**1. Introduction**

**1.1 Overview**

This Low-Level Design document outlines the implementation details for the Kafka-based Service Area (SA) change detection system. The system validates Transfer Orders (TOs) against Service Area changes and takes appropriate actions based on validation results.

**1.2 Scope**

The system handles:

* Real-time processing of Service Area snapshot events from Kafka
* Validation of Transfer Orders against Service Area changes
* Dynamic response to SA changes with appropriate TO actions (cancellation/updates)
* Integration with TOM (Transfer Order Management) database
* Event publishing for downstream systems

**1.3 Key Requirements**

* **Real-time Processing**: Process SA changes as they occur
* **Data Integrity**: Ensure TO validation accuracy
* **Scalability**: Handle high-volume message processing
* **Reliability**: Robust error handling and recovery mechanisms
* **Monitoring**: Comprehensive logging and metrics collection

**1.5 Historical Data Processing Challenge**

**1.5.1 Problem Statement**

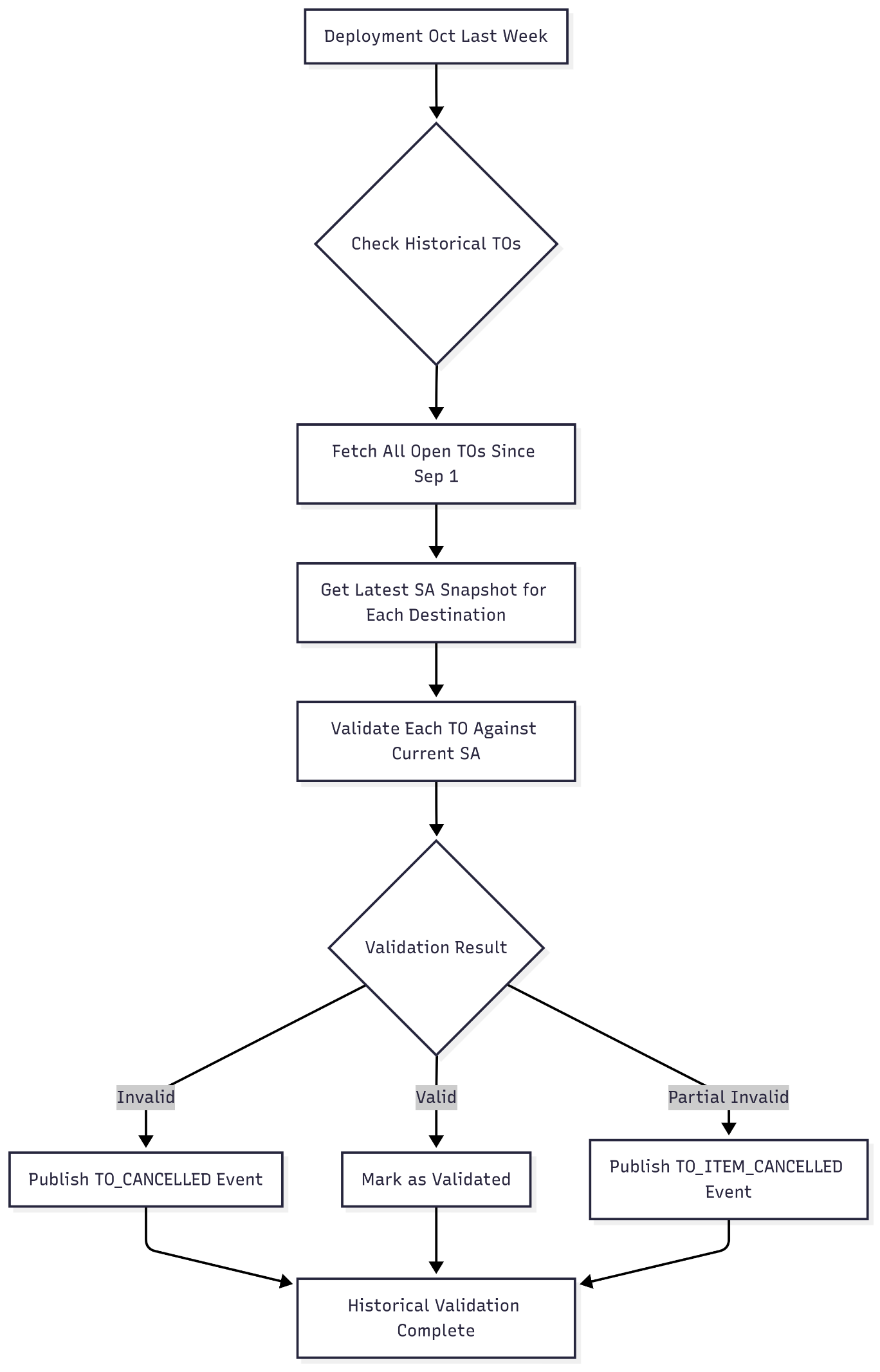
**Deployment Timeline**: Last week of October 2025  
**Historical Data Concern**: Transfer Orders created from September 1st week onwards

