Case-Event-Relay Technical Report

Executive Summary

The Case-Event-Relay is a critical infrastructure service in Datadog's case management domain that implements the transactional outbox pattern to ensure reliable event delivery from PostgreSQL to Kafka. It acts as a bridge between the Case Management database and the event streaming platform, reading domain events from the `domain_event` table and publishing them to Kafka topics for consumption by downstream event handlers. The service uses an active-passive deployment model with distributed locking to prevent duplicate event delivery while maintaining high availability through automatic failover.

1 Service Overview

1.1 Primary Functions

1. **Event Relay**: Reads unread domain events from PostgreSQL and publishes them to Kafka topics
2. **Outbox Pattern Implementation**: Ensures reliable event delivery through transactional storage before Kafka publication
3. **Distributed Lock Management**: Maintains exclusive processing lock across multiple replicas using active-passive pattern
4. **Event Acknowledgment**: Tracks Kafka delivery confirmations and removes acknowledged events from the database
5. **Backlog Monitoring**: Continuously reports unread event counts for observability and alerting
6. **Corrupt Event Isolation**: Identifies and marks corrupt events to prevent pipeline blocking

1.2 Key Characteristics

Deployment Model

- Active-Passive: Single active pod processes events at any time
- Distributed Lock: Uses PostgreSQL table for coordination across replicas
- Automatic Failover: Lock acquisition timeout of 20 seconds for pod failure recovery
- Lock Heartbeat: 10-second interval to maintain ownership

Performance Configuration

- Polling Period: 250ms between database reads

- Batch Size: Up to 1000 events per read cycle
- Ack Batch Size: Up to 1000 events deleted per acknowledgment cycle
- Ack Cycle: 20ms tick for responsive acknowledgment processing
- Kafka Partitions: 32 partitions for parallel downstream processing

1.3 Key Metrics

- **Replicas**: 1 per datacenter (active-passive via lock)
- **Resource Allocation**: 1 CPU core, 1-2Gi memory
- **Throughput**: Handles thousands of events per second
- **Latency**: Typically sub-second from database write to Kafka delivery
- **Topics Produced**: domain-events (Case Management), domain-events-oncall (On-Call)
- **Event Types**: 60+ domain event types (case lifecycle, assignments, integrations, etc.)
- **Deployment Environments**: Staging, Production, Government (FIPS-compliant)

2 Architecture Overview

2.1 System Architecture

The case-event-relay service sits between the Case Management database and Kafka, implementing a reliable event relay pattern. It reads events from a transactional outbox table, publishes them to Kafka, and removes them only after delivery confirmation.

2.2 Component Interaction

The service consists of six primary components that work together in a coordinated pipeline:

LockController

- Manages distributed lock acquisition and maintenance
- Heartbeats every 10 seconds to maintain exclusive ownership
- Only allows one pod across all replicas to process events
- Enables active-passive high availability

Reader

- Queries `domain_event` table for unread events
- Filters: `read_on IS NULL`, `is_dirty IS FALSE`, lock ownership
- Reads up to 1000 events per batch
- Marks corrupt events as dirty to prevent blocking

Processor

- Orchestrates the read-produce cycle
- Polls every 250ms for new events
- Prevents concurrent processing
- Only runs when lock is owned

Producer

- Encodes events as protobuf with version prefix (v0)
- Partitions by aggregate_id (UUID) across 32 partitions
- Maintains trace context for distributed tracing
- Sends to Kafka asynchronously

Acknowledger

- Receives Kafka delivery confirmations
- Batch deletes up to 1000 acknowledged events
- Publishes end-to-end latency metrics
- Runs on 20ms tick cycle

MetricsGenerator

- Reports count of unread events every 10 seconds
- Only active on lock owner
- Provides backlog visibility

3 Event Flow Architecture

The event flow follows a producer-relay-consumer pattern with guaranteed delivery semantics.

