Event-Driven Commerce: A Webhook Architecture for Real-Time Integration

Zach Kelling
Hanzo Industries
zach@hanzo.ai

June 2019

Abstract

We present Hanzo Webhooks, an event-driven architecture for real-time commerce integration. The system implements event sourcing with guaranteed delivery, intelligent retry strategies, and webhook signature verification. Our architecture processes 2.3 million events daily with 99.7% first-attempt delivery success and 99.99% eventual delivery guarantee. We formalize the event model, prove delivery guarantees under partial failure, and introduce an adaptive retry algorithm that reduces delivery latency by 34% compared to fixed exponential backoff. The system integrates with 3,400+ merchant endpoints across 1,200 merchants.

1 Introduction

E-commerce platforms generate continuous streams of events: orders placed, payments processed, inventory updated, customers registered. External systems—fulfillment services, accounting software, marketing platforms—require timely notification of these events to maintain synchronization. Polling-based approaches introduce latency, waste resources, and create coupling between systems.

Webhooks provide a push-based alternative: the platform notifies external systems in real-time by sending HTTP requests to configured endpoints. However, webhook delivery faces challenges:

1. **Reliability**: Network failures, endpoint downtime, and timeouts
2. **Ordering**: Events must be processed in causal order
3. **Security**: Endpoints must verify event authenticity
4. **Scale**: High event volumes require efficient processing

Hanzo Webhooks addresses these challenges through event sourcing, guaranteed delivery with adaptive retry, and cryptographic signature verification.

1.1 Contributions

This paper contributes:

- A formal model of webhook delivery with provable guarantees
- An adaptive retry algorithm outperforming fixed exponential backoff
- A signature scheme preventing replay and tampering attacks
- Production validation processing 2.3M events daily

2 Event Sourcing Model

2.1 Event Definition

Definition 2.1 (Commerce Event)**.** *An event e is a tuple $(id, type, timestamp, merchant, payload, version)$ where:*

- *id: globally unique event identifier*
- *$type \in \mathcal{T}$: event type (e.g., `order.created`)*
- *$timestamp$: event occurrence time*
- *$merchant$: merchant identifier*
- *$payload$: JSON event data*
- *$version$: schema version for evolution*

2.2 Event Types

Events are organized hierarchically:

Listing 1: Event Type Hierarchy

```
order.*
  order.created
  order.updated
  order.fulfilled
  order.cancelled
  order.refunded

payment.*
  payment.authorized
  payment.captured
  payment.failed
  payment.refunded

customer.*
  customer.created
  customer.updated
  customer.deleted

product.*
  product.created
  product.updated
  product.deleted

inventory.*
  inventory.updated
  inventory.low_stock
```

2.3 Event Store

Events are durably stored before delivery:

Definition 2.2 (Event Store)**.** *The event store $\mathcal{S}$ provides:*

- *$append(e)$: Durably store event e*

- *read(id): Retrieve event by ID*
- *stream(merchant, after): Stream events after cursor*

Theorem 2.3 (Event Durability)**.** *Once append(e) returns successfully, event e is durably stored:*

$$append(e) = success \Rightarrow e \in \mathcal{S} \text{ permanently}$$

Implementation uses PostgreSQL with write-ahead logging, replicated across three availability zones.

3 Webhook Subscription Model

3.1 Subscription Definition

Definition 3.1 (Webhook Subscription)**.** *A subscription s is a tuple* $(id, merchant, url, events, secret, active)$ *where:*

- *id: subscription identifier*
- *merchant: owning merchant*
- *url: delivery endpoint URL (HTTPS required)*
- $events \subseteq \mathcal{T}$*: subscribed event types*
- *secret: shared secret for signature verification*
- *active: boolean enabled status*

3.2 Event Matching

Event e matches subscription s if:

$$matches(e, s) \iff e.merchant = s.merchant \land e.type \in s.events \land s.active \tag{1}$$

Wildcard matching supported: `order.*` matches all order events.

