Below is a **copy-pasteable, extremely detailed project outline** you can hand to a fresh LLM chat to build the full system. It includes **repo structure**, **Docker Compose**, **file-by-file responsibilities**, **APIs**, **schemas**, **storage**, **Redis logic**, **Kafka topics**, and a **stepwise build plan**.

Project: Internal Developer Event Platform (Ingestion + Processing + Admin Ops)

Goal

Build an internal, developer-facing event platform with:

- **Fast ingestion API** (Go) with **API-key auth**, **rate limiting**, **idempotency**
- **Async pipeline** via **Kafka** (`events.raw`, `events.dlq`)
- **Processor** (Java Spring Boot) that validates/enriches events and stores them in **Postgres**
- **DLQ** routing for poison messages
- **Admin API** (can live in processor service) exposing operational metrics
- **React admin dashboard** consuming Admin API
- **Python load generator** simulating producers (normal, duplicates, malformed, multi-tenant)
- Everything runs locally via **Docker Compose** for infra (Kafka/Redis/Postgres)

---

1) Repo Structure (Monorepo)

```
pulse-event-platform/
  README.md
  .gitignore

  infra/
    docker-compose.yml
    kafka-init/
      create-topics.sh

  ingest-go/
  go.mod
  cmd/
    ingest/
      main.go
  internal/
    config/
      config.go
    http/
      router.go
      middleware_auth.go
      middleware_ratelimit.go
      handler_events.go
      models.go
      responses.go
    redis/
      client.go
      ratelimit.go
      idempotency.go
    kafka/
      producer.go
    metrics/
      metrics.go
    util/
      time.go
      uuid.go
  Dockerfile (optional later)

processor-java/
  pom.xml
  src/main/java/com/yourorg/processor/
    ProcessorApplication.java
    config/
      KafkaConfig.java
      PostgresConfig.java
      MetricsConfig.java
    kafka/
      RawEventConsumer.java
      DlqProducer.java
      EventPublisher.java (optional)
    service/
      EventProcessorService.java
      ValidationService.java
      EnrichmentService.java
    db/
```

```
      EventEntity.java
      EventRepository.java
      Migrations.md (notes)
    api/
      AdminController.java
      DtoModels.java
    observability/
      LagService.java
      HealthIndicators.java
    util/
      JsonUtil.java
  src/main/resources/
    application.yml
    db/migration/
      V1__init.sql
  Dockerfile (optional later)

dashboard/
  package.json
  next.config.js
  src/
    main.tsx
    App.tsx
    api/
      client.ts
      types.ts
    pages/
      Overview.tsx
      Pipeline.tsx
      Dlq.tsx
      Search.tsx
    components/
      MetricCard.tsx
      Chart.tsx (optional)
      Table.tsx
    styles/
      index.css

loadgen/
  requirements.txt
  loadgen.py
  scenarios/
    baseline.json
    spike.json
```

Key point: For MVP, you can run services directly on Windows terminals (Go/Java/React/Python) while Docker runs Kafka/Redis/Postgres.

---

2) Infrastructure: Docker Compose (Kafka + Redis + Postgres)

