

Benchmarks

This directory contains performance benchmarks for the SMS-C system using Benchee.

Available Benchmarks

1. Raw SMS Benchmark (`raw_sms_bench.exs`)

Benchmarks the `submit_message_raw` API endpoint using real SMS PDUs.

Features:

- Uses real SMS PDUs (add your PDUs to the `@sample_pdus` list in the file)
- Disables duplicate detection by clearing fingerprints before each iteration
- Outputs both console and HTML reports

Usage:

```
mix run benchmarks/raw_sms_bench.exs
```

Output: `benchmarks/output/raw_sms_benchmark.html`

2. Message API Benchmark (`message_api_bench.exs`)

Benchmarks various message API operations including insert, retrieval, and routing.

Features:

- Tests `insert_message` (simple and with routing)
- Tests `get_messages_for_smsc`
- Tests `list_message_queues`

- Pre-populates database with test data for realistic scenarios

Usage:

```
mix run benchmarks/message_api_bench.exs
```

Output: `benchmarks/output/message_api_benchmark.html`

Configuration

All benchmarks use Benchee with the following default settings:

- Warmup: 2 seconds
- Time: 10 seconds
- Memory time: 2 seconds
- Extended statistics enabled
- HTML reports auto-generated

Outputs

HTML benchmark reports are generated in `benchmarks/output/` and include:

- Detailed performance metrics
- Comparison charts
- Memory usage statistics
- Statistical analysis

SMS-C API Reference

[← Back to Documentation Index](#)

Complete reference for all SMS-C REST API endpoints with request/response examples.

Table of Contents

- [API Overview](#)
- [Authentication](#)
- [Common Response Formats](#)
- [Status Endpoint](#)
- [Message Queue API](#)
- [Raw SMS PDU API](#)
- [Location Management API](#)
- [Frontend Registration API](#)
- [Campaigns API](#)
- [Event Logging API](#)
- [MMS Message API](#)
- [SS7 Event API](#)
- [Error Codes](#)
- [Rate Limiting](#)
- [Best Practices](#)

API Overview

The SMS-C REST API provides programmatic access to message submission, routing, and management functions.

Base URL

```
https://api.example.com:8443/api
```

Default Port: 8443 (configurable) **Protocol:** HTTPS (TLS required in production)

Content Type

All requests and responses use JSON:

```
Content-Type: application/json
```

API Versioning

The current API is version 1 (implicit). Future versions will use URL versioning:

```
https://api.example.com:8443/api/v2/...
```

Authentication

TLS Client Certificates (Recommended)

Production deployments should use TLS client certificate authentication:

```
curl --cert client.crt --key client.key \  
https://api.example.com:8443/api/status
```

API Key Authentication

Custom API key authentication via `X-API-Key` header:


```
curl -H "X-API-Key: your_api_key_here" \  
https://api.example.com:8443/api/status
```

IP Whitelisting

Restrict API access to trusted IP addresses at the firewall level.

Common Response Formats

Success Response

```
{  
  "data": {  
    ...  
  }  
}
```

Error Response

```
{  
  "errors": {  
    "detail": "Error message describing what went wrong"  
  }  
}
```

List Response

```
{  
  "data": [  
    {...},  
    {...}  
  ]  
}
```

Status Endpoint

Health check endpoint for monitoring and load balancers.

Get API Status

Request:

```
GET /api/status
```

Response (200 OK):

```
{
  "status": "ok",
  "application": "OmniMessage",
  "timestamp": "2025-10-30T12:34:56Z"
}
```

Example:

```
curl https://api.example.com:8443/api/status
```

Use Cases:

- Load balancer health checks
- Monitoring system connectivity
- Service availability verification

Message Queue API

Core message submission and management endpoints.

List Messages

Retrieve messages from the queue.

Request:

```
GET /api/messages
```

Optional Headers:

- `smsc: frontend_name` - Filter by destination SMSC
- `include-unrouted: true|false|1|0` - Include messages without location registration (default: false)
 - `false` (default): Only return messages with explicit routing or location registration
 - `true`: Include messages without location registration (backward compatible mode)

Query Parameters (list mode — only when no `smsc` header is sent):

- `status` - Filter by a single status (default: all). One of: `queued`, `backoff`, `sent`, `delivered`, `expired`, `dropped`, `auto_replied`, `balance_rejected`, `deleted`
- `limit` - Max records to return (default: 100, max: 1000)
- `offset` - Records to skip for pagination (default: 0)

Results are returned newest-first by message id. **The endpoint is paginated by default:** without `limit` it returns at most 100 records, never the whole store.

Response (200 OK) — a JSON array of messages:

```
[
  {
    "id": 12345,
    "source_msisdn": "+15551234567",
    "destination_msisdn": "+447700900000",
    "message_text": "Hello World",
    "source_smsc": "api_client",
    "dest_smsc": "uk_gateway",
    "status": "queued",
    "send_time": "2025-10-30T12:00:00Z",
    "deliver_time": null,
    "delivery_attempts": 0,
    "inserted_at": "2025-10-30T12:00:00Z"
  }
]
```

Examples:

Get pending messages for specific SMSC (only with explicit routing or location):

```
curl -H "smsc: uk_gateway" \
  https://api.example.com:8443/api/messages
```

Get pending messages including unrouted messages (backward compatible):

```
curl -H "smsc: uk_gateway" \
  -H "include-unrouted: true" \
  https://api.example.com:8443/api/messages
```

Get all delivered messages:

```
curl "https://api.example.com:8443/api/messages?
status=delivered&limit=50"
```

Get Single Message

Retrieve details for a specific message.

Request:

```
GET /api/messages/:id
```

Response (200 OK):

```
{
  "data": {
    "id": 12345,
    "source_msisdn": "+15551234567",
    "destination_msisdn": "+447700900000",
    "message_body": "Hello World",
    "source_smsc": "api_client",
    "dest_smsc": "uk_gateway",
    "source_imsi": null,
    "dest_imsi": null,
    "message_parts": 1,
    "message_part_number": 1,
    "tp_data_coding_scheme": "00",
    "tp_user_data_header": null,
    "status": "queued",
    "send_time": "2025-10-30T12:00:00Z",
    "deliver_time": null,
    "expires": "2025-10-31T12:00:00Z",
    "deadletter": false,
    "delivery_attempts": 0,
    "deliver_after": "2025-10-30T12:00:00Z",
    "raw_data_flag": false,
    "raw_sip_flag": false,
    "raw_pdu": null,
    "inserted_at": "2025-10-30T12:00:00Z",
    "updated_at": "2025-10-30T12:00:00Z"
  }
}
```

Example:

```
curl https://api.example.com:8443/api/messages/12345
```

Submit Message (Synchronous)

Submit a message and receive the message ID immediately.

Request:

```
POST /api/messages
Content-Type: application/json
```

Body:

```
{
  "source_msisdn": "+15551234567",
  "destination_msisdn": "+447700900000",
  "message_body": "Hello World",
  "source_smsc": "api_client"
}
```

Optional Fields:

- `dest_smsc` - Override routing decision
- `send_time` - Schedule for future delivery (ISO 8601)
- `message_parts` - Total parts for multi-part message
- `message_part_number` - Part number (1-indexed)
- `tp_data_coding_scheme` - SMS DCS (default: "00")
- `source_imsi` - Source subscriber IMSI
- `dest_imsi` - Destination subscriber IMSI

Response (201 Created):

```
{
  "data": {
    "id": 12345,
    "source_msisdn": "+15551234567",
    "destination_msisdn": "+447700900000",
    "message_body": "Hello World",
    "source_smsc": "api_client",
    "dest_smsc": "uk_gateway",
    "status": "queued",
    "send_time": "2025-10-30T12:00:00Z",
    "inserted_at": "2025-10-30T12:00:00Z"
  }
}
```

Example:

```
curl -X POST https://api.example.com:8443/api/messages \
-H "Content-Type: application/json" \
-d '{
  "source_msisdn": "+15551234567",
  "destination_msisdn": "+447700900000",
  "message_body": "Hello World",
  "source_smsc": "api_client"
}'
```

Performance: ~70 messages/second, 14ms average response time

Use When:

- Need message ID immediately
- Processing messages/second
- Require immediate confirmation

Submit Message (Asynchronous)

Submit a message with high throughput (batch processing).

Request:

```
POST /api/messages/create_async
Content-Type: application/json
```

Body: Same as synchronous endpoint

Response (202 Accepted):

```
{
  "data": {
    "status": "accepted",
    "message": "Message queued for processing"
  }
}
```

Example:

```
curl -X POST
https://api.example.com:8443/api/messages/create_async \
-H "Content-Type: application/json" \
-d '{
  "source_msisdn": "+15551234567",
  "destination_msisdn": "+447700900000",
  "message_body": "Bulk notification message",
  "source_smsc": "bulk_api"
}'
```

Performance: ~4,650 messages/second, 0.22ms average response time

Latency: Message appears in database within 100ms (configurable)

Use When:

- High-volume bulk messaging (> 100 msg/sec)
- Don't need message ID in API response
- Throughput more important than instant confirmation

Update Message

Partially update message fields.

Request:

```
PATCH /api/messages/:id
Content-Type: application/json
```

Body:

```
{
  "dest_smsc": "alternate_gateway",
  "deliver_after": "2025-10-30T14:00:00Z"
}
```

Updatable Fields:

- `dest_smsc` - Change destination
- `deliver_after` - Delay delivery
- `message_body` - Update message text
- `status` - Change status

Response (200 OK):

```
{
  "data": {
    "id": 12345,
    "dest_smsc": "alternate_gateway",
    "deliver_after": "2025-10-30T14:00:00Z",
    ...
  }
}
```

Example:

```
curl -X PATCH https://api.example.com:8443/api/messages/12345 \
  -H "Content-Type: application/json" \
  -d '{
    "dest_smsc": "backup_gateway"
  }'
```

Mark Message Delivered

Mark a message as successfully delivered.

Request:

```
POST /api/messages/:id/mark_delivered
Content-Type: application/json
```

Body:

```
{
  "dest_smsc": "uk_gateway"
}
```

Response (200 OK):

```
{
  "data": {
    "id": 12345,
    "status": "delivered",
    "deliver_time": "2025-10-30T12:05:30Z",
    "dest_smsc": "uk_gateway",
    ...
  }
}
```

Example:

```
curl -X POST
https://api.example.com:8443/api/messages/12345/mark_delivered \
-H "Content-Type: application/json" \
-d '{
  "dest_smsc": "uk_gateway"
}'
```

Use Case: Called by frontend systems after successful delivery

Increment Delivery Attempt

Record a failed delivery: increments the attempt counter, moves the message to `backoff`, and schedules the next retry.

Request:

```
PUT /api/messages/:id
```

Response (200 OK) — the updated message:

```
{
  "id": 12345,
  "status": "backoff",
  "delivery_attempts": 2,
  "deliver_after": "2025-10-30T12:08:00Z"
}
```

Backoff: exponential, **capped at 30 minutes**, and never scheduled past the message's `expires`:

```
deliver_after = min(now + min(2^attempts, 30) minutes, expires)
```

The message stays in `backoff` until `deliver_after` passes, then becomes eligible again. It keeps retrying every ≤ 30 minutes until `expires` is reached, after which it is no longer served (failed deliveries are not dead-lettered).

Example:

```
curl -X PUT https://api.example.com:8443/api/messages/12345
```

Use Case: Called by frontend after delivery failure to schedule retry

Delete Message

Remove message from queue.

Request:

```
DELETE /api/messages/:id
```

Response (204 No Content)

Example:

```
curl -X DELETE https://api.example.com:8443/api/messages/12345
```

Warning: Deleting messages removes them permanently. Use with caution.

Raw SMS PDU API

Submit SMS messages as raw PDU (Protocol Data Unit) for maximum compatibility with legacy systems.

Submit Raw SMS (Synchronous)

Request:

```
POST /api/messages_raw  
Content-Type: application/json
```

Body:

```
{
  "pdu": "0001000B916407007009F0000004D4F29C0E",
  "source_smsc": "legacy_system"
}
```

PDU Format: Hex-encoded SMS TPDU (Transport Protocol Data Unit)

Response (201 Created):

```
{
  "data": {
    "id": 12346,
    "source_msisdn": "+447700900000",
    "destination_msisdn": "+447700900000",
    "message_body": "Test",
    "source_smsc": "legacy_system",
    "raw_pdu": "0001000B916407007009F0000004D4F29C0E",
    ...
  }
}
```

Example:

```
curl -X POST https://api.example.com:8443/api/messages_raw \
-H "Content-Type: application/json" \
-d '{
  "pdu": "0001000B916407007009F0000004D4F29C0E",
  "source_smsc": "legacy_system"
}'
```

Submit Raw SMS (Asynchronous)

Request:

```
POST /api/messages_raw/async
Content-Type: application/json
```

Body: Same as synchronous

Response (202 Accepted):

```
{
  "data": {
    "status": "accepted",
    "message": "PDU queued for processing"
  }
}
```

Example:

```
curl -X POST https://api.example.com:8443/api/messages_raw/async \
-H "Content-Type: application/json" \
-d '{
  "pdu": "0001000B916407007009F00000004D4F29C0E",
  "source_smsc": "legacy_gateway"
}'
```

PDU Handling

The system automatically:

1. Decodes PDU using SMS standards (3GPP TS 23.040)
2. Extracts phone numbers, message text, DCS
3. Detects delivery reports (CP-ACK, RP-ACK, etc.)
4. Performs IMSI to MSISDN lookup if needed
5. Applies routing rules
6. Stores original PDU for reference

Delivery Report Detection:

- CP-ACK, CP-ERROR - Connection Protocol acknowledgments

- RP-ACK, RP-ERROR, RP-SMMA - Relay Protocol responses
- Delivery reports are logged but not stored as messages

Location Management API

Manage subscriber location information for mobile-terminated message delivery.

List Locations

Request:

```
GET /api/locations
```

Response (200 OK):

```
{
  "data": [
    {
      "id": 1,
      "msisdn": "+15551234567",
      "imsi": "001001000000001",
      "location": "msc1.region1.example.com",
      "ran_location": "cell_tower_12345",
      "imei": "123456789012345",
      "ims_capable": true,
      "csfb": false,
      "registered": true,
      "expires": "2025-10-30T13:00:00Z",
      "user_agent": "Samsung Galaxy",
      "inserted_at": "2025-10-30T12:00:00Z",
      "updated_at": "2025-10-30T12:00:00Z"
    }
  ]
}
```

Example:

```
curl https://api.example.com:8443/api/locations
```

Get Location

Request:

```
GET /api/locations/:id
```

Response (200 OK):

```
{
  "data": {
    "id": 1,
    "msisdn": "+15551234567",
    "imsi": "001001000000001",
    ...
  }
}
```

Example:

```
curl https://api.example.com:8443/api/locations/1
```

Create/Update Location

Creates new location or updates existing based on IMSI (unique identifier).

Request:

```
POST /api/locations
Content-Type: application/json
```

Body:


```
{
  "msisdn": "+15551234567",
  "imsi": "001001000000001",
  "location": "msc1.region1.example.com",
  "ran_location": "cell_tower_12345",
  "imei": "123456789012345",
  "ims_capable": true,
  "csfb": false,
  "registered": true,
  "expires": "2025-10-30T13:00:00Z",
  "user_agent": "Samsung Galaxy"
}
```

Required Fields:

- `imsi` - Unique subscriber identifier
- `msisdn` - Phone number

Optional Fields:

- `location` - MSC/VLR address
- `ran_location` - Cell tower/sector ID
- `imei` - Device identifier
- `ims_capable` - IMS VoLTE capability
- `csfb` - Circuit-switched fallback flag
- `registered` - Currently registered
- `expires` - Registration expiry
- `user_agent` - Device model/info

Response (201 Created or 200 OK):

```
{
  "data": {
    "id": 1,
    "msisdn": "+15551234567",
    ...
  }
}
```

Example:

```
curl -X POST https://api.example.com:8443/api/locations \
-H "Content-Type: application/json" \
-d '{
  "msisdn": "+15551234567",
  "imsi": "0010010000000001",
  "location": "msc1.region1.example.com",
  "ims_capable": true,
  "registered": true
}'
```

Use Case: Called by mobility management systems (HSS, MME, etc.) when subscriber registers

Update Location

Request:

```
PATCH /api/locations/:id
Content-Type: application/json
```

Body: Partial update with any location fields

Response (200 OK):

```
{
  "data": {
    "id": 1,
    ...
  }
}
```

Example:

```
curl -X PATCH https://api.example.com:8443/api/locations/1 \
-H "Content-Type: application/json" \
-d '{
  "location": "msc2.region2.example.com",
  "ran_location": "cell_tower_67890"
}'
```

Delete Location

Request:

```
DELETE /api/locations/:id
```

Response (204 No Content)

Example:

```
curl -X DELETE https://api.example.com:8443/api/locations/1
```

Use Case: Called when subscriber de-registers or times out

Frontend Registration API

Track and manage frontend SMSC connections.

