Real-Time Truck Cycle Data Pipeline - Business Logic Documentation

1. Executive Summary

This document outlines the business logic for a real-time data pipeline service that ingests GPS data from cycle trucks, processes it to track operational cycles, and stores results in PostgreSQL. The system uses Python3 with amazon-kclpy 3.0.1 for Kinesis data consumption and works with existing asset and dump region data.

Key Objectives

- Track complete truck operational cycles from loading to dumping
- Detect outlier cycles when trucks dump outside designated regions
- Provide real-time insights into mining operation efficiency
- Use only available data sources without complex configuration

---

2. System Architecture Overview

2.1 Data Sources

- **Primary Input**: Kinesis Data Stream (real-time GPS data from cycle trucks)
- **Loader Data**: PostgreSQL asset table (loader unit locations)
- **Dump Region Data**: Redis (dump region polygon definitions)
- **Output Storage**: PostgreSQL asset_cycle_vlx table (cycle tracking records)

2.2 Core Components

- **Data Ingestion Service**: amazon-kclpy 3.0.1 for Kinesis consumption
- **Location Analysis Engine**: Distance calculation and polygon containment logic
- **Cycle State Manager**: Segment transition detection and cycle completion tracking

---

3. Business Logic Flow - Detailed Steps

3.1 Data Ingestion and Reference Data Loading

Step 1-2: GPS Data Consumption

Input Data Structure:

- `asset_guid`: Unique identifier for the cycle truck
- `timestamp`: UTC+0 timestamp
- `longitude`: GPS longitude coordinate
- `latitude`: GPS latitude coordinate
- `speed`: Vehicle speed
- `site_guid`: Site identifier for filtering reference data

Process:

- amazon-kclpy 3.0.1 consumes individual records from Kinesis Data Stream
- Parse and validate basic data structure integrity

Step 3: Site-Specific Reference Data Loading

PostgreSQL Query for Loaders:

```sql
SELECT asset_guid, latitude, longitude, site_guid
FROM asset
WHERE site_guid = ? AND asset_type = 'LOADER'
```

Redis Query for Dump Regions:

```
Key: dump_regions:{site_guid}
Value: JSON array containing region_guid, region_name, site_guid,
region_location
```

Purpose:

- Load all loaders for the site to calculate distances
- Load all dump regions for the site to check containment
- Enable comparison against ALL reference locations

3.2 Data Validation

Step 4: Basic Data Validation

- Validate GPS coordinate ranges (latitude: -90 to 90, longitude: -180 to 180)
- Verify timestamp format and chronological order
- Confirm asset_guid exists in asset table for the specified site
- Validate site_guid format and existence in system
- Filter out duplicate timestamps for the same asset
- Reject obviously invalid speeds (>150 km/h) to filter GPS device errors

3.3 Historical Data Analysis

Step 5: Asset History Check

Database Query:

```sql
SELECT * FROM asset_cycle_vlx
WHERE cycle_truck_asset_guid = ?
ORDER BY current_timestamp DESC
LIMIT 1
```

Decision Logic:

- **No Historical Data**: Apply initial data classification
- **INPROGRESS Record**: Compare with current GPS for transitions
- **COMPLETE/OUTLIER Record**: Start new cycle based on current location

Step 6-7: Initial Data Classification (First-Time Assets)

For assets with no historical records:

1. **Calculate distance to ALL loaders** in the site
2. **Check containment in ALL dump regions** using Shapely Point.within()
3. **Classification Logic:**
    - Within 50m of loader → `current_segment = 'LOAD_TIME'`
    - Within dump region → `current_segment = 'DUMP_TIME'`
    - Neither condition → `current_segment = NULL` (wait for next GPS point)

3.4 Location Analysis and Segment Classification

Step 8: Comprehensive Location Checking

Critical Rule: For every GPS point, check against ALL loaders AND ALL dump regions in the site.

