



MHI Gateway

Webhook Integration Guide

Secure delivery status processing for SMS API v1

Contract version 1.0.0-rc1
Document version 1.0
Publication date 28 July 2026
Classification Public integration documentation

MICRO HEALTH INITIATIVE

Document Control

Field	Value
Document title	MHI Gateway Webhook Integration Guide
Document identifier	MHI-SMS-WEBHOOK-GUIDE-V1
Contract version	1.0.0-rc1
Document version	1.0
Status	Milestone 3
Audience	Application developers, platform engineers, security engineers, and operators
Authoritative specification	docs/api/openapi.yaml
Companion API reference	API Reference
Companion integration guide	Developer Integration Guide

This implementation guide explains how to receive and process the frozen MHI Gateway delivery webhook securely. If this document and OpenAPI differ, OpenAPI governs. The [API Reference](#) is the field-level reference; this document focuses on receiver design and operation.

Revision History

Version	Date	Status	Description
1.0	28 July 2026	Initial	Official webhook implementation guide for contract 1.0.0-rc1

Table of Contents

Document Control	2
Revision History	2
Table of Contents	3
1 Introduction	5
1.1 Scope	5
1.2 Security objective	6
2 Webhook Overview	6
2.1 When a webhook is sent	6
2.2 Delivery guarantees and limits	6
3 Delivery Lifecycle	7
3.1 Status lifecycle	7
3.2 Webhook delivery flow	7
3.3 Recovery model	7
4 Webhook Configuration	7
4.1 Create or replace	7
4.2 Configuration fields	8
4.3 Read the safe configuration	8
4.4 Secret rotation	8
5 Webhook Payload	8
5.1 Field reference	9
5.2 Two different timestamps	9
5.3 Raw body requirement	9
6 Request Headers	9
7 Signature Verification	10
7.1 Signature input	10
7.2 Verification order	10
7.3 Constant-time comparison	10
7.4 PHP example	11
7.5 Node.js example	11
7.6 Python example	12
7.7 Java example	12
7.8 Go example	13
7.9 C# example	13
8 Timestamp Validation	13

8.1 Validation requirements	13
8.2 Clock synchronization	14
8.3 Boundary behavior	14
9 Replay Protection	14
9.1 Replay identity	14
9.2 Atomic acceptance	14
9.3 Retention	14
10 Secure Processing Pipeline	14
10.1 Required order	15
10.2 Queue-first architecture	15
10.3 Payload validation	15
11 Error Handling	16
11.1 Recommended receiver decisions	16
11.2 Failure recovery	16
11.3 Safe error responses	16
12 Idempotent Processing	16
12.1 Processing key	16
12.2 Side effects	17
12.3 Unknown messages	17
13 Security Best Practices	17
13.1 HTTPS	17
13.2 Secret storage and least privilege	17
13.3 Timing attacks	17
13.4 Replay attacks	17
13.5 Secure logging	17
13.6 Network and runtime controls	17
14 Production Deployment	18
14.1 Production architecture	18
14.2 Deployment checklist	18
14.3 Monitoring	18
14.4 Alerting	19
14.5 Operational recovery	19
15 Troubleshooting	19
15.1 Every signature fails	19
15.2 Signatures fail after rotation	20
15.3 Valid-looking requests fail timestamp validation	20
15.4 Duplicate processing occurs	20
15.5 Webhook events appear to be missing	20

15.6 Receiver returns 2xx but work is lost	20
15.7 Receiver times out	20
16 Frequently Asked Questions	20
16.1 Which events are supported?	20
16.2 Are non-terminal updates sent?	20
16.3 Will a failed webhook be retried?	20
16.4 What acknowledges an event?	20
16.5 Can several webhook URLs be configured?	20
16.6 Can the secret be retrieved?	21
16.7 How is a secret rotated?	21
16.8 Must JSON be parsed before signature verification?	21
16.9 Is timestamp validation sufficient replay protection?	21
16.10 Why is idempotent processing still required?	21
16.11 How are missed events recovered?	21
17 Appendices	21
Appendix A: Verification checklist	21
Appendix B: Frozen webhook contract	22
Appendix C: Replay key pseudocode	22
Appendix D: Payload examples by terminal status	22
Delivered	22
Failed	22
Expired	22
Rejected	23
Appendix E: Related resources	23

1 Introduction

A webhook is an HTTPS request sent by one system to a receiver URL when a defined event occurs. For MHI Gateway, the event reports that an SMS message has first reached a terminal public status.

The receiver is responsible for:

- accepting the exact raw request body;
- validating the timestamp and HMAC signature before parsing JSON;
- rejecting replayed requests;
- handing verified events to durable processing before acknowledgment;
- applying the event idempotently;
- monitoring delivery, verification, queueing, and reconciliation.

This guide does not introduce webhook behavior beyond the frozen contract. Field definitions and endpoint response matrices are in the [API Reference](#).

1.1 Scope

This guide covers the single tenant delivery webhook configured through:

```
PUT /api/v1/webhook
GET /api/v1/webhook
```

The contract does not define multiple receiver URLs, event filters, a delete operation, a separate secret-rotation endpoint, or retry scheduling.