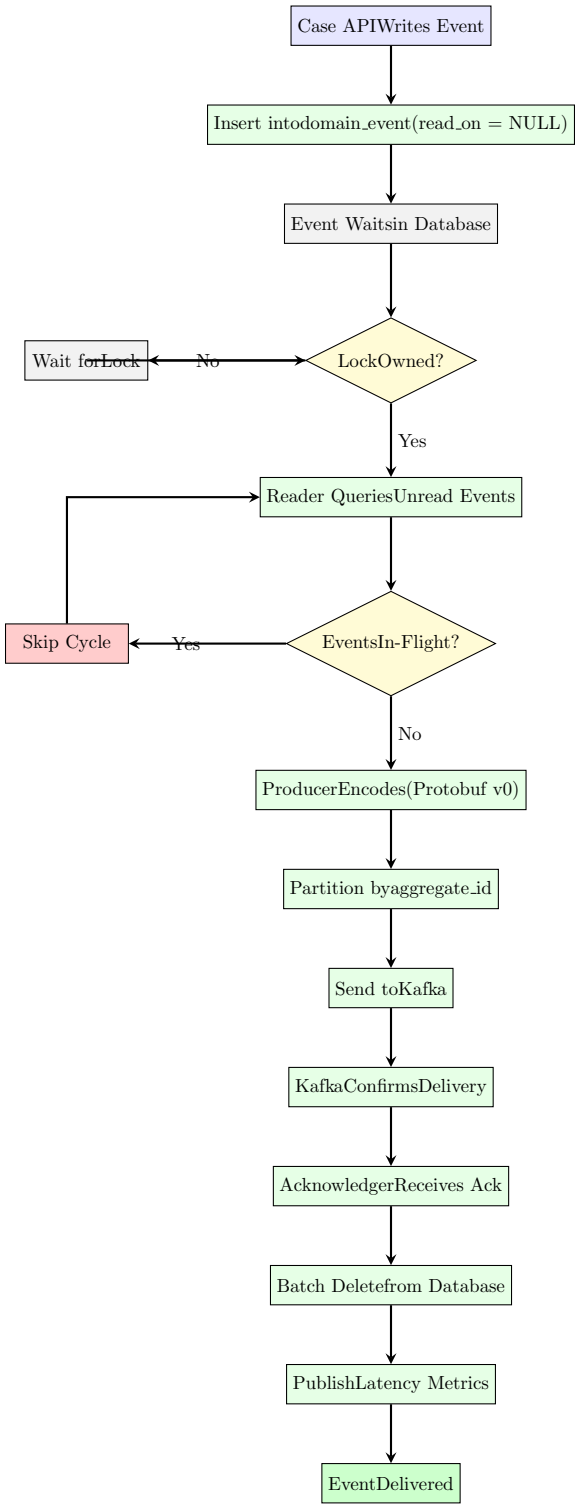

Figure 2: Event Processing Flow

4 Lock Management

The distributed lock mechanism ensures only one pod processes events at a time, preventing duplicate deliveries while enabling high availability.

