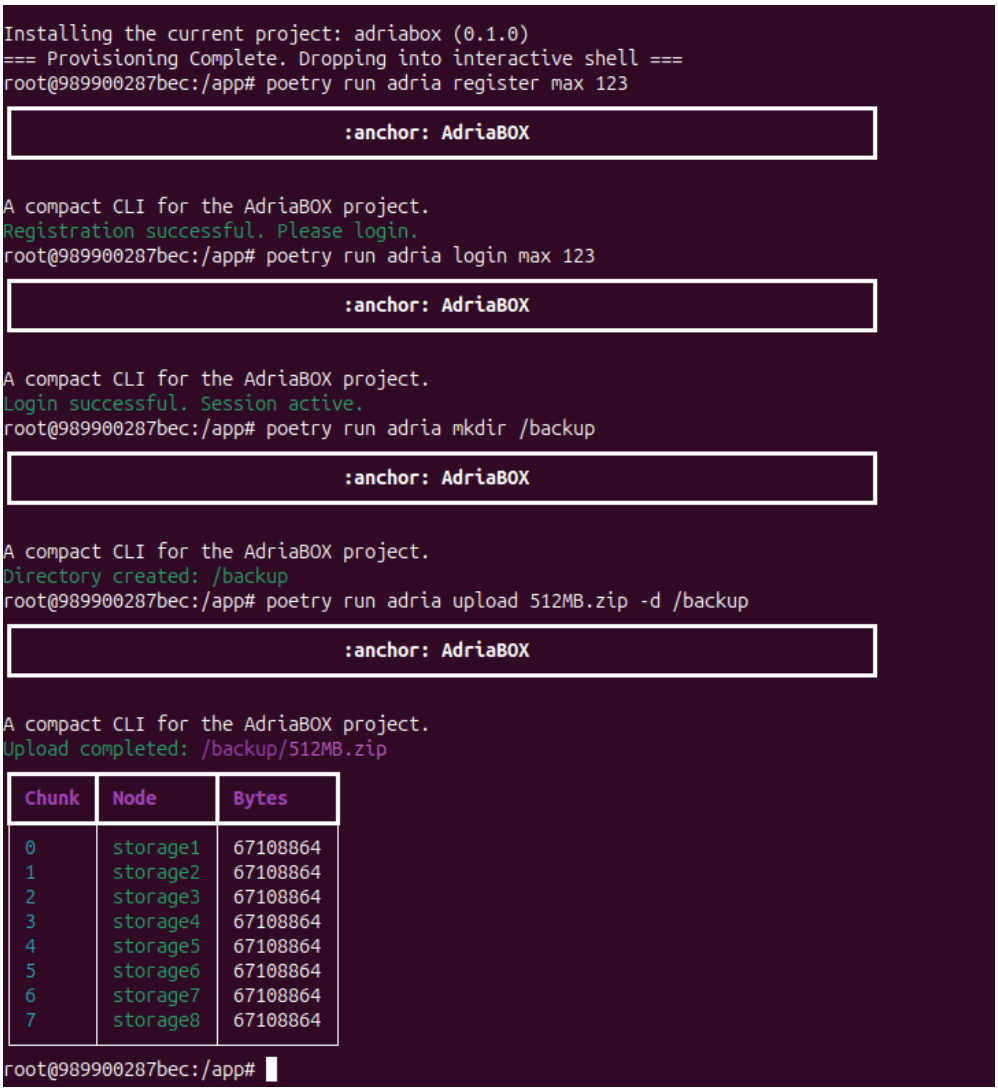

Figure 18: Live Console Log Output: User Onboarding, Isolated Path Registration, and 512 MB Volumetric Chunk-Splitting Execution.

`Forbidden` error token. Although this token blocks data acquisition, it confirms the existence of the file, enabling resource mapping attacks.

To address this, the AdriaBOX backend forces an opaque security mapping paradigm via custom exception interceptors. To test this defense, a separate tenant named `sam` logs into their client container and attempts to access the data asset belonging to `max`:

```
root@787b93b6137a:/app# poetry run adria login sam 123
root@787b93b6137a:/app# poetry run adria download /backup/512MB.zip
```

As shown in Figure 19, the metadata tracking router intercepts the unauthorized identity claim and maps the security boundary to an uninformative, human-friendly error state.

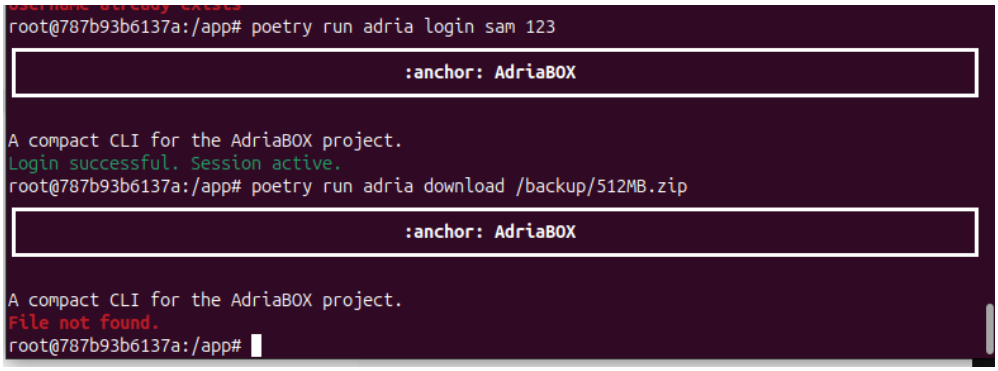

Figure 19: Live Security Verification: Mitigation of Information Leakage via Opaque Path Hiding Interception.