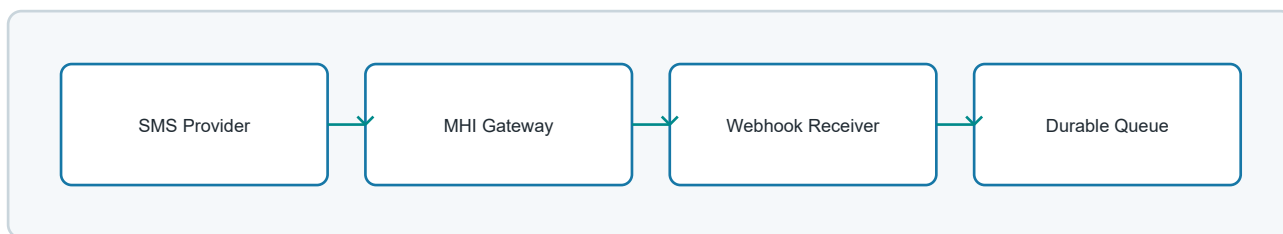
1.2 Security objective

The receiver must establish that:

1. the request timestamp is acceptable;
2. the signature matches the exact timestamp and raw body;
3. the request has not already been accepted;
4. the parsed event contains only approved fields and values;
5. business processing cannot apply the same outcome twice.

2 Webhook Overview

MHI Gateway sends an HTTP POST to the configured URL when a message first reaches **delivered**, **failed**, **expired**, or **rejected**.



2.1 When a webhook is sent

The gateway emits one immutable event only when a message first reaches one of the four terminal public statuses. It emits no event for:

queued
submitted
unknown

Duplicate delivery-receipt evidence and repeated observation of the same terminal outcome do not create another gateway event or delivery.

2.2 Delivery guarantees and limits

Behavior	Frozen contract
Event trigger	First transition to a terminal public status
Outbound timeout	10 seconds
Delivery attempts	Exactly one
Automatic retries	None
Retry scheduling	Not implemented in RC1
Acknowledgment	Any 2xx
Failure effect	Never rolls back persisted message delivery
Attempt records	Retained by the gateway for 90 days

Because the gateway does not retry a failed attempt, receiver availability and status reconciliation are operational requirements. A receiver must not depend on redelivery for recovery.

3 Delivery Lifecycle

3.1 Status lifecycle

The closed public status vocabulary is:

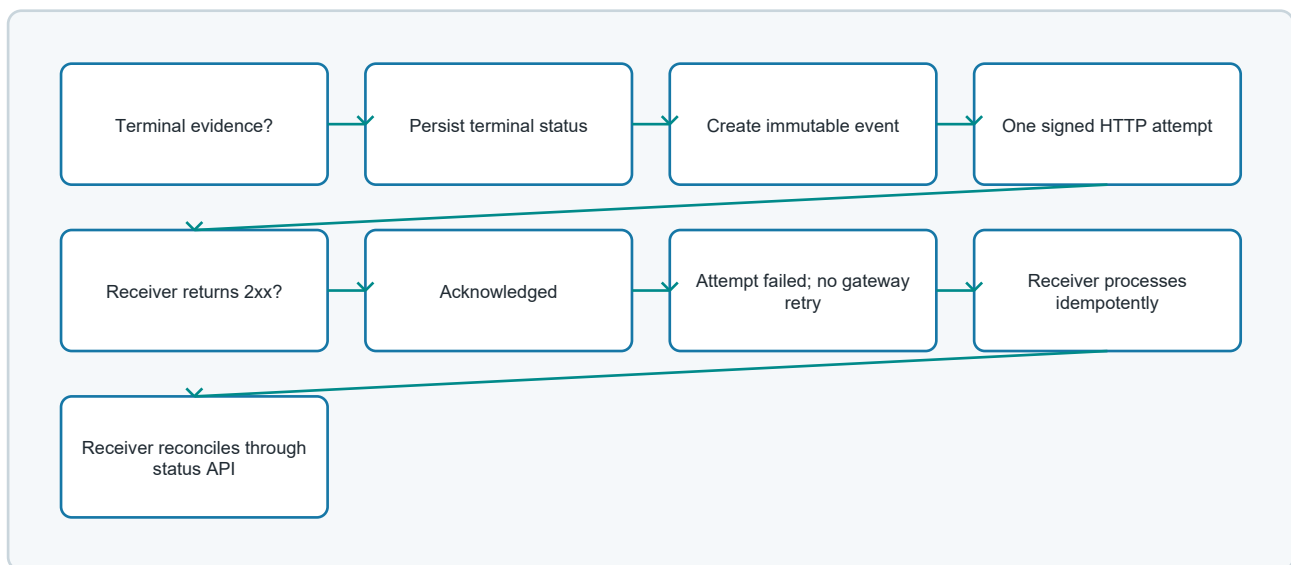
queued, submitted, delivered, failed, expired, rejected, unknown

Only terminal statuses produce delivery webhook events.

Status	Terminal	Webhook emitted
queued	No	No
submitted	No	No
unknown	No	No
delivered	Yes	Yes
failed	Yes	Yes
expired	Yes	Yes
rejected	Yes	Yes

Terminal states are immutable. Contradictory late terminal evidence does not rewrite public history.

3.2 Webhook delivery flow



3.3 Recovery model

Webhook delivery failure does not affect the persisted message outcome. Applications should reconcile non-terminal or locally incomplete records using:

GET /api/v1/messages/{message_id}/status

Reconciliation is the recovery path for an unavailable receiver, a failed durable handoff, or a non-2xx response because RC1 does not retry webhooks.

4 Webhook Configuration

4.1 Create or replace

PUT /api/v1/webhook creates the single tenant configuration when absent and completely replaces it when present.

```
curl --request PUT 'https://<API_HOST>/api/v1/webhook' \
--header 'Authorization: Bearer <API_TOKEN>' \
--header 'Accept: application/json' \
--header 'Content-Type: application/json' \
--header 'X-Correlation-ID: webhook-config-1' \
--data '{
  "url": "https://example.com/webhooks/mhi-gateway",
  "secret": "<AT_LEAST_32_RANDOM_BYTES>",
  "enabled": true
}'
```

4.2 Configuration fields

Field	Type	Required	Contract
url	string URI	Yes	HTTPS is mandatory in production
secret	string	Yes	At least 32 cryptographically random bytes; write-only
enabled	boolean	Yes	Enables or disables delivery

Unknown fields are rejected. The secret is never returned after configuration.

