Data Flow Architecture - CAN Telemetry Pipeline

Document: Data Flow Architecture Design
Issue: #68 - Design and Document Data Flow Architecture
Author: Abhishek Singh
Date: October 19, 2025
Version: 1.0
Status: Proposed → Review

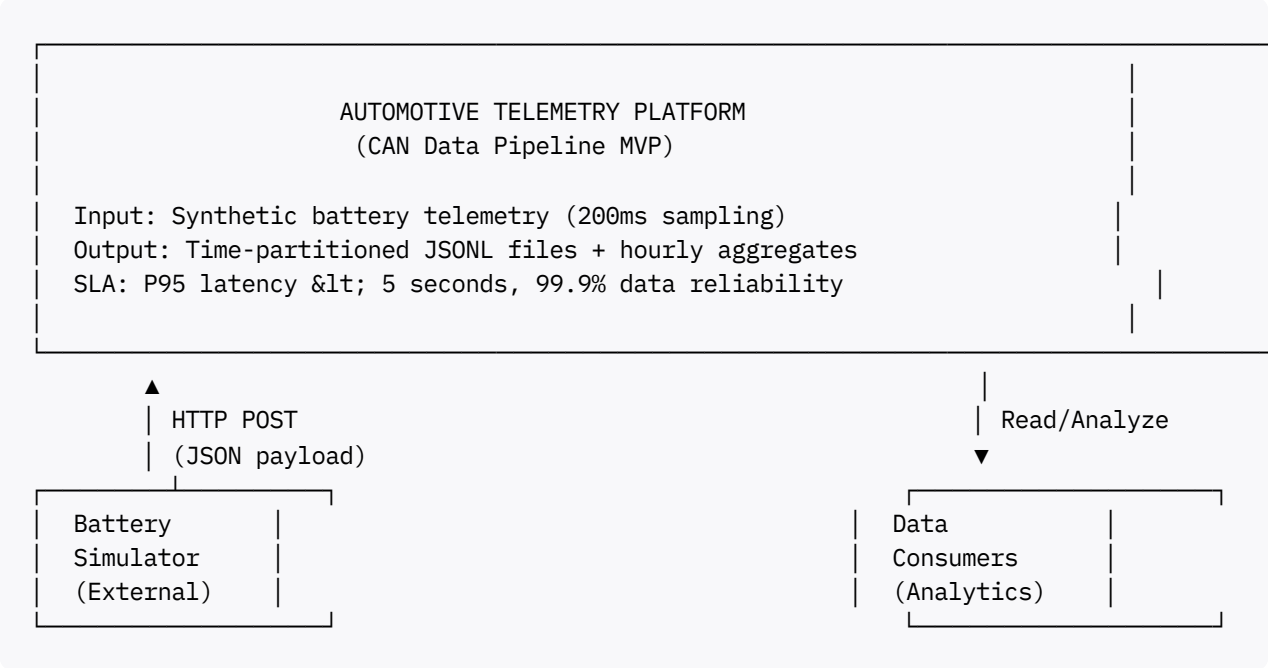
Document Purpose

This document provides a comprehensive architectural design for the **automotive telemetry data pipeline**, detailing how CAN bus data flows from simulation through ingestion, buffering, persistence, and aggregation. This architecture supports the **P01-Core-Pipeline-MVP** milestone and establishes the foundation for cloud-native scaling.

Target Audience: Development team, technical interviewers, system architects

1. System Overview

1.1 High-Level Architecture (Context Diagram - Level 0)



1.2 System Boundaries

In Scope:

