Malformed Payload Handling: Service Adjustments and Configurations

This document details the adjustments required across your data platform components to gracefully handle malformed payloads in both **local (Docker Compose)** and **AWS cloud environments**. It outlines where and how payloads might terminate or be managed when they deviate from expected structures, focusing on ensuring data integrity while preventing pipeline failures.

1. Understanding Payload Termination Points

A "malformed" payload can fail at various stages. The goal is to capture, log, and potentially quarantine such data as early as possible, providing clear visibility into *where* the issue occurred.

- **Ingestion Layer (FastAPI / AWS Lambda + API Gateway):**
  - **Purpose:** The first line of defense. Ideally, malformed payloads (syntax errors, missing required fields, invalid data types) should be rejected here with clear error messages.
  - **Termination:** HTTP 400 Bad Request (for unparseable JSON) or 422 Unprocessable Entity (for valid JSON but invalid data/schema against defined models).
- **Streaming/Message Queue (Kafka / MSK):**
  - **Purpose:** Kafka is generally schema-agnostic. It will accept any byte array. Malformed *content* (e.g., invalid JSON string within a message) will likely pass through Kafka but will cause issues for downstream consumers.
  - **Termination:** *Rarely* terminates at Kafka itself unless there's a fundamental network or serialization issue before Kafka.
- **Processing Layer (Spark / AWS Glue / Amazon EMR):**
  - **Purpose:** Where data is read from Kafka/S3, parsed, validated, and transformed. Malformed data can cause parsing errors, schema mismatches, or data quality violations here.
  - **Termination:** Job failure (e.g., ParseException, AnalysisException) or, with graceful handling, records being dropped or quarantined.
- **Storage Layer (Delta Lake on MinIO / S3):**
  - **Purpose:** If the malformed data somehow makes it past processing, it can lead to corrupted files or schema evolution issues if not handled by Delta Lake's features.
  - **Termination:** Data might not be written, or written as nulls/partial records, or cause downstream read failures.
- **Metadata / Lineage / Observability:**
  - Malformed data affecting successful pipeline runs can lead to incomplete lineage, inaccurate metrics, or uncataloged data.

2. Service Adjustments and Missing Configurations

2.1. Ingestion Layer: FastAPI Ingestor (Local) / AWS Lambda + API Gateway (Cloud)

The FastAPI application, built with Pydantic, is excellent for schema validation. The Python code written for FastAPI can be largely reused when refactoring to AWS Lambda. The key is to ensure errors are robustly handled and communicated at this entry point.

- **Current State:** FastAPI with Pydantic models (for local development).
- **Adjustments Needed:**
    1. **Comprehensive Pydantic Models:**
        - **Strict Validation:** Ensure Pydantic models enforce all required fields, data types, and formats (e.g., datetime for timestamps, UUID for IDs, Decimal for financial amounts). Use Field(..., gt=0) for positive amounts, pattern regex for specific string formats.
        - **Extra.forbid:** Consider setting Config.extra = Extra.forbid in your Pydantic models to reject payloads with unexpected fields, preventing extraField malformations from passing silently.

```
from pydantic import BaseModel, Field, Extra
from datetime import datetime
import uuid

class FinancialTransaction(BaseModel):
    transaction_id: uuid.UUID # Use UUID type
    timestamp: datetime
    amount: float = Field(..., gt=0) # Must be positive
    currency: str
    description: str | None = None
    user_id: str

    class Config:
        extra = Extra.forbid # Reject extra fields

class InsuranceClaim(BaseModel):
    claim_id: uuid.UUID
    policy_number: str
    claim_date: str # Consider using date (YYYY-MM-DD) validation
    claim_amount: float = Field(..., gt=0)
    claim_type: str
    insured_name: str
    status: str

    class Config:
        extra = Extra.forbid
```

2. **Custom Exception Handling for Validation Errors:**
   - By default, FastAPI returns 422 Unprocessable Entity for Pydantic validation errors. This behavior is generally desirable. You can customize this to provide more user-friendly messages or log details.
   - **For FastAPI (local main.py):**
```
from fastapi import FastAPI, HTTPException, Request, status
from fastapi.exceptions import RequestValidationError
from fastapi.responses import JSONResponse
import logging

logger = logging.getLogger(__name__)

app = FastAPI()

@app.exception_handler(RequestValidationError)
async def validation_exception_handler(request: Request, exc:
RequestValidationError):
  # Log the full details of the malformed payload
  logger.error(f"Validation error for request to {request.url}: {exc.errors()}
on payload: {await request.json()}")
  return JSONResponse(
    status_code=status.HTTP_422_UNPROCESSABLE_ENTITY,
    content={"detail": "Payload validation failed.", "errors": exc.errors()},
  )

@app.exception_handler(HTTPException)
async def http_exception_handler(request: Request, exc: HTTPException):
  logger.error(f"HTTP exception for request to {request.url}: Status
{exc.status_code}, Detail: {exc.detail}")
  return JSONResponse(
    status_code=exc.status_code,
    content={"detail": exc.detail},
  )

# ... other routes ...
```
   - **For AWS Lambda:** The same Python exception handling logic can be implemented within your Lambda function. Validation errors would typically be caught by Pydantic, and you'd return an API Gateway-compatible error response (e.g., statusCode: 422, body: JSON.stringify(error_details)).
3. **JSON Parsing Error Handling (for corruptJson):**
   - FastAPI typically handles unparseable JSON by returning a 400 Bad Request. Ensure this is consistently logged.