4.3 Read the safe configuration

GET /api/v1/webhook returns:

```
{
  "url": "https://example.com/webhooks/mhi-gateway",
  "enabled": true,
  "created_at": "2026-07-19T12:00:00.000000Z",
  "updated_at": "2026-07-19T12:00:00.000000Z"
}
```

The response does not contain secret.

The safe read model contains only url, enabled, created_at, and updated_at.

4.4 Secret rotation

Replacing the configuration activates the newest secret immediately. The previous secret remains valid for receiver-side verification for exactly 24 hours. After that overlap, only the newest secret is valid.

During the overlap:

1. attempt verification with the active secret;
2. if it does not match, attempt verification with the previous secret;
3. apply the same timestamp and replay controls regardless of which secret matched;
4. remove the previous secret from the receiver after 24 hours.

Do not expose either secret in configuration responses, logs, metrics, traces, or error bodies.

5 Webhook Payload

The request body uses the OpenAPI DeliveryWebhookEvent schema.

```
{
  "message_id": "01JEXAMPLE0000000000000000",
  "status": "delivered",
  "client_reference": "order-1042",
  "delivered_at": "2026-07-19T12:30:44.987654Z",
  "timestamp": "2026-07-19T12:30:45.123456Z"
}
```

5.1 Field reference

Field	Type	Required	Nullable	Constraints	Meaning
message_id	string	Yes	No	^[0-9A-HJKMNP-TV-Z]{26}\$	Public message identifier
status	string	Yes	No	delivered, failed, expired, or rejected	Terminal public status
client_reference	string	No	No	No additional OpenAPI constraint	Client metadata when available
delivered_at	string date-time	No	No	UTC RFC 3339	Delivery time when present
timestamp	string date-time	Yes	No	UTC RFC 3339	Event timestamp

No additional payload properties are approved. Provider identifiers, SMPP sequence numbers, internal database identifiers, persistence enums, stack traces, and provider metadata are forbidden.

5.2 Two different timestamps

X-Webhook-Timestamp is an integer number of Unix seconds used in signature and freshness verification. Body timestamp is the event's UTC RFC 3339 timestamp. They have different formats and purposes.

delivered_at is optional and must not be assumed to exist for every terminal status or every event.

5.3 Raw body requirement

The HMAC covers the exact request-body bytes. Verification must use the raw body as received, before JSON parsing, reformatting, character conversion, or object serialization.

These inputs are not equivalent for signature purposes:

```
{"status":"delivered","message_id":"01JEXAMPLE0000000000000000","timestamp":"2026-07-19T12:30:45.123456Z"}

{
  "message_id": "01JEXAMPLE0000000000000000",
  "status": "delivered",
  "timestamp": "2026-07-19T12:30:45.123456Z"
}
```

They contain comparable JSON values but different byte sequences.

6 Request Headers

Every webhook request includes:

Header	Type	Required	Validation
Content-Type	string	Yes	application/json
X-Webhook-Timestamp	integer int64	Yes	Unix seconds
X-Webhook-Signature	string	Yes	^v1=[0-9a-f]{64}\$

Example:

```
POST /webhooks/mhi-gateway HTTP/1.1
Host: example.com
Content-Type: application/json
X-Webhook-Timestamp: 1784464245
X-Webhook-Signature: v1=0123456789abcdef0123456789abcdef0123456789abcdef0123456789abcdef
```

The signature header contains the lowercase hexadecimal HMAC-SHA256 digest, prefixed with v1=.

7 Signature Verification

7.1 Signature input

Build the input as the exact byte sequence:

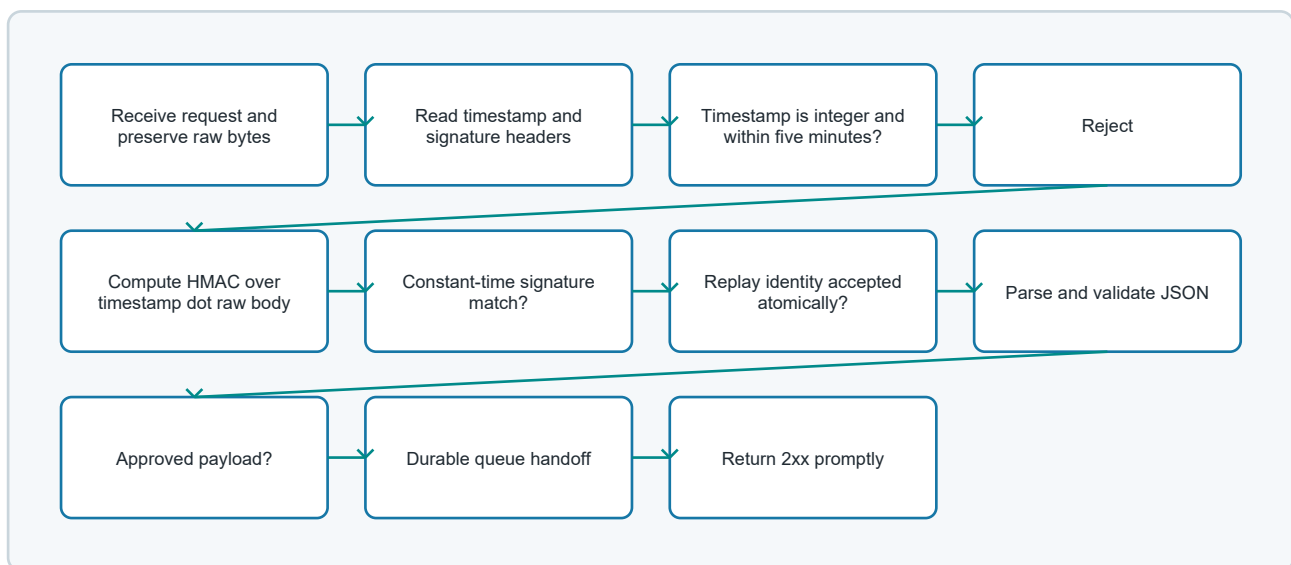
```
<timestamp>.<raw-request-body>
```

The timestamp is the exact X-Webhook-Timestamp header value. The separator is one ASCII full stop (.). The body is the unmodified request-body byte array.

Calculate:

```
digest = HMAC-SHA256(secret, timestamp + "." + raw_body)
expected = "v1=" + lowercase_hex(digest)
```

7.2 Verification order



Do not parse JSON before timestamp and signature verification.

7.3 Constant-time comparison

A normal string comparison may return as soon as it finds a differing character. The elapsed time can reveal information about how much of a supplied signature matches the expected value. Use the platform's timing-safe comparison primitive:

Language	Primitive
PHP	<code>hash_equals</code>
Node.js	<code>crypto.timingSafeEqual</code> after equal-length check
Python	<code>hmac.compare_digest</code>
Java	<code>MessageDigest.isEqual</code>
Go	<code>hmac.Equal</code>
C#	<code>CryptographicOperations.FixedTimeEquals</code>

7.4 PHP example

```
<?php

declare(strict_types=1);

$secret = getenv('WEBHOOK_SECRET') ?: '';
$timestamp = $_SERVER['HTTP_X_WEBHOOK_TIMESTAMP'] ?? '';
$signature = $_SERVER['HTTP_X_WEBHOOK_SIGNATURE'] ?? '';
$rawBody = file_get_contents('php://input');

if ($secret === '' || $rawBody === false || ! ctype_digit($timestamp)
    || abs(time() - (int) $timestamp) > 300) {
    http_response_code(401);
    exit;
}

$expected = 'v1=' . hash_hmac(
    'sha256',
    $timestamp . '.' . $rawBody,
    $secret,
);

if (! hash_equals($expected, $signature)) {
    http_response_code(401);
    exit;
}

$replayKey = hash('sha256', $timestamp . '.' . $signature . '.' . $rawBody);
// Atomically reserve $replayKey in a shared store before parsing.

$event = json_decode($rawBody, true, 512, JSON_THROW_ON_ERROR);
// Durably enqueue $event before acknowledging.
http_response_code(204);
```

7.5 Node.js example

```
const crypto = require('node:crypto');

function verifyWebhook(rawBody, headers, secret, nowSeconds) {
    const timestamp = headers['x-webhook-timestamp'] || '';
    const signature = headers['x-webhook-signature'] || '';
    const seconds = Number(timestamp);

    if (!secret || !Number.isInteger(seconds)
        || Math.abs(nowSeconds - seconds) > 300) {
        return false;
    }

    const digest = crypto.createHmac('sha256', secret)
        .update(timestamp)
        .update('.')
        .update(rawBody)
        .digest('hex');
    const expected = Buffer.from(`v1=${digest}`);
    const supplied = Buffer.from(signature);

    return expected.length === supplied.length
        && crypto.timingSafeEqual(expected, supplied);
}

// Preserve rawBody as a Buffer. Reserve a replay key atomically,
// parse JSON, durably enqueue the event, then return 204.
```

7.6 Python example

```
import hashlib
import hmac
import time

def verify_webhook(
    raw_body: bytes,
    timestamp: str,
    signature: str,
    secret: bytes,
) -> bool:
    try:
        timestamp_number = int(timestamp)
    except ValueError:
        return False

    if not secret or abs(time.time() - timestamp_number) > 300:
        return False

    expected = "v1=" + hmac.new(
        secret,
        timestamp.encode() + b"." + raw_body,
        hashlib.sha256,
    ).hexdigest()
    return hmac.compare_digest(expected, signature)

# Reserve a replay key atomically, parse raw_body as JSON, durably
# enqueue the event, and acknowledge with 204.
```

7.7 Java example

```
import java.nio.charset.StandardCharsets;
import java.security.MessageDigest;
import java.time.Instant;
import javax.crypto.Mac;
import javax.crypto.spec.SecretKeySpec;

static boolean verify(
    byte[] rawBody,
    String timestamp,
    String signature,
    byte[] secret
) throws Exception {
    long seconds;
    try {
        seconds = Long.parseLong(timestamp);
    } catch (NumberFormatException exception) {
        return false;
    }
    if (secret.length == 0
        || Math.abs(Instant.now().getEpochSecond() - seconds) > 300) {
        return false;
    }

    Mac mac = Mac.getInstance("HmacSHA256");
    mac.init(new SecretKeySpec(secret, "HmacSHA256"));
    mac.update(timestamp.getBytes(StandardCharsets.US_ASCII));
    mac.update((byte) '.');
    byte[] digest = mac.doFinal(rawBody);
    String expected = "v1=" + java.util.HexFormat.of().formatHex(digest);

    return MessageDigest.isEqual(
        expected.getBytes(StandardCharsets.US_ASCII),
        signature.getBytes(StandardCharsets.US_ASCII)
    );
}
```

After verification, reserve the replay identity atomically, validate the JSON payload, durably enqueue it, and return 2xx.

