

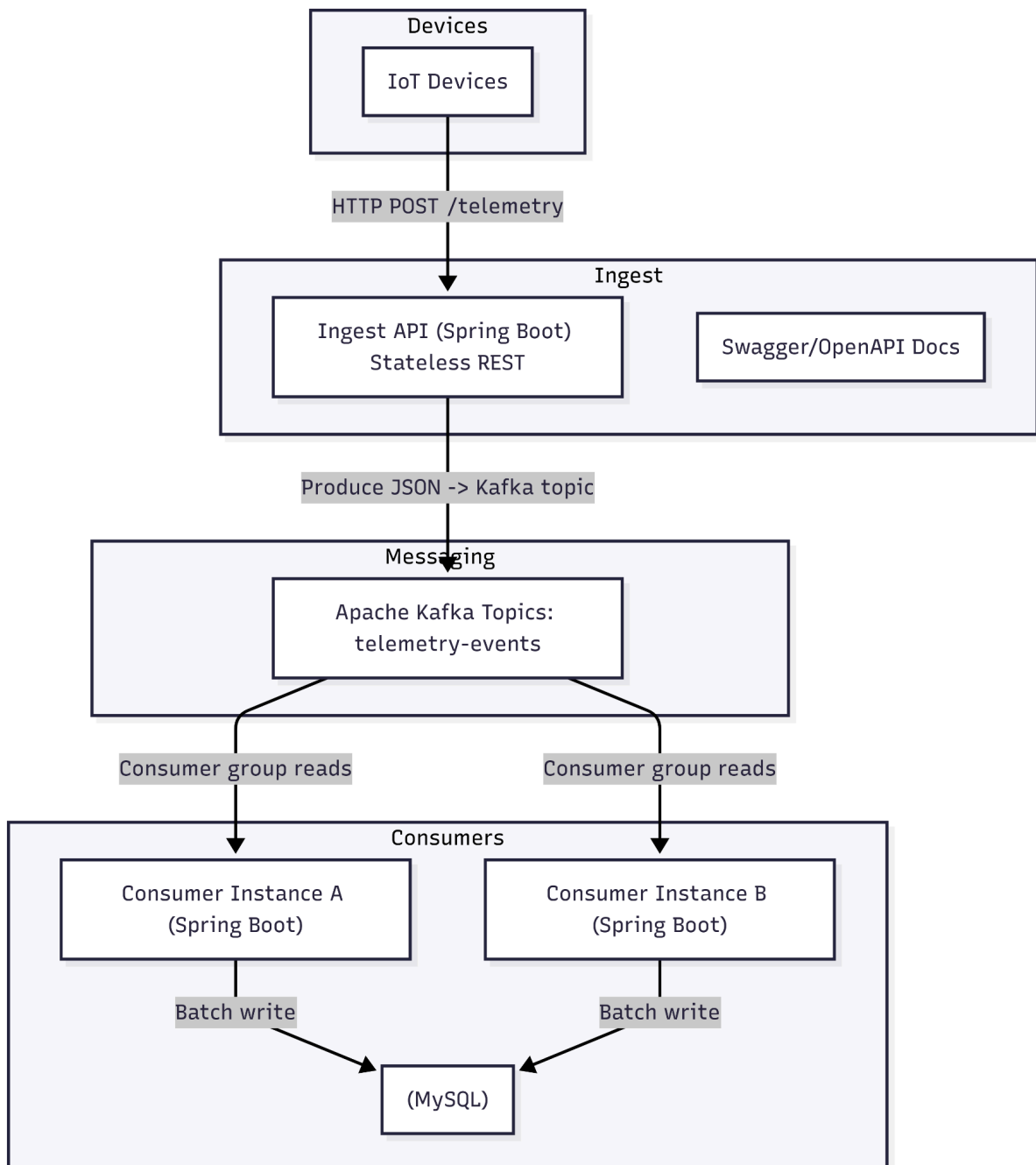
# IoT Data Processing System Comprehensive Documentation

(Uditha Nayanajith)

This document provides a detailed overview of the IoT data processing system, including architecture, component interactions, scaling strategies, and monitoring approaches.