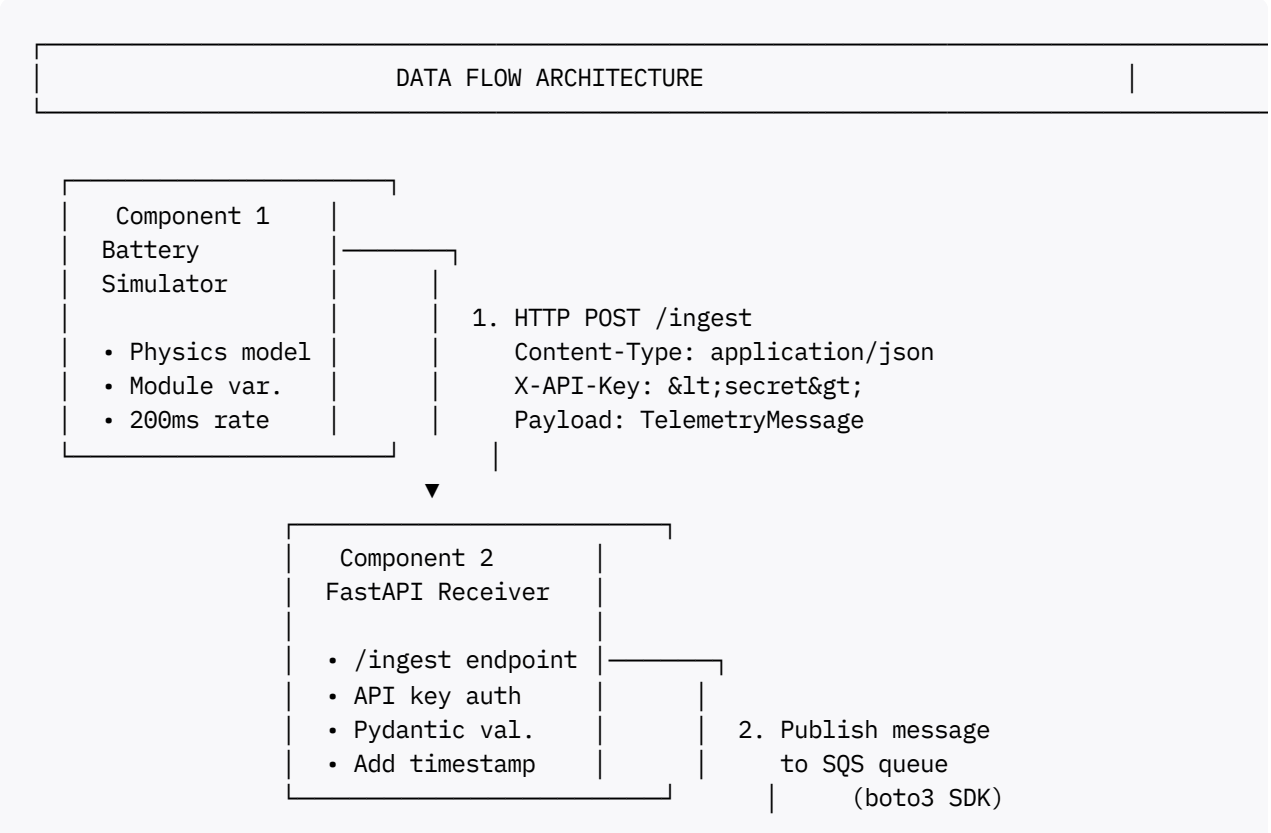
- Telemetry generation (battery simulation)
- HTTP ingestion with authentication
- Message buffering (AWS SQS)
- Batch processing and persistence (JSONL)
- Time-partitioned storage
- Hourly aggregation
- Latency tracking