Nearest Loader Selection Algorithm:

```python
def find_nearest_loader(truck_lat, truck_lon, site_loaders):
    nearest_loader = None
    min_distance = float('inf')

    for loader in site_loaders:
        distance = haversine_distance(truck_lat, truck_lon,
loader.latitude, loader.longitude)
        if distance <= 50 and distance < min_distance:
            min_distance = distance
            nearest_loader = loader

    return nearest_loader, min_distance
```

Active Dump Region Detection Algorithm:

```python
def find_active_dump_region(truck_lat, truck_lon,
site_dump_regions):
    truck_point = Point(truck_lon, truck_lat)

    for region in site_dump_regions:
        polygon = Polygon(region.region_location)
        if truck_point.within(polygon):
            return region

    return None
```

3.5 Segment Definitions and Transitions

Segment Classification Rules

🟢 **Load Time Segment**

- **Condition**: Distance ≤ 50m from nearest loader unit
- **Database Updates**: Record nearest_loader_asset_guid, set load_start_utc if transitioning from Empty Travel

🟡 **Load Travel Segment**

- **Condition**: Distance > 50m from ALL loader units AND previously was in Load Time
- **Database Updates**: Set load_end_utc, calculate load_duration

🔵 **Dump Time Segment**

- **Condition**: GPS point within any dump region polygon
- **Database Updates**: Record active_dump_region_guid, set dump_start_utc if transitioning from Load Travel

🟣 **Empty Travel Segment**

- **Condition**: Exited specific dump region AND previously was in Dump Time
- **Database Updates**: Set dump_end_utc, calculate dump_duration

❓ **NULL Segment**

- **Condition**: First-time asset NOT near loader AND NOT in dump region
- **Purpose**: Placeholder until asset moves to determinable location

3.6 Segment Transition Logic

Step 9: Basic Segment Transition Logic

- Compare previous record's segment with current GPS point's calculated segment
- **Critical Rule**: ALWAYS check current GPS position against ALL dump regions and ALL loaders, regardless of historical patterns
- **No Assumptions**: Never assume which dump region or loader a truck will use based on previous cycles

Normal Transitions:

- Load Time → Load Travel → Dump Time → Empty Travel → Load Time

Outlier Detection:

- **Trigger**: Load Travel → Load Time (skipping Dump Time and Empty Travel)
- **Interpretation**: Truck dumped outside designated dump regions
- **Action**: Mark cycle as OUTLIER status

Fixed Processing Parameters (applied to all sites):

- **Loader proximity threshold**: 50 meters
- **Cycle timeout**: 8 hours (mark as incomplete after this time)

3.7 Cycle Completion Detection

Step 10: Complete Cycle vs Outlier Detection

Normal Cycle Completion:

- **Trigger**: Empty Travel → Load Time transition
- **Action**: Mark `cycle_status = 'COMPLETE'`
- **Calculations**: All durations calculated, total_cycle_duration set

Outlier Cycle Detection:

- **Trigger**: Load Travel → Load Time transition (skipping Dump Time and Empty Travel)
- **Interpretation**: Truck dumped outside designated dump regions
- **Action**: Mark `cycle_status = 'OUTLIER'`
- **Database Impact**: Dump-related fields set to NULL

Step 11: Cycle Finalization and New Cycle Initialization

For Both Complete and Outlier Cycles:

1. Mark previous cycle with appropriate status
2. Calculate final durations and timestamps
3. Insert new record with incremented cycle_number
4. Initialize new cycle with INPROGRESS status