Create: `infra/docker-compose.yml`

```
services:
  postgres:
    image: postgres:16
    container_name: ep_postgres
    environment:
      POSTGRES_USER: ep_user
      POSTGRES_PASSWORD: ep_pass
      POSTGRES_DB: event_platform
    ports:
      - "5432:5432"
    volumes:
      - ep_postgres_data:/var/lib/postgresql/data
    healthcheck:
      test: ["CMD-SHELL", "pg_isready -U ep_user -d event_platform"]
      interval: 5s
      timeout: 5s
      retries: 10

  redis:
    image: redis:7
    container_name: ep_redis
    ports:
      - "6379:6379"
    command: ["redis-server", "--appendonly", "yes"]
    volumes:
      - ep_redis_data:/data
    healthcheck:
      test: ["CMD", "redis-cli", "ping"]
      interval: 5s
      timeout: 3s
      retries: 10

  kafka:
    image: bitnami/kafka:3.7
    container_name: ep_kafka
    ports:
      - "9092:9092"
    environment:
      - KAFKA_CFG_NODE_ID=1
      - KAFKA_CFG_PROCESS_ROLES=broker,controller
      - KAFKA_CFG_CONTROLLER_QUORUM_VOTERS=1@kafka:9093
      - KAFKA_CFG_LISTENERS=PLAINTEXT://:9092,CONTROLLER://:9093
      - KAFKA_CFG_ADVERTISED_LISTENERS=PLAINTEXT://localhost:9092
      - KAFKA_CFG_LISTENER_SECURITY_PROTOCOL_MAP=PLAINTEXT:PLAINTEXT,CONTROLLER:PLAINTEXT
      - KAFKA_CFG_CONTROLLER_LISTENER_NAMES=CONTROLLER
      - KAFKA_CFG_INTER_BROKER_LISTENER_NAME=PLAINTEXT
      - ALLOW_PLAINTEXT_LISTENER=yes
      - KAFKA_CFG_AUTO_CREATE_TOPICS_ENABLE=false
    healthcheck:
      test: ["CMD-SHELL", "kafka-topics.sh --bootstrap-server localhost:9092 --list 1>/dev/null"]
      interval: 10s
      timeout: 10s
      retries: 20

  kafka-init:
    image: bitnami/kafka:3.7
    container_name: ep_kafka_init
    depends_on:
      kafka:
        condition: service_healthy
    volumes:
      - ./kafka-init/create-topics.sh:/create-topics.sh:ro
    entrypoint: ["/bin/bash", "/create-topics.sh"]

volumes:
  ep_postgres_data:
  ep_redis_data:
```

Create: `infra/kafka-init/create-topics.sh`

```
#!/bin/bash
set -e

echo "Creating Kafka topics..."

kafka-topics.sh --bootstrap-server kafka:9092 --create --if-not-exists \
  --topic events.raw --partitions 6 --replication-factor 1

kafka-topics.sh --bootstrap-server kafka:9092 --create --if-not-exists \
  --topic events.dlq --partitions 3 --replication-factor 1

echo "Topics created."
```

What this does:

- Runs Postgres with a persistent volume (`ep_postgres_data`)
- Runs Redis with append-only persistence (optional but nice)
- Runs Kafka in KRaft mode (no Zookeeper)
- Creates topics explicitly (auto-create disabled)

---

3) Data Contracts (Event Schema)

Ingestion API input payload (producer sends)

POST `/events`

Headers:

- `X-API-Key: <tenant key>`
- `Idempotency-Key: <unique key per logical event>` (required for dedupe)
- `Content-Type: application/json`

Body (minimal):

```
{
  "event_id": "evt_123",
  "event_type": "user_login",
  "schema_version": 1,
  "occurred_at": "2026-01-02T20:00:00Z",
  "payload": { "user_id": "u1", "ip": "1.2.3.4" }
}
```

What gets published to Kafka (`events.raw`)

Wrap raw event with ingestion metadata (in Go before publishing):

```
{
  "tenant_id": "tenant_a",
  "received_at": "2026-01-02T20:00:01Z",
  "request_id": "req_uuid",
  "idempotency_key": "idem_abc",
  "event": { ...original event... }
}
```

---

4) Storage (Postgres)

Table `events`

Stored after validation/enrichment in processor.

Columns:

- `id` (uuid pk)
- `tenant_id` (text)
- `event_id` (text)
- `idempotency_key` (text)
- `event_type` (text)
- `schema_version` (int)
- `occurred_at` (timestamptz)
- `received_at` (timestamptz)
- `processed_at` (timestamptz)
- `payload` (jsonb)
- `status` (text) // e.g. "processed"
- indexes:
    - `(tenant_id, event_id)`

    - `(tenant_id, idempotency_key)`
    - `(event_type, occurred_at)`

Optional table `dlq_events`

If you want to persist DLQ samples in DB (optional for MVP). Otherwise, Admin API can consume DLQ topic to fetch sample messages.