List All Frontends

Request:

```
GET /api/frontends
```

Response (200 OK):

```
{
  "data": [
    {
      "id": 1,
      "frontend_name": "uk_gateway_1",
      "frontend_type": "smpp",
      "ip_address": "10.0.1.50",
      "hostname": "gateway1.uk.example.com",
      "uptime_seconds": 86400,
      "configuration": {
        "max_throughput": 1000,
        "bind_type": "transceiver"
      },
      "status": "active",
      "expires_at": "2025-10-30T12:02:00Z",
      "last_seen_at": "2025-10-30T12:00:30Z",
      "inserted_at": "2025-10-29T12:00:00Z",
      "updated_at": "2025-10-30T12:00:30Z"
    }
  ]
}
```

Example:

```
curl https://api.example.com:8443/api/frontends
```

List Active Frontends Only

Request:

```
GET /api/frontends/active
```

Response (200 OK): Same format, only active frontends

Example:

```
curl https://api.example.com:8443/api/frontends/active
```

Use Case: Get list of available destinations for routing

Get Frontend Statistics

Request:

```
GET /api/frontends/stats
```

Response (200 OK):

```
{
  "data": {
    "active_count": 5,
    "expired_count": 2,
    "unique_frontends": 7,
    "total_registrations": 1523
  }
}
```

Example:

```
curl https://api.example.com:8443/api/frontends/stats
```

Get Frontend History

Request:

```
GET /api/frontends/history/:name
```

Response (200 OK):

```
{
  "data": [
    {
      "id": 1,
      "frontend_name": "uk_gateway_1",
      "status": "active",
      "inserted_at": "2025-10-30T12:00:00Z",
      ...
    },
    {
      "id": 2,
      "frontend_name": "uk_gateway_1",
      "status": "expired",
      "inserted_at": "2025-10-29T12:00:00Z",
      ...
    }
  ]
}
```

Example:

```
curl
https://api.example.com:8443/api/frontends/history/uk_gateway_1
```

Register Frontend

Register or update frontend connection.

Request:

```
POST /api/frontends/register
Content-Type: application/json
```

Body:

```
{
  "frontend_name": "uk_gateway_1",
  "frontend_type": "smpp",
  "ip_address": "10.0.1.50",
  "hostname": "gateway1.uk.example.com",
  "uptime_seconds": 86400,
  "configuration": {
    "max_throughput": 1000,
    "bind_type": "transceiver",
    "system_id": "gateway1"
  }
}
```

Required Fields:

- `frontend_name` - Unique identifier for frontend
- `frontend_type` - Type: `smpp`, `sip`, `http`, etc.

Optional Fields:

- `ip_address` - Frontend IP
- `hostname` - Frontend hostname
- `uptime_seconds` - Uptime since start
- `configuration` - Custom config object

Response (201 Created):

```
{
  "data": {
    "id": 1,
    "frontend_name": "uk_gateway_1",
    "status": "active",
    "expires_at": "2025-10-30T12:01:30Z",
    ...
  }
}
```

Example:

```
curl -X POST https://api.example.com:8443/api/frontends/register \
-H "Content-Type: application/json" \
-d '{
  "frontend_name": "uk_gateway_1",
  "frontend_type": "smp",
  "ip_address": "10.0.1.50",
  "hostname": "gateway1.uk.example.com"
}'
```

Registration Timeout: 90 seconds (frontends must re-register every 60-90 seconds)

Use Case: Called periodically by frontend systems to maintain active status

Campaigns API

Bulk messaging: upload a named recipient list, then run a rate-limited campaign against it. See the [Bulk Messaging / Campaigns guide](#) for concepts (audience filter, drip rate, lifecycle, statistics).

Recipient Lists

Create a List

Request:

```
POST /api/campaign_lists
Content-Type: application/json
```

Each recipient may carry up to four template variables (`var1..var4`).

Body — supply recipients as a JSON array, a raw CSV string, or both. JSON recipients may be plain MSISDN strings or objects with `var1..var4`:


```
{
  "name": "VIP customers",
  "description": "High value subscribers",
  "recipients": [
    { "msisdn": "12025550101", "var1": "Alice", "var2": "Gold" },
    "12025550102"
  ]
}
```

For CSV, a header row naming an `msisdn` / `destination_msisdn` / `number` / `phone` column selects the MSISDN column (otherwise the first column is used); the **remaining columns become** `var1`..`var4` in column order:

```
{ "name": "Imported", "csv":
  "msisdn,first,plan\n12025550101,Alice,Gold\n12025550102,Bob,Silver\n"
}
```

Response (201 Created):

```
{ "id": 1, "name": "VIP customers", "recipient_count": 2 }
```

List / Get / Append / Delete Lists

```
GET    /api/campaign_lists      # all lists
GET    /api/campaign_lists/:id  # one list, includes
"recipients": [{msisdn, variables}, ...]
PATCH /api/campaign_lists/:id  # append recipients
({ "recipients": [...] } and/or { "csv": "..." })
DELETE /api/campaign_lists/:id  # delete the list and its
recipients
```

Campaigns

Create a Campaign

Request:

POST /api/campaigns
Content-Type: application/json

Body:

```
{
  "name": "Maintenance notice",
  "message": "Planned maintenance tonight 02:00-03:00.",
  "source_msisdn": "12345",
  "source_smsc": "IMS_SMSC",
  "list_id": 1,
  "audience": "active_only",
  "drip_tps": 50
}
```

Fields:

- `name`, `message`, `source_msisdn`, `source_smsc` — **required**
- `list_id` — recipient list. **Omit to target ALL currently active (registered) subscribers.**
- `audience` — `active_only` (default) or `all`. Ignored when `list_id` is omitted.
- `drip_tps` — messages submitted into the queue per second (default from config).
- `validity_hours` — message validity period in hours; sets each message's `expires` (default from config).
- `deliver_after` — optional ISO8601 time to hold messages until before delivery (scheduled send).

Response (201 Created): the campaign in `draft` status, including a live `stats` block.

Control a Campaign (start / pause / resume / cancel)

Request:

```
POST /api/campaigns/control
Content-Type: application/json
```

Body:

```
{ "campaign_id": 7, "action": "start" }
```

`action` is one of `start`, `pause`, `resume`, `cancel`. Returns the updated campaign (200), 404 if not found, or 422 if the action is invalid for the current state.

List / Get Campaigns

```
GET /api/campaigns      # all campaigns (newest first)
GET /api/campaigns/:id  # one campaign, including live stats
```

Response (GET /api/campaigns/:id):

```
{
  "id": 7,
  "name": "Maintenance notice",
  "list_id": 1,
  "audience": "active_only",
  "status": "running",
  "drip_tps": 50,
  "total_targets": 1000,
  "stats": {
    "total": 1000, "pending": 600, "queued": 300,
    "delivered": 90, "failed": 10, "skipped_inactive": 0,
    "dispatched": 400, "progress_percent": 40, "delivery_percent":
22
  }
}
```

Delete (Stop) a Campaign

```
DELETE /api/campaigns/:id
```

Deleting **stops** the campaign (cancels delivery) but **keeps the record** so its history and statistics remain visible; it returns the cancelled campaign (200). The record is purged automatically once it ages past the retention TTL.

Per-Recipient Results

```
GET /api/campaign_targets/:id          # :id is the campaign ID
GET /api/campaign_targets/:id?state=failed # filter by target state
```

Returns one entry per recipient with its `state` (`pending` / `queued` / `delivered` / `failed` / `skipped_inactive`), the submitted `message_ids`, and part counters.

Event Logging API

Track message lifecycle events.

Get Message Events

Request:

```
GET /api/events/:message_id
```

Response (200 OK):

```
{
  "data": [
    {
      "event_epoch": 1698672000,
      "name": "message_inserted",
      "description": "Message inserted into queue",
      "event_source": "node1@server.example.com"
    },
    {
      "event_epoch": 1698672001,
      "name": "message_routed",
      "description": "Routed to uk_gateway via route_id=42",
      "event_source": "node1@server.example.com"
    },
    {
      "event_epoch": 1698672005,
      "name": "message_delivered",
      "description": "Successfully delivered",
      "event_source": "node2@server.example.com"
    }
  ]
}
```

Example:

```
curl https://api.example.com:8443/api/events/12345
```

Event Types:

- `message_inserted` - Message created
- `message_routed` - Routing decision made
- `message_delivered` - Successful delivery
- `message_failed` - Delivery failed
- `message_dropped` - Dropped by route
- `auto_reply_sent` - Auto-reply triggered
- `number_translated` - Number transformation applied
- `routing_failed` - No route found
- `charging_failed` - Charging error

Record Event

Request:

```
POST /api/events
Content-Type: application/json
```

Body:

```
{
  "message_id": 12345,
  "name": "custom_event",
  "description": "Custom event description",
  "event_source": "external_system"
}
```

Response (201 Created):

```
{
  "data": {
    "message_id": 12345,
    "name": "custom_event",
    "description": "Custom event description",
    "event_source": "external_system",
    "event_epoch": 1698672010
  }
}
```

Example:

```
curl -X POST https://api.example.com:8443/api/events \
-H "Content-Type: application/json" \
-d '{
  "message_id": 12345,
  "name": "external_delivery_confirmed",
  "description": "Confirmed by downstream system"
}'
```

Event Retention: 7 days (configurable)

MMS Message API

Manage Multimedia Messaging Service (MMS) messages.

List MMS Messages

Request:

```
GET /api/mms_messages
```

Response (200 OK): Similar to SMS messages with additional MMS fields

Create MMS Message

Request:

```
POST /api/mms_messages  
Content-Type: application/json
```

Body:

```
{  
  "source_msisdn": "+15551234567",  
  "destination_msisdn": "+447700900000",  
  "subject": "Photo",  
  "content_type": "image/jpeg",  
  "content_location": "https://cdn.example.com/media/12345.jpg",  
  "message_size": 524288  
}
```

Response (201 Created): Full MMS message object

SS7 Event API

Track SS7 signaling events.

List SS7 Events

Request:

```
GET /api/ss7_events
```

Response (200 OK):

```
{
  "data": [
    {
      "id": 1,
      "event_type": "MAP_UPDATE_LOCATION",
      "imsi": "001001000000001",
      "msisdn": "+15551234567",
      "timestamp": "2025-10-30T12:00:00Z",
      ...
    }
  ]
}
```

Create SS7 Event

Request:

```
POST /api/ss7_events
Content-Type: application/json
```

Body:


```
{  
  "event_type": "MAP_UPDATE_LOCATION",  
  "imsi": "001001000000001",  
  "msisdn": "+15551234567"  
}
```

Response (201 Created): Full event object

Error Codes

HTTP Status Codes

Code	Meaning	Description
200	OK	Request successful
201	Created	Resource created successfully
202	Accepted	Request accepted for processing
204	No Content	Successful deletion
400	Bad Request	Invalid request format
401	Unauthorized	Authentication required
403	Forbidden	Insufficient permissions
404	Not Found	Resource doesn't exist
422	Unprocessable Entity	Validation errors
429	Too Many Requests	Rate limit exceeded
500	Internal Server Error	Server error
503	Service Unavailable	Temporarily unavailable

Error Response Format

```
{
  "errors": {
    "detail": "Validation failed: destination_msisdn is required"
  }
}
```

Common Error Messages

Error	Cause	Solution
"destination_msisdn is required"	Missing required field	Include destination_msisdn in request
"Invalid phone number format"	Malformed number	Use E.164 format: +15551234567
"Message too long"	Exceeds size limit	Split into multiple parts
"No route found"	Routing failed	Check routing configuration
"Charging failed"	OCS error	Verify charging system connectivity
"Message not found"	Invalid message ID	Verify ID exists
"Frontend not registered"	Unknown SMSC	Register frontend first

Rate Limiting

Default Limits

Endpoint	Limit	Window
POST /api/messages	100 req/sec	Per IP
POST /api/messages/create_async	1000 req/sec	Per IP
POST /api/messages_raw	100 req/sec	Per IP
GET /api/*	1000 req/sec	Per IP

Rate Limit Headers

```
X-RateLimit-Limit: 100
X-RateLimit-Remaining: 95
X-RateLimit-Reset: 1698672060
```

Rate Limit Exceeded

Response (429 Too Many Requests):

```
{
  "errors": {
    "detail": "Rate limit exceeded. Retry after 5 seconds."
  }
}
```

Best Practices

Message Submission

1. **Use Async for Bulk:** Use `/create_async` for > 100 msg/sec
2. **Include source_smsc:** Always identify your system
3. **Validate Numbers:** Use E.164 format (+country code)
4. **Handle Errors:** Implement retry logic for 5xx errors
5. **Check Routing:** Test routes before bulk submission

Frontend Integration

1. **Register Regularly:** Re-register every 60 seconds
2. **Poll for Messages:** Query with `smsc` header for your messages
3. **Use include-unrouted Wisely:** By default, only messages with explicit routing or location registration are returned. Set `include-unrouted: true` only if you need backward compatible behavior to receive all unrouted messages
4. **Mark Delivered:** Always call `mark_delivered` after success
5. **Increment on Failure:** Use PUT endpoint for retry logic
6. **Monitor Events:** Check event log for delivery issues

Performance

1. **Connection Pooling:** Reuse HTTP connections
2. **Batch Requests:** Group multiple messages per request
3. **Parallel Processing:** Make concurrent API calls
4. **Monitor Metrics:** Watch Prometheus for bottlenecks
5. **Set Timeouts:** Use 30-second timeout for API calls

Security

1. **Use TLS:** Always use HTTPS in production
2. **Validate Certificates:** Don't skip certificate validation

3. **Rotate API Keys:** Change keys regularly
4. **IP Whitelist:** Restrict to known sources
5. **Log API Activity:** Monitor for suspicious patterns

Error Handling

1. **Retry 5xx Errors:** Server errors are usually temporary
2. **Don't Retry 4xx:** Client errors need code fixes
3. **Exponential Backoff:** Wait longer between retries
4. **Circuit Breaker:** Stop after repeated failures
5. **Alert on Patterns:** Monitor error rates

Example Integration (Python)

```
import requests
import time

class SMSCClient:
    def __init__(self, base_url, api_key=None):
        self.base_url = base_url
        self.session = requests.Session()
        if api_key:
            self.session.headers.update({"X-API-Key": api_key})

    def submit_message(self, from_num, to_num, text,
async_mode=False):
        endpoint = "/messages/create_async" if async_mode else
"/messages"
        url = f"{self.base_url}{endpoint}"

        payload = {
            "source_msisdn": from_num,
            "destination_msisdn": to_num,
            "message_body": text,
            "source_smsc": "python_client"
        }

        try:
            response = self.session.post(url, json=payload,
timeout=30)
            response.raise_for_status()
            return response.json()["data"]
        except requests.exceptions.RequestException as e:
            print(f"API Error: {e}")
            return None

    def get_pending_messages(self, smsc_name,
include_unrouted=False):
        url = f"{self.base_url}/messages"
        headers = {"smsc": smsc_name}

        # Include unrouted messages if requested (backward
compatible mode)
        if include_unrouted:
            headers["include-unrouted"] = "true"
```

```

        try:
            response = self.session.get(url, headers=headers,
timeout=30)
            response.raise_for_status()
            return response.json()["data"]
        except requests.exceptions.RequestException as e:
            print(f"API Error: {e}")
            return []

    def mark_delivered(self, message_id, smsc_name):
        url = f"
{self.base_url}/messages/{message_id}/mark_delivered"
        payload = {"dest_smsc": smsc_name}

        try:
            response = self.session.post(url, json=payload,
timeout=30)
            response.raise_for_status()
            return True
        except requests.exceptions.RequestException as e:
            print(f"API Error: {e}")
            return False

# Usage
client = SMSCClient("https://api.example.com:8443/api",
api_key="your_key")

# Submit single message
result = client.submit_message("+15551234567", "+447700900000",
"Hello")
print(f"Message ID: {result['id']}")

# Submit bulk messages (async)
for i in range(1000):
    client.submit_message("+15551234567", f"+44770090{i:04d}",
f"Bulk {i}", async_mode=True)

# Frontend polling loop
while True:
    # Get messages with explicit routing or location registration
    messages = client.get_pending_messages("my_gateway")

    # Or use include_unrouted=True for backward compatible

```



```
behavior
    # messages = client.get_pending_messages("my_gateway",
include_unrouted=True)

    for msg in messages:
        # Deliver message via your protocol
        success = deliver_via_smpp(msg)

        if success:
            client.mark_delivered(msg["id"], "my_gateway")
        else:
            # Increment for retry
            requests.put(f"
{client.base_url}/messages/{msg['id']}")

    time.sleep(5) # Poll every 5 seconds
```

API Changelog

Version 1 (Current)

- Initial release
- Message queue CRUD
- Raw PDU submission
- Location management
- Frontend registration
- Event logging

Planned Features

- Batch message submission (single request, multiple messages)
- Message templates
- Scheduled delivery API
- Real-time webhooks for events
- GraphQL API endpoint
- OAuth2 authentication

For questions or issues with the API, check the [Troubleshooting Guide](#) or contact support.

ANSSI R226 Interception Compliance Documentation

Document Purpose: This document provides technical specifications required for ANSSI R226 authorization under Articles R226-3 and R226-7 of the French Penal Code for the OmniMessage SMS Service Center (SMSc).

