Redis "Everything Stream" Technical Gotchas - SOLVED

Concrete, implementable solutions for each identified failure mode.

---

1. SOLVED: Lock-Free Queue Implementation

Concrete Solution: Use LMAX Disruptor Pattern

cpp

```cpp
#include "disruptor/disruptor.h"

// Event structure with fixed size (no dynamic allocation)
struct TradingEvent {
    uint64_t sequence_id;
    uint64_t timestamp_ns;
    char correlation_id[64];
    char event_type[32];
    char data[512];  // Fixed size JSON payload
    std::atomic<bool> processed{false};
};

// Ring buffer with power-of-2 size
constexpr size_t RING_SIZE = 1024 * 1024;  // 1M events
disruptor::RingBuffer<TradingEvent, RING_SIZE> ring_buffer;

// TESTRADE publishes (single producer)
class TestradEventPublisher {
    disruptor::SequenceBarrier* barrier;

public:
    bool publish_event(const std::string& type, const nlohmann::json& data) {
        // Get next sequence (wait-free for single producer)
        long sequence = ring_buffer.next();

        // Get pre-allocated event slot
        TradingEvent& event = ring_buffer[sequence];

        // Populate (no allocation, just copy)
        event.sequence_id = sequence;
        event.timestamp_ns = rdtsc();  // CPU cycle counter
        strncpy(event.correlation_id, get_correlation_id().c_str(), 63);
        strncpy(event.event_type, type.c_str(), 31);

        std::string json_str = data.dump();
        strncpy(event.data, json_str.c_str(), 511);

        // Publish (memory barrier)
        ring_buffer.publish(sequence);

        return true;  // Never blocks
    }
};
```

```cpp
// Serialization consumers (multiple consumers)
class SerializationConsumer {
    std::string consumer_id;

public:
    void process_events() {
        long next_sequence = 0;

        while (running) {
            try {
                // Wait for available events
                long available = barrier->waitFor(next_sequence);

                // Process batch
                for (long seq = next_sequence; seq <= available; seq++) {
                    TradingEvent& event = ring_buffer[seq];

                    // Process if not already done by another consumer
                    bool expected = false;
                    if (event.processed.compare_exchange_strong(expected, true)) {
                        serialize_to_redis(event);
                    }
                }

                next_sequence = available + 1;

            } catch (const std::exception& e) {
                // Log but continue - never crash
                log_error("Serialization error: " + std::string(e.what()));
                std::this_thread::sleep_for(std::chrono::milliseconds(1));
            }
        }
    }
};
```

Why This Solves It:

- **No dynamic allocation:** Fixed ring buffer, pre-allocated
- **No ABA problems:** Sequence numbers prevent reuse issues
- **Cache-friendly:** Ring buffer has excellent locality
- **Proven in production:** LMAX processes 25M+ events/sec