---

4. Database Schema

4.1 Primary Table: asset_cycle_vlx

```sql
CREATE TABLE asset_cycle_vlx (
    -- Primary Key and Cycle Identification
    id SERIAL PRIMARY KEY,
    cycle_truck_asset_guid VARCHAR(255) NOT NULL,
    cycle_number INTEGER NOT NULL,
```

```
    cycle_status VARCHAR(20) NOT NULL, -- 'COMPLETE',
'INPROGRESS', 'OUTLIER', 'INCOMPLETE'
    site_guid VARCHAR(255) NOT NULL,

    -- Current GPS Data Point Information
    current_timestamp TIMESTAMP NOT NULL,
    current_longitude FLOAT NOT NULL,
    current_latitude FLOAT NOT NULL,
    current_speed FLOAT,
    current_segment VARCHAR(20), -- 'LOAD_TIME', 'LOAD_TRAVEL',
'DUMP_TIME', 'EMPTY_TRAVEL', NULL

    -- Nearest Loader Information (from asset table)
    nearest_loader_asset_guid VARCHAR(255),
    nearest_loader_location VARCHAR(255), -- '(lat, lon)'
    nearest_loader_distance FLOAT, -- Distance in meters

    -- Active Dump Region Information (from Redis)
    active_dump_region_guid VARCHAR(255),
    active_dump_region_name VARCHAR(255),
    active_dump_region_location TEXT, -- '[(lat,lon)...]'

    -- Cycle Timing Information
    load_start_utc TIMESTAMP,
    load_end_utc TIMESTAMP,
    dump_start_utc TIMESTAMP,
    dump_end_utc TIMESTAMP,

    -- Duration Calculations (in seconds)
    load_travel_duration INTEGER,
    empty_travel_duration INTEGER,
    dump_duration INTEGER,
    load_duration INTEGER,
    total_cycle_duration INTEGER,

    -- Audit Fields
```

```
    created_date TIMESTAMP DEFAULT CURRENT_TIMESTAMP,
    updated_date TIMESTAMP DEFAULT CURRENT_TIMESTAMP
);
```

4.2 Reference Table: asset

```sql
CREATE TABLE asset (
    asset_guid VARCHAR(255) PRIMARY KEY,
    asset_type VARCHAR(50), -- 'LOADER', 'TRUCK', etc.
    latitude FLOAT,
    longitude FLOAT,
    site_guid VARCHAR(255)
);
```

4.3 Performance Indexes

```sql
-- Primary operational indexes
CREATE INDEX idx_asset_cycle_vlx_site_asset_status
ON asset_cycle_vlx(site_guid, cycle_truck_asset_guid,
cycle_status);

CREATE INDEX idx_asset_cycle_vlx_active_cycles
ON asset_cycle_vlx(cycle_status, current_timestamp)
WHERE cycle_status = 'INPROGRESS';

-- Asset reference indexes
CREATE INDEX idx_asset_site_type
ON asset(site_guid, asset_type);
```

---

5. Error Handling and Edge Cases

Basic Data Issues

- **Invalid GPS Data**: Handle coordinates outside valid ranges, reject speeds >150 km/h
- **Temporal Issues**: Manage timestamp duplicates or out-of-sequence data
- **Missing References**: Handle cases where asset_guid or site_guid don't exist in tables

Data Stream Interruptions

Python
```python
def handle_data_stream_interruption(asset_guid,
last_known_record, current_gps):
    """
    Handle cases where GPS data stream is interrupted
    Simple recovery logic based on gap duration
    """
    gap_duration = current_gps.timestamp -
last_known_record.current_timestamp

    if gap_duration > timedelta(hours=4):
        # Large gap - mark previous cycle as incomplete and start
fresh
        mark_cycle_incomplete(last_known_record,
reason="DATA_STREAM_INTERRUPTION")
        return classify_initial_segment(current_gps)

    # Continue normal processing for smaller gaps
    return None
```

Incomplete Cycles

- Handle cases where assets go offline mid-cycle
- Implement timeout mechanisms for abandoned cycles (8 hours standard timeout)
- Mark incomplete cycles with appropriate status and reason codes

Python
```python
def check_cycle_timeouts():
    """
    Check for cycles that have been in progress too long
    """
    timeout_threshold = datetime.utcnow() - timedelta(hours=8)
```

```
stale_cycles = query_database("""
    SELECT * FROM asset_cycle_vlx
    WHERE cycle_status = 'INPROGRESS'
    AND current_timestamp < ?
""", [timeout_threshold])

for cycle in stale_cycles:
    mark_cycle_incomplete(cycle, reason="TIMEOUT")
```

Geographic Edge Cases

- Handle GPS points near loader/dump region boundaries
- Use standard 50m threshold for all loaders
- Standard polygon containment for dump regions