7.8 Go example

```
func verifyWebhook(
    rawBody []byte,
    timestamp string,
    signature string,
    secret []byte,
    now time.Time,
) bool {
    seconds, err := strconv.ParseInt(timestamp, 10, 64)
    if err != nil || len(secret) == 0 ||
        abs64(now.Unix()-seconds) > 300 {
        return false
    }

    mac := hmac.New(sha256.New, secret)
    mac.Write([]byte(timestamp))
    mac.Write([]byte{'.'})
    mac.Write(rawBody)
    expected := []byte("v1=" + hex.EncodeToString(mac.Sum(nil)))

    return hmac.Equal(expected, []byte(signature))
}

func abs64(value int64) int64 {
    if value < 0 {
        return -value
    }
    return value
}
```

Use the raw request bytes. After verification, reserve a replay identity in an atomic shared store, validate JSON, durably enqueue the event, and return 2xx.

7.9 C# example

```
using System.Security.Cryptography;
using System.Text;

static bool VerifyWebhook(
    byte[] rawBody,
    string timestamp,
    string signature,
    byte[] secret,
    DateTimeOffset now)
{
    if (secret.Length == 0
        || !long.TryParse(timestamp, out long seconds)
        || Math.Abs(now.ToUnixTimeSeconds() - seconds) > 300) {
        return false;
    }

    byte[] prefix = Encoding.ASCII.GetBytes(timestamp + ".");
    byte[] input = new byte[prefix.Length + rawBody.Length];
    Buffer.BlockCopy(prefix, 0, input, 0, prefix.Length);
    Buffer.BlockCopy(rawBody, 0, input, prefix.Length, rawBody.Length);

    byte[] digest = HMACSHA256.HashData(secret, input);
    byte[] expected = Encoding.ASCII.GetBytes(
        "v1=" + Convert.ToHexString(digest).ToLowerInvariant());
    byte[] supplied = Encoding.ASCII.GetBytes(signature);

    return expected.Length == supplied.Length
        && CryptographicOperations.FixedTimeEquals(expected, supplied);
}
```

Read the HTTP body as bytes. Reserve the replay identity atomically, validate the JSON payload, durably enqueue it, and return 2xx.

8 Timestamp Validation

X-Webhook-Timestamp is Unix time in seconds and is part of the signature input. Reject the request when the timestamp is more than five minutes before or after the receiver's current time.

```
absolute(receiver_unix_seconds - webhook_unix_seconds) <= 300
```

8.1 Validation requirements

- Require an integer value.

- Use seconds, not milliseconds.
- Compare with the receiver's current trusted clock.
- Reject timestamps outside the allowed window before parsing JSON.
- Do not substitute body timestamp for X-Webhook-Timestamp.

8.2 Clock synchronization

Keep production receiver clocks synchronized. A clock offset greater than the allowed window causes valid requests to be rejected. Monitor clock health on every webhook receiver instance.

8.3 Boundary behavior

The approved receiver examples accept timestamps whose absolute difference is at most 300 seconds and reject values greater than 300 seconds.

9 Replay Protection

A valid signature proves knowledge of the secret but does not prove that the signed request is new. An attacker or intermediary that captures a valid request could submit the same signed bytes again within the accepted timestamp window.

9.1 Replay identity

The approved receiver examples derive a stable digest from:

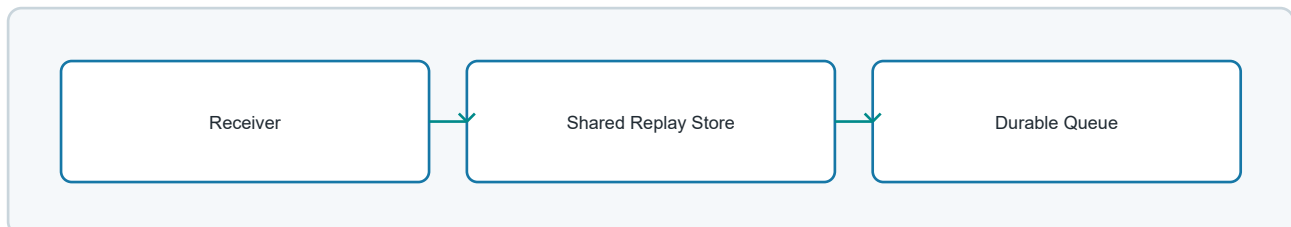
```
timestamp + "." + signature + "." + raw_body
```

For example:

```
replay_key = SHA-256(timestamp + "." + signature + "." + raw_body)
```

The replay key is receiver-side metadata. It is not a webhook payload field.