Classification: Regulatory Compliance Documentation

Target Authority: Agence nationale de la sécurité des systèmes d'information (ANSSI)

Regulation: R226 - Protection of Correspondence Privacy and Lawful Interception

1. DETAILED TECHNICAL SPECIFICATIONS

1.1 Commercial Technical Datasheet

Product Name: OmniMessage SMSc (SMS Service Center) **Product Type:** Telecommunications Message Center **Primary Function:** SMS message routing, storage, and delivery **Network Protocols:** REST API (HTTPS), SMS protocols (SMPP, IMS, SS7/MAP via external frontends) **Deployment Model:** On-premises server application **Technology Stack:** Elixir/Erlang, Phoenix Framework, Mnesia, MySQL/PostgreSQL

Core Capabilities

Message Processing:

- Centralized SMS message queue with REST API
- Protocol-agnostic design supporting SMPP, IMS, SS7/MAP frontends
- Dynamic routing engine with prefix-based routing
- Retry logic with exponential backoff
- Message expiration and dead letter queue handling
- Call Detail Record (CDR) generation and archival
- Performance: ~1,750 messages/second insert rate, 150 million messages/day capacity

Message Storage:

- **Active Message Queue:** Mnesia in-memory database with optional disc persistence
 - Primary storage: RAM for ultra-fast access (sub-millisecond latency)
 - Disc backup: `disc_copies` mode writes to disk for crash recovery
 - Automatic recovery: Messages survive system restarts
 - Retention: Configurable (default 24 hours), then automatic cleanup
- **Long-term CDR Archive:** MySQL/PostgreSQL database (separate from message queue)
 - CDRs written when messages are delivered, expired, failed, or rejected
 - SQL database used ONLY for CDR export/archival, NOT for active message operations
 - No performance impact on message routing (async write)
- **Two-tier Architecture Benefits:**
 - Active queue: Blazing fast (1,750 msg/sec) with no SQL bottleneck
 - CDR archive: Long-term retention (months/years) for billing and lawful interception
 - Clean separation: Message operations never touch SQL
- Cluster support for high availability (Mnesia replication across nodes)

Network Interfaces:

- **REST API:** HTTPS (port 8443) for external frontend communication
- **Control Panel:** HTTPS (port 8086) for web-based management

- **Frontend Protocols:** SMPP, IMS, SS7/MAP (via external gateway applications)
- **Database:** MySQL/PostgreSQL for CDR storage

Routing and Processing:

- Dynamic SMS routing with runtime configuration updates
- Prefix-based matching (calling/called numbers)
- Source SMSC and type filtering
- Priority and weight-based load balancing
- Number translation and normalization
- ENUM (E.164 Number Mapping) DNS lookup support
- Auto-reply and message drop capabilities
- Per-route charging control (OCS integration over Diameter Ro)

□ **Complete architecture and features documented in [README.md](#)**

1.2 Interception Capabilities

1.2.1 Message Acquisition

SMS Message Capture:

- The OmniMessage SMSc processes all SMS messages between subscribers and external networks
- Full access to message metadata and content including:
 - Source MSISDN (mobile number)
 - Destination MSISDN (mobile number)
 - Source IMSI (International Mobile Subscriber Identity)
 - Destination IMSI
 - Message body (text content)
 - Raw PDU (Protocol Data Unit) data
 - TP-DCS (Data Coding Scheme) information
 - Message encoding (GSM7, UCS-2, 8-bit, Latin-1)
 - Multipart message indicators and reassembly data
 - User Data Header (UDH) information

Message Metadata Acquisition:

- Complete Call Detail Records (CDR) stored in database with:
 - Message ID (unique identifier)
 - Calling number (source MSISDN)
 - Called number (destination MSISDN)
 - Submission timestamp (when message entered system)
 - Delivery timestamp (when message was delivered)
 - Expiry timestamp (when message expired if undeliverable)
 - Status (delivered, expired, failed, rejected)
 - Delivery attempts count
 - Message parts (for concatenated/multi-part SMS)
 - Source SMSC identifier
 - Destination SMSC identifier
 - Origin node (Erlang cluster node name)
 - Destination node (for distributed deployments)
 - Deadletter flag (retry exhaustion indicator)

□ **Complete CDR schema documented in [CDR_SCHEMA.md](#)**

Message Queue Access:

- Real-time message queue monitoring
- REST API endpoints for message retrieval
- Database queries for historical message search
- Filter capabilities by:
 - Phone number (source/destination)
 - SMSC gateway
 - Time range
 - Message status
 - Delivery attempts

□ **Complete API documentation in [API_REFERENCE.md](#)**

1.2.2 Data Processing Capabilities

Message Storage Architecture (Two-Tier System):

The SMSc uses a sophisticated two-tier storage architecture that separates operational message processing from long-term archival:

Tier 1: Active Message Queue (Mnesia)

- **Purpose:** Real-time message routing and delivery operations
- **Technology:** Erlang Mnesia distributed database
- **Storage Mode:** In-memory with disc_copies backup
 - Primary storage in RAM for maximum speed
 - Automatic disc synchronization for crash recovery
 - Messages persist across system restarts
- **Performance:** Sub-millisecond read/write operations
- **Retention:** Short-term (default 24 hours), configurable
- **Cleanup:** Automatic archival to CDR database, then deletion from Mnesia
- **Operations:** All message queue operations (insert, update, delivery status, routing)
- **Critical Feature:** SQL database is NEVER queried during message routing/delivery

Tier 2: CDR Archive (MySQL/PostgreSQL)

- **Purpose:** Long-term storage for billing, analytics, and lawful interception
- **Technology:** Traditional SQL database (MySQL or PostgreSQL)
- **Write Trigger:** CDRs written ONLY when messages reach final state:
 - Message delivered successfully
 - Message expired (exceeded validity period)
 - Message permanently failed
 - Message rejected by routing rules
- **Write Mode:** Asynchronous batch writing (no impact on message routing performance)
- **Retention:** Long-term (months to years), configurable per regulatory requirements
- **Operations:** Historical queries, reporting, compliance, lawful interception
- **Access:** SQL queries, REST API (future), CSV/JSON export

Key Architectural Benefits:

1. **Performance:** Active routing operations never touch SQL (no database bottleneck)
2. **Scalability:** Mnesia handles 1,750+ messages/second without SQL overhead
3. **Reliability:** Disc_copies mode ensures no message loss on crash
4. **Compliance:** CDR database provides permanent audit trail
5. **Separation of Concerns:** Operational data vs. archival data clearly separated

Message Lifecycle:

1. Message submitted → Stored in Mnesia (RAM + disc backup)
2. Message routed → Mnesia query (ultra-fast)
3. Message delivered/expired → CDR written to SQL (async)
4. After 24h → Message deleted from Mnesia (cleanup worker)
5. CDR remains in SQL → Available for lawful interception queries (years)

Data Retention and Retrieval:

- Configurable message body retention or deletion for privacy
- Binary data preservation (raw PDU storage in both Mnesia and CDR)
- Full-text search capability (if enabled on CDR database)
- Indexed CDR fields for fast lawful interception queries

Frontend Tracking:

- Real-time tracking of external SMSC frontends (SMPP, IMS, MAP gateways)
- Frontend registration with heartbeat monitoring
- Health status tracking (active/expired)
- Uptime/downtime history
- IP address and hostname tracking
- Frontend-specific configuration logging

1.2.3 Analysis Capabilities

Real-Time Monitoring:

- Web UI dashboard showing:
 - Active message queue
 - Message submission and delivery
 - Routing decisions and gateway selection
 - Frontend gateway status
 - System resource utilization
- Prometheus metrics integration for operational monitoring
- Performance metrics (throughput, latency, success rates)

▢ **Complete monitoring guide in [OPERATIONS_GUIDE.md](#)** ▢ **Metrics documentation in [METRICS.md](#)**

Historical Analysis:

- CDR database queryable by:
 - Time range
 - Calling/called party number
 - Message status
 - SMSC gateway
 - Delivery attempts
 - Message content (full-text search if enabled)
- Statistical analysis capabilities:
 - Message volume by hour/day/month
 - Success/failure rates by route
 - Average delivery times
 - Multi-part message analysis
 - Failed delivery patterns

Subscriber Tracking:

- Message history by phone number (MSISDN)
- IMSI-based tracking (when available from IMS/MAP frontends)
- Call pattern analysis
- Communication party correlation

- Temporal analysis (message frequency, timing patterns)

Network Analytics:

- Route performance metrics
- Gateway availability and health
- Message flow visualization
- Cluster node distribution (multi-node deployments)
- Delivery attempt analysis
- Retry pattern analysis

Number Intelligence:

- E.164 number normalization
- Country/region identification from number prefix
- Number translation and rewriting rules
- ENUM DNS lookup for routing intelligence
- Prefix-based routing decisions

□ **Number translation guide in [number_translation_guide.md](#)** □ **Routing guide in [sms_routing_guide.md](#)**

1.3 Countermeasure Capabilities

1.3.1 Privacy Protection Mechanisms

Communication Confidentiality:

- HTTPS/TLS for REST API communications
- Certificate-based authentication
- Database connection encryption (TLS support)
- Configurable message body deletion after delivery

Access Control:

- Web UI access control
- API authentication mechanisms

- Database access controls
- Frontend registration authentication

Audit Logging:

- Complete system event logging
- Message submission/delivery logging
- Configuration change tracking
- Administrative action logging
- Structured logging with configurable levels

1.3.2 Data Protection Features

Message Privacy:

- Configurable message body deletion after delivery
- Message body excluded from UI display (optional)
- Message body excluded from exports (optional)
- CDR message body field can be set to NULL for privacy

Database Security:

- MySQL table encryption support (ENCRYPTION='Y')
- PostgreSQL transparent data encryption support
- Database access role separation
- Read-only user accounts for analytics
- Restricted access to message content

System Hardening:

- Minimal exposed network ports
- TLS certificate management
- Secure configuration storage
- Environment-based configuration separation
- Cluster security with Erlang distribution protocol

1.4 Storage Architecture: Mnesia + SQL Two-Tier Design

Overview

The OmniMessage SMSc employs a unique two-tier storage architecture specifically designed to separate high-performance operational message processing from long-term compliance and archival storage.

Tier 1: Mnesia In-Memory Message Queue

What is Mnesia?

- Distributed database built into Erlang/OTP runtime
- Hybrid storage: Primary in-memory with automatic disc backup
- ACID-compliant transactions
- Cluster replication across multiple nodes

Storage Mode: `disc_copies`

- **In-Memory Primary:** All active messages stored in RAM
 - Lightning-fast read/write operations (sub-millisecond)
 - No disk I/O during normal message routing operations
 - Enables 1,750+ messages/second throughput
- **Disc Backup (Automatic):** Mnesia synchronizes RAM to disk
 - Writes happen asynchronously in background
 - Disk copy updated on every transaction commit
 - Crash recovery: System restarts with all messages intact
 - Location: `Mnesia.*/*` directory in application data

Message Lifecycle in Mnesia:

1. Message arrives via REST API → Inserted into Mnesia RAM + disc backup
2. Routing engine queries Mnesia → Instant response (memory access)
3. External gateway polls for messages → Mnesia query (memory access)
4. Gateway updates delivery status → Mnesia update (memory + disc)
5. After delivery/expiry → Message marked for cleanup

6. Cleanup worker (24h default) → Message deleted from Mnesia

Critical Performance Feature:

- **ZERO SQL database queries** during active message routing/delivery
- SQL is completely bypassed for operational message processing
- This eliminates the traditional SMS-C bottleneck (database I/O)

Tier 2: SQL Database for CDR Export/Archival

What is CDR (Call Detail Record)?

- Permanent audit record of message metadata and content
- Written to MySQL or PostgreSQL database
- Used for billing, analytics, compliance, and lawful interception

When CDRs are Written: CDR records are created **ONLY** when messages reach a final state:

- ☐ Message delivered successfully
- ☐ Message expired (exceeded validity period without delivery)
- ☐ Message permanently failed (invalid number, routing error)
- ☐ Message rejected (routing rules, validation failure)

How CDRs are Written:

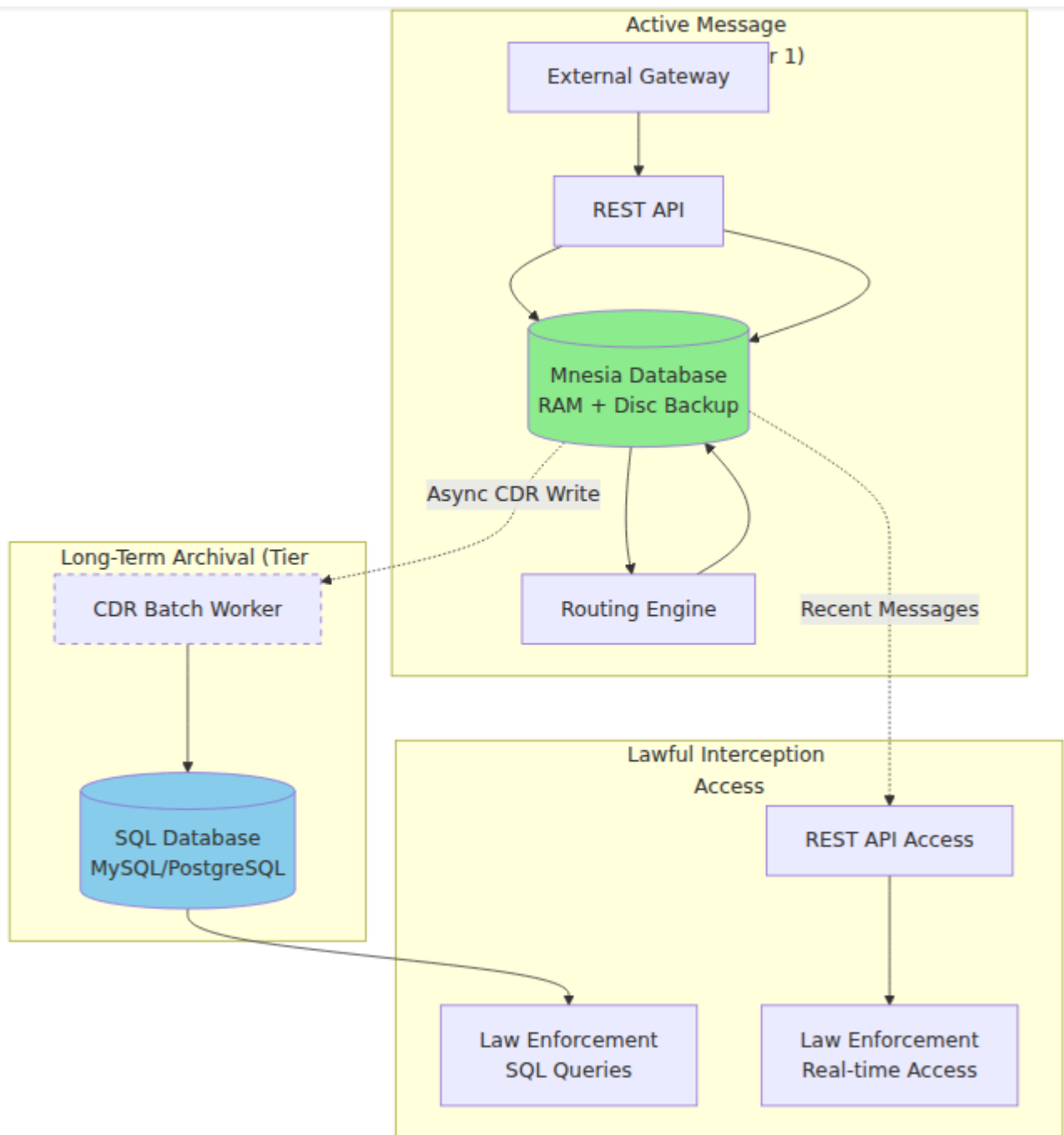
- **Asynchronous batch writing:** CDRs written in background worker process
- **No blocking:** Message routing never waits for SQL write
- **Batched inserts:** Multiple CDRs grouped (default 100) and written together
- **Flush interval:** 100ms default (configurable)
- **Error handling:** Failed CDR writes logged, message processing continues

```
# Configuration in config/runtime.exs
config :sms_c,
  batch_insert_batch_size: 100,          # Batch size for CDR
writes
  batch_insert_flush_interval_ms: 100    # Flush interval
```

SQL Database Purpose:

- ☐ NOT used for: Active message queue operations
- ☐ NOT used for: Message routing decisions
- ☐ NOT used for: Real-time message delivery
- ☐ ONLY used for: Long-term CDR archival and historical queries
- ☐ ONLY used for: Lawful interception queries (months/years of history)
- ☐ ONLY used for: Billing and analytics reports

Architecture Diagram



Legend:

- Solid lines: Synchronous operations (real-time)
- Dashed lines: Asynchronous operations (background)
- Green: High-performance tier (in-memory)
- Blue: Archival tier (persistent SQL)

Lawful Interception Implications

Recent Messages (< 24 hours):

- Accessible via Mnesia (REST API queries)
- Ultra-fast retrieval
- Full message content available
- Real-time monitoring possible

Historical Messages (> 24 hours):

- Accessible via SQL database (CDR table)
- Standard SQL query performance
- Full message metadata always available
- Message body available (unless privacy mode enabled)

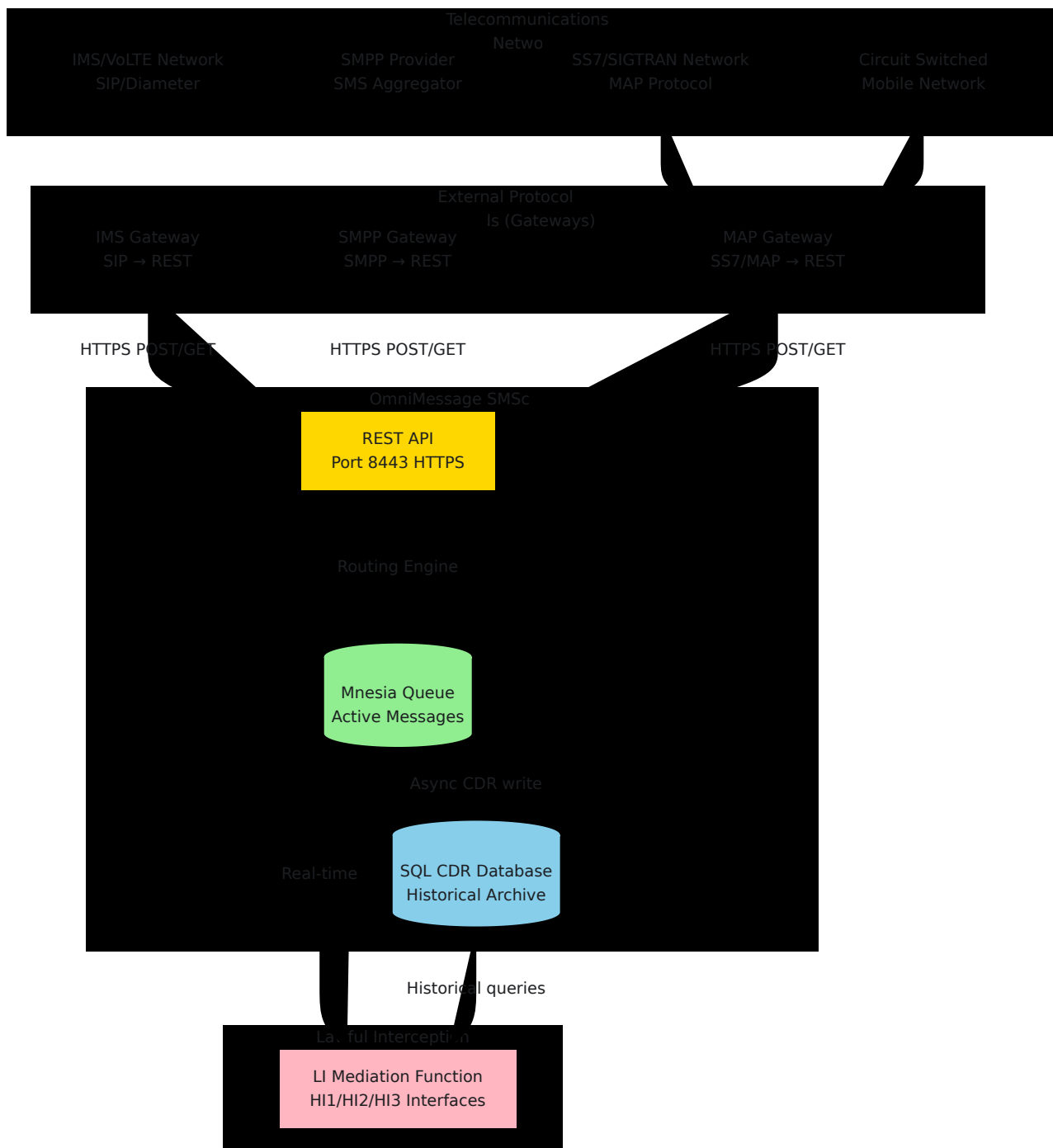
Compliance Benefits:

1. **No data loss:** Disc_copies mode ensures messages survive crashes
2. **Permanent audit trail:** CDRs retained for years in SQL database
3. **Performance:** Lawful interception queries don't impact message routing
4. **Flexibility:** Recent messages (Mnesia) + historical messages (SQL) both accessible

1.5 Multi-Protocol Frontend Integration Architecture

The OmniMessage SMSc employs a protocol-agnostic core design that interfaces with external protocol-specific gateways (frontends) via a unified REST API. This architecture allows lawful interception to capture messages regardless of which telecommunications protocol was used to send or receive them.