---

2. SOLVED: Poison Pill Event Handling

Concrete Solution: Circuit Breaker + Dead Letter Queue

cpp

```cpp
class PoisonPillSafeSerializer {
    std::atomic<int> consecutive_failures{0};
    std::atomic<bool> circuit_open{false};
    std::chrono::steady_clock::time_point last_failure;

    // Dead letter file for manual inspection
    std::ofstream poison_log{"poison_pills.jsonl", std::ios::app};
    std::mutex poison_log_mutex;

public:
    bool serialize_to_redis(const TradingEvent& event) {
        // Circuit breaker check
        if (circuit_open.load()) {
            if (should_retry_circuit()) {
                circuit_open = false;
                consecutive_failures = 0;
            } else {
                return false;  // Skip during circuit open
            }
        }

        try {
            // Validate event structure first
            if (!validate_event(event)) {
                record_poison_pill(event, "validation_failed");
                return false;
            }

            // Parse JSON (this is where poison pills usually fail)
            nlohmann::json json_data;
            try {
                json_data = nlohmann::json::parse(event.data);
            } catch (const nlohmann::json::parse_error& e) {
                record_poison_pill(event, "json_parse_error: " + std::string(e.what()));
                return false;
            }

            // Build Redis event
            nlohmann::json redis_event;
            redis_event["metadata"]["eventId"] = generate_uuid();
            redis_event["metadata"]["correlationId"] = event.correlation_id;
            redis_event["metadata"]["timestamp"] = event.timestamp_ns;
            redis_event["metadata"]["eventType"] = event.event_type;
```

```
            redis_event["data"] = json_data;

            // Publish to Redis (with timeout)
            auto future = std::async(std::launch::async, [&]() {
                return redis_client.xadd("events", redis_event.dump());
            });

            if (future.wait_for(std::chrono::milliseconds(100)) == std::future_status::timeout)
                record_poison_pill(event, "redis_timeout");
                return false;
            }

            // Success - reset failure counter
            consecutive_failures = 0;
            return true;

        } catch (const std::exception& e) {
            record_poison_pill(event, "exception: " + std::string(e.what()));

            // Circuit breaker logic
            int failures = ++consecutive_failures;
            if (failures >= 10) {
                circuit_open = true;
                last_failure = std::chrono::steady_clock::now();
            }

            return false;
        }
    }

private:
    bool validate_event(const TradingEvent& event) {
        return event.timestamp_ns > 0 &&
               strlen(event.event_type) > 0 &&
               strlen(event.correlation_id) > 0 &&
               strlen(event.data) > 0;
    }

    void record_poison_pill(const TradingEvent& event, const std::string& error) {
        std::lock_guard<std::mutex> lock(poison_log_mutex);

        nlohmann::json poison_record;
        poison_record["timestamp"] = std::chrono::system_clock::now().time_since_epoch().count(
        poison_record["sequence_id"] = event.sequence_id;
```

```cpp
        poison_record["correlation_id"] = event.correlation_id;
        poison_record["event_type"] = event.event_type;
        poison_record["error"] = error;
        poison_record["raw_data"] = std::string(event.data);

        poison_log << poison_record.dump() << std::endl;
        poison_log.flush();

        // Also send to monitoring
        metrics.increment_counter("poison_pills_total", {{"error_type", error}});
    }

    bool should_retry_circuit() {
        auto now = std::chrono::steady_clock::now();
        return (now - last_failure) > std::chrono::seconds(30);
    }
};
```

Why This Solves It:

- **Never crashes:** All exceptions caught and logged
- **Circuit breaker:** Stops processing when too many failures
- **Dead letter queue:** All poison pills saved for manual analysis
- **Timeout protection:** Redis calls can't hang forever

---

3. SOLVED: Redis Consumer Group Restart

Concrete Solution: Checkpoint-Based Resume + Idempotency

cpp

```cpp
class RobustRedisConsumer {
    std::string group_name = "intellisense_group";
    std::string consumer_name;
    std::unordered_set<std::string> processed_correlations;
    std::string checkpoint_file;

public:
    RobustRedisConsumer(const std::string& name)
        : consumer_name(name)
        , checkpoint_file("checkpoint_" + name + ".dat") {
        load_checkpoint();
    }

    void start_consuming() {
        // First, claim any abandoned messages from failed consumers
        claim_abandoned_messages();

        // Then start normal consumption
        while (running) {
            try {
                auto messages = redis_client.xreadgroup(
                    "GROUP", group_name, consumer_name,
                    "STREAMS", "events", ">",  // Only new messages
                    "BLOCK", "1000",
                    "COUNT", "10"
                );

                for (const auto& msg : messages) {
                    if (process_message_safely(msg)) {
                        // Only ACK after successful processing
                        redis_client.xack("events", group_name, msg.id);
                        save_checkpoint(msg.id);
                    }
                }

            } catch (const std::exception& e) {
                log_error("Consumer error: " + std::string(e.what()));
                std::this_thread::sleep_for(std::chrono::seconds(1));
                // Continue - don't crash
            }
        }
    }
```

```cpp
private:
    bool process_message_safely(const RedisMessage& msg) {
        try {
            // Parse event
            nlohmann::json event = nlohmann::json::parse(msg.data);
            std::string correlation_id = event["metadata"]["correlationId"];

            // Idempotency check
            if (processed_correlations.count(correlation_id)) {
                return true;  // Already processed, but ACK anyway
            }

            // Process the event
            bool success = analyze_trading_event(event);

            if (success) {
                // Mark as processed
                processed_correlations.insert(correlation_id);

                // Clean old correlations (memory management)
                if (processed_correlations.size() > 100000) {
                    cleanup_old_correlations();
                }
            }

            return success;

        } catch (const std::exception& e) {
            log_error("Message processing error: " + std::string(e.what()));
            return false;  // Don't ACK failed messages
        }
    }

    void claim_abandoned_messages() {
        try {
            // Find messages idle for > 1 minute
            auto pending = redis_client.xpending("events", group_name, "-", "+", "10");

            for (const auto& entry : pending) {
                if (entry.idle_time > 60000) {  // 1 minute in ms
                    // Claim abandoned message
                    auto claimed = redis_client.xclaim("events", group_name, consumer_name,
                                                       "60000", entry.id);
```

```cpp
                    for (const auto& msg : claimed) {
                        if (process_message_safely(msg)) {
                            redis_client.xack("events", group_name, msg.id);
                        }
                    }
                }
            }
        } catch (const std::exception& e) {
            log_error("Claim abandoned messages failed: " + std::string(e.what()));
        }
    }

    void save_checkpoint(const std::string& message_id) {
        std::ofstream file(checkpoint_file);
        file << message_id << std::endl;

        // Also save processed correlations (for restart)
        for (const auto& correlation : processed_correlations) {
            file << correlation << std::endl;
        }
    }

    void load_checkpoint() {
        std::ifstream file(checkpoint_file);
        if (!file.is_open()) return;

        std::string line;
        bool first_line = true;

        while (std::getline(file, line)) {
            if (first_line) {
                // First line is last message ID
                first_line = false;
            } else {
                // Rest are processed correlations
                processed_correlations.insert(line);
            }
        }
    }
};
```