9.2 Atomic acceptance



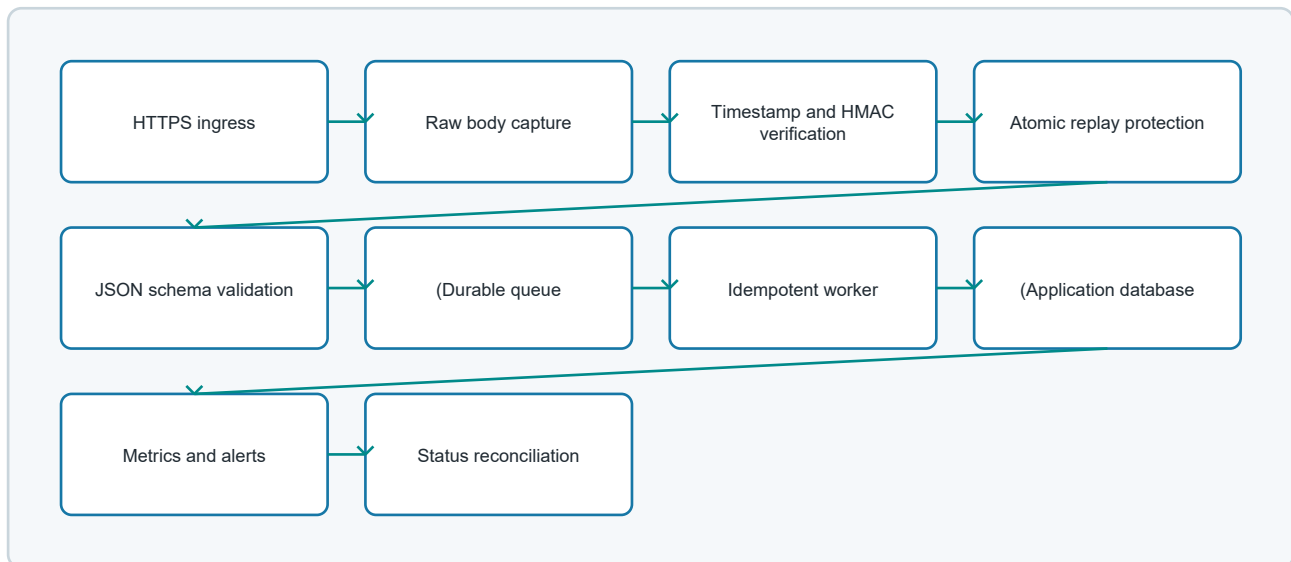
The check and insert must be one atomic operation in a shared store. A process-local set or temporary file is suitable only for demonstration because it does not coordinate multiple instances reliably.

9.3 Retention

Use bounded retention for replay identities. The contract and approved examples do not define a specific receiver-store retention duration. Do not claim an undocumented gateway guarantee from the receiver's retention policy.

10 Secure Processing Pipeline

Keep verification at the edge and business processing behind durable storage.



10.1 Required order

1. Terminate HTTPS.
2. Read the body once as raw bytes.
3. Read both webhook headers.
4. Validate timestamp syntax and freshness.
5. Compute HMAC over the exact signed input.
6. Compare signatures in constant time.
7. Reserve the replay identity atomically.
8. parse JSON;
9. validate the approved payload fields and values;
10. durably enqueue or store the event;
11. return any 2xx promptly;
12. process the event idempotently outside the request path.

10.2 Queue-first architecture

The receiver should acknowledge only after the event is durably handed off. Do not wait for downstream business workflows inside the 10-second gateway timeout.

If durable handoff fails, return a non-2xx response. The gateway will record the failed attempt but will not retry it; recover through status reconciliation.

10.3 Payload validation

After cryptographic verification:

- require `message_id`, `status`, and `timestamp`;
- accept only the four terminal status values;
- treat `client_reference` and `delivered_at` as optional;
- reject unapproved fields according to the frozen schema;
- keep public identifiers separate from internal database identifiers.

11 Error Handling

The webhook receiver defines its own HTTP error responses. The MHI Gateway contract defines only that any 2xx acknowledges the event.

11.1 Recommended receiver decisions

Condition	Receiver action	Gateway behavior
Missing or malformed timestamp	Reject without parsing body	Attempt fails; no retry
Timestamp outside five minutes	Reject without parsing body	Attempt fails; no retry
Missing or invalid signature	Reject without parsing body	Attempt fails; no retry
Replay identity already exists	Reject or safely suppress without reprocessing	No additional gateway retry
Invalid JSON after verification	Reject	Attempt fails; no retry
Unapproved payload	Reject	Attempt fails; no retry
Durable handoff unavailable	Return non-2xx	Attempt fails; no retry
Durable handoff succeeds	Return any 2xx promptly	Attempt acknowledged

The approved sample receivers use 401 for failed security checks and 400 for invalid JSON; one sample uses 409 for a replay. These are receiver implementation choices, not response codes defined by the webhook contract.

11.2 Failure recovery

Because automatic retry is absent:

- alert on verification, replay, queue, and non-2xx failures;
- restore the receiver or queue;
- reconcile affected messages through the status API;
- apply recovered terminal outcomes idempotently;
- retain safe operational evidence for incident investigation.

11.3 Safe error responses

Do not return the expected signature, secret, raw body, parsed message content, stack trace, provider evidence, or internal identifiers. A minimal empty error response is sufficient.

12 Idempotent Processing

Replay protection protects the HTTP ingress. Idempotent business processing protects downstream state.

12.1 Processing key

Use the public `message_id` with the immutable terminal status when applying the event. The contract emits only the first terminal outcome and does not rewrite it with a conflicting terminal state.

An idempotent worker should:

1. load the local message record by public `message_id` within the correct tenant context;
2. check whether the same terminal outcome was already applied;
3. if already applied, complete without repeating side effects;
4. otherwise persist the terminal outcome and an application processing record

atomically;

5. trigger downstream work through an idempotent handoff.

12.2 Side effects

Protect notifications, order updates, analytics, and other side effects with their own idempotency boundaries. Do not assume that a queue guarantees exactly one worker execution.

12.3 Unknown messages

If a verified event contains a `message_id` not yet available locally, preserve the verified event for controlled reconciliation. Do not invent a message resource or discard the event without operational evidence.

13 Security Best Practices

13.1 HTTPS

Production webhook URLs must use HTTPS. Validate certificates and restrict ingress to the receiver path and method required by the integration.

13.2 Secret storage and least privilege

- Generate at least 32 cryptographically random bytes.
- Store current and overlapping previous secrets in a secret manager.
- Grant secret access only to the verification component.
- Do not expose secrets to workers that process already verified events.
- Remove the previous secret after the 24-hour overlap.

13.3 Timing attacks

Use constant-time comparison for signatures. In Node.js and C#, check byte-array lengths before calling a fixed-time primitive that requires equal lengths.