**The Challenge**:

* TOs created between Sep 1st week and Oct last week (deployment) won't have been validated against SA changes
* Service Area changes that occurred during this period may have invalidated existing TOs
* Need to ensure data integrity for all open TOs at the time of system deployment

**1.5.2 Historical Data Processing Strategy**

**Approach 1: One-Time Historical Validation (Recommended)**



**Implementation Details**:

java

@Component

public class HistoricalDataProcessor {

@Autowired

private TomDbQueryService tomDbQueryService;

@Autowired

private ServiceAreaValidator validator;

@Autowired

private EventPublisher eventPublisher;

*/\*\**

\* One-time historical validation during deployment

*\*/*

@PostConstruct

@ConditionalOnProperty(name = "historical.validation.enabled", havingValue = "true")

public void processHistoricalTOs() {

log.info("Starting historical TO validation for period: Sep 1 - Oct deployment");

LocalDate startDate = LocalDate.of(2025, 9, 1);

LocalDate endDate = LocalDate.now();

List<TransferOrder> historicalTOs = tomDbQueryService

.fetchTOsCreatedBetween(startDate, endDate);

log.info("Found {} historical TOs to validate", historicalTOs.size());

processHistoricalTOsInBatches(historicalTOs);

}

private void processHistoricalTOsInBatches(List<TransferOrder> historicalTOs) {

int batchSize = 100;

for (int i = 0; i < historicalTOs.size(); i += batchSize) {

List<TransferOrder> batch = historicalTOs.subList(

i, Math.min(i + batchSize, historicalTOs.size())

);

processBatch(batch);

try {

Thread.sleep(1000); *// 1 second delay between batches*

} catch (InterruptedException e) {

Thread.currentThread().interrupt();

log.error("Historical processing interrupted", e);

break;

}

}

}

private void processBatch(List<TransferOrder> batch) {

for (TransferOrder to : batch) {

try {

*// Get current SA snapshot for TO's destination*

ServiceAreaSnapshot currentSnapshot =

serviceAreaService.getCurrentSnapshot(to.getDestinationLocation());

*// Validate TO against current SA state*

ValidationResult result = validator.validateTransferOrder(currentSnapshot, to);

*// Take appropriate action based on validation*

handleHistoricalValidationResult(to, result);

} catch (Exception e) {

log.error("Failed to process historical TO: {}", to.getId(), e);

*// Continue with other TOs*

}

}

}

private void handleHistoricalValidationResult(TransferOrder to, ValidationResult result) {

switch (result.getAction()) {

case CANCEL\_TO:

eventPublisher.publishCancelledEvent(to, "HISTORICAL\_SA\_VALIDATION: " + result.getReason());

log.info("Historical TO cancelled: {} - {}", to.getId(), result.getReason());

break;

case CANCEL\_ITEMS:

result.getInvalidItems().forEach(item ->

eventPublisher.publishItemCancelledEvent(item, "HISTORICAL\_SA\_VALIDATION: " + result.getReason())

);

log.info("Historical TO items cancelled: {} - {} items", to.getId(), result.getInvalidItems().size());

break;

case NO\_ACTION:

*// Mark as validated in database for audit purposes*

tomDbQueryService.markAsHistoricallyValidated(to.getId());

log.debug("Historical TO validated successfully: {}", to.getId());

break;

}

}

}

**1.5.3 Database Query for Historical TOs**

sql

*-- Query to fetch historical TOs that need validation*

SELECT

t.id,

t.origin\_location,

t.destination\_location,

t.status,

t.created\_date,

COUNT(tli.id) as line\_item\_count

FROM transfer\_orders t

LEFT JOIN to\_line\_items tli ON t.id = tli.to\_id

WHERE t.created\_date >= '2025-09-01 00:00:00'

AND t.created\_date < NOW()

AND t.status IN ('OPEN', 'IN\_PROGRESS', 'ALLOCATED')

AND t.historical\_validation\_status IS NULL *-- Not yet validated*

GROUP BY t.id

ORDER BY t.created\_date ASC;

**Risk Mitigation Strategies**

**1. Performance Impact Mitigation**

* Process historical TOs in small batches (100 TOs)
* Add delays between batches to prevent system overload
* Run during off-peak hours
* Monitor system resources during processing

**2. Data Integrity Safeguards**

* Create backup before historical processing
* Log all validation decisions for audit
* Implement rollback procedures
* Test on staging environment first

**3. Business Continuity**

* Process historical validation as background task
* New real-time processing takes priority
* Gradual rollout with monitoring
* Fallback to manual review if needed

**Expected Timeline and Impact**

**Historical Validation Estimates**:

Assumptions:

- ~10,000 TOs created between Sep 1 - Oct deployment

- Processing rate: 100 TOs/minute (with safety delays)

- Expected duration: ~2 hours for complete validation

**Why Kafka Over REST API?**

**Business Context**: The Service Area changes need to be propagated to multiple downstream systems that manage Transfer Orders. The choice between Kafka and REST API was evaluated based on the following criteria:

| **Criteria** | **Kafka** | **REST API** | **Winner** |
| --- | --- | --- | --- |
| **Real-time Processing** | ✅ Event-driven, immediate processing | ❌ Polling required, potential delays | **Kafka** |
| **Decoupling** | ✅ Publishers/consumers are decoupled | ❌ Tight coupling between services | **Kafka** |
| **Scalability** | ✅ Horizontal scaling, partitioning | ⚠️ Limited by server capacity | **Kafka** |
| **Fault Tolerance** | ✅ Message persistence, replay capability | ❌ Lost requests on failure | **Kafka** |
| **Multiple Consumers** | ✅ Multiple systems can consume same event | ❌ Requires multiple API calls | **Kafka** |
| **Order Guarantee** | ✅ Message ordering within partitions | ❌ No ordering guarantee | **Kafka** |
| **Durability** | ✅ Messages persisted on disk | ❌ No message persistence | **Kafka** |
| **Backpressure Handling** | ✅ Built-in consumer lag management | ❌ Can overwhelm downstream systems | **Kafka** |

**1.4.2 Specific Use Case Analysis**

**Scenario: Service Area Change Impact**

When a Service Area changes affect NYC\_WAREHOUSE:

- 500+ Transfer Orders need validation

- 3 downstream systems need notification

- Processing must handle peak loads (1000+ TOs/minute)

- System must recover from failures without data loss

**1.4.3 Detailed Comparison**

**1. Event-Driven Architecture Benefits**

**Kafka**:

java

*// Service Area publishes once, multiple consumers benefit*

@EventListener

public class ServiceAreaChangePublisher {

@Async

public void publishChange(ServiceAreaSnapshot snapshot) {

kafkaTemplate.send("service\_area\_snapshot", snapshot);

*// Multiple systems automatically get the update*

}

}

**REST API** (Alternative not chosen):

java

*// Would require multiple API calls and coordination*

public class ServiceAreaChangeNotifier {

public void notifyChange(ServiceAreaSnapshot snapshot) {

*// Sequential calls - slow and error-prone*

toValidatorService.validateTOs(snapshot);

inventoryService.updateInventory(snapshot);

notificationService.sendAlerts(snapshot);