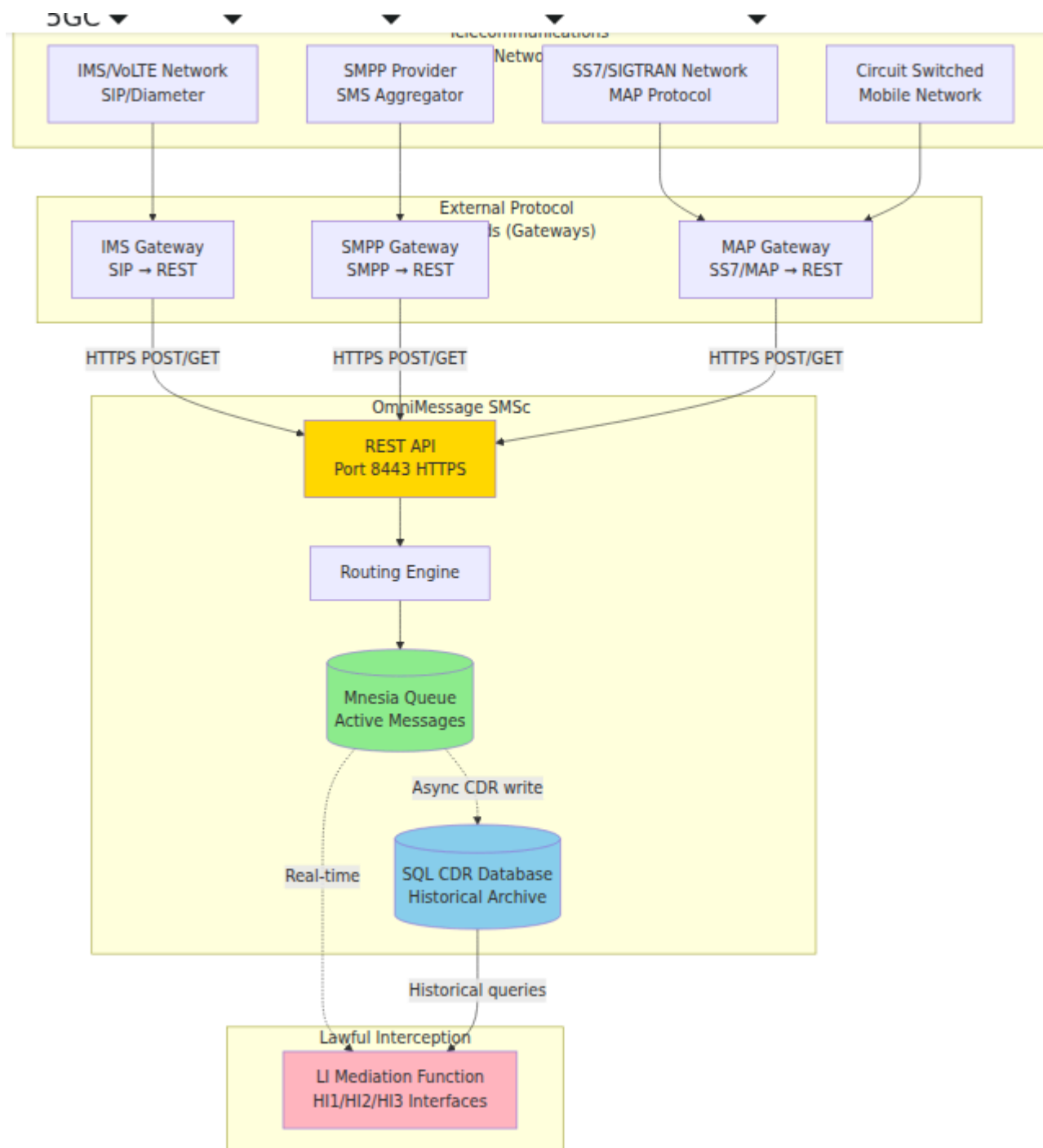
Architecture Overview



Frontend Protocol Integration Details

1. IMS/SIP Frontend Integration

IMS networks use SIP protocol for SMS-over-IP messaging. The IMS gateway translates between SIP and the SMSc REST API.

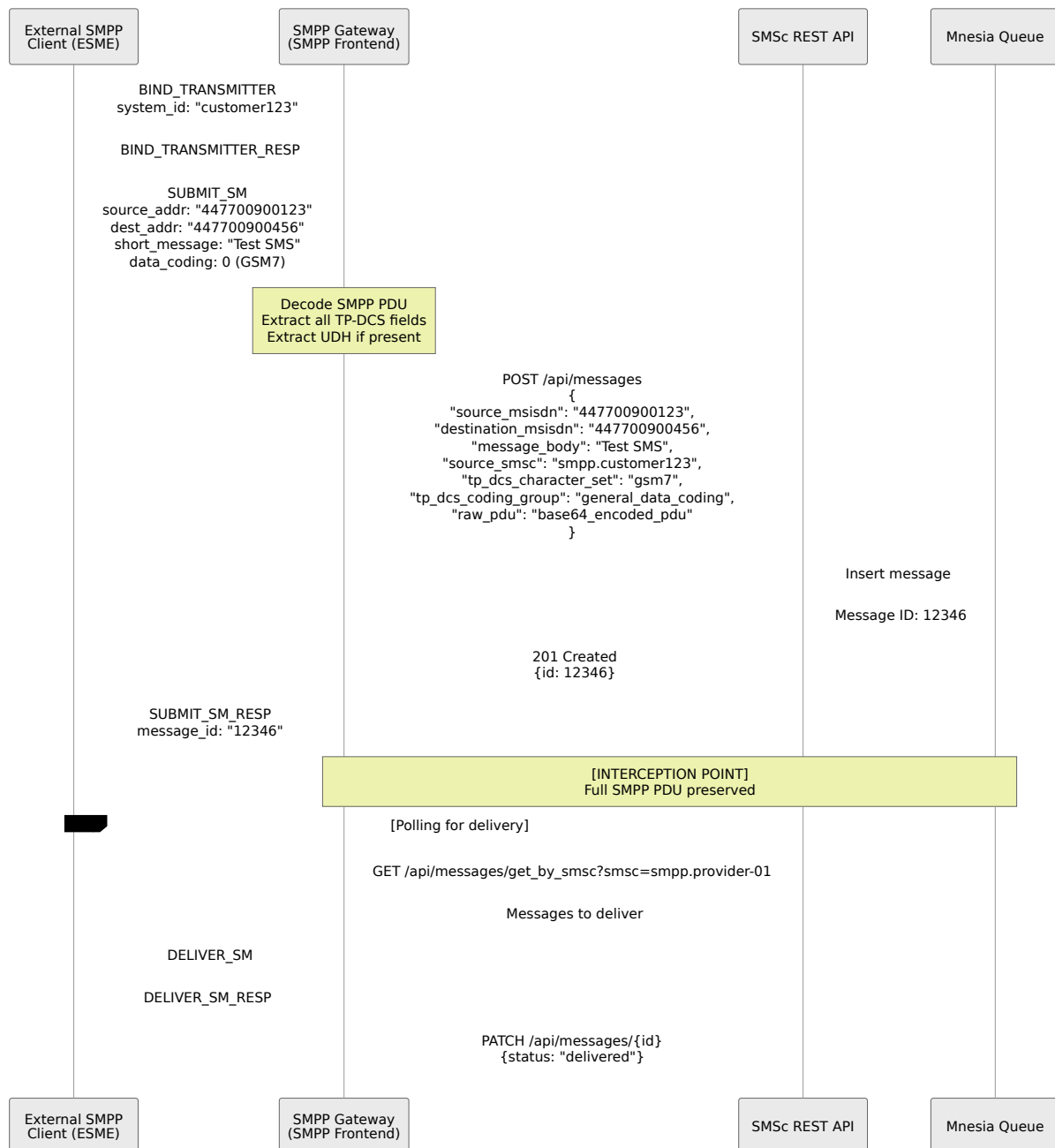


IMS-Specific Interception Data:

- Source/Destination IMSI (from IMS registration)
- P-Asserted-Identity SIP headers
- SIP Call-ID for correlation
- IMS network location (P-Access-Network-Info)
- Subscriber profiles from IMS HSS

2. SMPP Frontend Integration

SMPP is the industry-standard protocol for SMS aggregators and service providers. The SMPP gateway translates PDU-based SMPP messages to REST API calls.



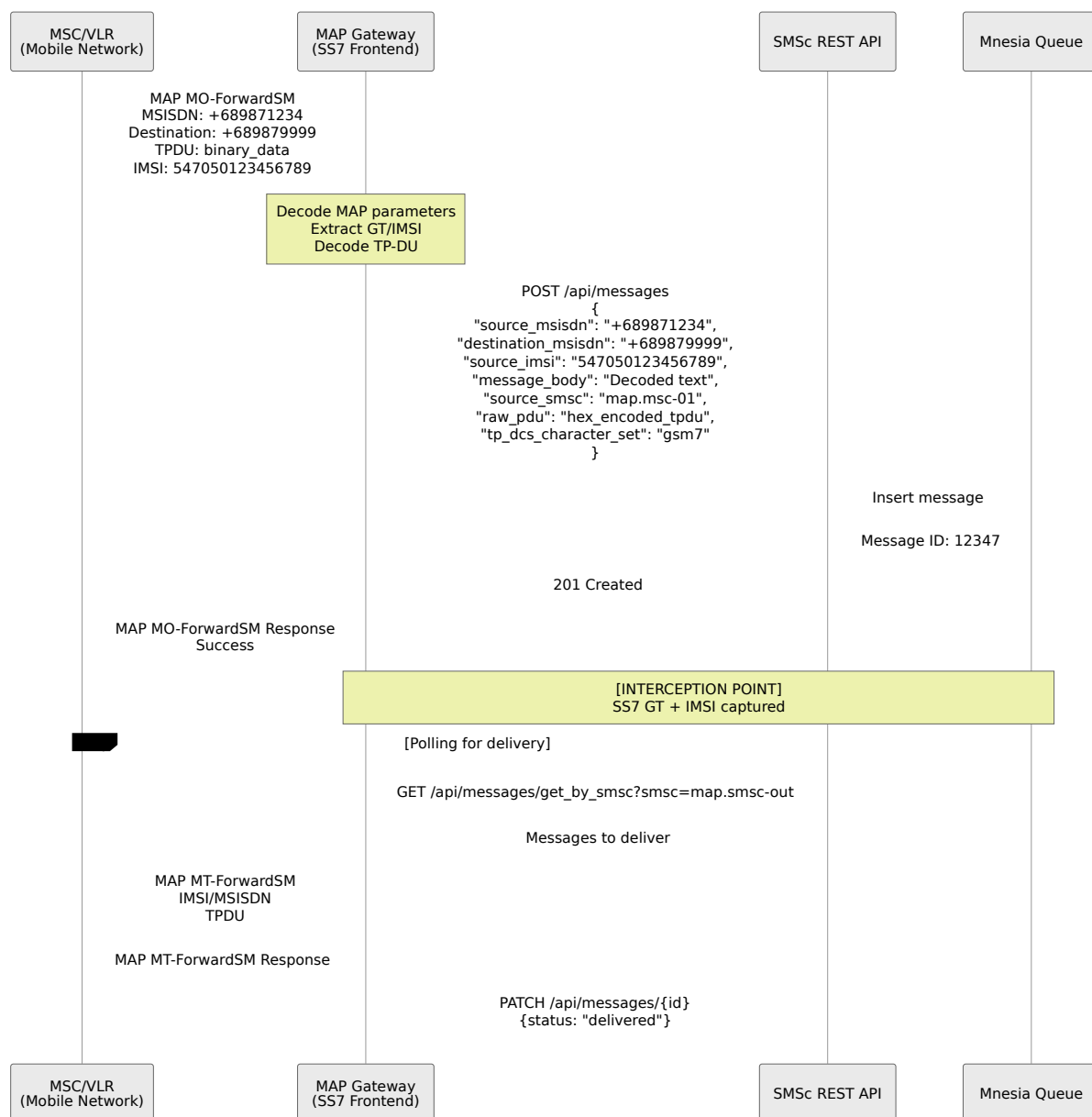
SMPP-Specific Interception Data:

- Complete SMPP PDU (binary format preserved)
- Data Coding Scheme (DCS) details
- User Data Header (UDH) for concatenated messages
- ESME system_id (customer identification)
- TON/NPI numbering plan information

- Registered delivery flags

3. SS7/MAP Frontend Integration

Legacy circuit-switched networks use SS7 MAP protocol for SMS. The MAP gateway translates between SS7 signaling and REST API.



SS7/MAP-Specific Interception Data:

- IMSI from MAP messages
- Global Title (GT) addresses
- MSC/VLR address (network element identification)
- SCCP calling/called party addresses

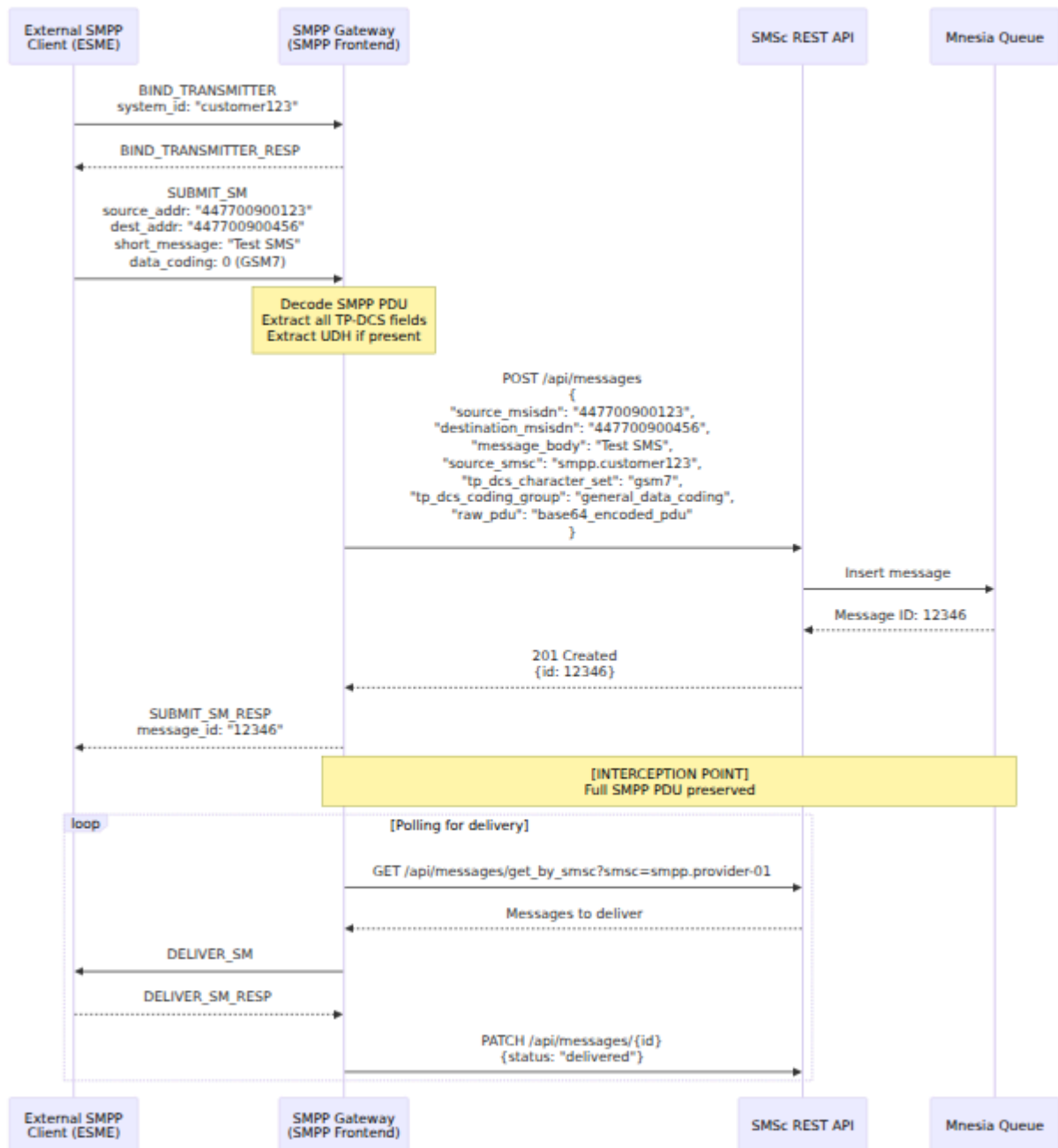
- MAP operation codes
- TP-User-Data binary format