13.4 Replay attacks

Enforce both the five-minute timestamp window and an atomic shared replay store. Timestamp validation alone does not prevent reuse within the accepted window.

13.5 Secure logging

Never log:

- webhook secrets;
- `X-Webhook-Signature`;
- raw request bodies;
- message content;
- provider identifiers or internal persistence details;
- stack traces in public responses.

Safe operational fields may include a locally generated request identifier, public `message_id` after successful verification and parsing, terminal status, verification outcome category, queue outcome, elapsed time, and receiver instance identifier.

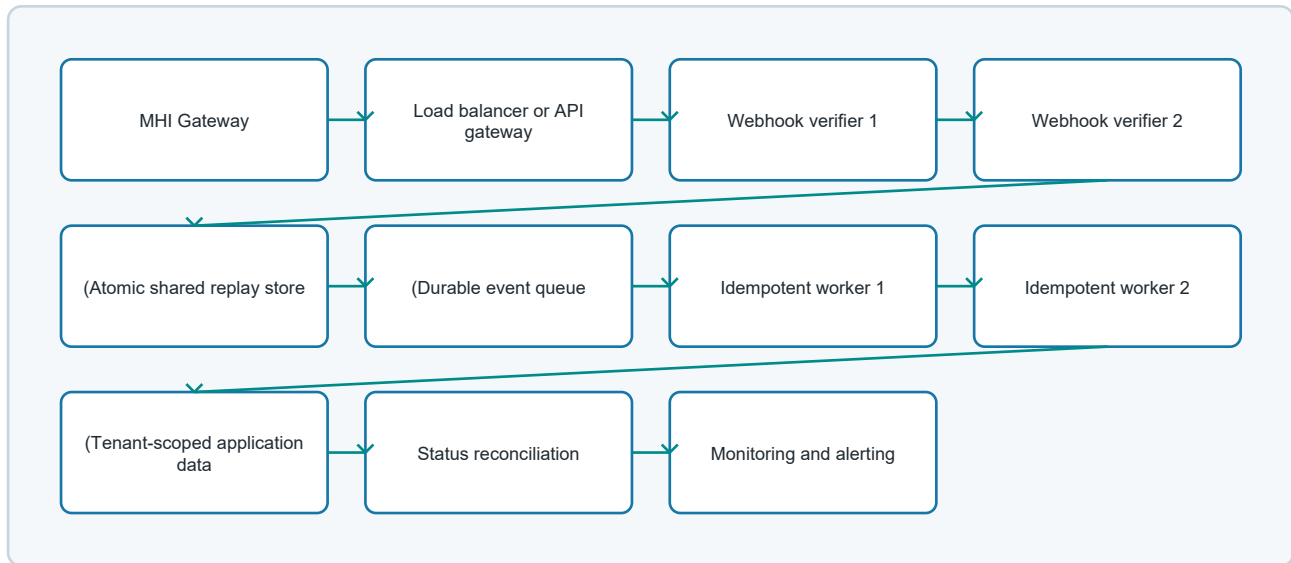
13.6 Network and runtime controls

- Keep receiver clocks synchronized.

- Bound body size to the approved event schema needs.
- Apply request timeouts below the 10-second outbound timeout.
- Isolate the public verification edge from business workers.
- Encrypt queue and database traffic where applicable.
- Restrict queue producers and consumers by least privilege.

14 Production Deployment

14.1 Production architecture



14.2 Deployment checklist

- The configured production URL uses HTTPS.
- Current and previous secrets are available only to verifiers.
- Raw-body access is tested in the deployed framework.
- The exact HMAC input is verified with known test bytes.
- Clock synchronization is monitored.
- Replay acceptance is atomic across all receiver instances.
- Payload schema validation rejects unknown fields and statuses.
- Durable handoff completes before acknowledgment.
- Business processing is idempotent.
- Request handling completes within the 10-second gateway timeout.
- Status reconciliation is operational.
- Logs and traces redact signatures, secrets, and bodies.

14.3 Monitoring

Monitor:

- request count and acknowledgment count;
- timestamp rejection count;
- signature rejection count;

- replay rejection count;
- JSON and schema validation failures;
- durable handoff failures and latency;
- queue age and depth;
- worker success, duplicate suppression, and failure;
- clock offset;
- status reconciliation backlog and failure.

14.4 Alerting

Alert on sustained or material:

- absence of expected webhook traffic;
- increases in timestamp or signature failures;
- replay detections;
- any durable handoff outage;
- response latency approaching 10 seconds;
- queue backlog growth;
- worker processing failures;
- reconciliation gaps.

The permitted sources do not define numeric alert thresholds. Set them from the receiver's capacity and operational objectives.

14.5 Operational recovery

When delivery or processing fails:

1. preserve safe receiver and queue evidence;
2. restore ingress, replay storage, queueing, or workers;
3. identify locally incomplete messages;
4. retrieve current status through the compact status endpoint;
5. apply terminal outcomes idempotently;
6. verify that backlog and reconciliation alerts clear.

15 Troubleshooting

15.1 Every signature fails

Check:

- the correct environment's secret is loaded;
- the raw body is captured before parsing;
- the literal `.` separator is included;
- `X-Webhook-Timestamp` is used exactly as received;
- HMAC-SHA256 produces lowercase hexadecimal output;
- the expected value includes the `v1=` prefix;
- no proxy or middleware changes the body bytes before verification.

15.2 Signatures fail after rotation

The newest secret is active immediately. Verify with the active secret and, for exactly 24 hours after replacement, the previous secret. Confirm that receiver instances received the updated secret set.