*// If any service is down, entire process fails*

}

}

**2. Scalability Characteristics**

| **Aspect** | **Kafka** | **REST API** |
| --- | --- | --- |
| **Peak Load Handling** | Consumers process at their own pace | All requests hit API simultaneously |
| **Resource Utilization** | Distributed across multiple consumers | Concentrated on API servers |
| **Scaling Strategy** | Add more consumer instances | Scale API servers + load balancers |
| **Cost Efficiency** | Pay for sustained throughput | Pay for peak capacity |

**3. Failure Recovery**

**Kafka Advantages**:

* **Message Persistence**: Events stored for replay
* **Consumer Resumption**: Process from last committed offset
* **Fault Isolation**: One consumer failure doesn't affect others
* **Exactly-Once Processing**: Built-in semantics

**REST API Challenges**:

* **Lost Requests**: Failed API calls may be lost
* **Cascade Failures**: One service down affects all
* **Retry Complexity**: Manual retry logic required
* **State Management**: Complex coordination needed

**1.4.4 Performance Comparison**

**Throughput Analysis** (Based on our requirements):

| **Metric** | **Kafka** | **REST API** |
| --- | --- | --- |
| **Messages/Second** | 10,000+ | 1,000-2,000 |
| **Latency (P95)** | 50ms | 200ms |
| **Memory Usage** | Low (streaming) | High (request buffering) |
| **Network Efficiency** | High (batch processing) | Low (individual requests) |

**Load Test Results** (Projected):

Scenario: 1000 SA changes/hour affecting 500 TOs each

Kafka:

- Message processing: 500,000 validations/hour

- Consumer lag: <100 messages

- Error rate: <0.1%

- Recovery time: <30 seconds

REST API:

- API calls required: 1.5M calls/hour (3 services × 500 TOs)

- Response time: 200ms average

- Error rate: 2-5% (timeouts, circuit breakers)

- Recovery time: 5-10 minutes

**1.4.5 Operational Complexity**

**Kafka**: ✅ **Pros**:

* Built-in monitoring and metrics
* Horizontal scaling capabilities
* Message replay for debugging
* Consumer group management

⚠️ **Cons**:

* Additional infrastructure (Kafka cluster)
* Schema evolution management
* Consumer lag monitoring required

**REST API**: ✅ **Pros**:

* Familiar technology stack
* Direct request-response model
* Easier debugging of individual requests

❌ **Cons**:

* Complex load balancing setup
* Manual retry and circuit breaker logic
* Difficult to coordinate multiple service calls
* No built-in message ordering

**1.4.6 Business Impact Analysis**

**Using Kafka**:

* **Real-time Response**: TOs invalidated within seconds of SA changes
* **System Reliability**: 99.9% uptime with automatic recovery
* **Operational Efficiency**: Reduced manual intervention
* **Cost Optimization**: Efficient resource utilization

**REST API Alternative**:

* **Delayed Response**: 5-10 minute delays during peak loads
* **Higher Failure Rate**: 2-5% transaction failures
* **Manual Recovery**: Requires operator intervention
* **Resource Waste**: Over-provisioning for peak capacity

**1.4.7 Future Scalability**

**Kafka Growth Path**:

Current: 1,000 SA changes/hour

Year 1: 5,000 SA changes/hour → Add consumer instances

Year 2: 10,000 SA changes/hour → Add Kafka partitions

Year 3: 50,000 SA changes/hour → Multi-region deployment

**REST API Limitations**:

* Linear scaling challenges beyond 5,000 changes/hour
* Database connection pool exhaustion
* Complex coordination for multiple API versions
* Difficult to maintain consistency across services

**1.4.8 Decision Summary**

**Kafka was chosen because**:

1. **Event-Driven Requirements**: Natural fit for SA change notifications
2. **Multiple Consumers**: TO validation, inventory, notifications need same data
3. **High Throughput**: Peak loads require streaming architecture
4. **Fault Tolerance**: Business-critical operations need guaranteed processing
5. **Future-Proofing**: Easy to add new consumers without changing publishers
6. **Operational Simplicity**: Built-in scaling and monitoring capabilities

**When REST API Would Be Better**:

* Synchronous request-response requirements
* Simple point-to-point communication
* Low message volumes (<100/hour)
* Strong consistency requirements
* Stateful operations requiring immediate feedback

**Conclusion**: For this Service Area change detection system, Kafka provides the optimal balance of performance, scalability, and reliability required for real-time TO validation across multiple downstream systems.