---

5) Redis Responsibilities (Ingestion Service)

Redis keys (namespaced):

Rate limiting

Key: `rl:{tenant_id}:{minute_bucket}`

- Use INCR + EXPIRE
- Example limit: 300 req/min per tenant
- If over limit, respond `429 Too Many Requests`

Idempotency

Key: `idem:{tenant_id}:{idempotency_key}`

- On first request: SETNX + EXPIRE (TTL 30 minutes)
- If already exists: treat as duplicate
    - Return 200/202 with `{ "duplicate": true }` OR return 409 (your choice; recommend returning 202 with duplicate flag for friendliness)

---

6) Services: Responsibilities & File-by-File Expectations

A) Go Ingestion Service (`ingest-go/`)

Config (env vars)

- `INGEST_PORT=8080`
- `REDIS_ADDR=localhost:6379`
- `KAFKA_BROKERS=localhost:9092`
- `RATE_LIMIT_PER_MIN=300`
- `IDEMPOTENCY_TTL_SECONDS=1800`
- `API_KEYS=tenant_a:key_a,tenant_b:key_b` (simple config for MVP)

Required endpoints

1. `POST /events`

- Auth with API key
- Rate limit
- Validate JSON minimally
- Idempotency check
- Publish to Kafka `events.raw`
- Return 202 with request_id and status

2. `GET /health`

- returns ok + dependencies basic check (optional)

3. `GET /metrics` (optional)

- simple JSON metrics (requests, errors, latency p95 approximate) OR expose Prometheus later

File expectations

- `cmd/ingest/main.go`: wire config, init Redis + Kafka producer, start HTTP server
- `internal/config/config.go`: load env vars, parse API keys mapping
- `internal/http/router.go`: register routes + middleware chain
- `internal/http/middleware_auth.go`: read `X-API-Key`, map to tenant_id, attach tenant to request context
- `internal/http/middleware_ratelimit.go`: call redis ratelimit logic, block if exceeded
- `internal/http/handler_events.go`: parse request, call idempotency, publish to kafka, respond
- `internal/redis/client.go`: redis connection init
- `internal/redis/ratelimit.go`: INCR/EXPIRE logic
- `internal/redis/idempotency.go`: SETNX + TTL

- `internal/kafka/producer.go`: producer init + `PublishRawEvent(...)`
- `internal/http/models.go`: request/response structs
- `internal/http/responses.go`: consistent JSON error format
- `internal/metrics/metrics.go`: measure latency per request, counts (in-memory is fine)

Response formats

Success (new event):

{ "status": "accepted", "request_id": "req_uuid", "duplicate": false }

Duplicate:

{ "status": "accepted", "request_id": "req_uuid", "duplicate": true }

Errors (standard):

{ "error": { "code": "RATE_LIMITED", "message": "Too many requests" } }

B) Java Processor Service (`processor-java/`)

Config (application.yml)

- Kafka bootstrap: `localhost:9092`
- topic names: `events.raw`, `events.dlq`
- Postgres: host `localhost`, db `event_platform`, user/pass from compose
- consumer group: `event-processor`
- retry policy: basic (MVP: 0-1 retry then DLQ)

Core behavior

Consume `events.raw`:

1. Deserialize wrapper
2. Validate required fields + schema_version allowed
3. Enrich: set processed_at timestamp
4. Insert into Postgres

5. If fails validation/deserialization → publish original message + reason to `events.dlq`

Admin endpoints (can live in same Spring Boot app)

- `GET /admin/overview`
  - events per minute (last 1m, 5m)
  - error counts (if tracked)
- `GET /admin/top-event-types?sinceMinutes=1440`
- `GET /admin/event/search?tenant=...&eventId=...`
- `GET /admin/event/by-idempotency?tenant=...&idempotencyKey=...`
- `GET /admin/dlq/sample?limit=20` (either read from DLQ topic or from dlq table)
- `GET /admin/health`
- `GET /admin/kafka/lag` (simple lag estimate)

File expectations

- `kafka/RawEventConsumer.java`: `@KafkaListener(topics="events.raw")` receives message
- `service/ValidationService.java`: checks required fields & version
- `service/EnrichmentService.java`: adds processed_at and possibly normalized fields
- `service/EventProcessorService.java`: orchestrates validate → enrich → persist; catches exceptions, routes to DLQ
- `kafka/DlqProducer.java`: publishes to `events.dlq` with reason
- `db/EventEntity.java`: JPA entity mapping for events table
- `db/EventRepository.java`: Spring Data repository queries for search endpoints
- `api/AdminController.java`: REST endpoints for dashboard
- `observability/LagService.java`: compute consumer lag (MVP: expose placeholder or use Kafka AdminClient to fetch offsets)
- `resources/db/migration/V1__init.sql`: create schema