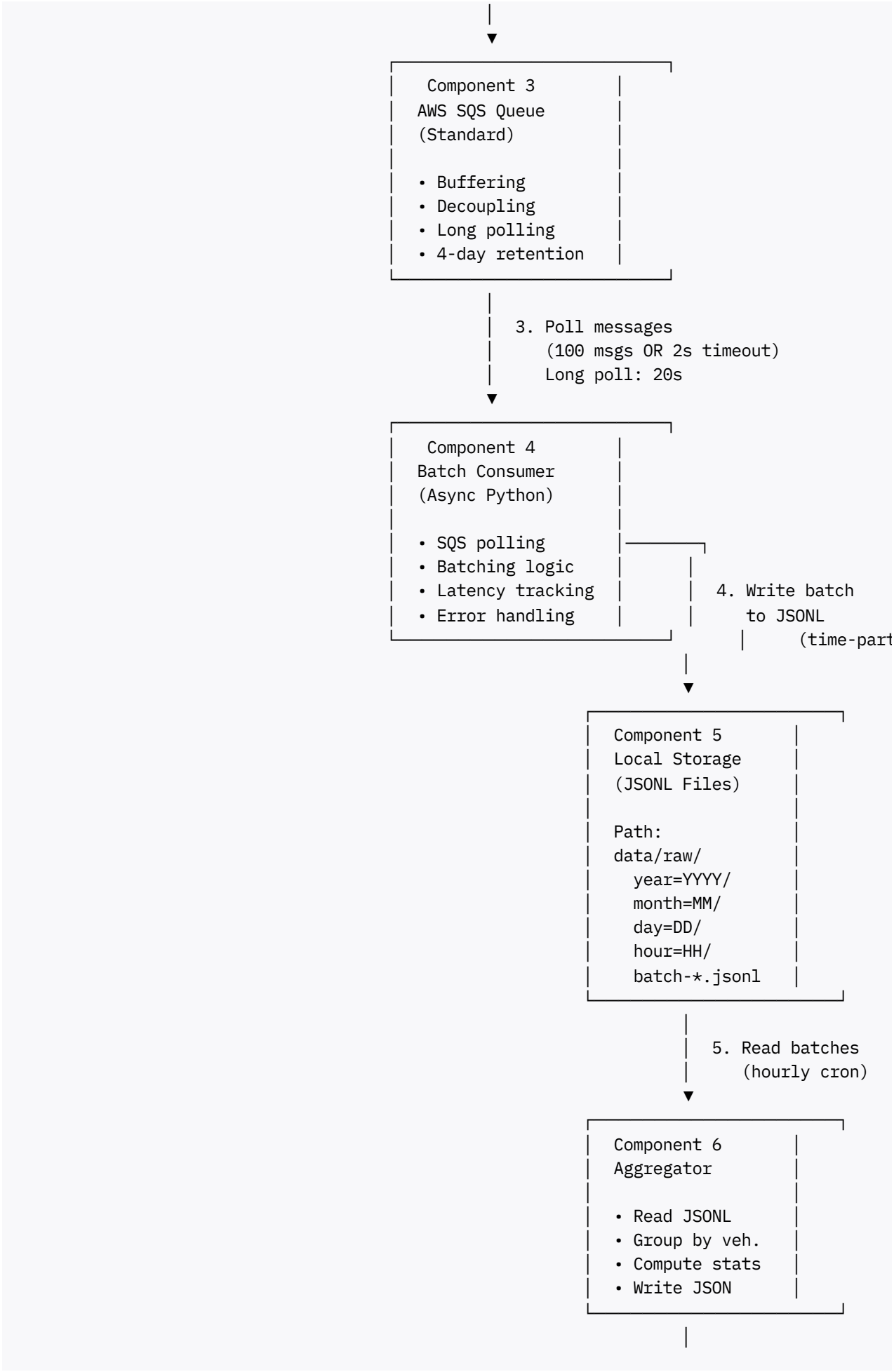
Out of Scope (Deferred):

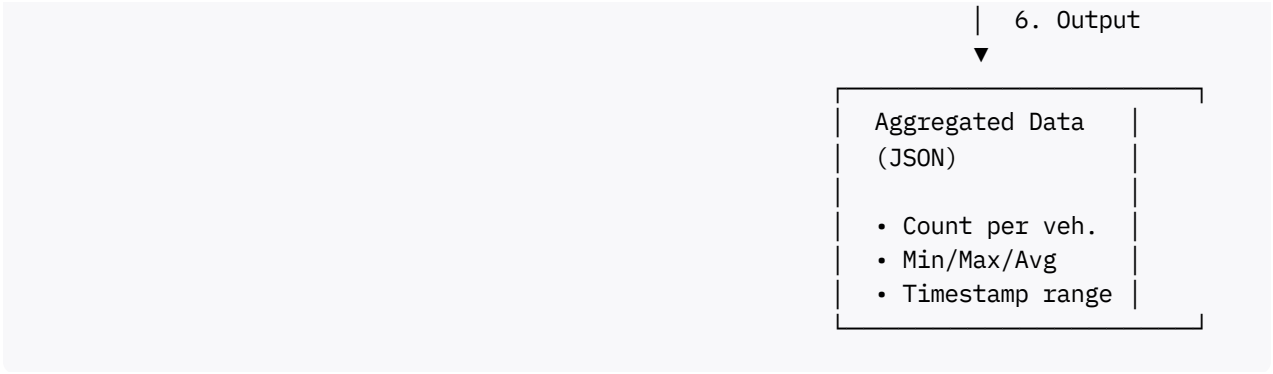
- Real-time streaming analytics
- Multi-region deployment
- Advanced monitoring dashboards
- Parquet conversion
- Data lake integration