Concurrent Processing

- Ensure thread-safe operations for multiple assets
- Handle race conditions in database updates with basic retry logic
- Process assets sequentially to maintain state consistency

```python
def update_cycle_with_retry(cycle_id, updates, max_retries=3):
    """
    Update cycle record with basic retry logic
    """
    for attempt in range(max_retries):
        try:
            result = update_database_record(cycle_id, updates)
            if result:
                return result
        except Exception as e:
            if attempt == max_retries - 1:
                raise
            time.sleep(0.1)  # Brief pause before retry
```

```
    raise Exception("Failed to update after maximum retries")
```

---

6. Performance Considerations

Database Optimization

- Use appropriate indexes for fast INPROGRESS record lookup
- Implement connection pooling for database operations
- Consider partitioning for large datasets by site_guid and date

SQL
```sql
-- Essential indexes for performance
CREATE INDEX idx_asset_cycle_vlx_site_asset_status
ON asset_cycle_vlx(site_guid, cycle_truck_asset_guid,
cycle_status);

CREATE INDEX idx_asset_cycle_vlx_active_cycles
ON asset_cycle_vlx(cycle_status, current_timestamp)
WHERE cycle_status = 'INPROGRESS';
```

Memory Management

- **Basic Reference Caching**: Cache loader and dump region data to minimize database/Redis queries
- Monitor memory usage for high-throughput scenarios

Python
```python
class SimpleCache:
    def __init__(self, ttl_seconds=300):  # 5 minute cache
        self.cache = {}
        self.timestamps = {}
        self.ttl = ttl_seconds
```

```
    def get_loaders(self, site_guid):
        """Get loaders for site with basic caching"""
        key = f"loaders_{site_guid}"

        # Check if cache is valid
        if (key in self.timestamps and
            time.time() - self.timestamps[key] < self.ttl):
            return self.cache[key]

        # Reload from database
        loaders = query_database(
            "SELECT * FROM asset WHERE site_guid = ? AND
asset_type = 'LOADER'",
            [site_guid]
        )

        self.cache[key] = loaders
        self.timestamps[key] = time.time()
        return loaders
```

Processing Optimization

- **Sequential Asset Processing**: Process GPS points for each asset in timestamp order
- **Basic Connection Pooling**: Reuse database connections efficiently
- **Simple Error Handling**: Log errors and continue processing

---

7. Monitoring and Alerting

Basic Logging System

Python

```python
import logging
import json
from datetime import datetime
```

```python
def setup_basic_logging():
    """Configure simple file-based logging"""
    logging.basicConfig(
        level=logging.INFO,
        format='%(asctime)s - %(levelname)s - %(message)s',
        handlers=[

logging.FileHandler('/var/log/truck-cycle/processor.log'),
            logging.StreamHandler()
        ]
    )

def log_cycle_event(event_type, asset_guid, details):
    """Log cycle events in JSON format"""
    log_entry = {
        'timestamp': datetime.utcnow().isoformat(),
        'event_type': event_type,
        'asset_guid': asset_guid,
        'site_guid': details.get('site_guid'),
        'cycle_number': details.get('cycle_number'),
        'previous_segment': details.get('previous_segment'),
        'current_segment': details.get('current_segment')
    }

    logging.info(json.dumps(log_entry))
```

Essential Monitoring

Python
```python
def check_stuck_cycles():
    """Find cycles that have been in progress too long"""
    timeout_threshold = datetime.utcnow() - timedelta(hours=8)

    stuck_cycles = query_database("""
```

```
        SELECT cycle_truck_asset_guid, cycle_number,
current_timestamp
        FROM asset_cycle_vlx
        WHERE cycle_status = 'INPROGRESS'
        AND current_timestamp < ?
    """, [timeout_threshold])

    if stuck_cycles:
        alert_message = f"Found {len(stuck_cycles)} stuck cycles"
        logging.warning(alert_message)
        # Send alert to monitoring system

    return stuck_cycles

def get_basic_metrics():
    """Get basic operational metrics"""
    metrics = query_database("""
        SELECT
            COUNT(*) as total_cycles,
            SUM(CASE WHEN cycle_status = 'COMPLETE' THEN 1 ELSE 0
END) as complete_cycles,
            SUM(CASE WHEN cycle_status = 'OUTLIER' THEN 1 ELSE 0
END) as outlier_cycles,
            SUM(CASE WHEN cycle_status = 'INCOMPLETE' THEN 1 ELSE
0 END) as incomplete_cycles
        FROM asset_cycle_vlx
        WHERE created_date >= NOW() - INTERVAL '24 hours'
    """)

    return metrics
```