Why This Solves It:

- **Idempotency:** Uses correlation IDs to detect duplicates

- **Crash recovery:** Checkpoints state to disk
- **Abandoned message recovery:** Claims messages from failed consumers
- **Memory management:** Cleans up old correlation tracking

---

4. SOLVED: Redis Memory Management

Concrete Solution: Auto-Trimming + Memory Monitoring

```redis
# Redis configuration (redis.conf)
maxmemory 32gb
maxmemory-policy allkeys-lru

# Stream auto-trimming configuration
# Keep last 1M entries OR last 24 hours, whichever is smaller
stream-node-max-entries 100
stream-node-max-bytes 4096

# Memory optimization
hash-max-ziplist-entries 512
hash-max-ziplist-value 64
```

cpp

```cpp
class RedisMemoryManager {
    redis::client redis_client;
    std::chrono::seconds check_interval{30};

public:
    void start_monitoring() {
        std::thread([this]() {
            while (running) {
                try {
                    manage_memory();
                    std::this_thread::sleep_for(check_interval);
                } catch (const std::exception& e) {
                    log_error("Memory manager error: " + std::string(e.what()));
                }
            }
        }).detach();
    }

private:
    void manage_memory() {
        // Check memory usage
        auto memory_info = redis_client.info("memory");
        double memory_usage = get_memory_usage_percent(memory_info);

        if (memory_usage > 80.0) {
            // Aggressive trimming
            trim_streams_aggressively();
            alert("Redis memory usage high: " + std::to_string(memory_usage) + "%");
        } else if (memory_usage > 60.0) {
            // Normal trimming
            trim_streams_normally();
        }

        // Check fragmentation
        double fragmentation = get_fragmentation_ratio(memory_info);
        if (fragmentation > 1.5) {
            // Trigger memory defragmentation
            redis_client.memory_doctor();
        }

        // Clean up old consumer groups
        cleanup_inactive_consumer_groups();
    }
```

```cpp
    void trim_streams_aggressively() {
        // Keep only last 100K events
        redis_client.xtrim("events", "MAXLEN", "~", "100000");
    }

    void trim_streams_normally() {
        // Keep last 1M events
        redis_client.xtrim("events", "MAXLEN", "~", "1000000");
    }

    void cleanup_inactive_consumer_groups() {
        try {
            auto groups = redis_client.xinfo_groups("events");

            for (const auto& group : groups) {
                auto consumers = redis_client.xinfo_consumers("events", group.name);

                bool all_inactive = true;
                for (const auto& consumer : consumers) {
                    // If any consumer active in last hour, keep group
                    if (consumer.idle < 3600000) {  // 1 hour in ms
                        all_inactive = false;
                        break;
                    }
                }

                if (all_inactive && group.name != "intellisense_group") {
                    redis_client.xgroup_destroy("events", group.name);
                    log_info("Cleaned up inactive group: " + group.name);
                }
            }
        } catch (const std::exception& e) {
            log_error("Group cleanup failed: " + std::string(e.what()));
        }
    }
};
```

Why This Solves It:

- **Automatic trimming:** Prevents unbounded growth
- **Memory alerts:** Early warning when usage high
- **Fragmentation handling:** Triggers defragmentation when needed

