1. System Objective and Operational Context

The system models a *carrier-grade session orchestration platform* operating across edge, core, and cloud layers.

Its job is to establish, manage, and persist subscriber sessions while coordinating multiple independent services that each control critical resources. These services are autonomous, failure-prone, and communicate only through network calls.

The system must therefore guarantee:

- Atomicity across services
- Consistency of shared session state
- Correct behavior under partial failures
- Observability of all decisions

This context defines the requirements and the architectural process flow.

2. Request Ingress and Boundary Enforcement (Edge Layer)

Process Flow

1. A session creation request enters the system at the Edge Gateway.
2. The Edge Gateway:

- Parses and validates the request structure
- Normalizes request fields (subscriber ID, traffic class, protocol)
- Cryptographically signs the request payload using an HMAC secret

3. The signed request is forwarded to the Session Manager.

Why This Matters

The Edge Gateway establishes a hard trust boundary. No internal service accepts unsigned payloads, and signatures are verified before any state mutation occurs.

Requirements Satisfied

Secure inter-service communication
Byzantine fault rejection (invalid or tampered messages are discarded)
Clear separation between external and internal system trust domains

This prevents malformed or malicious requests from propagating into the core system.

3. Leader Election and Coordination Authority (Core Layer)

Process Flow

1. Multiple Session Manager instances may be running for availability.
2. Each instance competes for leadership using a Redis-based lock with TTL.
3. Exactly one instance becomes the leader at any time.
4. The leader periodically refreshes its lease.
5. Non-leader instances:

- Reject transaction coordination requests
- Return a retry hint to the caller

Why This Matters
Distributed transactions require a single coordinator. Without strict leadership, the system would risk split-brain coordination, double commits, or inconsistent decisions.

Requirements Satisfied
- Explicit coordination model
- Consistency under replication
- Prevention of split-brain behavior

Leadership ensures that all transaction decisions originate from one authoritative source.

4. Distributed Shared Memory (DSM) and Session State
Process Flow
1. Once the leader accepts a request:
- A session record is written to DSM with status `PENDING`
2. Each DSM write:

- Is versioned
- Uses compare-and-swap semantics
3. DSM reads and writes are used throughout the transaction lifecycle.
4.
Why This Matters
DSM acts as a durable coordination substrate shared across services and instances. Session state is never implicit or transient.

Requirements Satisfied
- Shared state consistency
- Race-condition prevention
- Crash recovery capability

If the coordinator crashes mid-transaction, the session state remains visible and recoverable.

5. Transaction Initialization and Persistence
Process Flow
1. A unique transaction ID and session ID are generated.
2. A minimal persistent record is written to a local database:
- Transaction ID
- Protocol (2PC or 3PC)
- Initial state
3. This happens before contacting participants.
Why This Matters

Persisting intent early ensures the system can:

- Reconstruct incomplete transactions
- Audit decisions after failures
- Avoid “phantom” sessions

Requirements Satisfied

- Durability
- Auditability
- Failure recovery support

6. Two-Phase Commit (2PC) Execution

Phase 1: Prepare

Process Flow

1. The coordinator sends `prepare` requests to:
   - Resource Manager
   - Billing Service
2. Each participant:
   - Validates availability
   - Locks required resources
   - Responds with a vote (yes or no)

Guarantees

No participant commits prematurely

Resources are tentatively reserved but not finalized

Phase 2: Commit or Abort

Process Flow

1. If all participants vote yes:
   - The coordinator issues commit commands
2. If any participant fails or times out:
   - The coordinator issues abort commands
3. All participants release or finalize resources accordingly.

Requirements Satisfied

- Atomicity across services
- No partial success
- Strong consistency

2PC ensures that the system never enters a partially committed state.

7. Three-Phase Commit (3PC) Execution

Process Flow

1. Prepare Phase: Same as 2PC
2. Pre-Commit Phase:
   - Participants acknowledge readiness
   - State becomes locally durable
3. Commit Phase:
   - ***** Final commit is issued

Why This Matters
3PC reduces blocking by allowing participants to make safe decisions even if the coordinator fails after pre-commit.

Requirements Satisfied
- Advanced fault tolerance
- Reduced coordinator dependency
- Demonstration of non-blocking commit protocol

This shows awareness of distributed systems limitations beyond basic 2PC.

8. Scheduling and Traffic-Aware Execution
Process Flow
1. Incoming session requests are placed into a scheduler queue.
2. Requests are assigned priorities based on traffic class:
   - Voice traffic receives higher priority than data traffic
3. Worker threads pull tasks based on priority ordering.
4. Queue delay is measured for observability.

Why This Matters
Telecom systems must enforce quality of service, not just correctness.

Requirements Satisfied
- Resource allocation fairness
- Priority-based scheduling
- Realistic traffic modeling

The system does not treat all traffic equally by design.

9. Fault Injection and Controlled Failure Handling
Process Flow
1. Fault parameters are dynamically configurable:
   - Artificial latency
   - Random aborts
2. Faults are injected at runtime without code changes.
3. During execution:
   - Timeouts are enforced
   - Failed participants cause transaction aborts
4. Abort paths:
   - Release all locks
   - Update DSM state
   - Persist abort outcome

Why This Matters
This validates that the system behaves correctly under stress, not just in the happy path.

Requirements Satisfied
- Fault tolerance
- Graceful rollback
- Failure isolation

Injected failures never corrupt shared state.

10. Deadlock Detection and Resolution

Process Flow

1. Multiple transactions attempt to acquire distributed locks.
2. Locks are acquired in conflicting orders.
3. A timeout-based mechanism detects lack of progress.
4. One transaction is aborted.
5. Locks are released.

Why This Matters

Deadlocks are inevitable in distributed resource allocation systems.

Requirements Satisfied

- Deadlock detection
- Deadlock resolution
- System liveness guarantees

The system continues making progress even under contention.

11. Event Emission and Ordering

Process Flow

1. On every commit or abort:
   - An event is emitted to a Redis stream
2. Events include:
   - Session ID
   - Transaction outcome
   - Timestamp
3. Streams preserve per-session ordering.

Why This Matters

This enables asynchronous consumers (analytics, monitoring) to reason about system behavior.

Requirements Satisfied

- Event-driven architecture
- Ordered event delivery
- Traceability of decisions

12. Observability and Metrics Collection

Process Flow

1. Each request records:
   - Latency
   - Status code
   - Endpoint
2. Each transaction records:
   - Protocol used
   - Commit or abort outcome
3. System-level metrics capture:
   - Leadership status
   - Deadlock counts
   - Scheduler queue delays

Why This Matters

Correctness must be provable, not assumed.

Requirements Satisfied

- Quantitative verification
- Performance analysis
- Operational transparency

13. End-to-End Consistency Guarantee

Final Process Summary

From ingress to completion:

- Every request is authenticated
- Every transaction is coordinated by a single leader
- Every state change is durable
- Every failure triggers rollback
- Every decision is observable

Overall Requirements Met

- Atomicity
- Consistency
- Fault tolerance
- Deadlock safety
- Observability