        - **For AWS Lambda + API Gateway:** API Gateway itself can perform basic JSON schema validation, or the Lambda function will receive an unparseable body, leading to an error in the Lambda runtime or your parsing logic. Ensure robust try-except blocks around json.loads() calls.
    4. **Enhanced Logging:**
        - Log incoming raw payloads (before validation, if desired for debugging malformations) and any validation errors with full details. This is crucial for forensic analysis.
        - Use a structured logger (e.g., json_logging for FastAPI, Python's logging module configured for JSON output, or native CloudWatch Logs for Lambda).

2.2. Streaming/Message Queue: Apache Kafka (Local) / Amazon MSK (Cloud)

Kafka primarily acts as a byte buffer. It will not inherently validate the *content* of messages unless you implement custom serializers/deserializers with schema validation at the producer/consumer level.

- **Current State:** Kafka/MSK accepts any byte array.
- **Adjustments Needed (Conceptual, beyond docker-compose.yml / AWS Provisioning):**
    1. **Schema Registry (e.g., Confluent Schema Registry or AWS Glue Schema Registry):**
        - **Purpose:** To enforce schema on Kafka/MSK messages at the producer (FastAPI/Lambda) level. If the producer tries to send a message that doesn't conform to the registered schema, it will be rejected *before* hitting Kafka/MSK.
        - **Benefit:** Shifts validation left, preventing malformed data from ever entering Kafka topics.
        - **Configuration:** Requires adding Schema Registry to your environment (e.g., another service in docker-compose.yml for local; provisioned AWS Glue Schema Registry for cloud) and configuring FastAPI/Lambda (producer) and Spark/Glue (consumer) to use it with Avro or Protobuf serializers. This is a significant architectural addition for strict data contracts at the message queue layer.

2.3. Processing Layer: Apache Spark (Local) / AWS Glue ETL / Amazon EMR (Cloud)

Spark jobs are crucial for robust data parsing and transformation. Even if the ingestion layer allows some malformed data (e.g., due to configuration errors or very subtle issues not caught by API validation), Spark needs to handle it. The principles here apply to PySpark jobs whether run locally, on EMR, or as Glue ETL jobs.

- **Current State:** Spark reads from Kafka/S3 and writes to Delta Lake.

- **Adjustments Needed:**
  1. **JSON/CSV Reading Modes:**
     - When reading JSON (or CSV), use specific mode options to control error handling:
       - mode("PERMISSIVE") (default): Inserts null for fields that cannot be parsed. No job failure.
       - mode("DROPMALFORMED"): Drops the entire row if it contains malformed data.
       - mode("FAILFAST"): Fails the Spark job immediately upon encountering a malformed record. (This is generally *not* desired for production pipelines handling continuous streams).
     - **Recommendation:** Use PERMISSIVE or DROPMALFORMED to prevent job failures. If using PERMISSIVE, downstream transformations must handle potential null values.
     - **Example (for streaming_consumer.py on local Spark, or equivalent Glue/EMR job):**

```
import pyspark.sql.functions as F
from pyspark.sql.types import StructType, StructField, StringType,
DoubleType, TimestampType

# ... (SparkSession creation, kafka_brokers, kafka_topic, output_path setup)
...

# When reading from Kafka (local Spark or EMR with Kafka connector)
# For Glue, you might read directly from Kinesis/MSK or S3
df = spark \
    .readStream \
    .format("kafka") \
    .option("kafka.bootstrap.servers", kafka_brokers) \
    .option("subscribe", kafka_topic) \
    .load()

# Define your schema explicitly for better control and nullable fields
if kafka_topic == "raw_financial_transactions":
    schema = StructType([
        StructField("transaction_id", StringType(), True),
        StructField("timestamp", TimestampType(), True),
        StructField("amount", DoubleType(), True),
        StructField("currency", StringType(), True),
        StructField("description", StringType(), True),
        StructField("user_id", StringType(), True),
        # Ensure all fields that might be missing or malformed are nullable
(True)
```

```
  ])
elif kafka_topic == "raw_insurance_claims":
  schema = StructType([
    StructField("claim_id", StringType(), True),
    StructField("policy_number", StringType(), True),
    StructField("claim_date", StringType(), True), # Keep as string if raw
format is inconsistent, then validate later
    StructField("claim_amount", DoubleType(), True),
    StructField("claim_type", StringType(), True),
    StructField("insured_name", StringType(), True),
    StructField("status", StringType(), True),
  ])
else:
  schema = None # Handle unknown topics or default to basic string
schema

if schema:
  # Use from_json to parse, with the permissive mode
  # The 'value' column from Kafka is binary, cast to STRING
  parsed_df = df.selectExpr("CAST(value AS STRING) as json_string") \
          .withColumn("data", F.from_json(F.col("json_string"), schema,
{"mode": "PERMISSIVE"})) \
          .select("data.*", F.col("json_string").alias("original_raw_json")) #
Keep raw json_string for bad records
else:
  parsed_df = df.selectExpr("CAST(value AS STRING) as original_raw_json")
# Fallback for unknown schemas

# Separate good and bad records
# A "bad" record here is one where any of the schema fields are null AFTER
parsing (due to permissive mode),
# or if the entire 'data' column is null (meaning the JSON itself was
unparseable or totally mismatched schema).
# We also check for critical fields that MUST NOT be null if they were
supposed to be parsed.
good_records_df = parsed_df.filter(F.col("original_raw_json").isNotNull() &
F.col("transaction_id").isNotNull()) # Example check for a critical field
bad_records_df = parsed_df.filter(F.col("original_raw_json").isNull() |
F.col("transaction_id").isNull()) # Capture what failed the 'good' check
```