- **Consumer group cleanup:** Removes inactive groups

---

5. SOLVED: Schema Versioning

Concrete Solution: Version Registry + Backwards Compatibility

cpp

```cpp
class EventSchemaRegistry {
    struct SchemaVersion {
        std::string version;
        std::function<nlohmann::json(const nlohmann::json&)> validator;
        std::function<nlohmann::json(const nlohmann::json&)> migrator;
    };

    std::map<std::string, std::vector<SchemaVersion>> schemas;

public:
    EventSchemaRegistry() {
        register_schemas();
    }

    bool validate_and_migrate(nlohmann::json& event) {
        std::string event_type = event["metadata"]["eventType"];
        std::string version = event["metadata"]["version"];

        auto schema_it = schemas.find(event_type);
        if (schema_it == schemas.end()) {
            log_error("Unknown event type: " + event_type);
            return false;
        }

        // Find version handler
        for (const auto& schema_version : schema_it->second) {
            if (schema_version.version == version) {
                // Validate
                if (!schema_version.validator(event)) {
                    return false;
                }

                // Migrate to latest version if needed
                if (version != get_latest_version(event_type)) {
                    event = migrate_to_latest(event, event_type);
                }

                return true;
            }
        }

        log_error("Unsupported version: " + event_type + " " + version);
        return false;
```

```cpp
    }

private:
    void register_schemas() {
        // OrderCreated v1
        schemas["OrderCreated"].push_back({
            "v1",
            [](const nlohmann::json& event) {
                return event["data"].contains("orderId") &&
                       event["data"].contains("symbol");
            },
            [](const nlohmann::json& event) { return event; }  // No migration needed
        });

        // OrderCreated v2 (added quantity field)
        schemas["OrderCreated"].push_back({
            "v2",
            [](const nlohmann::json& event) {
                return event["data"].contains("orderId") &&
                       event["data"].contains("symbol") &&
                       event["data"].contains("quantity");
            },
            [](const nlohmann::json& event) { return event; }  // Already v2
        });

        // Add migration from v1 to v2
        register_migration("OrderCreated", "v1", "v2", [](nlohmann::json event) {
            // Add default quantity if missing
            if (!event["data"].contains("quantity")) {
                event["data"]["quantity"] = 0;  // Default value
            }
            event["metadata"]["version"] = "v2";
            return event;
        });
    }

    std::function<nlohmann::json(const nlohmann::json&)> migration_v1_to_v2;

    void register_migration(const std::string& event_type,
                           const std::string& from_version,
                           const std::string& to_version,
                           std::function<nlohmann::json(nlohmann::json)> migrator) {
        // Store migration function for later use
        migration_v1_to_v2 = migrator;
```

```cpp
    }

    nlohmann::json migrate_to_latest(const nlohmann::json& event,
                                     const std::string& event_type) {
        std::string current_version = event["metadata"]["version"];
        nlohmann::json migrated_event = event;

        // Simple migration chain (v1 -> v2)
        if (current_version == "v1") {
            migrated_event = migration_v1_to_v2(migrated_event);
        }

        return migrated_event;
    }

    std::string get_latest_version(const std::string& event_type) {
        auto it = schemas.find(event_type);
        if (it != schemas.end() && !it->second.empty()) {
            return it->second.back().version;  // Last version is latest
        }
        return "v1";
    }
};
```

Why This Solves It:

- **Version validation:** Rejects unknown versions safely
- **Automatic migration:** Converts old events to new format
- **Backwards compatibility:** Supports multiple versions simultaneously
- **Extensible:** Easy to add new versions and migrations

Periodically clean up old consumer groups

XGROUP DESTROY events old_group_name

````
**Monitoring & Alerting:**
```bash
# Memory usage alerts
redis-cli INFO memory | grep used_memory_human

# Stream length monitoring
redis-cli XLEN events

# Consumer group lag (custom script)
redis-cli XINFO GROUPS events
````

Risk Mitigation:

- Configure aggressive stream trimming policies
- Monitor Redis memory usage and set alerts at 80%
- Implement automatic consumer group cleanup
- Use Redis clustering if single instance can't handle load

---

6. Schema Evolution and Versioning Gotchas

Problem: Event Schema Changes Break Consumers

Research Findings:

- Schema changes can create poison pills for older consumers
- "Support multiple versions of same event for transition periods"
- JSON flexibility vs. validation trade-offs
- Breaking changes require careful rollout

Specific Gotchas:

1. **Breaking schema changes**: New required fields break old consumers
2. **Version mismatches**: Consumers don't know how to handle new versions
3. **Rollback scenarios**: Need to support old schemas during rollbacks
4. **Performance impact**: Schema validation adds latency

Solutions:
````