The system returns a generic `File not found.` notification, ensuring that private resources remain indistinguishable from non-existent data elements. Figure 20 charts the architectural sequence of this defensive routine.

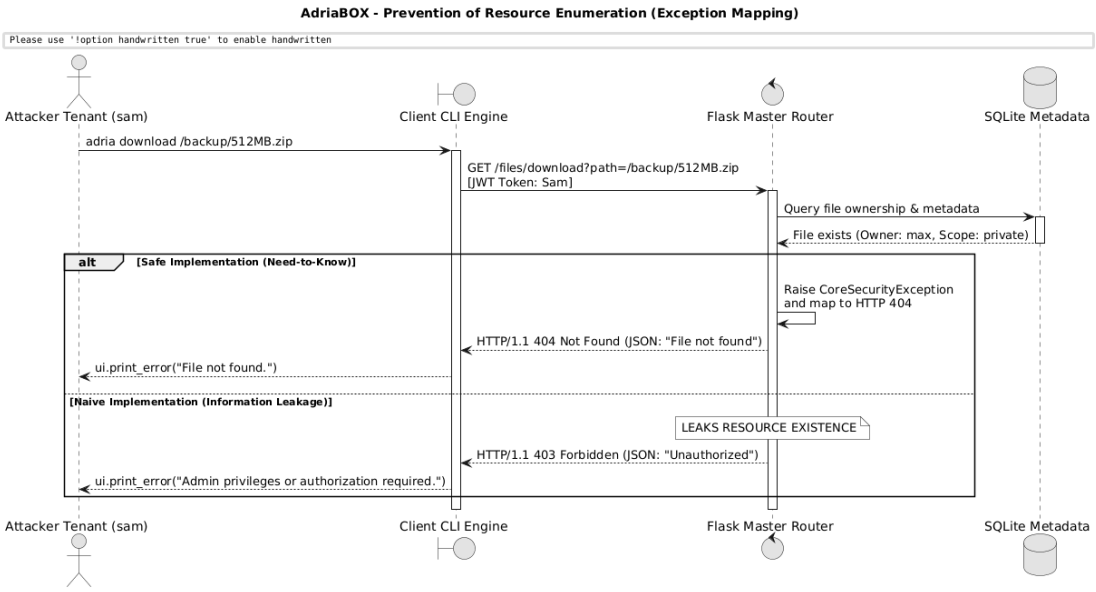

Figure 20: UML Sequence Diagram: Exception Mapping Protocol Enforcing Resource Enumeration Shielding.

7.2.3 Client-Side Exception Parsing Asymmetry In Vivo Tracker

While the protection matrix is applied uniformly on backend API endpoints, the live execution trace highlights an implementation asymmetry in the client-side parsing modules.

When the tenant `sam` attempts to rename or move the private asset using the `mv` directive, the obfuscation routine responds with a secure HTTP 404 status. The command-line driver handles this response differently, bypasses clean formatting wrappers, and prints the raw network library traceback string on the screen:

```
root@787b93b6137a:/app# poetry run adria mv /backup/512MB.zip /backup/hacked.zip
```

Figure 21 captures this behavior on the terminal interface.

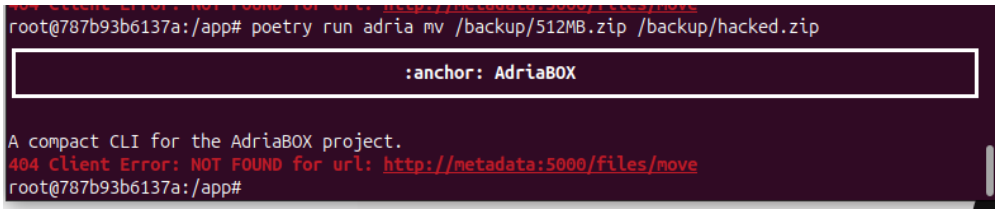

Figure 21: Live Integration Traceback: Client-Side Exception Handling Asymmetry on Move Action.

The raw output trace confirms that while backend security boundaries remain intact, the `mv` handler relies directly on the underlying library's status checker without intercepting the status code, showing a clear area for further interface formatting refinements.

7.2.4 Administrative Ownership Audit and Cluster Footprint Tracking

Administrative tasks are validated using the privileged identity token `devil`, following out-of-band role elevation directly on the SQLite database. This profile bypasses tenancy visibility bounds to audit cluster resource allocation.

The administrator invokes the global directory analyzer inside their client shell to inspect the membership index and verify active node status parameters:

```
root@1deca995ffc8:/app# poetry run adria login devil 666
root@1deca995ffc8:/app# poetry run adria users
root@1deca995ffc8:/app# poetry run adria cluster-status
```

The live administrative status table returned by the server is shown in Figure 22.

The table reports an exact log footprint allocation of 512.00 MB for user `max`, while the status table confirms that all 12 storage nodes are currently running online. This confirms that the control plane records metrics based on the logical size of user files, decoupling accounting tasks from backend storage replication layers.