2. Detailed Component Architecture (Level 1 DFD)

2.1 Component Diagram




Component 3
AWS SQS Queue
(Standard)
• Buffering
• Decoupling
• Long polling
• 4-day retention
3. Poll messages
(100 msgs OR 2s timeout)
Long poll: 20s
Component 4
Batch Consumer
(Async Python)
• SQS polling
• Batching logic
• Latency tracking
• Error handling
4. Write batch
to JSONL
(time-part
Component 5
Local Storage
(JSONL Files)
Path:
data/raw/
year=YYYY/
month=MM/
day=DD/
hour=HH/
batch-*.jsonl
5. Read batches
(hourly cron)
Component 6
Aggregator
• Read JSONL
• Group by veh.
• Compute stats
• Write JSON



2.2 Data Stores

DS1: AWS SQS Queue

- Type: Standard Queue (FIFO not required)
- Retention: 4 days (345,600 seconds)
- Visibility timeout: 30 seconds
- Message size: Max 256 KB
- Purpose: Decouple ingestion from processing

DS2: Local JSONL Files (data/raw/)

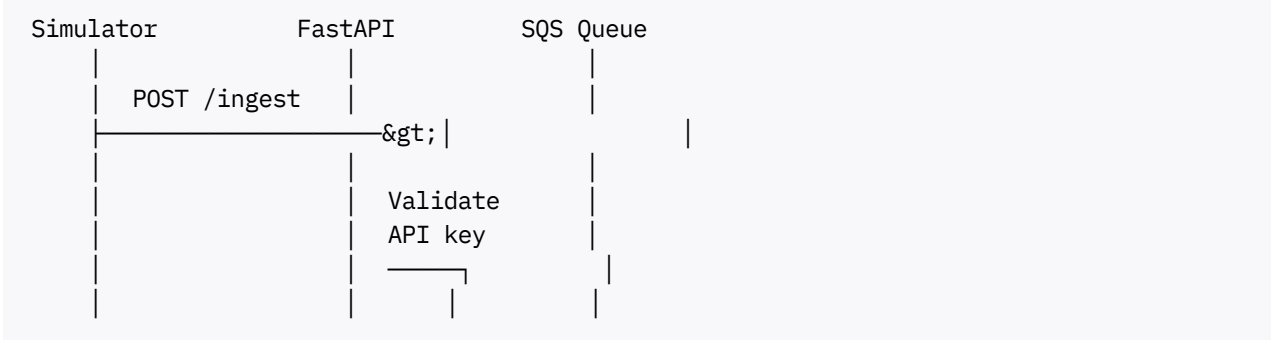
- Format: Line-delimited JSON
- Partitioning: year/month/day/hour
- Rotation: New file every 5 MB
- Purpose: Durable persistence before S3 migration

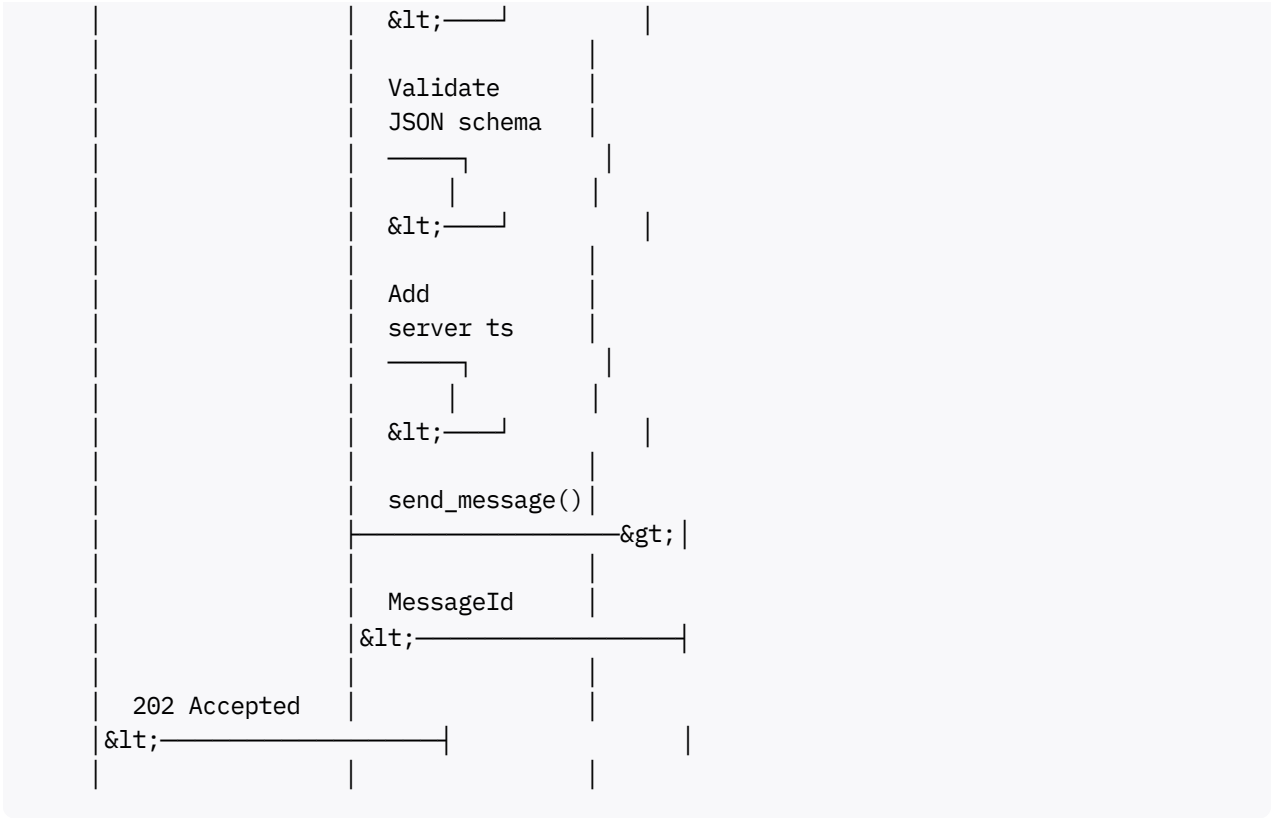
DS3: Aggregated Results (data/aggregated/)