Simple Alert Conditions

- **Stuck Cycles**: Assets with cycles >8 hours INPROGRESS
- **High Error Rate**: Database connection failures or processing errors
- **System Health**: Basic uptime and availability monitoring

8. Business Rules Summary

Core Processing Rules

1. **One GPS point = One database operation** in asset_cycle_vlx (INSERT or UPDATE)
2. **Check historical data first**: Query for any previous records for the asset (not just INPROGRESS)
3. **Initial data handling**: For first-time assets, use NULL segment if location is indeterminate
4. **ALWAYS check ALL locations**: For every GPS point, check against ALL loaders and ALL dump regions in the site
5. **No historical assumptions**: Never assume which dump region or loader a truck will use based on previous cycles
6. **Fixed business parameters**: Use 50m loader proximity and 8-hour cycle timeout for all sites
7. **Basic data filtering**: Reject speeds >150 km/h and invalid coordinates
8. **Nearest loader selection**: Calculate distances to ALL loaders → Select nearest ≤ 50m
9. **Active dump region detection**: Check GPS point against ALL dump regions → Identify any containing region
10. **Store only selected**: Record only the nearest loader and active dump region in database
11. **Calculate durations**: Only when segment transitions occur
12. **Mark COMPLETE**: When full sequence (Load Time → Load Travel → Dump Time → Empty Travel) is detected
13. **Mark OUTLIER**: When Load Travel → Load Time transition occurs (skipping Dump Time and Empty Travel)
14. **Mark INCOMPLETE**: When cycle timeout (8 hours) is exceeded
15. **Immediate new cycle**: Start new cycle immediately after completion or outlier detection

Cycle Status Determination

Status Definitions:

- **COMPLETE**: Full sequence completed (Load Time → Load Travel → Dump Time → Empty Travel → Load Time)
- **OUTLIER**: Incomplete sequence (Load Travel → Load Time, skipping dump regions)
- **INPROGRESS**: Currently active cycle
- **INCOMPLETE**: Cycle abandoned due to timeout or data stream loss

State Transition Rules

Valid Transitions:

- Load Time → Load Travel (truck moves >50m from loader)
- Load Travel → Dump Time (truck enters dump region)
- Load Travel → Load Time (outlier: dumped outside regions)
- Dump Time → Empty Travel (truck exits dump region)
- Empty Travel → Load Time (truck approaches loader, completes cycle)

---

9. Implementation Considerations

9.1 Technology Stack Requirements

- **Python 3.8+** with amazon-kclpy 3.0.1
- **PostgreSQL 12+** for main data storage (asset table and asset_cycle_vlx table)
- **Redis 6+** for dump region data storage
- **Shapely 2.1.1** for geometric calculations (polygon containment)

9.2 Available Data Sources

- **PostgreSQL asset table**: Contains loader units with asset_guid, latitude, longitude, site_guid
- **Redis dump regions**: Contains dump region definitions with region_guid, region_name, site_guid, region_location (4-point polygons)
- **Kinesis GPS stream**: Real-time truck data with asset_guid, timestamp, latitude, longitude, speed, site_guid

9.3 Fixed Business Parameters

- **Loader proximity threshold**: 50 meters (works for most mining equipment)
- **Cycle timeout**: 8 hours (reasonable for mining operations)
- **Speed filter**: Reject GPS points with speed >150 km/h (obvious errors)
- **No minimum segment durations**: Allow rapid transitions if GPS data shows them

9.4 Deployment Simplicity

- **Standard Database**: PostgreSQL with basic indexes
- **File-based Logging**: Simple log file outputs for debugging
- **Basic Error Handling**: Log errors and continue processing
- **No Configuration Files**: All parameters are hardcoded constants

---

This simplified business logic provides **reliable truck cycle tracking** using only the **available data sources** (asset table and Redis dump regions) with **fixed, practical business**

parameters that work across different mining operations without requiring complex configuration management or stakeholder input.