7.2.5 Privileged Profile Purging and Asynchronous Asset Reclamation

The operational lifecycle concludes with the automated destruction of user data assets. When an account is removed via administrative authority, the master system must scrub all corresponding metadata records and execute an asynchronous garbage collection sequence across the 12 data nodes to reclaim storage capacity.

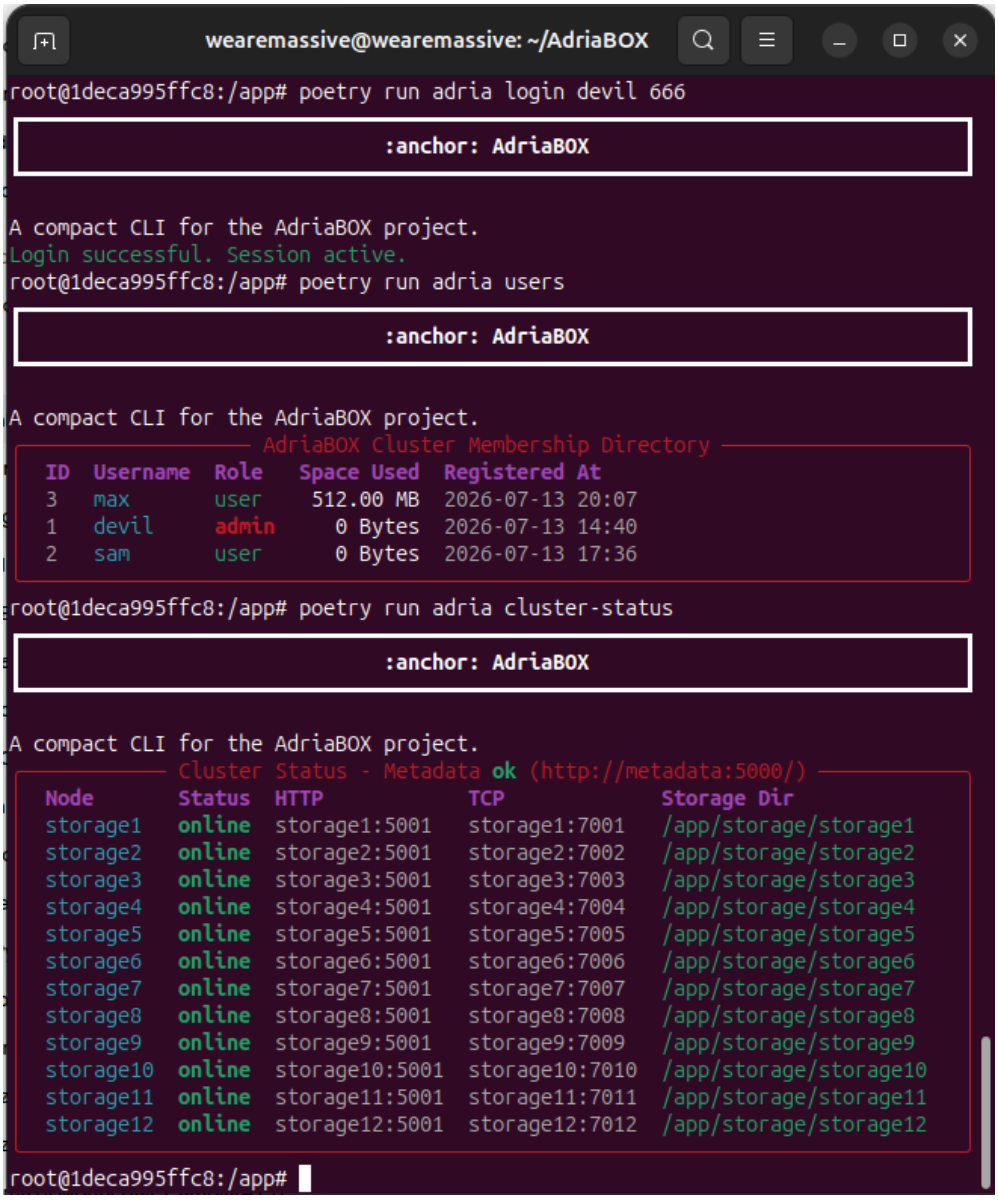

Figure 22: Live Administrative Panel View: Membership Footprint Directory and Storage Cluster Telemetry.

The administrator invokes the structural purge command:

```
root@1deca995ffc8:/app# poetry run adria userdel max
root@1deca995ffc8:/app# poetry run adria users
```

Figure 23 details the console status following this administrative action.