- Format: JSON
- File naming: aggregated-<timestamp>.json
- Purpose: Hourly analytics output

3. Data Flow Sequences (Level 2 Detail)

3.1 Sequence 1: Message Ingestion





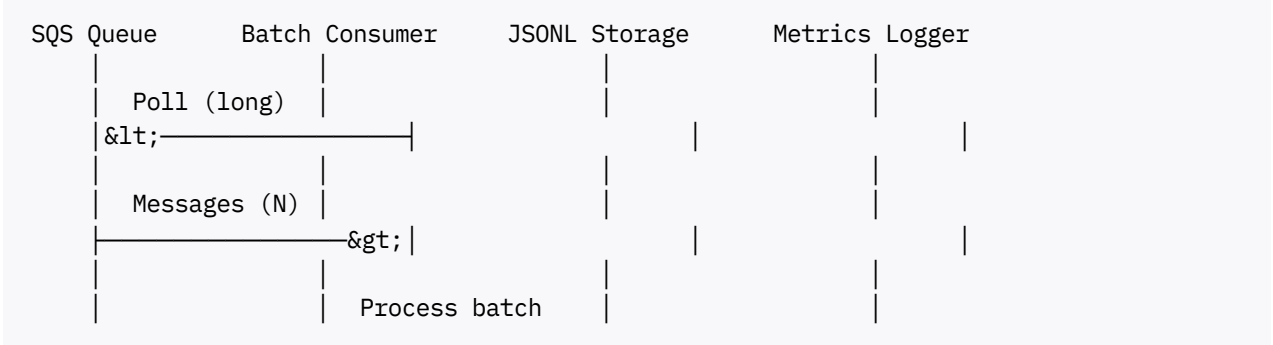
Key Operations:

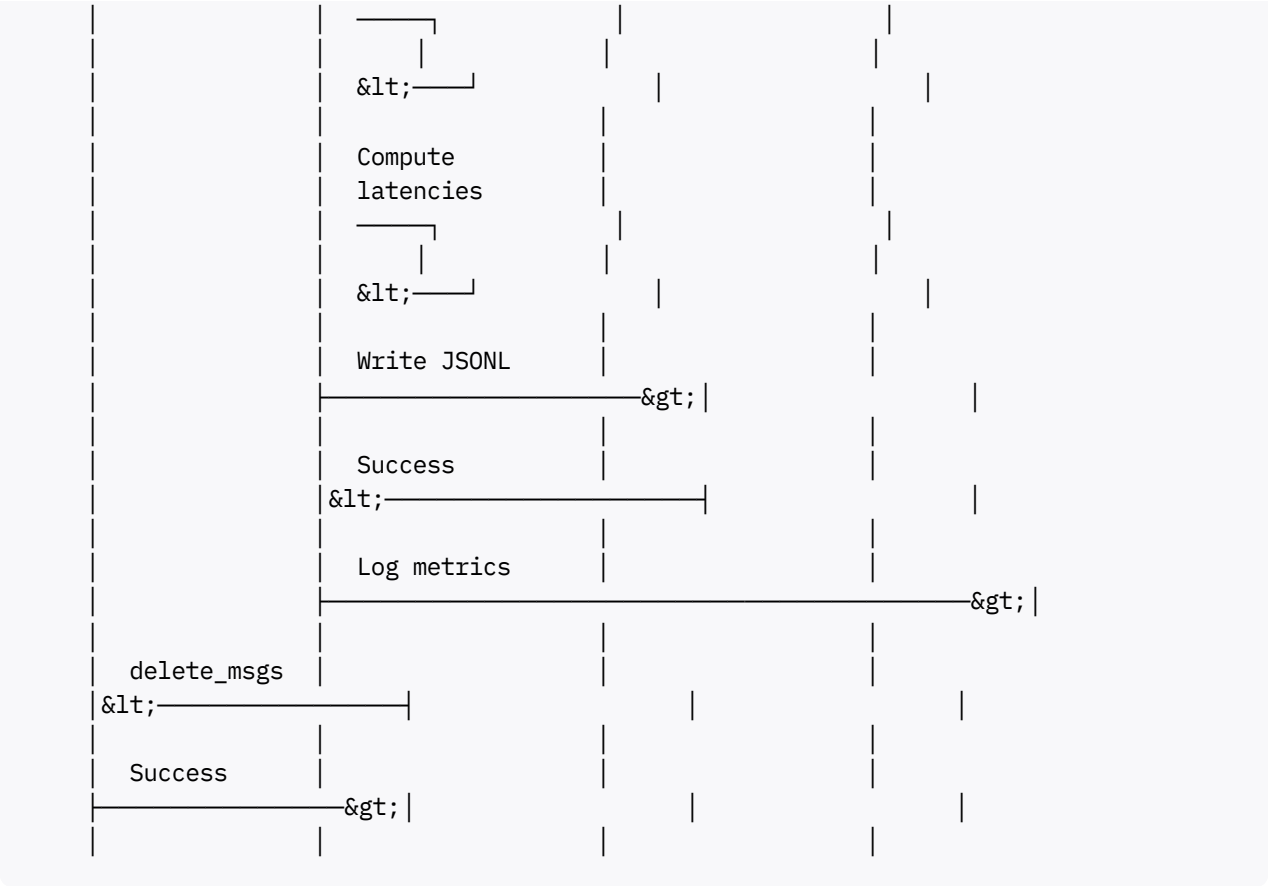
1. Simulator sends HTTP POST with JSON payload
2. FastAPI validates API key (X-API-Key header)
3. Pydantic validates message schema
4. Server adds `ingest_received_at` timestamp
5. boto3 publishes message to SQS
6. Return 202 Accepted (async acknowledgment)

Latency Targets:

- API validation: < 5ms
- SQS publish: < 50ms (P95)
- Total ingestion: < 100ms (P95)

3.2 Sequence 2: Batch Processing





Key Operations:

1. Consumer polls SQS with long polling (20s wait)
2. Receive up to 100 messages OR 2-second timeout
3. Process batch: deserialize, validate, timestamp
4. Calculate latency metrics (P50/P95/P99)
5. Write to time-partitioned JSONL file
6. Log batch metrics
7. Delete messages from SQS (only on success)

Batching Logic:

- Batch size: 100 messages
- Batch timeout: 2 seconds
- Trigger flush when batch size reaches 100 OR 2 seconds elapsed since first message

Error Handling:

- Failed write → Retry once
- Persistent failure → Log error, do NOT delete from SQS
- Message returns to queue after visibility timeout (30s)

4. Data Schema Definitions

4.1 Telemetry Message Schema (Input)

```
{
  "timestamp": "2025-10-19T18:15:23.456Z",
  "vehicle_id": "veh-00123",
  "can_id": "0x1F0",
  "payload": {
    "battery_soc": 85.5,
    "battery_voltage": 400.2,
    "battery_current": -15.3,
    "cell_voltage_min": 3.45,
    "cell_voltage_max": 3.67,
    "cell_temp_min": 28.5,
    "cell_temp_max": 31.2
  },
  "ingest_received_at": "2025-10-19T18:15:23.500Z"
}
```

Field Descriptions:

- `timestamp`: ISO 8601 UTC, client-side (may have clock skew)
- `vehicle_id`: Unique vehicle identifier (alphanumeric string)
- `can_id`: CAN message ID in hex format
- `payload`: Decoded CAN signals (battery telemetry)
- `ingest_received_at`: Server-side timestamp (authoritative)