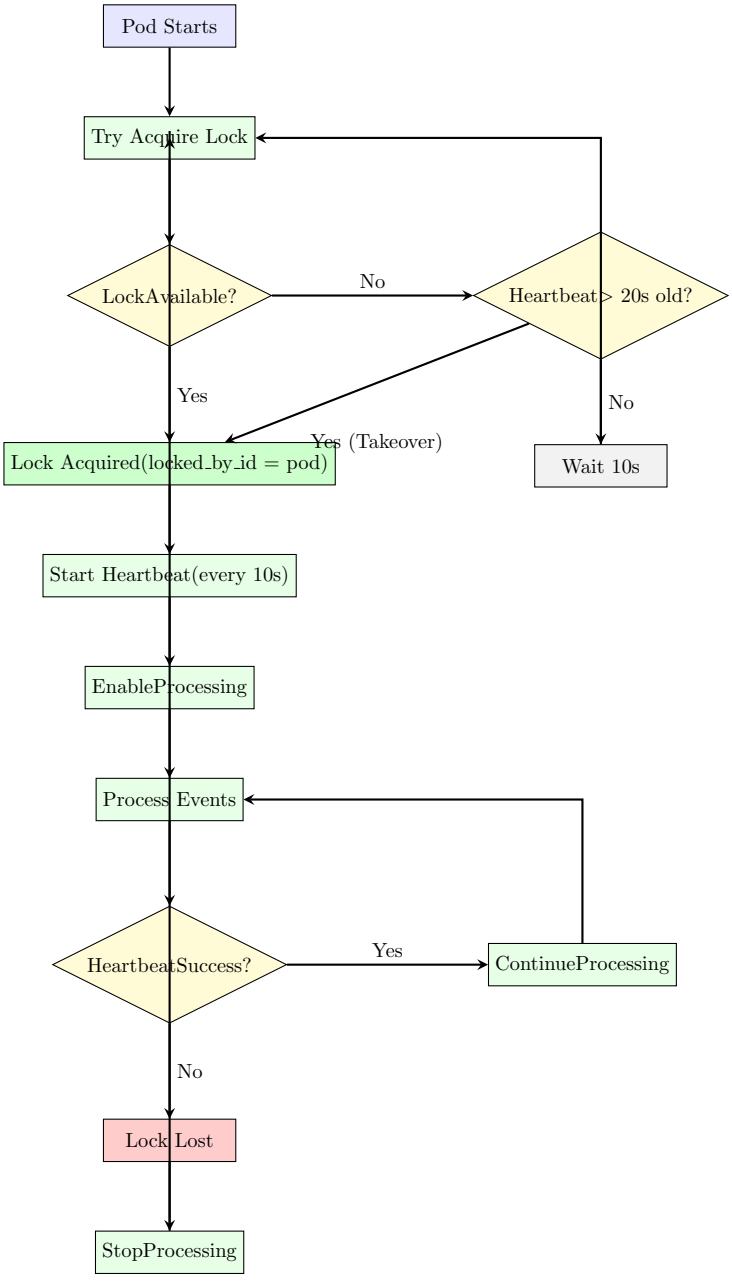

Figure 3: Lock Acquisition and Maintenance

4.1 Lock Behavior

Acquisition Conditions

- Lock table has no entry OR
- Existing lock's `latest_heartbeat_at` is older than 2x heartbeat period (20 seconds)
- Prevents split-brain: 2x period ensures stale lock before takeover

Heartbeat Mechanism

- Updates `latest_heartbeat_at` every 10 seconds
- Must verify current ownership (locked_by_id matches pod ID)
- Failed heartbeat immediately stops event processing
- Grace period: 20 seconds for failover

Failover Scenario

1. Active pod crashes or becomes unresponsive
2. Heartbeat stops updating
3. After 20 seconds, standby pod detects stale heartbeat
4. Standby pod acquires lock
5. New owner begins processing within 10 seconds (next heartbeat cycle)
6. Total failover time: 20-30 seconds

5 Database Integration

5.1 Schema Overview

domain_event Table

```
CREATE TABLE domain_event (
    id                  BIGSERIAL PRIMARY KEY,
    aggregate_id        UUID NOT NULL,
    event_type          TEXT NOT NULL,
    wire                BYTEA NOT NULL,
    read_on             TIMESTAMP,
    written_at          TIMESTAMP NOT NULL DEFAULT NOW(),
    is_dirty            BOOLEAN NOT NULL DEFAULT FALSE,
    org_id              BIGINT NOT NULL,
    aggregate_version   BIGINT NOT NULL
);

CREATE INDEX domain_event_scrapper_idx
ON domain_event (written_at DESC, aggregate_version DESC)
WHERE read_on IS NULL AND is_dirty IS FALSE;
```

Column Descriptions

- `id`: Unique event identifier (auto-increment)
- `aggregate_id`: Entity UUID (case, project, etc.) - used for Kafka partitioning
- `event_type`: Domain event type (CaseCreated, StatusChanged, etc.)
- `wire`: Protobuf-encoded event payload
- `read_on`: Timestamp when event was read (NULL = unread)
- `written_at`: Event creation timestamp
- `is_dirty`: Flag for corrupt/unprocessable events
- `org_id`: Organization identifier for multi-tenancy
- `aggregate_version`: Entity version for optimistic locking

event_relay_lock Table

```
CREATE TABLE event_relay_lock (
    latest_heartbeat_at  TIMESTAMP NOT NULL,
    locked_by_id         TEXT NOT NULL
);

-- Single row table (enforced by application logic)
```

Column Descriptions

- `latest_heartbeat_at`: Last successful heartbeat timestamp
- `locked_by_id`: Pod identifier of current lock owner

5.2 Query Patterns

Read Unread Events

```
SELECT id, aggregate_id, event_type, wire, org_id, written_at
FROM domain_event
WHERE read_on IS NULL
  AND is_dirty IS FALSE
  AND EXISTS (
      SELECT 1 FROM event_relay_lock
      WHERE locked_by_id = $1 -- current pod ID
        AND latest_heartbeat_at > NOW() - INTERVAL '20 seconds'
  )
ORDER BY written_at ASC, aggregate_version ASC
LIMIT 1000;
```

Batch Delete Acknowledged Events

```
DELETE FROM domain_event
WHERE id = ANY($1::BIGINT[]);
-- $1 is array of up to 1000 event IDs
```

Mark Corrupt Events as Dirty

```
UPDATE domain_event
SET is_dirty = TRUE
WHERE id = ANY($1::BIGINT[]);
```

Count Unread Events

```
SELECT COUNT(*)
FROM domain_event
WHERE read_on IS NULL AND is_dirty IS FALSE;
```