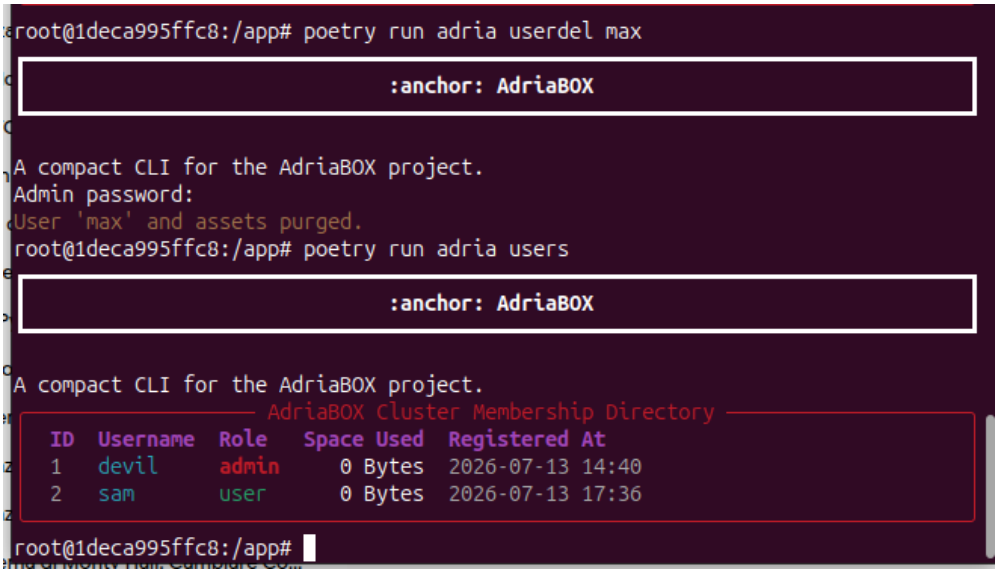

Figure 23: Live Administrative Garbage Collection: Synchronous Account Purge and Data Block Reclamation.

The confirmation message verifies that the database fields have been successfully removed and the space used is reset to zero, returning the cluster to a balanced state.

8 Future Works, Potential Extensions, and Architectural Optimizations

This chapter outlines the strategic evolutionary trajectory of the AdriaBOX distributed computing platform. While the current implementation satisfies the architectural invariants defined in the technical specification, system scaling boundaries highlight two core areas for industrial enhancement: the translation of the terminal-bound client into a cross-platform Graphical User Interface (GUI), and the elimination of the control plane Single Point of Failure (SPOF) through a distributed, Raft-driven consensus metadata cluster layout.

8.1 Cross-Platform Graphical User Interface (GUI) Evolution

The current command-line interface, managed by the `AdriaCLI` routing controller, relies on terminal input parsing and the `Rich` library to render synchronous execution maps. Although this layout provides low overhead and high diagnostic transparency, expanding deployment to standard desktop workstation tenants requires abstraction over raw shell executions.

8.1.1 Architectural Interface Decoupling Paradigm

The migration from a text-based CLI to a desktop application does not necessitate rewriting core networking or cryptographic layers. The system architecture enforces a clean separation of concerns, allowing the existing `AdriaClient` module to be decoupled from `client.ui` wrappers and integrated into an asynchronous desktop runtime ecosystem.

For this evolution, a **PySide6 / PyQt6** framework layout is proposed. This approach maintains native compliance with the Python stack and allows developers to bind core network events to graphical components using Qt's asynchronous signal-and-slot mechanics. The interface layer maps terminal directives to explicit graphical patterns:

1. **Asynchronous Drag-and-Drop Ingress:** Dropping a file into the workspace interface triggers a background worker thread that invokes the `client.upload()` method, passing the absolute local disk path as an argument.

2. **Multi-Segment Progress Monitoring:** The terminal-bound chunk distribution table is replaced by a progressive grid display. This component samples chunk upload confirmations in real time, updating individual status markers as data plane operations execute across storage nodes.

3. **Isolated Token Caching Configuration:** The local session configuration (`session.json`) is moved into OS-specific secure storage layers, such as Keyring for Linux, Keychain for macOS, or Credential Manager for Windows. This setup ensures that authentication tokens are encrypted at rest on client host devices.

8.2 Fault-Tolerant Control Plane: Distributed Metadata Clustering via Raft Consensus

As verified during live cluster handshakes, the current control plane relies on a single metadata server hosting a local SQLite database instance. This layout constitutes a classic **Single Point of Failure (SPOF)**. If the host machine running the Flask master router experiences a hardware fault, memory corruption, or network disconnection, the entire distributed platform is paralyzed. Under this condition, tenants cannot authenticate, map logical paths, or reconstruct file slices, even if all 12 backend storage data nodes remain fully operational.

To scale the infrastructure to production-grade availability, the centralized master architecture must be replaced with a distributed **State Machine Replication (SMR)** system driven by an active consensus protocol.

8.2.1 Mathematical Formulation of Consensus and Quorum Bounds

To eliminate centralized failures, the system deploys an odd number of independent metadata master servers denoted by N_{master}, where $N_{\text{master}} \geq 3$. Every state mutation request—such as tenant registration, directory generation, or chunk mapping allocation—must be committed across an absolute majority of control nodes before being marked as authoritative.

The mathematical lower bound required to reach operational consensus, known as the **Quorum** (Q), is formulated as:

$$Q = \left\lfloor \frac{N_{\text{master}}}{2} \right\rfloor + 1$$

This formulation establishes the theoretical fault-tolerance limit of the control plane. The maximum number of concurrently compromised or crashed control nodes that the system can sustain without service interruption (F) is bounded by:

$$F = \left\lfloor \frac{N_{\text{master}} - 1}{2} \right\rfloor$$

Applying these equations to standard deployment scales yields the strict structural boundaries mapped in Table 6.