4.2 JSONL Storage Format

```
{"timestamp":"2025-10-19T18:15:23.456Z","vehicle_id":"veh-00123"...}
{"timestamp":"2025-10-19T18:15:23.656Z","vehicle_id":"veh-00124"...}
{"timestamp":"2025-10-19T18:15:23.856Z","vehicle_id":"veh-00125"...}
```

Characteristics:

- One JSON object per line
- No formatting/indentation (compact)
- Newline-delimited (\n)
- Human-readable, grep-able

4.3 Aggregated Output Schema

```
{
  "aggregation_timestamp": "2025-10-19T18:00:00Z",
  "time_range": {
    "start": "2025-10-19T17:00:00Z",
    "end": "2025-10-19T17:59:59Z"
  },
  "vehicles": [
    {
      "vehicle_id": "veh-00123",
      "message_count": 1800,
      "battery_soc": {
        "min": 82.1,
        "max": 85.5,
        "avg": 83.8
      },
      "battery_voltage": {
        "min": 398.5,
        "max": 402.1,
        "avg": 400.3
      }
    }
  ]
}
```

5. Component Specifications

5.1 Component 1: Battery Simulator

Responsibility: Generate realistic synthetic CAN telemetry

Implementation:

- Language: Python 3.9+
- Framework: Custom battery physics model
- Module: `src/generator/simulator.py`

Outputs:

- HTTP POST to `/ingest` endpoint
- Rate: Configurable (default 200ms per vehicle)
- Format: JSON (TelemetryMessage schema)

Configuration:

```
NUM_VEHICLES = 10
SAMPLING_RATE_MS = 200
DURATION_SECONDS = 3600
```

5.2 Component 2: FastAPI Receiver

Responsibility: Accept and validate incoming telemetry

Implementation:

- Framework: FastAPI (async)
- Module: `src/receiver/app.py`
- Authentication: API key (custom header)

Endpoints:

- `POST /ingest` - Accept telemetry
- `GET /health` - Health check

Request Flow:

1. Validate `X-API-Key` header
2. Parse JSON body → Pydantic model
3. Add `ingest_received_at` timestamp
4. Publish to SQS (async)
5. Return 202 Accepted

5.3 Component 3: AWS SQS Queue

Configuration:

- Queue name: telemetry-queue-dev
- Queue type: Standard (not FIFO)
- Message retention: 4 days (345,600 seconds)
- Visibility timeout: 30 seconds
- Long polling: 20 seconds

Why SQS Standard vs FIFO:

- **Cost**: $0.40/million vs $0.50/million
- **Throughput**: Unlimited vs 3,000/sec
- **Ordering**: Not required for analytics
- **Duplicates**: Acceptable (handled downstream)

5.4 Component 4: Batch Consumer

Implementation:

- Language: Python 3.9+ (async)
- Module: `src/consumer/batch_consumer.py`

Processing Logic:

```
async def consume_loop():
    while True:
        messages = await poll_sqs(max_messages=100, wait_time=20)

        if messages or batch_timeout_reached():
            process_batch(messages)
            write_jsonl(messages)
            log_metrics(messages)
            delete_messages(messages)
```

5.5 Component 5: Local Storage (JSONL)

Path Structure:

```
data/raw/
  year=2025/
    month=10/
      day=19/
        hour=14/
          batch-1729344896123.jsonl
          batch-1729344898456.jsonl
```

File Naming: `batch-<unix_timestamp_ms>.jsonl`

5.6 Component 6: Aggregator

Implementation:

- Module: `src/aggregator/hourly_stats.py`
- Trigger: Cron (hourly at :05 past the hour)

Processing:

1. Identify JSONL files for previous hour
2. Stream-read lines (memory-efficient)
3. Group by vehicle_id
4. Compute aggregates (count, min, max, avg)
5. Write JSON output

6. Operational Characteristics

6.1 Latency Targets

Metric	Target	Measurement
Ingestion Latency	< 100ms (P95)	ingest_received_at - timestamp
Queue Latency	< 2s (P95)	SQS FirstReceive - Sent
Processing Latency	< 1s (P95)	Poll → JSONL write
End-to-End Latency	< 5s (P95)	timestamp → JSONL write