5.3 Performance Optimizations

- **Index**: Partial index on (written_at DESC, aggregate_version DESC) where unread and not dirty
- **Connection Pool**: 10 max connections (idle and open)
- **Batch Operations**: Deletes up to 1000 events per transaction
- **Lock Verification**: Embedded in read query for atomic ownership check
- **Dirty Flag**: Prevents repeated processing of corrupt events

6 Kafka Integration

6.1 Producer Configuration

```
[dd.kafka.producer]
topic = domain-events                # Case Management tenant
partition_count = 32                 # Number of partitions
linger_ms = 10                       # Batching delay
ack_timeout_ms = 100                 # Ack wait timeout
compression_codec = snappy           # Message compression
max_in_flight_requests = 5           # Parallel requests
idempotence = true                   # Exactly-once semantics
```

6.2 Partitioning Strategy

Events are partitioned by `aggregate_id` (UUID) to ensure ordering guarantees:

```
partition := int(aggregate_id.ID()) % partition_count
```

```
// Example:
// aggregate_id = "550e8400-e29b-41d4-a716-446655440000"
// partition_count = 32
// partition = hash(UUID) % 32 = 17
```

Benefits

- All events for a case/project go to the same partition
- Maintains event ordering per aggregate
- Enables parallel processing across partitions
- Simplifies downstream consumer state management

6.3 Message Format

Messages are protobuf-encoded with a version prefix:

```
// Wire format: [version_byte][protobuf_payload]
// version_byte = 0x00 (v0 encoding)

type KafkaMessage struct {
    Key       []byte  // aggregate_id as bytes
    Value     []byte  // version prefix + protobuf
    Headers   map[string]string {
        "trace.trace_id":  "12345...",
        "trace.span_id":   "67890...",
        "org_id":          "123456",
        "event_type":      "CaseCreated"
    }
}
```

6.4 Topics and Tenants

Table 1: Kafka Topics by Tenant

Topic	Tenant	Purpose
domain-events	case-management	Case Management domain events
domain-events-oncall	on-call	On-Call product domain events

6.5 Event Types

Over 60 domain event types are relayed through Kafka:

- CaseCreated
- CaseDeleted
- CaseArchived
- CaseUnarchived
- StatusChanged
- PriorityChanged
- TitleChanged
- DescriptionChanged
- CaseAssigned
- CaseUnassigned
- CommentAdded
- CommentUpdated
- CommentDeleted
- AttachmentCreated
- AttachmentDeleted
- JiraIssueAdded
- JiraIssueRemoved
- ServiceNowTicketAdded
- ServiceNowTicketRemoved
- IncidentLinked
- IncidentUnlinked
- AutomationRuleCreated
- AutomationRuleUpdated
- AutomationRuleEnabled
- AutomationRuleDisabled
- CellCreated
- LinkCreated
- LinkDeleted
- InsightsAdded
- InsightsRemoved
- ProjectUpdated
- ProjectDeleted