2. **Bad Records Path / Quarantine Zone:**
    - Direct malformed records to a separate "quarantine" or "dead-letter" storage location (e.g., s3a://bad-records-bucket/financial_data_bad/). This

prevents them from polluting good data and allows for later inspection/reprocessing.

- **Example (streaming_consumer.py continuation for local Spark/EMR):**

```
# Write good records to raw Delta Lake
query_good = good_records_df \
    .writeStream \
    .format("delta") \
    .outputMode("append") \
    .option("checkpointLocation", f"{output_path}/_checkpoint") \
    .trigger(processingTime="1 minute") \
    .start(output_path)

# Write bad records to a separate bad records path
# For simplicity, write as text, could be more structured like JSON Lines or
Avro for re-processing
query_bad = bad_records_df.select("original_raw_json") \
    .writeStream \
    .format("text") \
    .outputMode("append") \
    .option("checkpointLocation",
f"s3a://bad-records-bucket/{kafka_topic}_bad_checkpoint") \
    .trigger(processingTime="1 minute") \
    .start(f"s3a://bad-records-bucket/{kafka_topic}_bad")
```

- **docker-compose.yml (Local) / Terraform (Cloud) adjustment:** You'll need to create a bad-records-bucket in MinIO (mc mb minio/bad-records-bucket;) locally, or an S3 bucket in AWS using Terraform for this purpose.

3. **Data Quality Checks (Advanced):**
   - Integrate libraries like [Great Expectations] or [Deequ] within your Spark/Glue jobs. These can define and validate expectations on data (e.g., amount is always positive, currency is in a predefined list).
   - **Benefit:** Catches logical errors beyond schema/parsing, allows for profiling and detailed reports on data quality.
   - **Action:** Requires installing these libraries in your Spark environment (Docker image for local, Glue job dependencies, or EMR cluster configuration) and adding specific data quality tasks to your PySpark jobs.
4. **Robust Logging and Metrics:**
   - Log counts of good vs. bad records (e.g., good_records_df.count(), bad_records_df.count()).
   - Emit custom metrics (e.g., using spark_metrics or pushgateway for Prometheus locally, or CloudWatch custom metrics for Glue/EMR) for "malformed_records_total" per topic/pipeline.

2.4. Observability Stack: Grafana (Local) / Amazon Managed Grafana (Cloud) & Grafana Alloy (Local) / AWS Distro for OpenTelemetry (ADOT) (Cloud)

Visibility is key to reacting to malformed data.

- **Current State:** Basic metrics collection.
- **Adjustments Needed:**
  1. **Alerting for API Errors:**
     - Configure Prometheus/CloudWatch alerts for high rates of 4xx (especially 422 Unprocessable Entity or 400 Bad Request) HTTP responses from the FastAPI/Lambda ingestor.
  2. **Dashboard for Bad Records:**
     - Create Grafana dashboards (local) or Amazon Managed Grafana dashboards (cloud) to visualize the "bad record count" metrics emitted by your Spark/Glue jobs. This provides a clear view of data quality issues.
  3. **Logging Aggregation:**
     - Ensure all application logs (FastAPI, Spark, Lambda, Glue) are centralized (e.g., via Grafana Alloy to Loki for local; or CloudWatch Logs to Amazon OpenSearch Service for cloud). This allows quick searching for specific error messages related to malformed data parsing or validation failures across the entire pipeline.