DLQ message format (publish reason)

```
{
  "failed_at": "2026-01-02T20:05:00Z",
  "reason": "VALIDATION_FAILED: missing event_type",
  "original": { ...raw kafka message... }
}
```

C) React Dashboard (`dashboard/`)

Pages (MVP)

1. Overview
- cards: events last minute, events last 5 minutes, top event type today
- health status (green/red)
2. Pipeline
- kafka lag
- processed events per minute (last 5m)
- DLQ count
3. DLQ
- list sample DLQ messages
- show reason + expand payload
4. Search
- search by event_id or idempotency_key
- show event details from DB

API client

- Base URL: `http://localhost:8081` (if processor/admin runs on 8081)
- `api/client.ts`: fetch wrapper
- `api/types.ts`: TS types

File expectations

- `pages/Overview.tsx`: fetch `/admin/overview`
- `pages/Pipeline.tsx`: fetch `/admin/kafka/lag`, `/admin/overview`
- `pages/Dlq.tsx`: fetch `/admin/dlq/sample`
- `pages/Search.tsx`: fetch search endpoints, render event

D) Python Load Generator (`loadgen/`)

Purpose

Simulate producers:

- sustained traffic
- duplicates

- malformed events
- multiple API keys/tenants
- measure latency and error rate

Behavior

- Accept CLI args: `--rps`, `--minutes`, `--duplicate-rate`, `--bad-rate`, `--tenants`
- Generate `Idempotency-Key` deterministically for duplicates
- Print summary stats: success, duplicates, rate-limited, invalid, avg latency, p95 approx

File expectations

- `loadgen.py`: main script
- `requirements.txt`: `requests`

---

7) Ports & Local Run Plan (Windows)

- Docker services:
  - Postgres: `localhost:5432`
  - Redis: `localhost:6379`
  - Kafka: `localhost:9092`
- Apps:
  - Go ingest: `localhost:8080`
  - Java processor/admin: `localhost:8081`
  - React dashboard: `3000` (Next)
  - Loadgen hits Go ingest on `8080`

---

8) Step-by-Step Build Order (What the new LLM should do)

Step 1 — Bring up infra

- Implement `infra/docker-compose.yml` and `create-topics.sh`

- Run: `docker compose up -d` from `infra/`
- Verify:
    - Postgres up
    - Redis ping
    - Kafka topics exist

Step 2 — Go ingest minimal

- `POST /events` publishes wrapper to Kafka `events.raw`
- No rate limit/idempotency yet

Step 3 — Java processor consumes and writes Postgres

- Create migrations and events table
- Consume raw events and insert
- Send malformed to DLQ

Step 4 — Add Redis rate limiting

- per-tenant limit with INCR/EXPIRE

Step 5 — Add Redis idempotency

- SETNX with TTL
- duplicates return accepted+duplicate flag

Step 6 — Admin endpoints

- Add overview queries from Postgres
- Add search endpoints
- Add DLQ sample (basic)

Step 7 — React dashboard

- Build pages to show overview, pipeline, dlq, search

Step 8 — Loadgen + failure demo

- Python script load test

- Simulate stopping processor and observing lag/DLQ

---

9) “Definition of Done” (MVP Acceptance Criteria)

1. `POST /events` requires API key and idempotency key
2. Rate limiting returns 429 for over-quota tenants
3. Duplicate idempotency keys do not republish to Kafka
4. Processor stores valid events in Postgres
5. Invalid events end up in Kafka `events.dlq` (and show in DLQ page)
6. Admin API returns:
    - events/min last 1 and 5 minutes
    - top event types today
    - search by event_id/idempotency
    - DLQ sample list
7. Dashboard renders those endpoints
8. Loadgen can demonstrate:
    - duplicates ignored
    - malformed goes DLQ
    - sustained throughput
    - rate limiting works

---

10) Notes / Constraints (avoid scope creep)

- No “exactly once” claims
- Kafka provides durability + ordering per partition, not global ordering
- Idempotency is “best effort” with Redis TTL (good enough for MVP)
- Keep schemas simple; just enforce required fields + version

---

Also generate: a **README.md** that explains how to run everything