...and 28 more

7 Datacenter Deployment Strategy

7.1 Multi-Environment Deployment

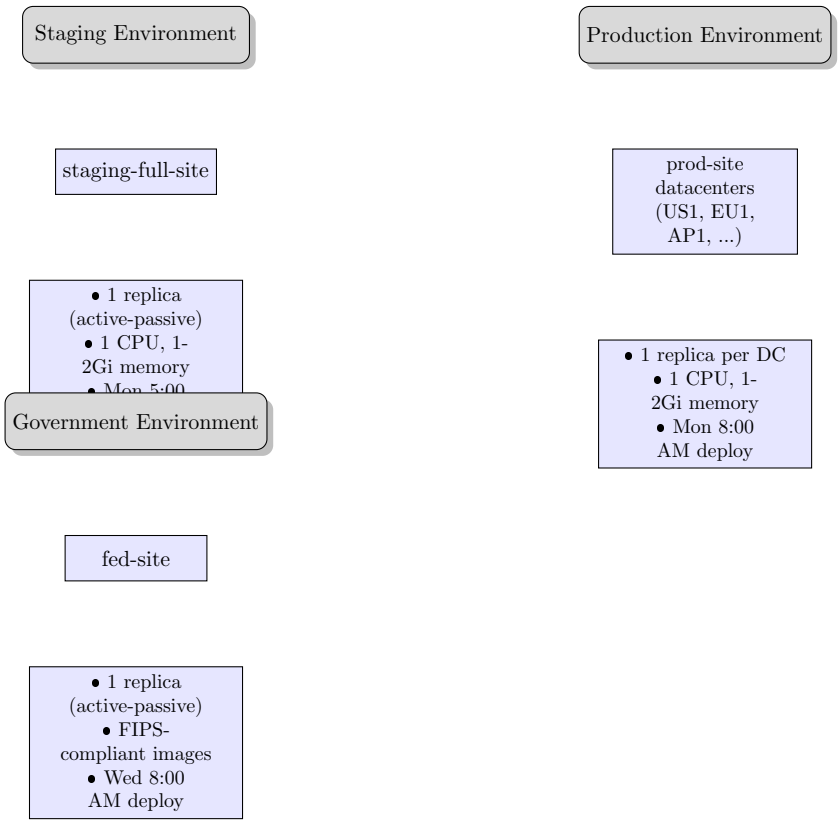

Figure 4: Deployment Strategy Across Environments

7.2 Deployment Configuration

Table 2: Deployment Settings by Environment

Setting	**Staging**	**Production**	**Government**
Datacenters	staging-full-site	prod-site (multiple)	fed-site
Replicas	1	1 per DC	1
Active Pods	1 (via lock)	1 per DC (via lock)	1 (via lock)
Deployment Strategy	RollingUpdate	RollingUpdate	RollingUpdate
Max Surge	1	1	1
Max Unavailable	0	0	0
CPU Request/Limit	1 / 1	1 / 1	1 / 1
Memory Request/Limit	1Gi / 2Gi	1Gi / 2Gi	1Gi / 2Gi
Image	caseeventrelay	caseeventrelay	caseeventrelayfips
Schedule	Mon 5:00 AM	Mon 8:00 AM	Wed 8:00 AM
Health Gates	Basic monitoring	20min health check	20min health check

7.3 High Availability Strategy

Active-Passive Model

- Multiple replicas deployed per datacenter

- Only one replica actively processes events (lock owner)
- Other replicas act as warm standbys
- Automatic failover within 20-30 seconds on pod failure
- No event duplication due to distributed lock

Zero-Downtime Deployments

- RollingUpdate strategy with maxSurge: 1, maxUnavailable: 0
- New pod starts and attempts lock acquisition
- Old pod maintains lock until termination
- Lock gracefully transfers during pod shutdown
- Minimal event processing interruption (sub-second)

Health Checks

- Liveness Probe: /liveness on port 8080 (15s period, 3 failure threshold)
- Readiness Probe: /readiness on port 8080 (10s period, 3 failure threshold)
- Termination Grace Period: 30 seconds for graceful shutdown

8 Performance and Observability

8.1 Key Metrics Collected

Event Processing Metrics

```
dd.case-event-relay.reads (count)
Tags: success:true/false
Description: Number of events read from database

dd.case-event-relay.acks (count)
Description: Number of events acknowledged and deleted

dd.case-event-relay.marked_dirty (count)
Description: Number of corrupt events marked as dirty

dd.case-event-relay.latency_ms (distribution)
Tags: kafka_partition
Description: End-to-end latency from written_at to Kafka ack
```

System Health Metrics

```
dd.case-event-relay.unread_events (gauge)
Description: Count of unread events in database
Purpose: Backlog monitoring and alerting

dd.case-event-relay.is_lock_owner (gauge)
Values: 0 or 1
Description: Lock ownership status
Purpose: Active pod identification
```

Database Metrics

```
dd.case.pg.query_duration (distribution)
Tags: query_name, success
Description: Database query latency

dd.case.pg.connection_pool.idle_connections (gauge)
dd.case.pg.connection_pool.open_connections (gauge)
Description: Connection pool health
```

8.2 Dashboards

Production Dashboard

- URL: `https://app.datadoghq.com/dashboard/cq2-3tg-f2t`
- Panels: Throughput, latency percentiles, unread events, lock ownership, error rates

Staging Dashboard

- URL: `https://ddstaging.datadoghq.com/dashboard/vbh-qu6-ydn`

8.3 Tracing

APM Integration

- Service: case-event-relay
- Trace Propagation: Maintains context from original API request through Kafka

- Key Spans:
  - `acknowledger.main_loop` - Ack processing
  - `process_events` - Main processing loop
  - `reader.read` - Database reads
  - `producer.produce` - Kafka production
  - `domain_events_repo.*` - Repository operations