15.3 Valid-looking requests fail timestamp validation

Confirm that the header is Unix seconds rather than milliseconds and that the receiver clock is synchronized. Body `timestamp` is not the freshness value.

15.4 Duplicate processing occurs

Confirm that replay check-and-insert is atomic and shared across all receiver instances. Also confirm that the queue worker applies terminal outcomes and side effects idempotently.

15.5 Webhook events appear to be missing

Confirm:

- the configuration is enabled;
- the safe configuration contains the expected URL;
- the message reached one of the four terminal statuses;
- the receiver responded within 10 seconds;
- ingress and durable handoff were available.

The gateway does not emit events for `queued`, `submitted`, or `unknown`, and it does not retry a failed attempt. Reconcile through the status endpoint.

15.6 Receiver returns 2xx but work is lost

The request was acknowledged before durable handoff. Move queue or database storage before the 2xx response and keep business processing asynchronous.

15.7 Receiver times out

Remove business processing from the request path. Verify, reserve the replay identity, durably enqueue, and acknowledge promptly within the 10-second outbound timeout.

16 Frequently Asked Questions

16.1 Which events are supported?

One delivery status event is supported. Its status is one of `delivered`, `failed`, `expired`, or `rejected`.

16.2 Are non-terminal updates sent?

No. The gateway does not emit webhook events for `queued`, `submitted`, or `unknown`.

16.3 Will a failed webhook be retried?

No. RC1 performs exactly one delivery attempt and has no retry scheduling.

16.4 What acknowledges an event?

Any HTTP 2xx response.

16.5 Can several webhook URLs be configured?

No. The contract supports exactly one configuration per tenant.

16.6 Can the secret be retrieved?

No. It is write-only and is never returned.

16.7 How is a secret rotated?

Replace the complete configuration with PUT `/api/v1/webhook`. The new secret is active immediately; the previous secret remains valid for receiver verification for 24 hours.

16.8 Must JSON be parsed before signature verification?

No. Verify the exact raw bytes first. Parsing or reserializing changes the signed input.

16.9 Is timestamp validation sufficient replay protection?

No. It limits request age but does not prevent the same valid request from being submitted again within the accepted window. Use an atomic replay store.

16.10 Why is idempotent processing still required?

It prevents repeated side effects from receiver replays, queue redelivery, or local recovery operations. It is separate from signature verification.

16.11 How are missed events recovered?

Retrieve current message status using GET `/api/v1/messages/{message_id}/status` and apply the terminal outcome idempotently.

17 Appendices

Appendix A: Verification checklist

Order	Check	Reject before parsing on failure
1	Raw body bytes captured	Yes
2	X-Webhook-Timestamp present and integer	Yes
3	Timestamp within five minutes	Yes
4	X-Webhook-Signature matches required format	Yes
5	HMAC-SHA256 calculated over exact signed input	Yes
6	Constant-time comparison succeeds	Yes
7	Replay identity atomically reserved	Yes
8	JSON parsing succeeds	Parsing occurs here
9	Payload matches <code>DeliveryWebhookEvent</code>	Yes
10	Durable handoff succeeds	Yes
11	Any 2xx returned promptly	Acknowledges event

Appendix B: Frozen webhook contract

Item	Value
Configuration count	One per tenant
Configuration write	PUT /api/v1/webhook
Configuration read	GET /api/v1/webhook
Event method	POST
Required payload fields	message_id, status, timestamp
Optional payload fields	client_reference, delivered_at
Event statuses	delivered, failed, expired, rejected
Timestamp header	X-Webhook-Timestamp, Unix seconds
Signature header	X-Webhook-Signature, v1=<lowercase hex HMAC-SHA256>
Signature input	<timestamp>.<raw-request-body>
Timestamp tolerance	Five minutes
Secret overlap	Previous secret valid for 24 hours
Outbound timeout	10 seconds
Delivery attempts	Exactly one
Automatic retry	None
Attempt retention	90 days
Acknowledgment	Any 2xx

Appendix C: Replay key pseudocode

```
timestamp = request.header("X-Webhook-Timestamp")
signature = request.header("X-Webhook-Signature")
raw_body = request.raw_bytes()

verify_timestamp(timestamp, tolerance_seconds=300)
verify_hmac(secret, timestamp + "." + raw_body, signature)

replay_key = sha256(timestamp + "." + signature + "." + raw_body)
atomic_insert_or_reject(replay_key)

event = parse_and_validate_delivery_webhook_event(raw_body)
durably_enqueue(event)
return_2xx()
```

Appendix D: Payload examples by terminal status

Delivered

```
{
  "message_id": "01JEXAMPLE0000000000000000",
  "status": "delivered",
  "client_reference": "order-1042",
  "delivered_at": "2026-07-19T12:30:44.987654Z",
  "timestamp": "2026-07-19T12:30:45.123456Z"
}
```

Failed

```
{
  "message_id": "01JEXAMPLE0000000000000000",
  "status": "failed",
  "client_reference": "order-1042",
  "timestamp": "2026-07-19T12:30:45.123456Z"
}
```

Expired

```
{
  "message_id": "01JEXAMPLE0000000000000000",
  "status": "expired",
  "timestamp": "2026-07-19T12:30:45.123456Z"
}
```

Rejected

```
{
  "message_id": "01JEXAMPLE0000000000000000",
  "status": "rejected",
  "timestamp": "2026-07-19T12:30:45.123456Z"
}
```

Appendix E: Related resources

- [OpenAPI 3.1](#)
- [API v1 Contract](#)
- [Developer Integration Guide](#)
- [API Reference](#)
- [Webhook receiver examples](#)

End of MHI Gateway Webhook Integration Guide v1.0