Unified Interception Across All Protocols

Key Benefit for Lawful Interception: Regardless of which protocol was used (IMS/SIP, SMPP, or SS7/MAP), all messages converge in the SMSc core with normalized data structure, enabling:

1. **Protocol-Agnostic Monitoring:** Single interception point captures all message types
2. **Unified CDR Format:** All protocols write to same CDR schema
3. **Cross-Protocol Correlation:** Track messages across protocol boundaries
4. **Complete Metadata Preservation:** Protocol-specific fields preserved in CDR

Data Flow Summary:



Protocol Identification in CDR:

- `source_smsc` field indicates frontend protocol (e.g., "ims.gateway-01", "smpp.customer123", "map.msc-01")
- Enables filtering and analysis by protocol type
- Lawful interception queries can target specific protocols or all protocols

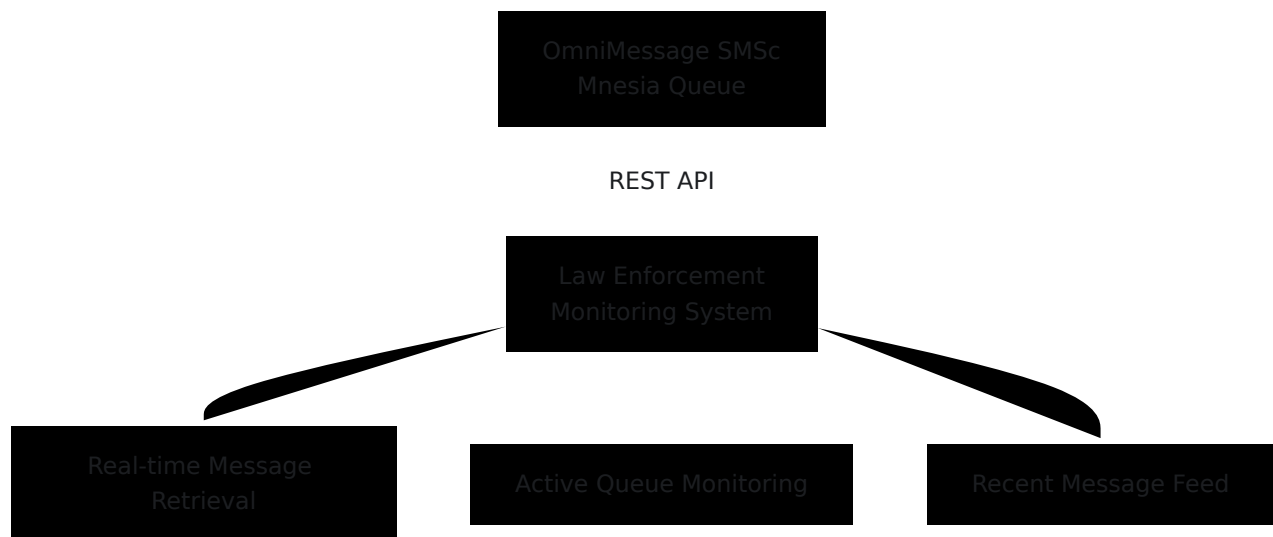
1.6 Technical Architecture for Lawful Interception

Lawful Interception Integration Points

The two-tier storage architecture provides multiple access points for lawful interception, optimized for both real-time monitoring (Mnesia) and historical analysis (SQL).

1. REST API Access for Recent Messages (Mnesia):

Access to active messages in the Mnesia queue (typically last 24 hours):



API Endpoints for Real-Time Interception:

- `GET /api/messages` - List active messages with filtering
- `GET /api/messages/{id}` - Get specific message details (from Mnesia)
- `GET /api/messages/get_by_smsc?smsc=X` - Get messages by gateway
- All queries hit Mnesia (in-memory) for instant response

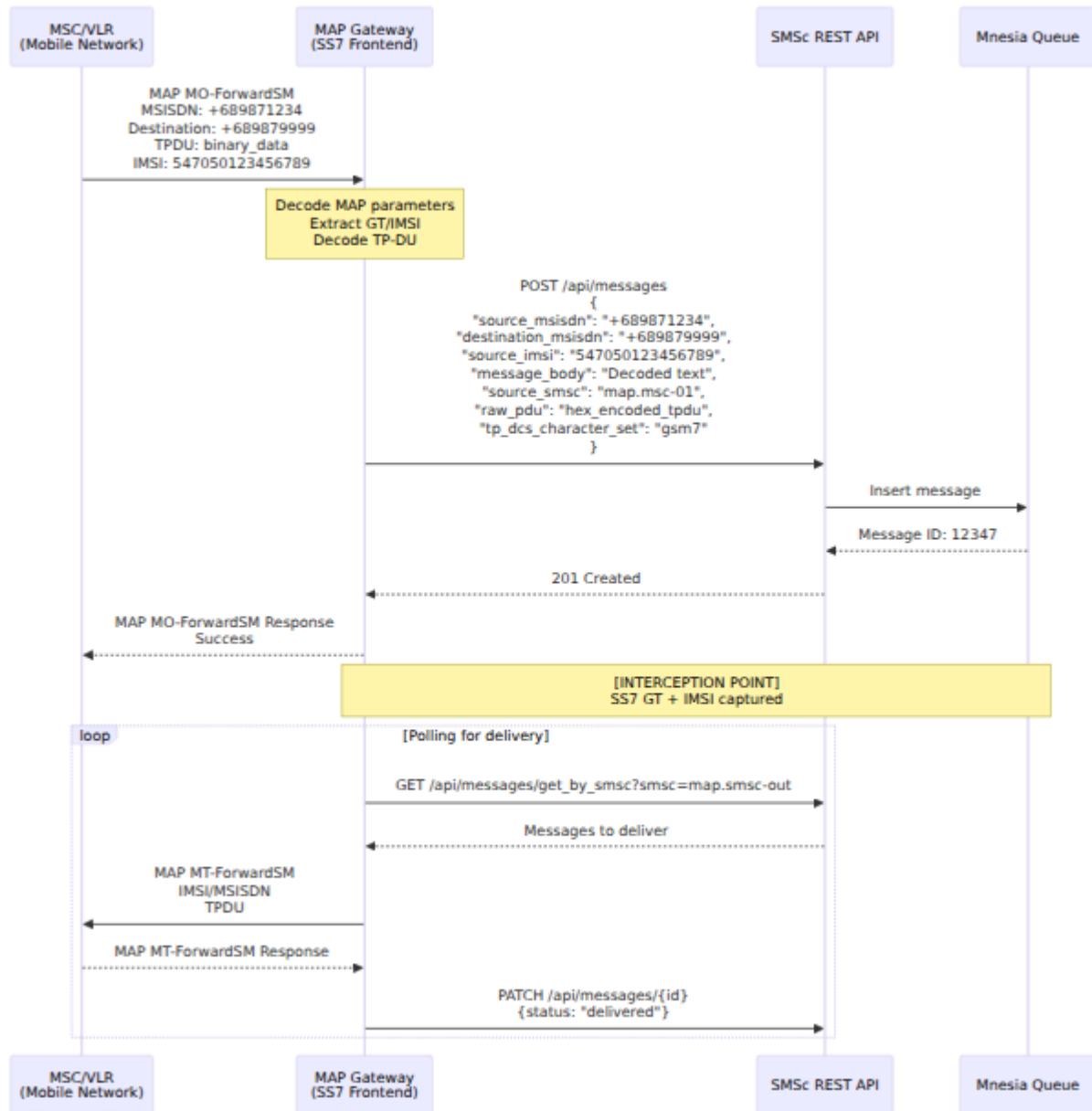
Note: These endpoints query the active Mnesia message queue, providing access to messages currently being processed or recently delivered (within retention period).

Query Parameters:

- Filter by source/destination MSISDN
- Filter by time range
- Filter by SMSC gateway
- Filter by message status
- Sort and pagination support

2. CDR Database Direct Access for Historical Messages (SQL):

Access to archived messages in the SQL database (all delivered/expired/failed messages):



Direct SQL Access:

- Read-only database credentials for authorized systems
- SQL query access to `cdrs` table (permanent audit trail)
- **Access Method:** Standard SQL client (mysql, psql, DBeaver, etc.)
- **Data Source:** Only archived messages (not active queue)
- Indexed fields for efficient searching:
 - `calling_number` (indexed) - Source phone number

- `called_number` (indexed) - Destination phone number
- `message_id` (indexed) - Unique message identifier
- `submission_time` (indexed) - When message entered system
- `status` (indexed) - Final delivery status
- `dest_smsc` (indexed) - Gateway used for delivery

Note: CDR database contains permanent records of all processed messages. This is the primary data source for historical lawful interception queries (months/years of data).

3. Real-Time Message Feed (PubSub):

- Phoenix PubSub integration for real-time events
- Message submission notifications
- Message delivery notifications
- Message status change events
- Configurable event filtering by criteria
- WebSocket support for live monitoring

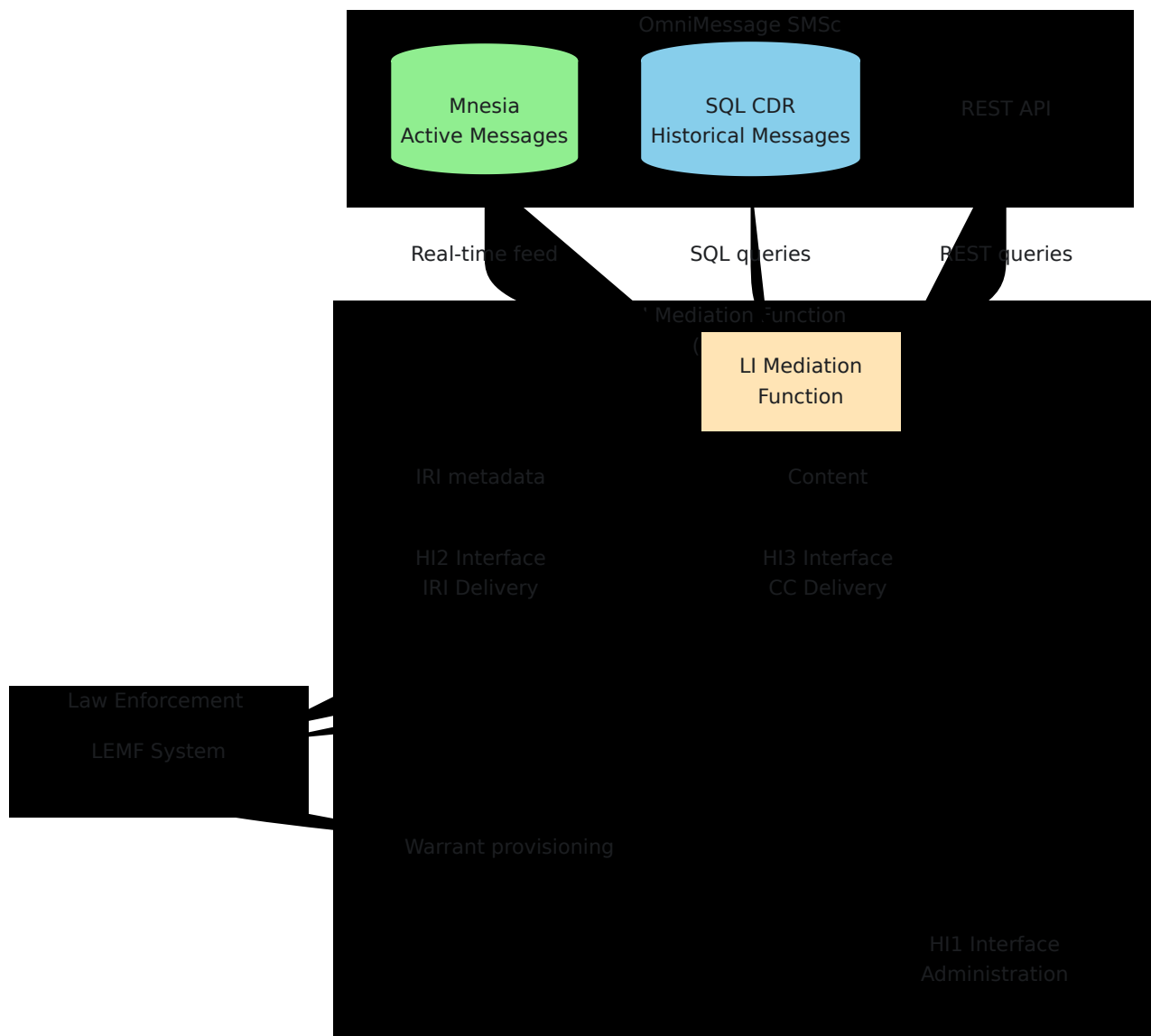
4. Batch Export Interface:

- CSV export of CDR records
- JSON export for programmatic access
- Configurable export fields
- Time-range based exports
- Privacy-aware exports (optional message body exclusion)

ETSI Lawful Interception Standard Interfaces

The OmniMessage SMSc provides the foundation for implementing ETSI-compliant lawful interception interfaces. While the SMSc core does not natively implement HI1/HI2/HI3 handover interfaces, it provides all necessary data access points that can be integrated with external Lawful Interception Mediation Function (LIMF) systems.

Standard ETSI LI Interfaces:



Interface Descriptions:

HI1 Interface - Administration Function:

- **Purpose:** Warrant and target provisioning from law enforcement to interception system
- **Direction:** LEMF → LIMF (bidirectional)
- **Functions:**
 - Activate/deactivate interception for specific targets (MSISDNs, IMSIs)
 - Set interception duration and validity period
 - Configure filtering criteria (phone numbers, time windows)
 - Retrieve interception status
- **Integration with SMSG:**
 - LIMF maintains target list (warrant database)

- LIMF queries SMSc CDR/API for matching messages
- LIMF filters based on HI1 provisioned criteria

HI2 Interface - IRI (Intercept Related Information) Delivery:

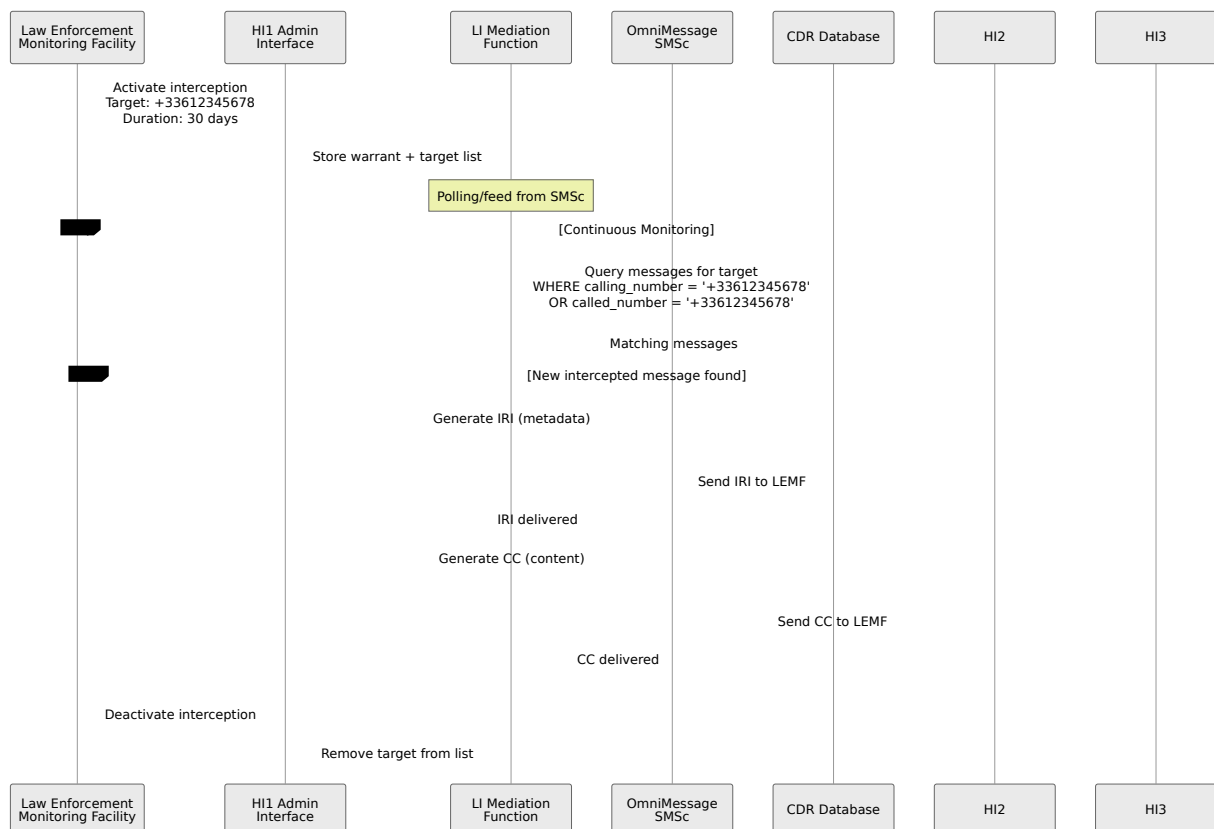
- **Purpose:** Deliver message metadata to law enforcement
- **Direction:** LIMF → LEMF (one-way)
- **Data Format:** ETSI TS 102 232-x compliant XML/ASN.1
- **Content from SMSc CDR:**
 - Message ID
 - Calling number (source MSISDN)
 - Called number (destination MSISDN)
 - IMSI (source and destination, if available)
 - Submission timestamp
 - Delivery timestamp
 - Message status (delivered/failed/expired)
 - Delivery attempts
 - SMSC gateway information (source/destination)
 - Network location (if available)
- **Integration with SMSc:**
 - LIMF queries CDR database for target phone numbers
 - LIMF transforms CDR records into ETSI IRI format
 - LIMF delivers IRI to LEMF via HI2