8.4 Alerting

Critical Alerts

1. **High Backlog**
   - Metric: `dd.case-event-relay.unread_events`
   - Threshold: $> 10{,}000$ events for > 10 minutes
   - Indicates: Throughput issue or downstream Kafka problem
2. **No Lock Owner**
   - Metric: `dd.case-event-relay.is_lock_owner`
   - Threshold: Sum across all pods $= 0$ for > 2 minutes
   - Indicates: All pods unable to acquire lock (database issue)
3. **High Latency**
   - Metric: `dd.case-event-relay.latency_ms (p99)`
   - Threshold: > 5 seconds for > 5 minutes
   - Indicates: Database or Kafka performance degradation
4. **High Dirty Event Rate**
   - Metric: `dd.case-event-relay.marked_dirty`
   - Threshold: > 10 events per minute for > 5 minutes
   - Indicates: Data corruption or protobuf schema issues

9 Reliability Patterns

9.1 Transactional Outbox Pattern

The service implements the transactional outbox pattern to ensure reliable event delivery:

Benefits

- Events stored in same database transaction as domain changes
- Database ACID properties guarantee event persistence
- Events only deleted after Kafka confirms delivery
- No message loss on relay service failures
- At-least-once delivery semantics

Implementation

1. Case API writes event to `domain_event` table in same transaction as business logic
2. Event relay reads unread events
3. Event relay publishes to Kafka
4. Kafka confirms delivery
5. Event relay deletes event from database
6. If any step fails, event remains in database for retry

9.2 Active-Passive High Availability

Design Principles

- Single active processor prevents duplicate deliveries
- Multiple replicas provide failover capability
- Distributed lock coordinates across replicas
- Automatic failover on active pod failure
- No split-brain due to 2x heartbeat grace period

Failure Scenarios

Active Pod Crashes

1. Active pod stops heartbeating
2. After 20 seconds, standby detects stale heartbeat
3. Standby acquires lock
4. Standby begins processing
5. Unread events processed normally
6. Recovery time: 20-30 seconds

Database Connection Loss

1. Read/heartbeat operations fail
2. Lock ownership lost
3. Processing stops
4. Connection pool attempts reconnection
5. On reconnect, attempt lock reacquisition
6. Processing resumes

Kafka Producer Failure

1. Kafka client throws error
2. Events remain in database (not deleted)
3. Kafka client auto-reconnects
4. Next read cycle retries undelivered events
5. Events delivered with slightly higher latency
6. Potential duplicate deliveries (at-least-once semantics)

9.3 Corrupt Event Handling

Detection and Isolation

- Protobuf decode errors during read
- Invalid event structure or missing required fields
- Events marked with `is_dirty = TRUE`
- Dirty events excluded from future reads
- Prevents pipeline blocking on single corrupt event
- Manual investigation and remediation required

Monitoring

- Metric: `dd.case-event-relay.marked_dirty`
- Alert on sustained dirty event rate
- Dashboards show dirty event counts over time

9.4 Backpressure Management

In-Flight Event Protection

- Processor checks for in-flight events before reading
- Only one batch processed at a time
- Prevents overwhelming Kafka producer
- Ensures ordered delivery within partitions

- Simple but effective flow control

Batch Sizing

- Read batch: 1000 events max
- Ack batch: 1000 events max
- Tuned for throughput vs memory usage balance
- Prevents large transaction contention in PostgreSQL

10 Technical Implementation Details

10.1 Core Components

Main Entry Point (main.go)

```
func main() {
    // Initialize database connection
    db := initDB()
    repo := domainevent.NewRepository(db)

    // Initialize Kafka producer
    producer := initKafkaProducer()

    // Create components
    lockCtrl := NewLockController(repo)
    reader := NewReader(repo)
    kafkaProducer := NewProducer(producer)
    acknowledger := NewAcknowledger(repo)
    metricsGen := NewMetricsGenerator(repo)
    processor := NewProcessor(
        lockCtrl, reader, kafkaProducer, acknowledger
    )

    // Start goroutines
    go lockCtrl.Run(ctx)
    go processor.Run(ctx)
    go acknowledger.Run(ctx)
    go metricsGen.Run(ctx)

    // Health check server
    http.HandleFunc("/liveness", livenessHandler)
    http.HandleFunc("/readiness", readinessHandler)
    http.ListenAndServe(":8080", nil)
}
```

LockController (lock_controller.go)

```
type LockController struct {
    repo            *domainevent.Repository
    heartbeatPeriod time.Duration // 10 seconds
    podID           string
    isOwner         atomic.Bool
}

func (lc *LockController) Run(ctx context.Context) {
    ticker := time.NewTicker(lc.heartbeatPeriod)
    for {
        select {
        case <-ticker.C:
            if lc.isOwner.Load() {
                err := lc.repo.HeartBeatTheLock(lc.podID)
                if err != nil {
                    lc.isOwner.Store(false)
                }
            } else {
                err := lc.repo.TryToAcquireTheLock(lc.podID)
                if err == nil {
                    lc.isOwner.Store(true)
                }
```

```
            }
        case <-ctx.Done():
            return
        }
    }
}
```