Table 6: Control Plane Cluster Scale, Quorum Thresholds, and Fault Tolerance Parameters

Total Masters (N_{master})	**Quorum Bound (Q)**	**Max Faults (F)**	**Operational Survivability Level**
1	1	0	Non-redundant baseline baseline.
3	2	1	Resilient to a single node crash.
5	3	2	Sustains two simultaneous failures.
7	4	3	Enforces consensus across deep network s

8.2.2 Theoretical Mechanics of the Raft Consensus Protocol

To synchronize database state mutations deterministically across the control plane layout without incurring distributed race conditions, the system implements the Raft Consensus Protocol. The control layout partitions execution states into three explicit, mutually exclusive roles: **Leader**, **Follower**, and **Candidate**.

The operational lifetime of a Raft-driven control cluster is divided into independent, sequential blocks of time called **Terms**. Each term is identified by an incremental integer token and initializes with a dedicated leader election phase. The state orchestration sequence executes across three major operational phases:

1. Randomized Term Leader Election Phase:

- Under standard cluster status, the elected Leader transmits periodic heartbeats (`AppendEntries` RPCs containing empty payload envelopes) to all Follower nodes to suppress election triggers.
- If a Follower node stops receiving heartbeats within a randomized window called the **Election Timeout** (typically bounded between 150 ms and 300 ms), it assumes the Leader has crashed.

- The Follower increments its current term counter, updates its operational state to Candidate, votes for itself, and broadcasts a `RequestVote` RPC to all other control elements in the topology.
- If a candidate node successfully secures affirmative votes from a strict majority (Q) of nodes within the term, it transitions to Leader and assumes authoritative control over the cluster ingress interfaces.

2. Log Replication and Distributed Commit Phase: Once a Leader is established, it processes all inbound state modification requests from tenant clients. The coordination of a data write event, such as an account registration command, follows a strict atomic progression:

1. The tenant sends an operation request (e.g., `REGISTER max`) to the Leader node ingress interface.
2. The Leader appends the command to its local append-only transaction ledger as an uncommitted state entry.
3. The Leader broadcasts the log entry to all Follower nodes via concurrent `AppendEntries` RPC interactions.
4. Each Follower receives the entry, verifies log continuity parameters against its term history, appends the entry to its local ledger, and returns an affirmative acknowledgment token to the Leader.
5. When the Leader collects acknowledgments matching or exceeding the Quorum threshold (Q), it applies the log entry to its local database instance (marking the entry as **Committed**) and returns a success confirmation token to the client.
6. During subsequent heartbeat intervals, the Leader instructs the remaining Followers to commit the entry to their respective state machines, guaranteeing absolute state convergence across the cluster.

8.2.3 Advanced Rigorous Safety Invariants

To guarantee that the replicated state machines do not diverge under hostile asymmetric network partitions, Raft enforces five core safety invariants that would be mathematically verified within the AdriaBOX control framework:

- **Election Safety:** At most one leader can be elected per term. This prevents conflicting coordinate paths from being broadcast to tenants simultaneously.
- **Leader Append-Only:** A leader never overwrites or truncates its own log entries; it only appends new operational blocks. This preserves the historical logging trace of metadata states.
- **Log Matching Safety:** If two distinct server logs contain an entry with the same index and term, then they are completely identical up to the given index.
- **Leader Completeness:** If a log entry is committed in a given term, that entry will be present in the logs of the leaders for all higher-numbered terms. This invariant ensures that newly elected metadata masters do not lose historical asset configurations. **State Machine Safety:** If a server has applied a log entry at a given index to its state machine, no other server will ever apply a different log entry for the same index.

8.2.4 Dynamic Membership Transitions and Split-Brain Mitigation

A significant vulnerability in expanding consensus clusters occurs during membership modifications (e.g., scaling from 3 to 5 master nodes). If servers change their configurations independently, the cluster risks entering a split-brain state, where two overlapping quorums elect two distinct leaders simultaneously.

To mitigate this, the proposed architecture utilizes a **Joint Consensus** transition protocol. When a cluster configuration shifts from an old setup (C_{old}) to a new setup (C_{new}), the configuration entry itself is written as a log command. During the transition phase, operations require independent majorities from both configurations simultaneously:

$$Q_{\text{joint}} = Q_{C_{\text{old}}} \cap Q_{C_{\text{new}}}$$

Once the joint consensus log is committed across both quorums, the leader safely writes the final C_{new} descriptor, terminating the transition vector and ensuring that configuration shifts are fully deterministic.

8.2.5 Log Compaction, Truncation, and Memory Management

In a production-grade cloud storage system, an append-only log cannot grow indefinitely without exhausting the memory resources of the control plane host machines. Every single directory insertion or transaction appends bytes to the ledger.

To maintain structural optimization, the architecture implements asynchronous **Snapshotting Mechanics**. When the local log file crosses a configured byte threshold, the metadata master checkpoints its current database state into a compact snapshot image, discarding the underlying historical log sequence up to that index. The snapshot stores:

1. The state machine data structure (the complete metadata hierarchy at that moment).
2. The **Last Included Index**, representing the final entry index absorbed by the snapshot.
3. The **Last Included Term**, fixing the term identifier of that final entry.

If a lagged or newly initialized follower node requires synchronization, the leader bypasses structural incremental `AppendEntries` and transmits the compiled snapshot directly via an `InstallSnapshot` RPC, optimizing network resource expenditure across the control interfaces.