2.5. Data Lineage: Spline (Local) / AWS Glue Data Catalog (Cloud) & Metadata: OpenMetadata (Local) / AWS DataZone (Cloud)

These tools help understand the impact of malformed data on the data ecosystem and maintain data governance.

- **Current State:** Captures lineage for successful Spark jobs.
- **Adjustments Needed:**
  1. **Lineage for Quarantined Data:** If you implement a "bad records path," ensure Spline (for local Spark) or custom lineage capture mechanisms (for Glue Data Catalog or OpenMetadata) track the flow of these malformed records to their quarantine zone. This provides traceability for investigations.
  2. **Data Quality in OpenMetadata/AWS DataZone:** OpenMetadata and AWS DataZone have native data quality features. Configure them to:
     - Ingest data quality results from Great Expectations/Deequ (local or cloud).
     - Flag tables or columns that frequently receive malformed data based on metrics or validation rule failures.

3. Infrastructure Configuration Adjustments (docker-compose.yml for Local / Terraform for Cloud)

While many adjustments are in application code, a few infrastructure configurations support the handling of malformed payloads.

1. **FastAPI Ingestor Dockerfile / Lambda Deployment Package:**
   - **Local (fastapi-app/Dockerfile):** Ensure your Dockerfile installs any necessary logging or validation libraries (e.g., uvicorn[standard]).
   - **Cloud (Lambda deployment package):** Ensure your Lambda deployment package (ZIP file or container image) includes all Python dependencies needed for Pydantic and custom error handling.
2. **MinIO Bucket for Bad Records (Local) / S3 Bucket for Bad Records (Cloud):**
   - **Local (docker-compose.yml):** Add a new bucket to your init-minio-buckets service for quarantining bad records.
     ```
     # ... inside init-minio-buckets entrypoint ...
         /usr/bin/mc mb minio/bad-records-bucket; # New bucket for malformed data
     # ...
     ```
   - **Cloud (Terraform):** Provision a dedicated S3 bucket using Terraform.
     ```
     resource "aws_s3_bucket" "bad_records_bucket" {
       bucket = "your-app-bad-records-${var.environment}"
       tags = {
         Environment = var.environment
         Project     = var.project_name
         Purpose     = "BadRecordsQuarantine"
       }
     }
     # ... other S3 bucket configurations (versioning, lifecycle, encryption) ...
     ```
3. **Spark/Glue/EMR Dependencies for Data Quality Libraries:**
   - **Local (docker-compose.yml):** If using Great Expectations or Deequ with Spark, you might need to add their packages to SPARK_SUBMIT_ARGS (for Spark-specific versions) or ensure they are bundled in your Spark job's image/dependencies.
     - Example for spark-master and spark-worker-1 (conceptual, check specific library requirements):
       ```
       SPARK_SUBMIT_ARGS: "--packages org.apache.spark:spark-sql-kafka-0-10_2.12:3.3.2,io.delta:delta-core_2.12:2.4.0 --conf spark.driver.extraJavaOptions=-Dlog4j.configurationFile=file:///opt/bitnami/spark/conf/log4j2.properties --conf spark.executor.extraJavaOptions=-Dlog4j.configurationFile=file:///opt/bitnami/spark/conf/log4j2.properties --py-files /opt/bitnami/spark/deps/great_expectations.zip" # Example with --py-files for Python lib
       ```
   - **Cloud (AWS Glue Job / EMR Cluster Configuration):**

        - **AWS Glue:** Specify additional Python libraries (e.g., Great Expectations) directly in the Glue job configuration, either via an S3 path or by bundling them. Ensure appropriate Spark versions for compatibility.
        - **Amazon EMR:** Include necessary libraries in the EMR cluster bootstrap actions or add them to the Spark configuration when submitting jobs.
4. **Logging Configuration (Optional but Recommended):**
    - **Local (docker-compose.yml):** For richer logging, you might mount custom log4j2.properties for Spark or Python logging configurations for FastAPI to /etc/ directories.
      ```
      # Example for Spark History Server to pick up custom log4j config
      # spark-master, spark-worker-1, spark-history-server
      volumes:
        - ./config/spark/log4j2.properties:/opt/bitnami/spark/conf/log4j2.properties:ro
      ```
    - **Cloud (CloudWatch Logs, S3 Logging):** Ensure all AWS services are configured to send their logs to CloudWatch Logs. Configure S3 bucket logging for access transparency.

By implementing these adjustments, your data platform will become significantly more robust in handling unexpected and malformed data across both local and AWS environments, providing better error detection, clear termination points, and the ability to investigate and reprocess problematic records without disrupting the entire data flow.