cpp

```cpp
// Versioned event handling
class VersionedEventProcessor {
    std::map<std::string, std::function<bool(const nlohmann::json&)>> handlers;

    VersionedEventProcessor() {
        // Register handlers for different versions
        handlers["OrderCreated.v1"] = &process_order_created_v1;
        handlers["OrderCreated.v2"] = &process_order_created_v2;
    }

    bool process_event(const nlohmann::json& event) {
        std::string event_type = event["metadata"]["eventType"];
        std::string version = event["metadata"]["version"];
        std::string key = event_type + "." + version;

        auto handler = handlers.find(key);
        if (handler != handlers.end()) {
            return handler->second(event);
        } else {
            log_unknown_version(event_type, version);
            return false;  // Unknown version, skip safely
        }
    }
};

// Schema validation
class EventValidator {
    bool validate_event(const nlohmann::json& event) {
        // Required metadata fields
        if (!event.contains("metadata") ||
            !event["metadata"].contains("eventType") ||
            !event["metadata"].contains("version")) {
            return false;
        }

        // Version-specific validation
        std::string version = event["metadata"]["version"];
        if (version == "v1") {
            return validate_v1_schema(event);
        } else if (version == "v2") {
            return validate_v2_schema(event);
        }
```

```
            return false;  // Unknown version
        }
    };
```

Risk Mitigation:

- Always include version in event metadata
- Maintain backwards compatibility for at least 2 versions
- Test schema changes with poison pill scenarios
- Implement gradual rollout for schema changes

---

7. Operational and Monitoring Gotchas

Problem: Production Visibility and Debugging

Specific Gotchas:

1. **Invisible failures**: Events silently dropped without monitoring
2. **Performance degradation**: Gradual slowdown hard to detect
3. **Correlation debugging**: Hard to trace event chains through system
4. **Capacity planning**: Don't know when to scale

Solutions:

cpp

```cpp
// Comprehensive metrics collection
class RedisStreamMetrics {
    void record_event_published(const std::string& event_type) {
        increment_counter("events_published_total", {{"type", event_type}});
    }

    void record_serialization_time(const std::chrono::microseconds& duration) {
        record_histogram("serialization_duration_us", duration.count());
    }

    void record_queue_depth(size_t depth) {
        set_gauge("queue_depth", depth);
    }

    void record_poison_pill(const std::string& error_type) {
        increment_counter("poison_pills_total", {{"error", error_type}});
    }
};

// Redis health monitoring
class RedisHealthMonitor {
    void check_redis_health() {
        auto info = redis_client.info("memory");
        auto memory_usage = parse_memory_usage(info);

        if (memory_usage > 0.8) {
            alert("Redis memory usage high: " + std::to_string(memory_usage));
        }

        auto replication_lag = get_replication_lag();
        if (replication_lag > std::chrono::seconds(5)) {
            alert("Redis replication lag: " + std::to_string(replication_lag.count()));
        }
    }
};
```

Critical Alerts:

- Queue depth > 1M events (backpressure building)
- Poison pill rate > 1% (data quality issues)
- Redis memory usage > 80% (scale up needed)

- Consumer lag > 1 minute (processing issues)
- Serialization latency > 1ms (performance degradation)

Implementation Strategy: Risk-First Approach

Phase 0: Proof of Concept (2 weeks)

1. **Single event type** (OrderCreated only)
2. **Simple queue** (std::queue with mutex, no lock-free yet)
3. **Basic Redis publishing** (no consumer groups)
4. **Validate core concept** before complexity

Phase 1: Production Foundation (4 weeks)

1. **Lock-free queue implementation** with extensive testing
2. **Poison pill handling** with dead letter queue
3. **Consumer group setup** with restart logic
4. **Monitoring and alerting** infrastructure

Phase 2: Scale and Optimize (4 weeks)

1. **Multiple event types** with schema versioning
2. **Performance optimization** (latency, throughput)
3. **Operational runbooks** and debugging tools
4. **Load testing** with production volumes

Risk Mitigation Checklist

- [ ] Queue stress tested with 10M+ events
- [ ] Poison pill scenarios tested and handled
- [ ] Consumer restart logic tested
- [ ] Redis failover tested
- [ ] Schema evolution tested
- [ ] Monitoring dashboards created
- [ ] Alerting thresholds tuned
- [ ] Rollback procedures documented

Success Criteria:

- Zero event loss during normal operation

- <100μs p99 latency for event publishing
- Recovery from any single component failure in <30 seconds
- 99.9% uptime during trading hours