HI3 Interface - CC (Content of Communication) Delivery:

- **Purpose:** Deliver actual message content to law enforcement
- **Direction:** LIMF → LEMF (one-way)
- **Data Format:** ETSI TS 102 232-x compliant
- **Content from SMSc:**
 - Message body (text content)
 - Raw PDU (binary SMS data)
 - Character encoding information
 - Multipart message segments
 - TP-DCS information

- User Data Header (UDH)
- **Integration with SMS:**
 - LIMF retrieves message content from CDR `message_body` field
 - LIMF retrieves raw PDU data if available
 - LIMF packages content in ETSI CC format
 - LIMF delivers CC to LEMF via HI3

Implementation Architecture:



SMS Data Mapping to LI Interfaces:

SMSc Data Field	HI2 (IRI)	HI3 (CC)	CDR Table Column
Message ID	☐ Correlation ID	☐ Reference	message_id
Calling Number	☐ Party A	-	calling_number
Called Number	☐ Party B	-	called_number
Submission Time	☐ Timestamp	-	submission_time
Delivery Time	☐ Completion	-	delivery_time
Status	☐ Result	-	status
Message Body	-	☐ Content	message_body
Raw PDU	-	☐ Binary	(Mnesia/CDR)
Source SMSC	☐ Network element	-	source_smsc
Dest SMSC	☐ Network element	-	dest_smsc
IMSI	☐ Subscriber ID	-	(Via frontends)

LIMF Integration Options:

Option 1: Polling Architecture

- LIMF periodically queries CDR database (every 1-60 seconds)
- SQL query filters by target phone numbers from HI1 warrant list
- Low complexity, easy to implement
- Slight delay between message delivery and LI delivery

Option 2: Real-Time Feed Architecture

- SMSc PubSub publishes message events
- LIMF subscribes to real-time message stream
- LIMF filters based on target list
- Near-zero latency for lawful interception
- Requires custom integration development

Option 3: Hybrid Architecture

- Recent messages: Real-time PubSub feed (< 24 hours)
- Historical messages: CDR database polling
- Optimal balance of latency and reliability

Interception Triggering Mechanisms

Target-Based Interception:

- Phone number matching (MSISDN)
- IMSI-based targeting (when available)
- Configurable watch lists
- Database views for target isolation
- API filtering by target identifiers

Event-Based Interception:

- All messages to/from specific numbers
- Messages via specific SMSC gateways
- Messages with specific characteristics (multi-part, failed delivery, etc.)
- Geographic routing (via ENUM or prefix matching)

Time-Based Interception:

- Date/time range filtering in CDR queries
- Retention period enforcement
- Automatic archival of old messages
- Configurable data retention policies

Example SQL Queries for Lawful Interception:

```
-- Get all messages for target number
SELECT * FROM cdrs
WHERE calling_number = '+33612345678'
      OR called_number = '+33612345678'
ORDER BY submission_time DESC;

-- Get messages in specific time window
SELECT * FROM cdrs
WHERE (calling_number = '+33612345678' OR called_number =
'+33612345678')
      AND submission_time BETWEEN '2025-11-01 00:00:00' AND '2025-11-
30 23:59:59'
ORDER BY submission_time;

-- Get conversation between two parties
SELECT * FROM cdrs
WHERE (calling_number = '+33612345678' AND called_number =
'+33687654321')
      OR (calling_number = '+33687654321' AND called_number =
'+33612345678')
ORDER BY submission_time;
```

2. ENCRYPTION AND CRYPTANALYSIS CAPABILITIES

2.1 Cryptographic Capabilities Overview

The OmniMessage SMSc implements cryptographic mechanisms for securing communications and protecting sensitive data. This section documents all cryptographic capabilities in accordance with ANSSI requirements.

2.2 Transport Layer Encryption

2.2.1 TLS/SSL Implementation

Supported Protocols:

- TLS 1.2 (RFC 5246)
- TLS 1.3 (RFC 8446) - Recommended
- SSL 2.0/3.0: NOT SUPPORTED (known vulnerabilities)
- TLS 1.0/1.1: DEPRECATED (not recommended)

Implementation:

- Erlang/OTP SSL/TLS library (cryptographically validated)
- Cowboy web server with TLS support
- Phoenix Framework HTTPS endpoints

Cipher Suites:

The system uses Erlang/OTP's default secure cipher suite selection, which includes:

Preferred - TLS 1.3:

- TLS_AES_256_GCM_SHA384
- TLS_AES_128_GCM_SHA256
- TLS_CHACHA20_POLY1305_SHA256

Supported - TLS 1.2:

- ECDHE-RSA-AES256-GCM-SHA384
- ECDHE-RSA-AES128-GCM-SHA256
- DHE-RSA-AES256-GCM-SHA384
- DHE-RSA-AES128-GCM-SHA256

Security Features:

- Perfect Forward Secrecy (PFS) via ECDHE/DHE key exchange
- Strong Diffie-Hellman groups (2048-bit minimum)
- Elliptic Curve Cryptography support
- Server Name Indication (SNI) support

Certificate Management:

- X.509 certificate support
- RSA key sizes: 2048-bit minimum, 4096-bit recommended
- ECDSA support
- Certificate chain validation
- Self-signed certificates (development only)
- External CA integration

TLS Configuration Location:

```
# config/runtime.exs
config :api_ex,
  api: %{
    enable_tls: true,
    tls_cert_path: "priv/cert/omnitouch.crt",
    tls_key_path: "priv/cert/omnitouch.pem"
  }
```

📖 Complete configuration reference in **CONFIGURATION.md**

Applications:

- HTTPS for REST API (port 8443)
- HTTPS for web control panel (port 8086)
- Database connections (MySQL/PostgreSQL over TLS)

2.3 Data Encryption at Rest

2.3.1 Database Encryption

MySQL/MariaDB Encryption:

- Table-level encryption support
- AES-256 encryption algorithm
- Transparent data encryption (TDE)

```
-- Enable encryption for CDR table  
ALTER TABLE cdrs ENCRYPTION='Y';
```

PostgreSQL Encryption:

- Transparent data encryption support
- Filesystem-level encryption
- Column-level encryption (pgcrypto extension)

2.3.2 Mnesia Disc Storage

Mnesia Database:

- Disc copies storage for message persistence
- File system-level encryption recommended (LUKS, dm-crypt)
- Memory protection via Erlang VM isolation

2.3.3 File System Encryption

Sensitive Data Storage:

- Configuration files: Filesystem encryption recommended
- Private keys: File permissions (0600) + filesystem encryption
- Log files: Configurable encryption for archived logs
- CDR exports: Encrypted storage for sensitive exports

Key Storage:

- TLS certificates and keys stored in `priv/cert/`
- File-based keystores with restricted permissions
- Secure key rotation procedures

2.4 Authentication and Access Control

2.4.1 API Authentication

REST API Security:

- HTTPS/TLS transport encryption mandatory
- Header-based authentication (SMSc header for frontend identification)
- IP-based access control (firewall level)
- Certificate-based client authentication (optional)

Frontend Registration:

- Unique frontend identification (name, type, IP, hostname)
- Heartbeat-based authentication
- Expiration-based session management (90-second timeout)
- Frontend tracking and monitoring

2.4.2 Database Authentication

Database Access Control:

- Username/password authentication
- TLS/SSL connection support
- IP-based connection restrictions
- Role-based access control (RBAC)

Configuration:

```
# config/runtime.exs
config :sms_c, SmsC.Repo,
  username: "omnitouch",
  password: "omnitouch2024", # Should use strong passwords in
production
  hostname: "localhost",
  ssl: true # Enable TLS for database connections
```

Access Control Recommendations:

```
-- Create read-only user for law enforcement access
CREATE USER 'li_readonly'@'%' IDENTIFIED BY 'secure_password';
GRANT SELECT ON sms_c.cdcs TO 'li_readonly'@'%';

-- Create limited user without message body access
CREATE USER 'analytics'@'%' IDENTIFIED BY 'secure_password';
GRANT SELECT (id, message_id, calling_number, called_number,
              source_smsc, dest_smsc, submission_time,
              delivery_time,
              status, delivery_attempts)
ON sms_c.cdcs TO 'analytics'@'%';
```

2.5 Cryptographic Algorithm Details

2.5.1 Hashing Algorithms

Available in Erlang/OTP:

- SHA-256, SHA-384, SHA-512 (recommended)
- SHA-1 (deprecated, legacy compatibility only)
- MD5 (deprecated, not used for security)
- BLAKE2 (available in modern OTP versions)

Usage:

- Message fingerprinting (duplicate detection)
- Data integrity verification
- Audit log integrity

2.5.2 Symmetric Encryption

Available Algorithms:

- AES (Advanced Encryption Standard)
 - AES-128-GCM
 - AES-256-GCM
 - AES-128-CBC
 - AES-256-CBC

- ChaCha20-Poly1305

Key Sizes:

- 128-bit (minimum)
- 256-bit (recommended)

Usage:

- TLS session encryption
- Database encryption at rest
- Optional message body encryption

2.5.3 Asymmetric Encryption**Supported Algorithms:**

- RSA (2048-bit minimum, 4096-bit recommended)
- ECDSA (Elliptic Curve Digital Signature Algorithm)
 - P-256, P-384, P-521 curves
- Ed25519 (EdDSA)

Usage:

- TLS certificate authentication
- Digital signatures
- Key exchange

2.6 SMS Protocol Security**2.6.1 SMS Message Encoding****Character Encoding Support:**

- GSM 7-bit (standard SMS encoding)
- UCS-2 (Unicode, 16-bit)
- 8-bit binary data
- Latin-1

TP-DCS (Data Coding Scheme):

- Message class indication
- Compression flags
- Coding group specification
- Character set identification

No Native SMS Encryption:

- SMS protocol does not provide end-to-end encryption
- Message content accessible at SMSc level
- Enables lawful interception as required

2.6.2 Protocol Security Considerations

SMPP Protocol (External Frontend):

- Username/password authentication at SMPP level
- TLS support available (SMPP over TLS)
- Bind authentication

IMS Protocol (External Frontend):

- SIP-based messaging
- SIP authentication mechanisms
- Integration with IMS core network security

SS7/MAP Protocol (External Frontend):

- SS7 network security
- MAP protocol authentication
- SCCP/TCAP layer security

Note: Protocol-specific security is implemented in external frontend gateways, not in the SMSc core.

2.7 Cryptanalysis and Security Assessment Capabilities

2.7.1 Protocol Analysis Tools

Built-in Debugging Capabilities:

- Comprehensive logging system
- Message flow tracing
- API request/response logging
- Database query logging
- Error and exception tracking

External Integration:

- Standard logging output (stdout/files)
- PCAP capture support for network analysis
- Database query logging for forensics
- Prometheus metrics export

2.7.2 Vulnerability Assessment Considerations

Known Limitations:

- SMS protocol inherently unencrypted (by design, enables lawful interception)
- Database credentials in configuration files (should use secrets management)
- Self-signed certificate support (development/testing only)

Security Hardening Recommendations:

- Use strong TLS cipher suites
- Implement database connection encryption
- Use external secrets management (Vault, AWS Secrets Manager)
- Regular security updates for Erlang/OTP and dependencies
- Firewall restrictions on API ports

- IP whitelisting for frontend access

Security Testing:

- Regular dependency vulnerability scanning
- Penetration testing support
- TLS configuration validation
- Database security audits
- Access control review

2.8 Key Management Infrastructure

2.8.1 Key Generation

TLS Certificate Generation:

```
# Generate private key (RSA 4096-bit)
openssl genrsa -out omnitouch.pem 4096

# Generate certificate signing request
openssl req -new -key omnitouch.pem -out omnitouch.csr

# Self-signed certificate (development)
openssl x509 -req -days 365 -in omnitouch.csr -signkey
omnitouch.pem -out omnitouch.crt

# Production: Obtain certificate from trusted CA
```

Random Number Generation:

- Erlang/OTP CSPRNG (Cryptographically Secure Pseudo-Random Number Generator)
- System entropy pool (/dev/urandom)
- Strong randomness for session keys, IDs, tokens

2.8.2 Key Storage and Protection

Private Key Storage:

- File system with restricted permissions (0600)
- Stored in `priv/cert/` directory
- PEM format (optionally encrypted)
- Secure backup procedures

Key Rotation:

- TLS certificate renewal (annually recommended)
- Database credential rotation
- API token rotation (if implemented)

2.8.3 Key Distribution

Certificate Distribution:

- Manual installation in `priv/cert/`
- Configuration file references
- ACME protocol support possible (Let's Encrypt)

Symmetric Key Distribution:

- Out-of-band key exchange for database credentials
- Diffie-Hellman key agreement in TLS
- No cleartext key transmission

2.9 Compliance and Standards

This section documents compliance with international telecommunications standards, regulatory frameworks, and security specifications applicable to SMS processing across all supported protocols.

2.9.1 SMS over SS7/MAP Protocol Compliance

3GPP and ETSI Standards:

- **3GPP TS 23.040:** Technical realization of Short Message Service (SMS) - Core SMS protocol specification

- **3GPP TS 23.038:** Alphabets and language-specific information - Character encoding (GSM7, UCS-2)
- **3GPP TS 29.002:** Mobile Application Part (MAP) specification - SS7 signaling for SMS
- **3GPP TS 23.003:** Numbering, addressing and identification - MSISDN, IMSI formats
- **ETSI TS 100 901:** Point-to-Point Short Message Service specification
- **ETSI TS 100 902:** Cell Broadcast Short Message Service specification

SS7 Signaling Standards:

- **ITU-T Q.711-Q.716:** Signaling Connection Control Part (SCCP)
- **ITU-T Q.771-Q.775:** Transaction Capabilities Application Part (TCAP)
- **ITU-T Q.701-Q.710:** Message Transfer Part (MTP) Levels 1-3
- **ETSI EN 300 356:** Signaling System No.7 - ISDN User Part (ISUP)

Security Standards for SS7/MAP:

- **GSMA FS.07:** SS7 and Diameter Signaling Security - Baseline security controls
- **GSMA FS.11:** SS7 Security Monitoring Guidelines
- **3GPP TS 33.117:** Catalogue of general security assurance requirements
- **ETSI TS 133 210:** Network domain security - IP network layer security

Lawful Interception for SS7/MAP:

- **ETSI TS 101 671:** Lawful Interception (LI); Handover interface for the lawful interception of telecommunications traffic
- **ETSI TS 102 232-1:** Lawful Interception (LI); Handover specification - Part 1: Handover interface for LI management
- **3GPP TS 33.107:** Lawful Interception architecture and functions for 3G networks

2.9.2 SMS over IMS Protocol Compliance

3GPP IMS Standards:

- **3GPP TS 23.228:** IP Multimedia Subsystem (IMS) - Stage 2 architecture

- **3GPP TS 24.229:** IP Multimedia Call Control Protocol - SIP and SDP procedures
- **3GPP TS 24.341:** Support of SMS over IP networks - Stage 3 specification
- **3GPP TS 23.204:** Support of Short Message Service (SMS) over generic 3GPP IP access - Stage 2
- **3GPP TS 29.228:** IP Multimedia (IM) Subsystem Cx and Dx interfaces

IMS Security Standards:

- **3GPP TS 33.203:** 3G security; Access security for IP-based services (IMS AKA)
- **3GPP TS 33.210:** 3G security; Network Domain Security (NDS); IP network layer security (IPsec)
- **3GPP TS 33.310:** Network Domain Security (NDS); Authentication Framework (AF)
- **ETSI TS 133 203:** Access security for IP-based services

SIP Protocol Standards:

- **RFC 3261:** SIP: Session Initiation Protocol - Core specification
- **RFC 3428:** SIP Extension for Instant Messaging - MESSAGE method
- **RFC 3325:** Private Extensions to SIP for Asserted Identity
- **RFC 5765:** Security Issues and Solutions in Peer-to-Peer Systems

Lawful Interception for IMS:

- **ETSI TS 102 232-5:** Lawful Interception (LI); Handover specification - Part 5: Service-independent LI for IP Multimedia Subsystem services
- **3GPP TS 33.107:** Lawful Interception requirements and architecture
- **3GPP TS 33.108:** Handover interface for Lawful Interception (LI)

2.9.3 SMPP Protocol Compliance

SMPP Specification:

- **SMPP v3.4:** Short Message Peer-to-Peer Protocol Specification - Industry standard
- **SMPP v5.0:** Extended SMPP protocol with enhanced features

SMPP Security Guidelines:

- **TLS over SMPP:** Transport layer security for SMPP connections (SMPP over TLS)
- **SMPP Bind Authentication:** System ID and password authentication
- **IP-based Access Control:** Network-level restrictions for SMPP connections

Interoperability Standards:

- **GSM 03.40 (ETSI TS 100 901):** Technical realization of SMS Point-to-Point (PP)
- **GSM 03.38 (ETSI TS 100 900):** Alphabets and language-specific information
- **GSM 04.11 (ETSI TS 100 942):** Point-to-Point SMS support on mobile radio interface

Message Encoding Compliance:

- **ITU-T T.50:** International Alphabet No. 5 (IA5)
- **ISO/IEC 8859-1:** Latin-1 character encoding
- **ISO/IEC 10646:** Universal Character Set (UCS-2/UTF-16)

2.9.4 Cryptographic Standards Compliance

TLS and Network Security:

