Event Stream Processor

Engineering Design Document

Author Maya Thornton, Staff Engineer – Platform Reliability
Reviewers Carlos Ibáñez (Architecture), Priya Mehta (Security), Dan Lee (SRE)
Version 1.2
Status DRAFT Pending Architecture Review
Date 2026-07-05
Confluence EDD-ESP-1.2

1. Overview & Goals

The **Event Stream Processor (ESP)** is a horizontally scalable microservice responsible for ingesting, filtering, transforming, and routing high-throughput event streams from internal producers (mobile clients, IoT sensors, backend services) to downstream consumers (analytics warehouse, alerting engine, third-party webhooks).

Goals:

- Handle $\geq 500{,}000$ events/sec sustained with $< 10\,\text{ms}$ p99 latency.
- Guarantee at-least-once delivery with idempotent consumer support.
- Provide a declarative routing DSL for schema-validated event pipelines.
- Achieve zero-downtime rolling deploys and automated canary promotion.

Non-goals: Stateful aggregations (owned by Analytics Service); long-term event storage (owned by the Data Lake).

2. Requirements

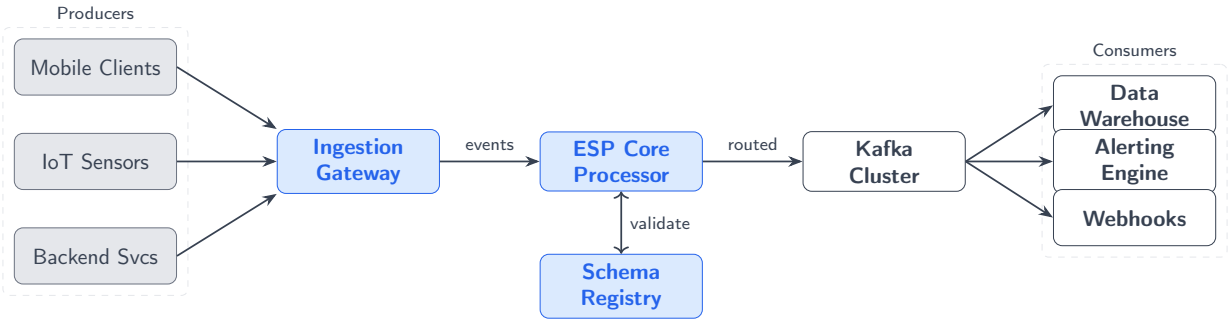
2.1 Functional

ID	Requirement
F-01	Accept events via REST (`/v1/ingest`) and gRPC (`EventIngester.Publish`).
F-02	Validate event envelopes against a registered Protobuf or JSON Schema.
F-03	Route events to one or more Kafka topics based on configurable filter expressions.
F-04	Apply transformation plugins (masking, enrichment, deduplication) in declared order.
F-05	Expose a web UI for pipeline authoring and live event-trace debugging.
F-06	Publish routing metrics (throughput, error rate, lag) to Prometheus.

2.2 Non-Functional

ID	Category	Requirement
NF-01	Performance	$\leq 10\,\text{ms}$ p99 ingest latency at 500 k ev/s.
NF-02	Availability	99.95% monthly uptime; graceful degradation under backpressure.
NF-03	Scalability	Horizontal autoscale 2–64 pods; stateless design.
NF-04	Security	mTLS between all services; PII fields masked before Kafka write.
NF-05	Observability	Structured JSON logs; distributed traces (OpenTelemetry).
NF-06	Compliance	GDPR: event TTL ≤ 90 days enforced at Kafka topic level.

3. System Architecture



The **Ingestion Gateway** terminates TLS, authenticates producers (API key / service account), and forwards validated raw events to the **ESP Core Processor** over an internal gRPC stream. The Core Processor evaluates declarative routing rules, applies transformation plugins, and publishes to **Kafka**. The **Schema Registry** (Confluent-compatible) is consulted on every event type; unknown schemas are rejected with `422 Unprocessable Entity`.

4. Detailed Design

4.1 Routing DSL

Pipelines are declared in YAML and compiled to a binary filter tree at startup:

```
pipeline:  iot-telemetry-v2
  filters:
    - field:  event_type
        eq:  sensor.reading
    transforms:  [mask_pii, add_region]
    route_to:  kafka://telemetry.raw
```

4.2 Transformation Plugin API

Plugins implement the `Transform` interface (Go):

```
type Transform interface {
    Name() string
    Apply(ctx context.Context, e *Event) (*Event, error)
}
```

Plugins are loaded as Go plugins at startup; hot-reload requires a rolling restart. Execution order follows declaration order in the pipeline YAML. A plugin returning an error triggers the dead-letter-queue (DLQ) path.

4.3 State & Persistence

ESP Core is intentionally stateless. Offset management is delegated to Kafka consumer groups. Pipeline configuration is stored in etcd with watch-based live propagation ($< 500\,\mathrm{ms}$ propagation SLO).

5. Interfaces & APIs

Endpoint	Method	Description
`/v1/ingest`	POST	Ingest a single event or batch (≤ 1000).
`/v1/pipelines`	GET	List all declared pipelines.
`/v1/pipelines/{id}`	PUT	Create or update a pipeline definition.
`/v1/pipelines/{id}`	DELETE	Drain and deactivate a pipeline.
`/v1/trace/{event_id}`	GET	Retrieve the routing trace for a past event.
`/metrics`	GET	Prometheus text exposition (internal only).
`/healthz`	GET	Liveness probe; `/readyz` for readiness.

gRPC: `EventIngester.Publish(stream EventEnvelope) returns (AckResponse)` – bidirectional streaming; backpressure via flow-control window.

6. Testing & Validation

- **Unit tests (Go):** $\geq 85\%$ line coverage enforced in CI; table-driven tests for all filter/transform combinations.
- **Integration tests:** Docker Compose environment with Kafka, etcd, and a mock Schema Registry; contract-tested via Pact.
- **Load test:** k6 scenario at 600 k ev/s for 10 min; p99 latency assertion $< 10\,\text{ms}$; automated via Argo Workflows post-deploy.
- **Chaos:** Gremlin fault injection – pod kill, 200 ms network delay, Kafka broker eviction – verifying at-least-once delivery is maintained.
- **Security scan:** `govulncheck` + Trivy image scan in every PR; high/critical CVEs block merge.

7. Risks

ID	Risk	Likelihood	Impact	Mitigation
R-01	Plugin ABI instability	Medium	High	Pin Go plugin build tags; integration-test every plugin in CI.
R-02	etcd watch storm	Low	Medium	Jitter propagation; circuit-breaker on watch re-subscribe.
R-03	Kafka lag spike	Medium	High	Auto-scale consumers; DLQ with exponential backoff replay.
R-04	Schema Registry outage	Low	High	Local LRU cache (5 min TTL); fail-open configurable per pipeline.
R-05	PII leakage in traces	Low	Critical	Trace data redacted by `mask_pii` before storage; field-level ACLs.