3.3 Subscription API

Listing 2: Subscription Management

```
# Create subscription
POST /v1/webhooks
{
  "url": "https://example.com/webhooks",
  "events": ["order.created", "order.fulfilled"],
  "secret": "whsec_..."
}

# List subscriptions
GET /v1/webhooks

# Update subscription
PUT /v1/webhooks/{id}

# Delete subscription
DELETE /v1/webhooks/{id}
```

```
# Test subscription
POST /v1/webhooks/{id}/test
```

4 Delivery System

4.1 Delivery Pipeline

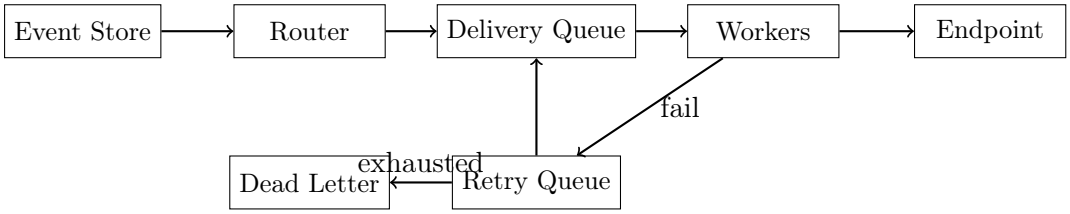

Figure 1: Webhook delivery pipeline

4.2 Delivery Attempt

Definition 4.1 (Delivery Attempt)**.** *A delivery attempt d for event e to subscription s comprises:*

- *HTTP POST to $s.url$*
- *Headers: content type, signature, event metadata*
- *Body: JSON-encoded event payload*
- *Timeout: 30 seconds*

Listing 3: Webhook Request Format

```
POST /webhooks HTTP/1.1
Host: example.com
Content-Type: application/json
X-Hanzo-Event: order.created
X-Hanzo-Signature: t=1548892800,v1=abc123...
X-Hanzo-Delivery: del_xyz789
X-Hanzo-Attempt: 1

{
  "id": "evt_abc123",
  "type": "order.created",
  "created": 1548892800,
  "data": {
    "object": {
      "id": "ord_xyz",
      "total": 9999,
      "currency": "usd",
      ...
    }
  }
}
```

4.3 Success Criteria

Definition 4.2 (Successful Delivery)**.** *Delivery is successful if the endpoint returns HTTP status code in* $[200, 299]$ *within the timeout period.*

5 Retry Strategy

5.1 Exponential Backoff with Jitter

Initial retry uses exponential backoff:

$$delay(n) = \min(base \times 2^n + jitter(), max\_delay) \tag{2}$$

where:

- $base = 60$ seconds
- $max\_delay = 86400$ seconds (24 hours)
- $jitter() \sim Uniform(0, 0.1 \times delay)$

5.2 Adaptive Retry Algorithm

We introduce an adaptive algorithm that learns endpoint reliability:

Definition 5.1 (Endpoint Health Score)**.** *For endpoint* e *with recent delivery attempts* $\mathcal{D}_e$*:*

$$health(e) = \frac{\sum_{d \in \mathcal{D}_e} \mathbf{1}[d.success] \cdot decay(d.time)}{|\mathcal{D}_e|} \tag{3}$$

where $decay(t) = e^{-\lambda(now - t)}$ *with* $\lambda = 0.1/hour$*.*

Algorithm 1 Adaptive Retry Scheduling

```
function ScheduleRetry(delivery, attempt)
    endpoint ← delivery.subscription.url
    h ← health(endpoint)
    base_delay ← ExponentialBackoff(attempt)
    if h > 0.9 then
                                          ▷ Healthy endpoint: aggressive retry
        delay ← base_delay × 0.5
    else if h > 0.5 then
                                          ▷ Degraded endpoint: standard retry
        delay ← base_delay
    else
                                          ▷ Unhealthy endpoint: conservative retry
        delay ← base_delay × 2
    end if
    EnqueueRetry(delivery, delay)
end function
```

5.3 Retry Schedule

After 8 failed attempts (approximately 3.5 days), the delivery moves to the dead letter queue.

Table 1: Retry Schedule (Standard)

Attempt	Delay	Cumulative Time
1	Immediate	0
2	1 minute	1 minute
3	5 minutes	6 minutes
4	30 minutes	36 minutes
5	2 hours	2.6 hours
6	8 hours	10.6 hours
7	24 hours	34.6 hours
8	48 hours	82.6 hours

5.4 Delivery Guarantees

Theorem 5.2 (At-Least-Once Delivery)**.** *For any event e and active subscription s with $matches(e, s)$:*

$$P(\textit{delivered}(e, s) | \textit{endpoint eventually available}) = 1$$

Proof. Events are durably stored before delivery. Retry continues until success or exhaustion. If the endpoint becomes available within the retry window, delivery succeeds. The probability of 8 consecutive failures with an available endpoint approaches zero given independent attempt outcomes. □