- **NIST SP 800-52:** Guidelines for the Selection, Configuration, and Use of TLS Implementations
- **NIST SP 800-131A:** Transitioning the Use of Cryptographic Algorithms and Key Lengths
- **RFC 7525:** Recommendations for Secure Use of TLS and DTLS
- **RFC 8446:** The Transport Layer Security (TLS) Protocol Version 1.3

Cryptographic Algorithm Standards:

- **FIPS 197:** Advanced Encryption Standard (AES)
- **FIPS 180-4:** Secure Hash Standard (SHA-2 family)

- **NIST SP 800-38D:** Recommendation for Block Cipher Modes of Operation: GCM Mode
- **RFC 7539:** ChaCha20 and Poly1305 for IETF Protocols

Key Management:

- **NIST SP 800-57:** Recommendation for Key Management
- **RFC 5280:** Internet X.509 Public Key Infrastructure Certificate and CRL Profile

2.10 Cryptanalysis Resistance

2.10.1 Design Principles

Defense Against Cryptanalysis:

- No custom/proprietary cryptographic algorithms
- Industry-standard, peer-reviewed algorithms only
- Regular security updates for cryptographic libraries
- Deprecation of weak algorithms
- Use of authenticated encryption (GCM, Poly1305)

2.10.2 Operational Security

Key Rotation:

- TLS certificate renewal procedures
- Session key rotation (per-session for TLS)
- Database credential rotation policies

Monitoring and Detection:

- Failed authentication logging
 - Certificate expiration monitoring
 - TLS handshake failure logging
 - Anomaly detection for encryption failures
 - Security event alerting
-

3. INTERCEPTION CONTROL AND AUTHORIZATION

3.1 Access Control for Lawful Interception

Administrative Authorization:

- System administrator access required for configuration
- Database-level access controls for CDR queries
- API access restricted by IP/authentication
- Audit logging of all access

Legal Framework Integration:

- Interception warrant tracking (external system integration)
- Target identifier authorization lists (database views)
- Time-limited queries (SQL WHERE clauses)
- Automatic enforcement via access policies

3.2 Data Retention and Privacy

Retention Policies:

- Active message retention: Configurable (default 24 hours in Mnesia)
- CDR retention: Configurable (typical 6 months to 2 years)
- Automatic archival from Mnesia to SQL
- Automatic purging of old CDRs (cron-based)

Privacy Protections:

- Message body deletion option after delivery
- Message body exclusion from UI/exports
- Database encryption at rest
- Access logging and monitoring
- Minimal data collection principle

Configuration:

```
# config/runtime.exs
config :sms_c,
  # Mnesia message retention before archival
  message_retention_hours: 24,

  # Delete message body after delivery for privacy
  delete_message_body_after_delivery: false, # Set true for
privacy mode

  # CDR writing control
  cdr_enabled: true,

  # Batch archival settings
  batch_insert_batch_size: 100,
  batch_insert_flush_interval_ms: 100
```

☐ See **CONFIGURATION.md** for all retention settings

3.3 Handover Interfaces for Law Enforcement

Standard Interfaces:

1. REST API Access:

- HTTPS endpoints for message retrieval
- JSON format data exchange
- Authentication and authorization
- Query filtering by target criteria

2. Direct Database Access:

- Read-only SQL credentials
- Standard SQL queries
- CDR table access
- Indexed search capabilities

3. Batch Export:

- CSV export format
- JSON export format
- Time-range based exports
- Configurable field selection

Delivery Formats:

IRI (Intercept Related Information):

- CDR metadata fields:
 - Message ID
 - Calling/called numbers
 - Timestamps (submission, delivery, expiry)
 - Status
 - Delivery attempts
 - SMSC routing information
 - Node information (cluster tracking)

CC (Content of Communication):

- Message body (text content)
- Raw PDU data
- Encoding information
- Multipart message assembly

Export Example:


```
# CSV export for law enforcement
mysql -u li_readonly -p -D sms_c -e "
SELECT
    message_id,
    calling_number,
    called_number,
    message_body,
    submission_time,
    delivery_time,
    status
FROM cdrs
WHERE (calling_number = '+33612345678' OR called_number =
'+33612345678')
    AND submission_time BETWEEN '2025-11-01' AND '2025-11-30'
ORDER BY submission_time
" --batch --silent | sed 's/\t/,/g' > interception_report.csv
```

4. SYSTEM SECURITY AND INTEGRITY

4.1 Application Security

Elixir/Erlang Security:

- Erlang VM isolation and sandboxing
- Process isolation and supervision
- Crash recovery and fault tolerance
- No buffer overflow vulnerabilities (managed runtime)

Dependency Management:

- Dependency version locking (mix.lock)
- Security vulnerability scanning
- Regular dependency updates
- Minimal dependency footprint

4.2 Network Security

Network Exposure:

- Minimal exposed ports:
 - 8443 (HTTPS REST API)
 - 8086 (HTTPS Control Panel)
- Firewall configuration recommended
- IP whitelisting for frontend access
- DMZ deployment for internet-facing services

Network Segmentation:

- Separate management network
- Isolated database network
- Frontend gateway network separation
- Cluster communication network (Erlang distribution)

4.3 Monitoring and Intrusion Detection

Logging Capabilities:

- Structured application logging
- Configurable log levels
- Log rotation and archival
- Syslog integration support
- Centralized logging (ELK stack compatible)

Security Event Monitoring:

- Failed authentication attempts
- Unusual message patterns
- Database connection failures
- TLS handshake failures
- System resource anomalies

Metrics and Alerting:

- Prometheus metrics export
- Message throughput monitoring
- Error rate tracking
- System resource utilization
- Custom alert rules

☐ **Complete monitoring documentation in [OPERATIONS_GUIDE.md](#) and [METRICS.md](#)**

4.4 High Availability and Disaster Recovery

Cluster Support:

- Erlang distributed cluster capability
- Mnesia replication across nodes
- Automatic failover
- Node discovery and joining

Data Redundancy:

- Mnesia disc_copies on all cluster nodes
- SQL database replication (MySQL/PostgreSQL native)
- CDR backup procedures
- Configuration backup

Recovery Procedures:

- Database backup and restore
 - Mnesia table recovery
 - Configuration restoration
 - Node replacement procedures
-

5. DOCUMENTATION REFERENCES

5.1 Technical Manuals

Available documentation in the project repository:

- **README.md** - System overview, architecture, and features
- **CONFIGURATION.md** - Complete configuration reference
- **API_REFERENCE.md** - REST API documentation
- **OPERATIONS_GUIDE.md** - Operational procedures and monitoring
- **CDR_SCHEMA.md** - Call Detail Record database schema
- **sms_routing_guide.md** - SMS routing configuration
- **number_translation_guide.md** - Number normalization
- **METRICS.md** - Prometheus metrics and monitoring
- **PERFORMANCE_TUNING.md** - Performance optimization
- **TROUBLESHOOTING.md** - Common issues and solutions

5.2 Security Certifications

- **Penetration Test Reports:** [To be provided upon request]
- **Security Audit Reports:** [To be provided upon request]
- **Vulnerability Assessments:** [To be provided upon request]
- **Erlang/OTP Cryptographic Validation:** Industry-standard cryptographic library

5.3 Compliance Documentation

- **ANSSI R226 Authorization Request:** This document
 - **Lawful Interception Compliance:** As required by French telecommunications regulations
 - **Data Protection Compliance:** GDPR considerations for message data
-

6. CONTACT INFORMATION

Vendor/Operator Information:

- Company Name: Omnitouch Network Services Pty Ltd
- Address: PO BOX 296, QUINNS ROCKS WA 6030, AUSTRALIA
- Contact Person: Compliance Team
- Email: compliance@omnitouch.com.au

Technical Security Contact:

- Name: Compliance Team
- Email: compliance@omnitouch.com.au

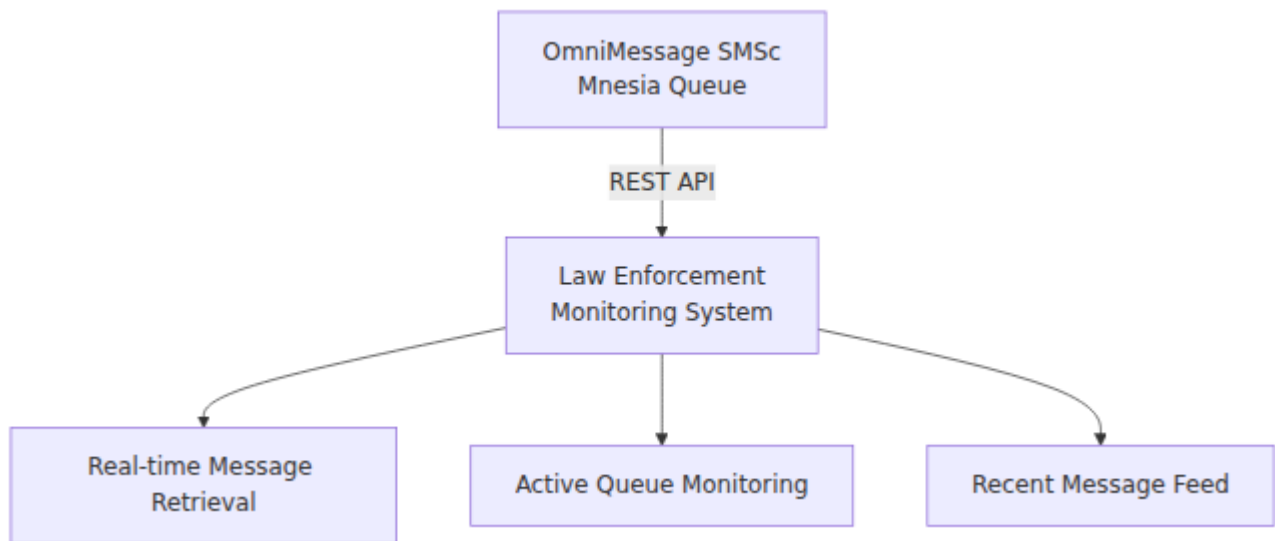
Legal/Compliance Contact:

- Name: Compliance Team
 - Email: compliance@omnitouch.com.au
-

APPENDICES

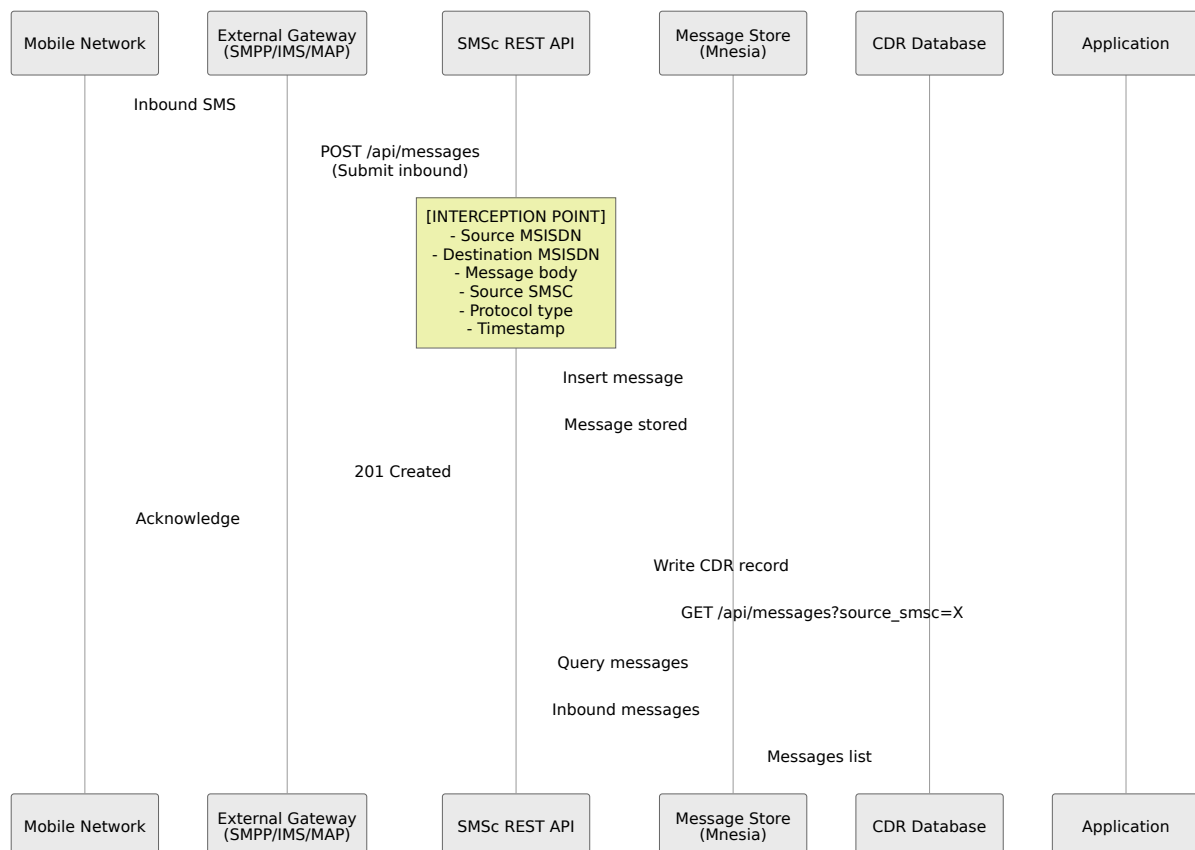
Appendix A: SMS Message Flow with Interception Points

A.1 Outbound SMS Flow (Mobile Terminated)



Legend: [INTERCEPTION POINT] = Points where lawful interception data is captured and stored

A.2 Inbound SMS Flow (Mobile Originated)



Appendix B: CDR Schema for Lawful Interception

The OmniMessage SMSc stores Call Detail Records in a SQL database (MySQL or PostgreSQL) for long-term retention and lawful interception access.

B.1 Key CDR Fields for Lawful Interception

Field Name	Type	Description
id	BIGINT	Auto-incrementing primary key
message_id	BIGINT	Unique message identifier
calling_number	VARCHAR(255)	Source MSISDN
called_number	VARCHAR(255)	Destination MSISDN
source_smsc	VARCHAR(255)	Source gateway identifier
dest_smsc	VARCHAR(255)	Destination gateway identifier
origin_node	VARCHAR(255)	Erlang cluster node (origination)
destination_node	VARCHAR(255)	Erlang cluster node (delivery)
submission_time	DATETIME	Message submission timestamp
delivery_time	DATETIME	Message delivery timestamp
expiry_time	DATETIME	Message expiry timestamp

Field Name	Type	Description
status	VARCHAR(50)	Message status (delivered/expired/failed/rejected)
delivery_attempts	INT	Number of delivery attempts
message_parts	INT	Number of SMS segments
deadletter	BOOLEAN	Dead letter queue flag
message_body	TEXT	SMS message content
inserted_at	DATETIME	CDR creation timestamp
updated_at	DATETIME	CDR update timestamp

📄 **Complete schema documentation with SQL examples in**
CDR_SCHEMA.md

B.2 CDR Query Examples for Lawful Interception

Query all messages for target number:

```
SELECT * FROM cdrs
WHERE calling_number = '+33612345678'
      OR called_number = '+33612345678'
ORDER BY submission_time DESC;
```

Query messages within time window:

```
SELECT * FROM cdrs
WHERE (calling_number = '+33612345678' OR called_number =
'+33612345678')
    AND submission_time BETWEEN '2025-11-01 00:00:00' AND '2025-11-
30 23:59:59'
ORDER BY submission_time;
```

Export to CSV for law enforcement:

```
.mode csv
.output /tmp/interception_report.csv
SELECT message_id, calling_number, called_number, message_body,
        submission_time, delivery_time, status
FROM cdrs
WHERE calling_number = '+33612345678'
ORDER BY submission_time DESC;
```

B.3 CDR Database Access Methods

1. Direct SQL Access:

- Read-only database credentials
- Standard SQL queries
- JDBC/ODBC connectivity
- Database client tools (MySQL Workbench, pgAdmin)

2. REST API Access:

- Future enhancement: REST API for CDR queries
- JSON format responses
- Authentication and authorization
- Query parameter filtering

3. Batch Export:

- CSV export via mysql/psql command-line
- Automated export scripts
- Scheduled exports via cron

B.4 CDR Retention and Privacy

Retention Configuration:

```
# config/runtime.exs
config :sms_c,
  # Delete message body after delivery (privacy mode)
  delete_message_body_after_delivery: false, # true for privacy

  # Enable/disable CDR writing
  cdr_enabled: true,

  # Mnesia to CDR archival settings
  message_retention_hours: 24
```

Privacy Options:

- Message body can be set to NULL after delivery
- Database table encryption (MySQL ENCRYPTION='Y')
- Column-level access restrictions
- Masked exports for analytics

Appendix C: REST API Reference for Interception

C.1 Message Retrieval Endpoints

Get all messages:

```
GET /api/messages
Authorization: Bearer <token>
```

Get messages by SMSC:

```
GET /api/messages/get_by_smsc?smsc=gateway-name
```

Get specific message:

GET /api/messages/{id}

Response Format:

```
{
  "status": "success",
  "data": [
    {
      "id": 12345,
      "message_id": 12345,
      "source_msisdn": "+33612345678",
      "destination_msisdn": "+33687654321",
      "message_body": "Message content here",
      "source_smsc": "ims.gateway",
      "dest_smsc": "smpp.provider",
      "status": "delivered",
      "delivery_attempts": 1,
      "inserted_at": "2025-11-29T10:30:00Z",
      "deliver_time": "2025-11-29T10:30:05Z",
      "expires": "2025-11-30T10:30:00Z"
    }
  ]
}
```

📄 Complete API documentation in [API_REFERENCE.md](#)

C.2 Filtering and Search

Query Parameters:

- `source_smsc` - Filter by source gateway
- `dest_smsc` - Filter by destination gateway
- Time-range filtering (via CDR database queries)
- Status filtering (via CDR database queries)