6.2 Throughput Targets

Scenario	Target	Notes
MVP (10 vehicles)	50 msg/s	200ms sampling
Scaling (100 vehicles)	500 msg/s	SQS handles
Future (1000 vehicles)	5,000 msg/s	Add consumers

6.3 Reliability

Data Loss Prevention:

1. SQS at-least-once delivery
2. Messages deleted only after successful write
3. 4-day retention for recovery
4. Idempotency: Duplicates acceptable

7. Monitoring & Observability

7.1 Key Metrics

Ingestion:

- Requests per second
- HTTP response codes
- Request latency (P50/P95/P99)

Queue (CloudWatch):

- ApproximateNumberOfMessages
- NumberOfMessagesSent/Received
- ApproximateAgeOfOldestMessage

Consumer:

- Batch processing rate

- Messages per second
- End-to-end latency
- Error rate

Storage:

- JSONL files per hour
- Data volume (MB/hour)
- Partition count

7.2 Logging Strategy

Log Levels:

- DEBUG: Message contents, async operations
- INFO: Batch processing, metrics
- WARNING: Retries, near-threshold
- ERROR: Failures, AWS API errors

8. Security Considerations

8.1 Authentication

API Key:

- Header: X-API-Key
- Storage: Environment variable
- Validation: FastAPI dependency
- Rotation: Monthly

8.2 AWS IAM Permissions

Receiver (SQS Publisher):

```
{
  "Effect": "Allow",
  "Action": ["sqs:SendMessage", "sqs:GetQueueUrl"],
  "Resource": "arn:aws:sqs:us-east-1:ACCOUNT:queue"
}
```

Consumer (SQS Consumer):

```
{
  "Effect": "Allow",
  "Action": ["sqs:ReceiveMessage", "sqs:DeleteMessage"],
```

```
  "Resource": "arn:aws:sqs:us-east-1:ACCOUNT:queue"
}
```

9. Cost Analysis

9.1 AWS Costs (Monthly)

SQS Standard:

- Requests: 130M/month (50 msg/s × 30 days)
- Cost: $0.40 per 1M requests
- **Total**: $52/month

Optimization:

- Long polling: 95% reduction in empty receives
- Batch processing: Fewer API calls
- Week 2 S3 lifecycle: 70% storage savings

10. Evolution & Upgrade Triggers

10.1 Migrate to Kinesis When:

Triggers:

- Sustained throughput > 10,000 msg/s
- Duplicate rate > 0.5%
- Need real-time streaming
- Multiple consumers

10.2 Switch to Parquet When:

Triggers:

- Query performance > 10 minutes
- Storage costs > $100/month
- Need columnar analytics

11. Testing Strategy

11.1 Unit Tests

Coverage Target: ≥95%

Test Cases:

- Battery model calculations
- Schema validation
- SQS publishing
- Batch processing
- Latency metrics
- JSONL writing

11.2 Integration Tests

End-to-End:

```
def test_pipeline():
    # 1. Start services
    # 2. Generate 1,000 messages
    # 3. Wait for processing
    # 4. Validate JSONL files
    # 5. Check P95 latency &lt; 5s
```

12. Interview Talking Points

Q: "Why SQS over Kinesis?"

A: "For MVP analytics use case:

- SQS provides simplicity, low ops overhead
- No strict ordering required
- Cost: $0.40/M vs Kinesis always-on cost
- Upgrade trigger: duplicate rate > 0.5%
- Documented in ADR 0001"

Q: "How to scale to 50K msg/s?"

A: "Evolution path:

1. Current (100 msg/s): Single consumer
2. 1,000 msg/s: Horizontal scaling

3. 10,000 msg/s: Kinesis Data Streams
4. 50,000 msg/s: Kinesis + Firehose
5. Bottlenecks: Network I/O, S3 throughput"

Q: "Explain batching strategy?"

A: "Dual-trigger batching:

- Flush on 100 msgs OR 2s timeout
- Trade-off: throughput vs latency
- P95 <5s drives 2s timeout
- Measured via P50/P95/P99 tracking"

13. References

- AWS SQS: https://docs.aws.amazon.com/sqs/
- FastAPI: https://fastapi.tiangolo.com/
- ADR 0001: docs/adr/0001-ingestion-transport.md

14. Approval & Sign-Off

Role	Name	Date	Status
Author	Abhishek Singh	Oct 19, 2025	✍ Drafted
Reviewer	Self Review	Oct 19, 2025	Pending

Document Status: Draft → Ready for Review
File Location: `docs/architecture/data-flow-architecture.md`
Closes Issue: #68