Processor (processor.go)

```
type Processor struct {
    lockCtrl    *LockController
    reader      *Reader
    producer    *Producer
    pollingPeriod time.Duration // 250ms
    inFlight    atomic.Bool
}

func (p *Processor) Run(ctx context.Context) {
    ticker := time.NewTicker(p.pollingPeriod)
    for {
        select {
        case <-ticker.C:
            if !p.lockCtrl.IsOwner() {
                continue
            }
            if p.inFlight.Load() {
                continue
            }

            p.inFlight.Store(true)
            events, err := p.reader.Read(ctx)
            if err != nil {
                p.inFlight.Store(false)
                continue
            }

            for _, event := range events {
                p.producer.Produce(event)
            }

            // Note: inFlight cleared by acknowledger
        case <-ctx.Done():
            return
        }
    }
}
```

Acknowledger (acknowledger.go)

```
type Acknowledger struct {
    repo       *domainevent.Repository
    ackChan    chan *Ack
    batchSize  int           // 1000
    tickPeriod time.Duration // 20ms
}

func (a *Acknowledger) Run(ctx context.Context) {
    ticker := time.NewTicker(a.tickPeriod)
    batch := make([]*Ack, 0, a.batchSize)
```

```
    for {
        select {
        case ack := <-a.ackChan:
            batch = append(batch, ack)
            if len(batch) >= a.batchSize {
                a.processBatch(batch)
                batch = batch[:0]
            }
        case <-ticker.C:
            if len(batch) > 0 {
                a.processBatch(batch)
                batch = batch[:0]
            }
        case <-ctx.Done():
            return
        }
    }
}

func (a *Acknowledger) processBatch(batch []*Ack) {
    ids := extractIDs(batch)
    err := a.repo.BatchDelete(ids)
    if err == nil {
        for _, ack := range batch {
            latency := time.Since(ack.WrittenAt)
            metrics.Distribution(
                "latency_ms",
                float64(latency.Milliseconds()),
                "kafka_partition", ack.Partition,
            )
        }
    }
}
```

10.2 Concurrency Model

- **Goroutines**: 4 main goroutines (LockController, Processor, Acknowledger, MetricsGenerator)
- **Synchronization**: Atomic booleans for lock ownership and in-flight state
- **Channels**: Kafka ack channel for asynchronous acknowledgment
- **Context**: Used for graceful shutdown across all goroutines
- **Stopper Pattern**: Coordinates shutdown sequence

10.3 Dependencies

Key Go Dependencies

- `github.com/confluentinc/confluent-kafka-go/kafka` - Kafka client
- `github.com/lib/pq` - PostgreSQL driver
- `google.golang.org/protobuf` - Protobuf encoding
- `gopkg.in/DataDog/dd-trace-go.v1` - Distributed tracing
- `github.com/DataDog/datadog-go/statsd` - Metrics

11 Operational Considerations

11.1 Scalability

Current Model

- Single active processor per datacenter
- Throughput limited by single-threaded read-produce cycle
- Sufficient for current case management load (< 10k events/sec)
- Kafka partitioning enables downstream parallel processing

Scaling Options
Vertical Scaling

- Increase CPU/memory allocation
- Increase batch sizes (read and ack)
- Tune polling period for higher frequency
- Limited by single-pod processing model

Horizontal Scaling (Future)

- Partition `domain_event` table by org_id or hash
- Run multiple relay instances per partition
- Requires schema changes and migration
- Enables 10x+ throughput increase

11.2 Monitoring Best Practices

Daily Checks

- Monitor `unread_events` gauge (should be near zero)
- Check `latency_ms p99` (should be < 1 second)
- Verify `is_lock_owner` sum = 1 per datacenter
- Review dirty event counts (should be zero)

Incident Response
High Backlog

1. Check Kafka producer health (metrics, logs)
2. Verify lock ownership (multiple owners = split-brain)
3. Check database query latency
4. Review recent case-api deployment (event surge)
5. Consider temporary batch size increase

No Active Processor

1. Check all pods (kubectl get pods)
2. Review pod logs for lock acquisition failures
3. Verify database connectivity
4. Check `event_relay_lock` table state
5. Manual lock reset if needed (UPDATE event_relay_lock SET latest_heartbeat_at = '1970-01-01')