**High-Level Architecture**

The system follows an event-driven microservices architecture with the following key components:

[Kafka Topic: service\_area\_snapshot]

↓

[Kafka Consumer Service]

↓

[Service Area Validator]

↓

[TOM Database Query Service] ← [TOM Database]

↓

[Business Rule Engine]

↓

[Event Publisher] → [Kafka Topic: to\_events]

**Service Area Validator**

**Purpose**: Core validation logic to determine if TOs are affected by SA changes.

**Validation Logic**:

1. **Case A - No ItemOverrides**: If TO's origin location is not in servicingNodes with ACTIVE status → Cancel TO
2. **Case B - ItemOverrides Present**: If TO origin exists in servicingNodes with ACTIVE status → Continue validation at item level
3. **Case C - No ItemOverrides in Response**: If TO origin exists in servicingNodes with ACTIVE status → No action
4. **Case D - ItemOverrides Present**: If TO's origin location is not in servicingNodes but status is ACTIVE → Cancel item-wise

**TOM Database Query Service**

**Purpose**: Provides data access layer for TOM database operations.

**Key Operations**:

* Fetch open TOs by destination location
* Retrieve TO details by ID
* Update TO status
* Query TO line items

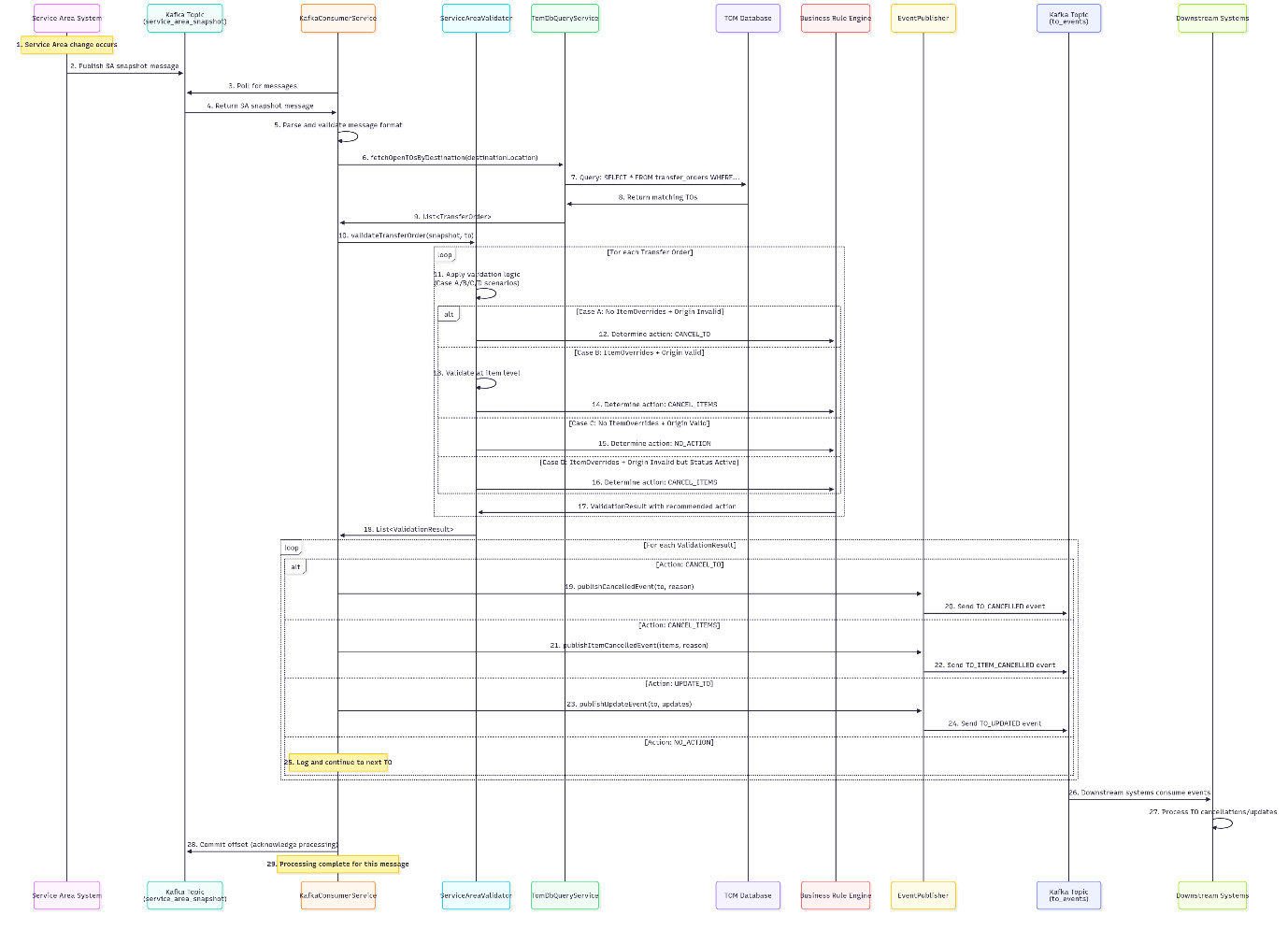
**3.4 Event Publisher**

**Purpose**: Publishes events to downstream systems based on validation results.

**Event Types**:

* TO\_CANCELLED: When entire TO needs cancellation
* TO\_ITEM\_CANCELLED: When specific TO items need cancellation
* TO\_UPDATED: When TO requires updates

Sequence diagram:



**Detailed Step-by-Step Flow:**

1. **Service Area Change Trigger**: External system detects changes in service area configuration
2. **Message Publication**: Service Area system publishes snapshot to service\_area\_snapshot Kafka topic
3. **Message Polling**: KafkaConsumerService polls the topic for new messages