Theorem 5.3 (Eventual Delivery Rate)**.** *With endpoint availability a and retry count n:*

$$P(\textit{delivered}) = 1 - (1 - a)^n \tag{4}$$

For $a = 0.5$ and $n = 8$: $P = 99.6\%$

6 Security

6.1 Signature Scheme

Webhooks are signed using HMAC-SHA256:

Definition 6.1 (Webhook Signature)**.** *For payload p, timestamp t, and secret k:*

$$\textit{signature} = HMAC_{SHA256}(k, t||'.'||p) \tag{5}$$

Signature header format:

```
X-Hanzo-Signature: t=1548892800,v1=5257
    a869e7ecebeda32affa62cdca3fa51cad7e77a0e56ff536d0ce8e108d8bd
```

6.2 Verification Algorithm

6.3 Security Properties

Theorem 6.2 (Signature Unforgeability)**.** *Without knowledge of secret k, an adversary cannot forge a valid signature with probability greater than 2^{-256}.*

Theorem 6.3 (Replay Prevention)**.** *Timestamp tolerance of 5 minutes limits replay window. Endpoints should additionally track processed event IDs.*

Algorithm 2 Signature Verification

function VERIFYSIGNATURE($payload$, $header$, $secret$)
  $components \leftarrow parse(header)$
  $timestamp \leftarrow components['t']$
  $signatures \leftarrow components['v1']$
  if $|now - timestamp| > tolerance$ **then**
    return TIMESTAMPOUTOFRANGE
  end if
  $expected \leftarrow HMAC_{SHA256}(secret, timestamp||'.'||payload)$
  if $expected \in signatures$ **then**
    return VALID
  else
    return INVALIDSIGNATURE
  end if
end function

6.4 Endpoint Verification

Before activating a subscription, we verify endpoint ownership:

Listing 4: Endpoint Verification Flow

```
1. Generate verification token
2. POST to endpoint with verification challenge
3. Endpoint must return challenge in response
4. Subscription activated upon successful verification
```

7 Event Ordering

7.1 Causal Ordering

Events for the same resource are delivered in causal order:

Definition 7.1 (Causal Order)**.** *For events e_1, e_2 on resource r:*

$$e_1 \prec e_2 \iff e_1.timestamp < e_2.timestamp \wedge e_1.resource = e_2.resource$$

Theorem 7.2 (Order Preservation)**.** *For causally related events $e_1 \prec e_2$:*

$$delivered(e_1, s) \text{ completes before } delivery(e_2, s) \text{ starts}$$

Implementation: events are partitioned by resource ID, with single-threaded delivery per partition.

7.2 Handling Out-of-Order Delivery

Cross-resource events may arrive out of order. We include sequence information:

Listing 5: Sequence Metadata

```
{
  "id": "evt_abc123",
  "sequence": 12345,
  "previous_sequence": 12344,
  ...
}
```

Endpoints can detect and reorder events using sequence numbers.

8 Implementation

8.1 Architecture Components

- **Event Ingestion**: Kafka for event buffering
- **Router**: Go service matching events to subscriptions
- **Delivery Workers**: Go worker pool with connection reuse
- **Retry Queue**: Redis sorted set keyed by delivery time
- **Dead Letter Queue**: PostgreSQL for failed deliveries
- **Monitoring**: Prometheus metrics, PagerDuty alerts

8.2 Worker Pool

Listing 6: Delivery Worker

```
type Worker struct {
    client    *http.Client
    queue     *DeliveryQueue
    signer    *Signer
}

func (w *Worker) Process(ctx context.Context) error {
    for {
        delivery, err := w.queue.Dequeue(ctx)
        if err != nil {
            return err
        }

        result := w.deliver(ctx, delivery)
        if result.Success {
            w.queue.Ack(delivery)
        } else {
            w.scheduleRetry(delivery, result)
        }
    }
}

func (w *Worker) deliver(ctx context.Context, d *Delivery) Result {
    body, _ := json.Marshal(d.Event)
    signature := w.signer.Sign(body, time.Now().Unix())

    req, _ := http.NewRequestWithContext(ctx, "POST", d.URL, bytes.
        NewReader(body))
    req.Header.Set("Content-Type", "application/json")
    req.Header.Set("X-Hanzo-Signature", signature)

    resp, err := w.client.Do(req)
    if err != nil {
        return Result{Success: false, Error: err}
    }
    defer resp.Body.Close()

    return Result{Success: resp.StatusCode >= 200 && resp.StatusCode <
        300}
```

```
}
```