11.3 Maintenance Operations

Draining Events

```
-- Check unread event count
SELECT COUNT(*) FROM domain_event
WHERE read_on IS NULL AND is_dirty IS FALSE;

-- Monitor until count reaches zero
-- Safe to perform maintenance once drained
```

Cleaning Dirty Events

```
-- Identify dirty events
SELECT id, aggregate_id, event_type, written_at
FROM domain_event
WHERE is_dirty IS TRUE
ORDER BY written_at DESC
LIMIT 100;

-- After investigation and remediation
DELETE FROM domain_event WHERE is_dirty IS TRUE;
```

Manual Lock Reset

```
-- Emergency lock reset (if all pods deadlocked)
UPDATE event_relay_lock
SET latest_heartbeat_at = '1970-01-01␣00:00:00';

-- Pods will detect stale heartbeat and compete for lock
```

11.4 Service Catalog

- **Service**: case-event-relay
- **Team**: case-management
- **On-Call**: `https://app.datadoghq.com/on-call/teams/bc33b578-f8a9-11ed-a001-da7ad0900002`
- **Slack**: #case-management
- **Repository**: `https://github.com/DataDog/dd-source/tree/main/domains/case_management/apps/case-event-relay`
- **Documentation**: `https://datadoghq.atlassian.net/wiki/spaces/CM/pages/3294330939/Case+Event+Relay`

12 Future Considerations

12.1 Performance Enhancements

- **Table Partitioning**: Partition `domain_event` by time range or hash for horizontal scaling
- **Read Optimization**: Materialized view for unread events to avoid index scan
- **Parallel Processing**: Multiple relay instances with partition-based work distribution
- **Compression**: Enable Kafka message compression for bandwidth reduction
- **Batch Tuning**: Dynamic batch sizing based on backlog depth

12.2 Reliability Improvements

- **Dead Letter Queue**: Automatic routing of repeatedly failing events to DLQ
- **Circuit Breaker**: Kafka producer circuit breaker for graceful degradation
- **Health Scoring**: Composite health score for more nuanced monitoring
- **Auto-Recovery**: Automatic dirty event analysis and retry

12.3 Operational Enhancements

- **Self-Service Metrics**: Per-org and per-event-type throughput dashboards
- **Canary Deployments**: Gradual rollout with synthetic event validation
- **Chaos Testing**: Automated fault injection for resilience validation
- **Lock Observability**: Detailed lock contention and handoff metrics

13 Conclusion

The Case-Event-Relay service serves as the critical bridge between the Case Management database and Kafka event streaming platform. Its implementation of the transactional outbox pattern ensures reliable, at-least-once delivery of domain events to downstream consumers while maintaining high availability through an active-passive deployment model.

Key Architectural Strengths:

- **Reliability**: Transactional outbox pattern prevents event loss
- **High Availability**: Active-passive with automatic failover in 20-30 seconds
- **Ordering Guarantees**: Partition-by-aggregate ensures per-entity event ordering
- **Simplicity**: Single-threaded processing eliminates concurrency bugs
- **Observability**: Comprehensive metrics for latency, throughput, and backlog
- **Fault Isolation**: Dirty event marking prevents pipeline blocking

Operational Excellence:

- **Zero-Downtime Deployments**: RollingUpdate with graceful lock handoff
- **Predictable Failover**: Well-defined 20-second lock timeout

- **Comprehensive Monitoring**: Metrics, tracing, and dashboards for all scenarios
- **Simple Troubleshooting**: Clear error paths and manual intervention procedures

Integration Characteristics:

- **Database**: Direct PostgreSQL integration with connection pooling
- **Kafka**: OAuth-authenticated, TLS-encrypted, idempotent producer
- **Downstream**: Enables 10+ event handler flavors and third-party integrations
- **Tracing**: Full distributed tracing from API to consumer

The service's architecture prioritizes correctness and reliability over raw throughput. The active-passive model with distributed locking ensures exactly-one active processor, preventing duplicate event deliveries while enabling automatic failover on pod failures. The transactional outbox pattern guarantees no event loss, even during relay service failures. As Case Management scales, the architecture provides clear paths for horizontal scaling through table partitioning and parallel relay instances.

This report provides a comprehensive technical overview of the Case-Event-Relay service, its transactional outbox implementation, lock management strategy, and operational characteristics across Datadog's global infrastructure.

This report documents the Case-Event-Relay service as of January 2026. For the latest implementation details, refer to the source code at:

`domains/case_management/apps/case-event-relay`