## System Architecture

### High-Level Architecture



The system follows a microservices architecture with clear separation of concerns:

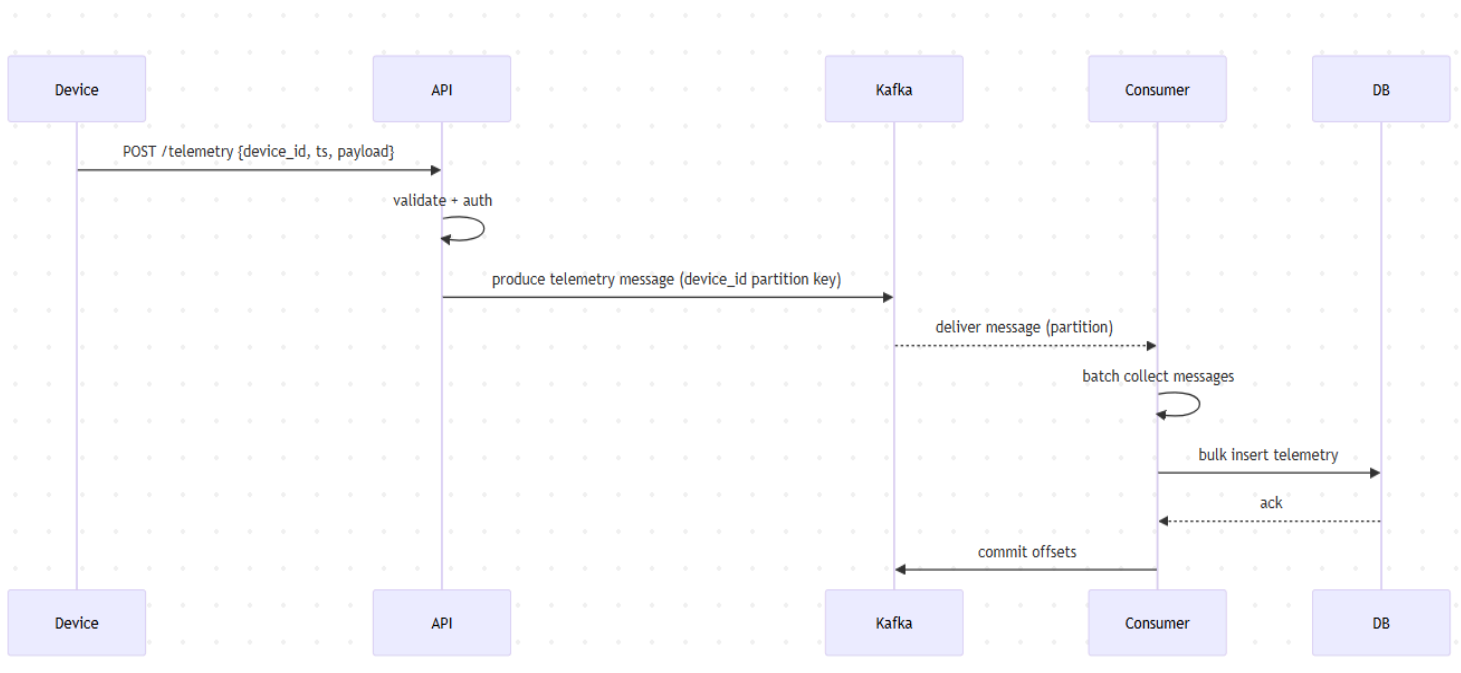
1. **IoT Ingest API:** Receives device registration and telemetry data via REST API
2. **Apache Kafka:** Acts as a message broker for asynchronous communication
3. **IoT Consumer Worker:** Processes telemetry messages from Kafka and stores them in the database
4. **MySQL Database:** Stores device information and telemetry data

### Data Flow

1. Devices register through the IoT Ingest API
2. Registered devices send telemetry data to the ingest API
3. The API validates devices and publishes telemetry to Kafka
4. Consumer workers read from Kafka, validate devices again, and store data in the database
5. Data is available for querying and analysis

## Sequence Diagram

### End-to-End Flow



The following sequence diagram illustrates how telemetry data flows through the system from device to database:

1. **Device**: Sends telemetry payload (e.g., JSON) via `POST /telemetry` request
2. **IoT Ingest API**: Validates device identity, checks schema, and accepts data
3. **Apache Kafka**: The API publishes the telemetry event into the `telemetry` topic
4. **IoT Consumer Worker**: Consumes messages in batches from Kafka, validates devices again, and prepares for persistence
5. **MySQL Database**: The worker writes telemetry data into the database for durable storage and later analysis
6. **Acknowledgments**: Once successfully persisted, offsets are committed in Kafka to mark the message as processed

### Key Points

- Ensures *at-least-once* delivery semantics by committing offsets only after persistence
- Supports horizontal scalability by running multiple consumer instances in the same group
- Provides resilience by decoupling producers (API) from consumers (workers) via Kafka

## Component Details

### IoT Ingest API Service

**Purpose:** Provides REST endpoints for device management and telemetry ingestion.

#### Key Components:

- **DeviceController**: Handles device registration and retrieval
- **TelemetryController**: Accepts telemetry data and publishes to Kafka
- **DeviceBusinessService**: Business logic for device operations
- **TelemetryProducerService**: Kafka producer service

