Kafka High-Throughput API Ingestion Documentation

Objective

Simulate high-throughput event ingestion using Kafka with Flask APIs, Kafka Producer/Consumer setup, load testing, and optional AWS Lambda integration.

Tech Stack

- **Apache Kafka + Zookeeper** (via Docker Compose)
- **Flask** for API endpoints
- **Python Kafka** (confluent_kafka) for producers and consumers
- **ThreadPoolExecutor** for concurrent load testing
- **Docker** for Kafka stack
- *(Optional)* AWS Lambda + API Gateway

Project Structure

```
kafka_ingestion_project/
├── app/
│   ├── producer_app.py       # Flask API: /register_event, /get_status
│   ├── consumer_app.py       # Kafka consumer: process events
│   └── templates/
│       └── index.html        # Optional frontend UI
├── docker-compose.yml        # Kafka + Zookeeper services
├── load_test.py              # Simulate high-load API calls
├── requirements.txt          # Python dependencies
└── run_all.py                # Combined launcher script
```

How to Run

One Command (Recommended)

python run_all.py

This will:

- Start Kafka + Zookeeper (make sure containers are already running)
- Launch the Flask API
- Launch the Kafka consumer
- Send 10,000+ simulated requests

Manual Steps (For Development)

1. Start Kafka and Zookeeper

$ docker-compose up -d

2. Install Python packages

$ pip install -r requirements.txt

3. Start Flask API

$ python app/producer_app.py

4. Start Kafka Consumer

$ python app/consumer_app.py

5. Run Load Test

$ python load_test.py

API Endpoints

Endpoint	Method	Description
/register_event	POST	Send data to Kafka topic
/get_status	GET	Returns system health

Kafka Topology

[Load Test] → [Flask API (/register_event)] → [Kafka Topic: events] → [Kafka Consumer(s)]

Features

- Structured logging
- Retry logic in consumers
- Concurrent event simulation using threads
- Simple one-command startup

Architectural Decisions

1. **Kafka for Streaming**: Kafka handles large-scale event ingestion reliably and persistently.
2. **Docker Compose**: Simplifies local Kafka setup for testing.
3. **Flask**: Lightweight and fast to implement for API simulation.
4. **Load Testing**: ThreadPoolExecutor simulates high-concurrency API usage.
5. **Consumer Design**: Scalable by nature; consumer logic supports multiple partitions.
6. **Offset Management**: Set to earliest to enable replay; consumers handle retries explicitly.

Challenges Faced

Area	Issue
Kafka Setup	Version mismatches & port mapping delays initial testing
Consumer Failures	Needed custom logic to prevent infinite retries
Load Testing	Hitting 10K RPM needed tuning of threads & batch sizes
Flask Limitations	Flask struggled with concurrency under stress; WSGI server preferred
Message Durability	Verifying persistence post-crash required volume configuration

Scope for Improvement

1. **Partitioning**: Use multiple Kafka partitions to achieve true consumer parallelism.
2. **Gunicorn/Uvicorn**: Replace Flask dev server for better concurrency.
3. **Persist Output**: Store processed messages in DB for better tracking.
4. **DLQ Support**: Add Dead Letter Queue for unprocessable events.
5. **Cloud Deployment**: Use AWS Lambda + MSK for cloud-native ingestion.
6. **Monitoring**: Add Prometheus + Grafana for real-time metrics.

Conclusion

This project demonstrates high-throughput event ingestion using Kafka and Flask with load testing support. While optimized for local testing, the architecture is extendable to distributed and cloud environments with minor additions. The system highlights real-world ingestion reliability patterns, including consumer resilience, durability guarantees, and replay logic.