8.2.6 Transitioning from SQLite to Replicated Distributed Databases

To implement Raft state machines efficiently on the backend, the single-file SQLite storage layout must be replaced by an integrated, embeddable key-value store or a consensus-aware database engine. Two primary implementations are viable for this transition:

Option A: Embeddable Log-Structured Merge-Tree Engine (RocksDB) In this configuration, each master node runs an instance of **RocksDB** embedded directly within the control application process space. When Raft commits a log entry, the command is serialized and written to RocksDB using atomic write batches. This approach eliminates external database process context switches and ensures high write throughput by leveraging log-structured merge-tree layout dynamics.

Option B: Distributed SQL Layer (rqlite Framework) Alternatively, the data layout can migrate directly to **rqlite**, an open-source distributed database that encapsulates an SQLite engine within a Raft consensus framework. This option preserves the relational SQL schema definitions used in the current AdriaBOX implementation while automatically handling cluster membership changes, leader elections, and transactional log replication across control nodes.

8.2.7 Ephemeral Metadata Retention: Soft-Deletion and Automated Trash Bin Garbage Collection

The current implementation of the asset eradication routine (`adria rm`) executes a synchronous, destructive purge across both the metadata engine and the distributed data nodes. While the client-side terminal mitigates accidental input via an interactive confirmation warning prompt, this layout risks irreversible data loss if authentication tokens are compromised or if an operator forces execution flags.

To enhance structural data safety, a **Soft-Deletion Framework** is proposed to replace immediate purging with an ephemeral trash bin retention layer. The architecture under this paradigm shifts across distinct execution phases:

1. **Metadata State Mutation (Logical Flagging):** When a tenant issues a delete directive, the master router intercepts the operation without emitting un-linking commands to the storage nodes. Instead, the entry status inside the SQL schema is updated with an active logical flag (`is_deleted = TRUE`) and assigned a high-precision epoch timestamp metadata attribute (`deleted_at`).

2. **Isolated Path Relocation:** The logical key reference of the file inside the object namespace is mutated, prepending a hidden system prefix (e.g., `/trash/tenant_id/...`). This action immediately evacuates the asset from the tenant's primary workspace directory tree, rendering it invisible during standard list operations while preserving chunk integrity.

3. **Asynchronous 30-Day Chronological Purging:** Rather than locking worker threads with immediate physical I/O operations, a lightweight background daemon process—orchestrated via a dedicated cron schedule or a Redis celery worker queue—is deployed on the control plane. This background task executes a daily database sweep using the following temporal condition constraint query:

$$\Delta t = t_{\text{current}} - t_{\text{deletion}} \geq 30 \text{ days}$$

4. **Deferred Physical Block Scavenging:** When an entry satisfies the 30-day age limit (Δt), the master marks the log state as permanently scrubbed. It then triggers an asynchronous, parallelized metadata cleanup routine across the worker nodes, dropping the physical 64 MB chunk segments from disk storage and reclaiming cluster storage capacity without blocking real-time user traffic.

This decoupled lifecycle approach balances operational resilience for the end-tenant with optimized, predictable storage reclamation intervals for the infrastructure cluster.

9 Self-Evaluation

This chapter contains the individual self-evaluation statements for each group member. In accordance with the academic guidelines, each section outlines individual responsibilities, objective role assessments, and a critical analysis of the product's engineering strengths and structural weaknesses from each developer's perspective.

9.1 Sajmir Buzi

Working on the design and development of AdriaBOX has been a deeply educational experience. Developing a system from scratch allowed me to face the real-world challenges of distributed systems, forcing me to turn the theoretical concepts learned in class into practical solutions. In particular, it was both challenging and rewarding to manage real memory constraints—such as the 4KB page limit for streaming—and to guarantee fault tolerance without compromising data consistency when nodes fail.

A fundamental strength of this project was the workflow I established with my colleague. We carefully planned the architecture right from the start, clearly defining how the different components would communicate with each other. This approach allowed us to write the code smoothly, preventing any major roadblocks or compatibility issues during integration.

We are so satisfied with the final result that we are considering looking beyond the academic scope: our idea for the future is to further engineer the product for potential commercialization. The first step in this direction will be developing a proper graphical user interface (GUI) to make AdriaBOX accessible and scalable even for non-technical users, turning a solid backend into a complete cloud service.

9.2 Massimiliano Pupillo

The experience of designing and deploying the AdriaBOX distributed architecture has represented a cornerstone in my academic and professional development. The most significant element of this project lies not only in the technical implementation of the distributed storage logic, but in the specific collaborative methodology established with my colleague. From the very inception of the infrastructure, our workflow was never partitioned in a dichotomous or fragmented manner through a rigid separation of modules; instead, we worked strictly side-by-side across every phase of the project, co-engineering everything from low-level network interactions to high-level storage persistence mappings.

Given our fundamentally different academic backgrounds—with Sajmir rooted in pure computer science and my trajectory anchored in electronic engineering—each of us approached architectural roadblocks from entirely distinct engineering perspectives. What truly formed and enriched my technical paradigm was this continuous exposure to and sharing of divergent problem-solving strategies. My perspective, naturally focused on physical hardware constraints, low-level virtual memory mapping configurations, and structural resource boundary optimization, integrated seamlessly with Sajmir's high-level software abstraction patterns and algorithmic reasoning. This mutual cross-contamination allowed us to discover innovative technical paths and analyze systemic efficiency at a level of granularity that neither of us could have reached in isolation.