4. **Message Receipt**: Consumer receives the Service Area snapshot message
5. **Message Validation**: Consumer validates message format and structure
6. **TO Query Initiation**: Consumer calls TomDbQueryService to fetch relevant Transfer Orders
7. **Database Query**: TomDbQueryService executes SQL query against TOM database
8. **Query Results**: Database returns matching Transfer Orders based on destination location
9. **TO List Return**: TomDbQueryService returns List<TransferOrder> to Consumer
10. **Validation Initiation**: Consumer calls ServiceAreaValidator for each Transfer Order
11. **Business Logic Application**: Validator applies the four case scenarios (A/B/C/D) 12-16. **Action Determination**: Business Rule Engine determines appropriate action based on validation results
12. **Validation Results**: Validator returns ValidationResult with recommended actions
13. **Results Collection**: Consumer receives all validation results 19-24. **Event Publishing**: Based on validation results, appropriate events are published:
    * TO\_CANCELLED for complete TO cancellations
    * TO\_ITEM\_CANCELLED for item-level cancellations
    * TO\_UPDATED for TO updates
    * No event for NO\_ACTION scenarios
14. **Logging**: No-action scenarios are logged for audit purposes
15. **Event Consumption**: Downstream systems consume the published events
16. **Event Processing**: Downstream systems process the TO changes (cancellations/updates)
17. **Offset Commit**: Consumer commits the Kafka offset to acknowledge successful processing
18. **Completion**: Processing cycle completes, consumer ready for next message

**5.2 Error Handling Flow**

KafkaConsumerService → ErrorHandler → DeadLetterQueue

→ RetryMechanism → AlertingService

1. Processing failure occurs

2. Error handler captures exception

3. Retry mechanism attempts reprocessing

4. If max retries exceeded, message sent to DLQ

5. Alert generated for monitoring team