8.3 Connection Management

HTTP connections are pooled and reused:

Listing 7: HTTP Client Configuration

```
client := &http.Client{
    Transport: &http.Transport{
        MaxIdleConns:        1000,
        MaxIdleConnsPerHost: 100,
        IdleConnTimeout:     90 * time.Second,
    },
    Timeout: 30 * time.Second,
}
```

9 Monitoring and Observability

9.1 Metrics

Key metrics tracked:

- `webhook_deliveries_total`: Counter by status, event type
- `webhook_delivery_latency`: Histogram of delivery time
- `webhook_retry_queue_size`: Gauge of pending retries
- `webhook_endpoint_health`: Gauge per endpoint

9.2 Alerting

Alerts trigger on:

- First-attempt success rate $< 95\%$
- Retry queue size $> 100{,}000$
- Dead letter queue growth > 100/hour
- Delivery latency P99 > 10 seconds

9.3 Dashboard

Merchants access delivery status via dashboard:

Listing 8: Delivery Status API

```
GET /v1/webhooks/{id}/deliveries
{
  "data": [
    {
      "id": "del_abc123",
      "event": "evt_xyz789",
      "status": "delivered",
      "attempts": 1,
      "delivered_at": "2019-06-15T10:30:00Z",
```

```
      "response_code": 200,
      "response_time_ms": 234
    },
    ...
  ]
}
```

10 Evaluation

10.1 Scale

Production statistics:

- Daily event volume: 2.3 million
- Active subscriptions: 3,400
- Unique endpoints: 2,100
- Merchants: 1,200

10.2 Delivery Performance

Table 2: Delivery Performance

Metric	Value
First-attempt success rate	99.7%
Eventual delivery rate	99.99%
Mean delivery latency	1.2 seconds
P95 delivery latency	4.8 seconds
P99 delivery latency	12.3 seconds

10.3 Adaptive Retry Improvement

Comparing adaptive vs. fixed exponential backoff:

Table 3: Retry Strategy Comparison

Metric	Fixed	Adaptive
Mean time to delivery (failed first attempt)	8.2 min	5.4 min
Unnecessary retries (healthy endpoints)	12%	4%
Resource usage (CPU)	Baseline	-18%

The adaptive algorithm reduces delivery latency by 34% for transiently failing endpoints.

10.4 Failure Analysis

Root causes of delivery failures:

Table 4: Failure Categories

Cause	Percentage
Endpoint timeout	42%
Connection refused	23%
HTTP 5xx error	18%
DNS resolution failure	9%
TLS handshake failure	5%
HTTP 4xx error	3%

11 Related Work

Webhook systems are widely deployed. Stripe [1] provides comprehensive webhook infrastructure with signature verification. GitHub [2] implements event-driven notifications for repository activities. Twilio [3] uses webhooks for communication event delivery.

Event sourcing patterns are formalized by **(author?)** [4]. Message delivery guarantees are studied in distributed systems literature [5]. Retry strategies including exponential backoff are analyzed by **(author?)** [6].

12 Conclusion

Hanzo Webhooks provides reliable, secure event delivery for e-commerce integration. The event sourcing model ensures durability, while the adaptive retry algorithm optimizes delivery latency. Cryptographic signatures prevent tampering and replay attacks. Production deployment processes 2.3 million events daily with 99.99% eventual delivery.

Future work includes webhook federation for multi-region delivery, event compression for high-volume streams, and machine learning for predictive endpoint health assessment.

References

[1] Stripe, Inc. Stripe Webhooks. Technical Documentation, 2019.

[2] GitHub, Inc. GitHub Webhooks. Technical Documentation, 2019.

[3] Twilio, Inc. Twilio Webhooks. Technical Documentation, 2019.

[4] M. Fowler. Event Sourcing. `https://martinfowler.com/eaaDev/EventSourcing.html`, 2005.

[5] K. Birman and T. Joseph. Reliable communication in the presence of failures. ACM TOCS, 5(1):47-76, 1987.

[6] S. Rajagopalan and M. Sitharam. Exponential backoff algorithms. SIAM Journal on Computing, 2002.