The professional and personal synergy resulting from this collaboration has been exceptional. We are so satisfied with the computational resilience and core stability of the AdriaBOX backend that we are actively exploring market avenues for potential commercialization, with the shared goal of building a technological venture and engineering future systems together.

A Appendix A: Repository Sanitization and State Persistence Management

During the lifecycle of a decentralized, containerized platform, enforcing a strict boundary between version-controlled source assets and dynamic runtime states is paramount. This appendix documents the core architectural sanitization issues identified during the development of AdriaBOX, focusing on state leakage and environmental contamination. It also provides the production-grade `.gitignore` framework developed to guarantee deterministic deployments.

A.1 Architectural Contaminations and Dynamic Leakage Issues

Three major configuration vulnerabilities were identified during the transition from a manual bare-metal implementation to a multi-container isolated virtualization layer:

1. **Database State Contamination (The Double-Devil Paradox):** By default, an unconfigured version control environment tracking SQLite database binaries (`metadata.db`) inevitably serializes dynamic records (such as active token sessions and pre-registered tenants) directly into the public repository.

   When a developer pulls the repository onto an unconfigured workspace, local database initializations are bypassed. The system instead loads dirty, stale, or privileged administrative profiles (such as the `devil` or `max` user accounts created in previous validation cycles), leading to silent state pollution and privilege boundary failure.

2. **High-Volume Storage Volume Bloat:** Local volume mounts mapping isolated storage node partitions (`storage_data/nodeX/`) generate physical chunk fragments on the host system filesystem. For example, uploading a single 512 MB file splits the binary into 8 chunks of 64 MB each.

   Allowing Git to index these temporary segments results in immediate repository bloat, dramatically slowing down download speeds and introducing significant data plane pollution.

3. **Cryptographic Key Leakage:** Generating high-entropy JWT boundary secrets (such as the 256-bit key derived via OpenSSL) directly inside local files presents a severe threat model. If these environments are not explicitly isolated, private deployment secrets will be committed, causing immediate identity claims compromises across all system endpoints.

A.2 Deterministic Sanitization Framework: Gitignore Matrix

To systematically eliminate these vulnerabilities, a custom sanitization blueprint was implemented. It strictly enforces the exclusion of dynamic states while protecting persistent initial database frames (`.gitkeep`) and immutable documentation assets.

The complete, production-grade version-control configuration is presented below:

```
# ======================================================================
# AdriaBOX Official Gitignore Layout
# ======================================================================

# Python bytecode and runtime cache
__pycache__/
*.py[cod]
*$py.class
*.pyc
```

```
*.pyo
*.pyd

# Testing, code coverage, and quality-tool artifacts
.pytest_cache/
.coverage
.coverage.*
coverage.xml
htmlcov/
.mypy_cache/
.ruff_cache/
.pyre/
.tox/
.nox/

# Virtual environments and package driver isolation
.venv/
venv/
env/
ENV/
.poetry/
.cache/

# Project distribution and packaging build outputs
build/
dist/
*.egg-info/
*.egg

# Critical local security boundaries and environments
.env
.env.*
!.env.example
.adriabox_session
session.json

# Platform runtime log frameworks and process identifiers
*.log
*.pid
*.out
server.out.log
server.err.log
server.error.log

# Dynamic persistent databases and runtime storage engines
# Explicitly blocking database persistence to prevent state contamination
*.db
*.sqlite
*.sqlite3
data/
src/metadata_server/data/*
!src/metadata_server/data/.gitkeep

# Local containerized storage node volumes and block segments
storage_data/
storage_nodes/
src/storage_node/data/*
```

```
!src/storage_node/data/my_first_file.txt

# Volumetric file artifacts and high-volume test archives
*.zip
*.tar.gz
*.tgz
*.rar
*.iso
512MB.zip
file_example.avi

# System workspace layouts and local editor noise
.vscode/
.idea/
.DS_Store
Thumbs.db
*.swp
*.swo
.project
.settings/
```

This configuration preserves zero-trust security invariants across the entire development team. It ensures that any fresh checkout initializes on a clean, empty state, forcing the user registration and JWT authorization pipelines to build deterministically from scratch.

B Appendix B: Total System Teardown and Bare-Metal Reset Protocol

To guarantee absolute idempotency during iterative infrastructure evaluations, a structural mechanism to discard the existing deployment and revert the host machine to a pristine bare-metal state is mandatory. This appendix outlines the destructive teardown manual engineered to purge volatile runtime containers, network allocation sockets, persistent virtual volumes, and localized source trees from the host shell environment.

B.1 Destructive Sequence Invariants

Executing an un-ordered asset deletion under Docker introduces execution deadlocks due to filesystem layer locking and interface socket tracking dependencies. To achieve an instantaneous, clean reset without operating system fragmentation, the removal sequence must strictly adhere to the following technological dependency hierarchy:

1. **Container Layout Cessation:** All active processes running inside the 12 virtual storage units and the metadata master must be terminated to release read/write file handle locks on the volume mounts.
2. **Data Plane Volumetric Purging:** Physical data blocks (such as the 64 MB segmented binary chunks) and the centralized SQLite infrastructure database must be completely erased by pruning the persistent virtual storage hardware pools.
3. **Network Isolation De-allocation:** Virtual layer-2 bridge subnets (`172.18.0.0/16`) must be unmapped to release host networking subsystem constraints and avoid internal interface collisions upon subsequent restarts.
4. **Code Base Eviction:** Finally, the root repository workspace is purged from the user's home directory partition to destroy local state configurations and tracking errors.