#### Configuration Details:

`server.port: 8080`

`spring.kafka.bootstrap-servers: localhost:9092`

`spring.datasource.url: jdbc:mysql://localhost:3306/iotdatabase`

## IoT Consumer Worker Service

**Purpose:** Consumes telemetry messages from Kafka and stores them in the database.

### Key Components:

- **TelemetryConsumer:** Kafka consumer with batch processing capabilities
- **DeviceRepository:** Data access for device validation
- **TelemetryRepository:** Data access for storing telemetry records

### Configuration Details:

server.port: 8081

spring.kafka.bootstrap-servers: localhost:9092

spring.kafka.consumer.group-id: iot-telemetry-group

spring.kafka.consumer.max-poll-records: 500

## Service Interconnection

### How Services Interact

#### 1. Device Registration Flow

1. Client sends POST request to `/api/v1/devices` on IoT Ingest API
2. API validates and saves device to MySQL database
3. Device information is now available for both services

#### 2. Telemetry Ingestion Flow

1. Device sends telemetry data to `/api/v1/telemetry` on IoT Ingest API
2. API validates that device is registered
3. API publishes telemetry data to Kafka topic `iot-telemetry`
4. IoT Consumer Worker reads messages from Kafka
5. Consumer validates device exists in database
6. Consumer saves telemetry data to MySQL

#### 3. Data Consistency

Both services share the same database for device information, ensuring consistency in device validation.

Kafka provides at-least-once delivery semantics, ensuring no telemetry data is lost.

# Scaling Strategies

## Horizontal Scaling Approaches

### IoT Ingest API Scaling

- Deploy multiple instances behind a load balancer
- Stateless design allows easy horizontal scaling
- Consider using API gateway for rate limiting and authentication

### IoT Consumer Worker Scaling

- Deploy multiple consumer instances with the same group ID
- Kafka automatically partitions the load across consumers
- Increase Kafka topic partitions to allow more parallel consumers
- Adjust concurrency setting for more threads per instance

### Database Scaling

- Implement read replicas for reporting queries
- Consider sharding by device ID for very large deployments
- Use connection pooling and optimize queries

### Kafka Scaling

- Increase topic partitions as needed
- Monitor consumer lag to identify bottlenecks
- Consider Kafka cluster expansion for high throughput scenarios

## Performance Optimization

Component	Optimization Strategy
IoT Ingest API	Async processing, connection pooling, efficient serialization
IoT Consumer Worker	Batch processing, database bulk inserts, connection pooling
Database	Indexing, query optimization, appropriate hardware
Kafka	Proper partitioning, compression, monitoring

# Monitoring and Observability

## Monitoring Strategy

### Application Metrics

- **IoT Ingest API** exposes Prometheus metrics via Spring Boot Actuator
- Key metrics: HTTP request rates, error rates, JVM metrics, custom telemetry counters
- **IoT Consumer Worker** tracks message processing rates, error rates, and database performance

### Infrastructure Monitoring

- Monitor server resources (CPU, memory, disk I/O)
- Database performance metrics (query times, connection pool usage)
- Kafka metrics (topic throughput, consumer lag, broker performance)

### Alerting

Set up alerts for:

- High error rates in either service
- High consumer lag in Kafka
- Database connection issues
- Service downtime

### Logging

- Structured logging with correlation IDs for tracing requests across services
- Centralized log aggregation (ELK stack, Splunk, etc.)
- Log retention policies for troubleshooting and compliance

## Key Performance Indicators

Metric	Description	Target
API Response Time	P95 response time for ingest API	< 200ms
Telemetry Processing Lag	Time between ingestion and storage	< 1 second
Kafka Consumer Lag	Messages waiting to be processed	< 1000 messages
Error Rate	Percentage of failed requests/operations	< 0.1%

# Deployment Considerations

## Containerization

Both services are containerized with Docker for consistent deployment across environments.

## Orchestration

For production deployment, consider using Kubernetes for:

- Automated scaling based on load
- Self-healing capabilities
- Rolling updates with zero downtime
- Resource management

## Environment Configuration

Use environment-specific configuration files or config management tools for:

- Database connection strings
- Kafka broker addresses
- External service URLs
- Feature flags

# Security Considerations

- Implement authentication for API endpoints
- Use TLS for all communications
- Secure Kafka with SASL/SSL authentication
- Database encryption at rest and in transit
- Regular security updates and vulnerability scanning

# Future Enhancements

- Add real-time analytics capabilities
- Implement device management dashboard
- Add support for more telemetry formats
- Implement data retention policies
- Add anomaly detection for telemetry data