B.2 Atomic Teardown Execution Script

The system operator executes the sequence on the host terminal using the atomic, step-by-step commands documented below:

```
# Step 1: Terminate the container grid and remove compiled runtime images
cd ~/AdriaBOX && sudo docker compose down -v --rmi all

# Step 2: Forcefully remove any orphan container processes remaining in the
    background
sudo docker rm -f $(sudo docker ps -aq) 2>/dev/null

# Step 3: Purge the persistent virtual storage volume layers to scrub database
    blocks
sudo docker volume prune -f

# Step 4: De-allocate the virtual bridge network interfaces and routing subnets
sudo docker network prune -f

# Step 5: Evict the root project repository directory and raw test assets from disk
cd ~ && sudo rm -rf ~/AdriaBOX
```

This teardown routing sequence completes the system automation framework. By wiping out all shared network layers and data blocks, it guarantees that any subsequent initialization

using the platform's automation framework will start from a reliable, pristine zero-state baseline.

References

References

[1] J. V. Guttag, *Introduction to Computation and Programming Using Python*, 2nd ed., Cambridge, MA, USA: The MIT Press, 2016.

[2] Z. A. Shaw, *Learn Python 3 the Hard Way: A Very Simple Introduction to the Terrifyingly Beautiful World of Computers and Code*, Boston, MA, USA: Addison-Wesley, 2017.

[3] J. Postel, "Transmission Control Protocol - DARPA Internet Program Protocol Specification," IETF, RFC 793, Sep. 1981. [Online]. Available: https://tools.ietf.org/html/rfc793

[4] IEEE and The Open Group, "The Open Group Base Specifications Issue 7 — POSIX.1-2008, Socket Interfaces," IEEE Std 1003.1-2008, 2018.

[5] Y. Nir and A. Langley, "ChaCha20 and Poly1305 for IETF Protocols," IETF, RFC 7539, May 2015. [Online]. Available: https://tools.ietf.org/html/rfc7539

[6] C. Percival, "Stronger Key Derivation via Sequential Memory-Hard Functions," in *Proceedings of BSDCan 2009 Conference*, May 2009. [Online]. Available: https://www.tarsnap.com/scrypt/scrypt.pdf

[7] Amazon Web Services, "Amazon Simple Storage Service (S3) Dynamic Architecture and Object Key Hierarchy," AWS Whitepapers, 2024. [Online]. Available: https://docs.aws.amazon.com/s3/

[8] M. Factor, K. Meth, A. Naor, O. Segall, and J. Satran, "Object storage: The future framework for scalable storage," *IBM Systems Journal*, vol. 44, no. 2, pp. 289–306, Apr. 2005.

[9] D. Ongaro and J. Ousterhout, "In search of an understandable consensus algorithm," in *Proceedings of the 2014 USENIX Conference on USENIX Annual Technical Conference (ATC'14)*, 2014, pp. 305–320.

[10] L. Lamport, "The part-time parliament," *ACM Transactions on Computer Systems (TOCS)*, vol. 16, no. 2, pp. 133–155, May 1998.

[11] S. Gilbert and N. Lynch, "Brewer's conjecture and the feasibility of consistent, available, partition-tolerant web services," *ACM SIGACT News*, vol. 33, no. 2, pp. 51–59, Jun. 2002.

[12] R. van Renesse and F. B. Schneider, "Chain replication for supporting high throughput and availability," in *Proceedings of the 6th Conference on Symposium on OSDI*, 2004, pp. 91–104.

[13] A. Silberschatz, P. B. Galvin, and G. Gagne, *Operating System Concepts*, 10th ed. Hoboken, NJ, USA: John Wiley & Sons, 2018. [Focus on Page-Size 4KB Architecture and IO Streaming Constraints].

[14] G. van Rossum and P. Eby, "Simple Generators," Python Enhancement Proposals, PEP 255, Jan. 2001. [Online]. Available: https://peps.python.org/pep-0255/

[15] S. Ghemawat, H. Gobioff, and S.-T. Leung, "The Google file system," in *Proceedings of the Nineteenth ACM Symposium on Operating Systems Principles (SOSP '03)*, 2003, pp. 29–43. [The Foundation of 64MB Chunk-Size Standardization].

[16] K. Shvachko, H. Kuang, S. Radia, and R. Chansler, "The Hadoop Distributed File System," in *Proceedings of the 2010 IEEE 26th Symposium on Mass Storage Systems and Technologies (MSST)*, 2010, pp. 1–10.

[17] D. Merkel, "Docker: lightweight Linux containers for consistent development and deployment," *Linux Journal*, vol. 2014, no. 239, Mar. 2014.

[18] Pallets Projects, "Flask Framework Documentation: WSGI Routing and Application Context Mechanics," 2023. [Online]. Available: https://flask.palletsprojects.com/