Future Enhancements:

- REST API for CDR queries
- Advanced filtering by phone number

- Date range filtering
- Full-text search on message body

Appendix D: Configuration Examples

D.1 TLS Certificate Configuration

Generate TLS Certificate:

```
# Generate 4096-bit RSA private key
openssl genrsa -out priv/cert/omnitouch.pem 4096

# Generate certificate signing request
openssl req -new -key priv/cert/omnitouch.pem -out
priv/cert/omnitouch.csr \
    -subj "/C=FR/ST=IDF/L=Paris/O=Omnitouch/CN=smc.example.com"

# Self-signed certificate (development/testing)
openssl x509 -req -days 365 -in priv/cert/omnitouch.csr \
    -signkey priv/cert/omnitouch.pem -out priv/cert/omnitouch.crt

# Production: Submit CSR to trusted CA for signing
```

Configure in application:

```
# config/runtime.exs
config :api_ex,
  api: %{
    enable_tls: true,
    tls_cert_path: "priv/cert/omnitouch.crt",
    tls_key_path: "priv/cert/omnitouch.pem"
  }
```

D.2 Database Encryption Configuration

MySQL Table Encryption:

```
-- Enable encryption for CDR table
ALTER TABLE cdrs ENCRYPTION='Y';

-- Verify encryption status
SELECT TABLE_NAME, CREATE_OPTIONS
FROM information_schema.TABLES
WHERE TABLE_SCHEMA = 'sms_c' AND TABLE_NAME = 'cdrs';
```

PostgreSQL Connection Encryption:

```
# config/runtime.exs
config :sms_c, SmsC.Repo,
  username: "omnitouch",
  password: "secure_password",
  hostname: "localhost",
  database: "sms_c",
  ssl: true,
  ssl_opts: [
    verify: :verify_peer,
    cacertfile: "/path/to/ca.crt",
    certfile: "/path/to/client.crt",
    keyfile: "/path/to/client.key"
  ]
```

D.3 Privacy Configuration

Enable Message Body Deletion:

```
# config/runtime.exs
config :sms_c,
  # Delete message body after successful delivery
  delete_message_body_after_delivery: true,

  # Hide message body in UI
  hide_message_body_in_ui: true,

  # Hide message body in exports
  hide_message_body_in_export: true
```

Create Privacy-Preserving Database View:

```
-- Create view without message bodies for general analytics
CREATE VIEW cdrs_metadata AS
SELECT
    id, message_id, calling_number, called_number,
    source_smsc, dest_smsc, origin_node, destination_node,
    submission_time, delivery_time, expiry_time,
    status, delivery_attempts, message_parts, deadletter,
    inserted_at, updated_at
FROM cdrs;

-- Grant access to analytics users
GRANT SELECT ON cdrs_metadata TO 'analytics'@'%';
```

Appendix E: Glossary

Regulatory and Standards Bodies

- **ANSSI:** Agence nationale de la sécurité des systèmes d'information - French National Cybersecurity Agency
- **ETSI:** European Telecommunications Standards Institute
- **3GPP:** 3rd Generation Partnership Project - Mobile telecommunications standards
- **IETF:** Internet Engineering Task Force - Internet standards body

Telecommunications Terms

- **SMSc:** SMS Service Center - Central system for SMS message routing and delivery
- **SMPP:** Short Message Peer-to-Peer protocol - Industry standard for SMS exchange
- **ESME:** External Short Message Entity - SMPP client application
- **IMS:** IP Multimedia Subsystem - All-IP network architecture for multimedia services
- **SIP:** Session Initiation Protocol - Signaling protocol for IMS messaging
- **P-CSCF:** Proxy Call Session Control Function - IMS network entry point

- **S-CSCF:** Serving Call Session Control Function - IMS session control
- **HSS:** Home Subscriber Server - IMS subscriber database
- **SS7/MAP:** Signaling System 7 / Mobile Application Part - Legacy mobile signaling protocols
- **MSC:** Mobile Switching Center - Circuit-switched network element
- **VLR:** Visitor Location Register - Subscriber location database
- **GT:** Global Title - SS7 addressing scheme
- **SCCP:** Signaling Connection Control Part - SS7 network layer
- **TCAP:** Transaction Capabilities Application Part - SS7 application layer
- **MSISDN:** Mobile Station International Subscriber Directory Number - Phone number
- **IMSI:** International Mobile Subscriber Identity - Unique subscriber identifier
- **E.164:** International numbering plan for telephone numbers
- **ENUM:** E.164 Number Mapping - DNS-based phone number to URI mapping
- **PDU:** Protocol Data Unit - Binary encoded SMS message
- **TP-DCS:** Transfer Protocol Data Coding Scheme - SMS encoding specification
- **TP-DU:** Transfer Protocol Data Unit - SMS-specific PDU format
- **UDH:** User Data Header - Header for concatenated/special SMS messages
- **TON/NPI:** Type of Number / Numbering Plan Indicator - Number format classification
- **GSM7:** GSM 7-bit default alphabet - Standard SMS character encoding
- **UCS-2:** Universal Character Set 2-byte - Unicode encoding for SMS

System Components

- **Mnesia:** Erlang distributed database system - In-memory/disc storage
- **CDR:** Call Detail Record - Billing and analytics record for messages
- **REST API:** Representational State Transfer - HTTP-based API architecture
- **Phoenix:** Elixir web framework
- **Cowboy:** Erlang HTTP server
- **Ecto:** Elixir database wrapper and query language
- **PubSub:** Publish-Subscribe messaging pattern

Lawful Interception

- **LI:** Lawful Interception - Legal monitoring of telecommunications
- **LIMF:** Lawful Interception Mediation Function - System that interfaces between telecom network and law enforcement
- **LEMF:** Law Enforcement Monitoring Facility - Law enforcement system receiving intercepted data
- **IRI:** Intercept Related Information - Call/message metadata for law enforcement
- **CC:** Content of Communication - Actual message content
- **HI1 Interface:** ETSI LI administrative handover interface - Warrant provisioning and target activation
- **HI2 Interface:** ETSI LI handover interface for IRI delivery - Metadata handover to law enforcement
- **HI3 Interface:** ETSI LI handover interface for CC delivery - Content handover to law enforcement
- **R226:** Articles R226-3 and R226-7 of French Penal Code governing interception equipment
- **ETSI:** European Telecommunications Standards Institute - Defines LI standards
- **ETSI TS 102 232:** Technical specification for lawful interception handover interfaces

Message Processing

- **MT:** Mobile Terminated - Outbound message to mobile subscriber
- **MO:** Mobile Originated - Inbound message from mobile subscriber
- **DLR:** Delivery Receipt - Confirmation of message delivery
- **Dead Letter:** Message that failed delivery after all retry attempts
- **Exponential Backoff:** Increasing retry delay (2min, 4min, 8min, etc.)

Security and Encryption

- **TLS:** Transport Layer Security - Encryption protocol
- **PFS:** Perfect Forward Secrecy - Cryptographic property for session key security
- **AES:** Advanced Encryption Standard
- **RSA:** Rivest-Shamir-Adleman - Public key cryptography

- **ECDSA:** Elliptic Curve Digital Signature Algorithm
- **SHA:** Secure Hash Algorithm
- **X.509:** Certificate standard
- **CA:** Certificate Authority
- **CSPRNG:** Cryptographically Secure Pseudo-Random Number Generator

Database Terms

- **MySQL:** Open-source relational database
- **PostgreSQL:** Open-source object-relational database
- **TDE:** Transparent Data Encryption
- **RBAC:** Role-Based Access Control

Document Version: 1.0 **Date:** 2025-11-29 **Prepared for:** ANSSI R226
Authorization Application **Document Classification:** Regulatory Compliance -
Confidential

Bulk Messaging / Campaigns

[← Back to Documentation Index](#)

Overview

The campaigns feature sends a single plaintext message to many recipients while protecting the SMSC from overload. You upload a named **recipient list** once, then run one or more **campaigns** against it. Each campaign drips messages into the normal message queue at a configurable rate, and reports live progress and delivery statistics.

Two reusable objects:

Object	What it is
Campaign list	A named, reusable set of recipient MSISDNs, each with up to four optional variables. Upload once (JSON array or CSV), reuse across campaigns.
Campaign	A send run: a list (or <i>all active subscribers</i>) + a message template, dripped into the queue.

Everything is exposed over the [HTTP API](#). All state is stored in Mnesia alongside the rest of the SMSC.

Targeting: lists and the active-subscriber default

A campaign's recipients come from one of two sources:

- **A recipient list** (`list_id`) — the MSISDNs you uploaded.

- **No list at all** — if you create a campaign **without** a `list_id`, it targets **all currently active (registered) subscribers** — every MSISDN with an unexpired registration in the location store at the moment the campaign starts. This is the easiest way to reach "everyone on the network right now".

Per-recipient variables and templating

Each list recipient can carry up to four variables — `var1`, `var2`, `var3`, `var4` — for things like first name, last name, or plan name. The campaign message is a **template**: `{{ var1 }}`...`{{ var4 }}` (and `{{ msisdn }}`) are spliced in for each recipient when the message is sent. Whitespace inside the braces is optional, and any placeholder with no value renders as an empty string.

The templating is deliberately minimal — variable substitution only (no conditionals or filters), so there is no extra dependency and no surprises in what subscribers receive.

Uploading variables via CSV — the MSISDN column is the one named `msisdn/destination_msisdn/number/phone` (or the first column if none is named); every other column becomes `var1...var4` in column order:

```
msisdn,first,last,plan
12025550101,Alice,Smith,Gold
12025550102,Bob,Jones,Silver
```

Uploading variables via JSON — recipient objects may set the vars directly:

```
{ "name": "VIPs", "recipients": [
  { "msisdn": "12025550101", "var1": "Alice", "var2": "Gold" }
] }
```

Example template:

Hi {{ var1 }}, your {{ var2 }} plan renews soon. Reply STOP to opt out.

For recipient 12025550101 above (var1=Alice, var2=Gold) this sends: Hi Alice, your Gold plan renews soon. Reply STOP to opt out.

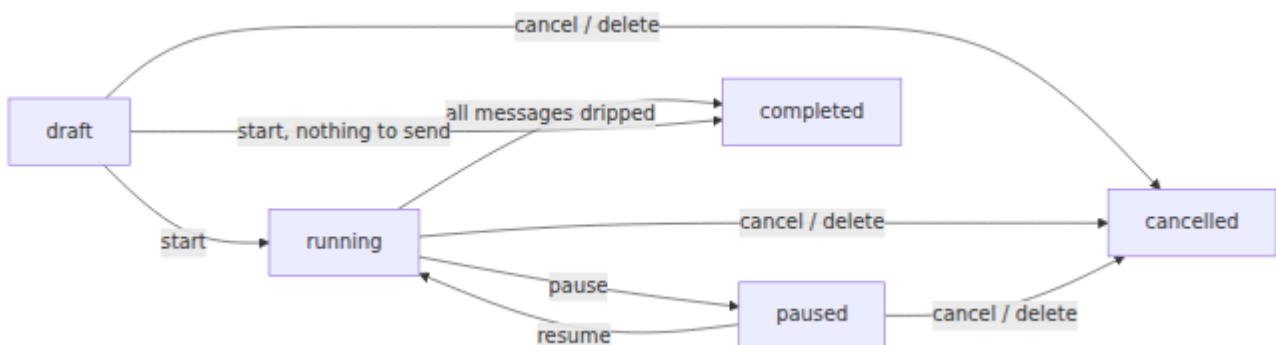
Audience filter

When a campaign uses a list, the `audience` flag decides who actually gets a message:

Audience	Behaviour
<code>active_only</code> (default)	Only list members that are registered (an unexpired location) at the moment the campaign starts. Everyone else is recorded as <code>skipped_inactive</code> and never sent to.
<code>all</code>	Every member of the list, regardless of registration.

The registration check is a **snapshot taken at start time** — subscribers who register later are not picked up. When a campaign has **no list**, the recipient set is already the active-subscriber base, so `audience` is implicitly `active_only` and any supplied value is ignored.

Lifecycle



1. **draft** — created, no targets yet.

2. **start** — the campaign snapshots its targets (from its list or the active subscriber base), then moves to **running**. If there is nothing to send it goes straight to **completed**.
3. **running** — the drip worker submits messages into the queue.
4. **pause / resume** — temporarily stop / continue dripping.
5. **completed** — every target has been dripped into the queue (delivery then continues asynchronously, and delivery stats keep updating).
6. **cancelled** — stopped permanently.

Deleting a campaign

Deleting a campaign **stops it but keeps the record** — it is cancelled rather than purged, so its history and final statistics stay visible. Stopped and completed campaigns are removed automatically once they age past the retention TTL (`campaign_retention_hours`, default 7 days).

Drip rate (overload protection)

Each campaign has a `drip_tps` (messages submitted into the queue per second, default from config). A background worker wakes on a fixed interval and submits at most `drip_tps × interval` messages per campaign per tick — so a one-million recipient campaign never floods the queue. Rate is **per campaign**; multiple running campaigns each drip at their own rate.

Progress and delivery statistics

Every campaign object includes a live `stats` block (poll `GET /api/campaigns/:id`):

Field	Meaning
total	Targets snapshotted at start
pending	Not yet dripped into the queue
queued	Submitted, awaiting a delivery report
delivered	Confirmed delivered
failed	Delivery failed / dead-lettered
skipped_inactive	Skipped because not registered at start
dispatched	queued + delivered + failed
progress_percent	Percent of targets no longer pending
delivery_percent	Percent of dispatched messages delivered

Delivery and failure counts update automatically as delivery reports flow back through the normal SMSC delivery path. Per-recipient detail (and filtering by state, e.g. everyone that failed) is available via `GET /api/campaign_targets/:id?state=failed`.

Configuration

```
config :sms_c, :campaigns,
  # Start the drip worker (set false to pause ALL campaign
  delivery)
  enabled: true,
  # Interval between drip ticks in milliseconds
  drip_tick_ms: 1000,
  # Per-campaign messages/second when a campaign does not set its
  own drip_tps
  default_drip_tps: 50,
  # Default message validity window when a campaign sets no
  validity_hours
  message_validity_hours: 72,
  # How long a stopped/completed campaign persists before it is
  purged (hours)
  campaign_retention_hours: 168
```

Validity and scheduled delivery

A campaign can set, at creation time:

- **validity_hours** — the message validity period. Each submitted message is stamped with an **expires** so undelivered messages lapse instead of lingering in the queue. Defaults to **message_validity_hours** from config.
- **deliver_after** — an optional ISO8601 time (must be in the future) for a scheduled send.

How scheduled delivery interacts with the drip. The campaign still starts dripping *immediately* at **drip_tps**, but each message's clock is shifted forward by a fixed offset $\text{delta} = \text{deliver_after} - \text{started_at}$. So a message dripped at real time t is stamped $\text{send_time}/\text{deliver_after} = t + \text{delta}$ and $\text{expires} = t + \text{delta} + \text{validity_hours}$. Because messages are inserted at **drip_tps** in real time, their per-message **deliver_after** values come out **staggered at drip_tps starting at deliver_after** — delivery is rate-limited rather than a thundering herd at the scheduled instant. With no **deliver_after**, $\text{delta} = 0$ and messages are dripped and delivered immediately.

Typical workflow

```
# 1. Upload a named recipient list with per-recipient variables (CSV)
curl -k -X POST https://smsc:8443/api/campaign_lists \
  -H 'Content-Type: application/json' \
  -d '{"name":"VIP customers",

"csv":"msisdn,first,plan\n12025550101,Alice,Gold\n12025550102,Bob,Sil
# => {"id": 1, "recipient_count": 2, ...}

# 2. Create a campaign against that list, templating in the variables
curl -k -X POST https://smsc:8443/api/campaigns \
  -H 'Content-Type: application/json' \
  -d '{"name":"Renewal notice","message":"Hi {{ var1 }}, your {{ var2
renews soon.",
      "source_msisdn":"12345","source_smsc":"IMS_SMSC",
      "list_id":1,"audience":"active_only","drip_tps":50}'
# => {"id": 7, "status": "draft", ...}

# 3. Start it
curl -k -X POST https://smsc:8443/api/campaigns/control \
  -H 'Content-Type: application/json' \
  -d '{"campaign_id":7,"action":"start"}'

# 4. Poll progress / delivery stats
curl -k https://smsc:8443/api/campaigns/7
# => {... "stats":
{"total":...,"delivered":...,"failed":...,"progress_percent":...}}
```

To message **all active subscribers**, omit `list_id` in step 2 (and skip step 1 entirely).

See also

- [Campaigns API reference](#)
- [SMS Routing Guide](#) — how dripped messages are routed/delivered
- [Configuration Reference](#)

CDR (Call Detail Record) Schema Reference

[← Back to Documentation Index](#)

Complete reference for the CDR database table used for long-term message storage, billing, and analytics.

Table of Contents

- [Overview](#)
- [Table Schema](#)
- [Field Descriptions](#)
- [SQL Examples](#)
- [Indexes](#)
- [Data Types by Database](#)
- [Privacy Considerations](#)
- [Retention and Archival](#)
- [Billing Integration](#)

Overview

The `cdrs` table stores Call Detail Records for all SMS messages processed by the system. A CDR is written the moment a message reaches a terminal state — `delivered`, `expired`, `failed`, or `rejected` (an out-of-balance `balance_rejected` is recorded as `rejected`) — with a retention-time sweep as a backstop for anything that ages out of Mnesia without a live write. See [When CDRs Are Written](#) for the full write path and idempotency guarantees.

CDRs provide long-term storage separate from the operational Mnesia database, enabling:

- Billing and invoicing
- Analytics and reporting
- Compliance and auditing
- Message history beyond Mnesia retention period

Table Schema

MySQL / MariaDB

```
CREATE TABLE cdrs (  
  id BIGINT AUTO_INCREMENT PRIMARY KEY,  
  
  -- Message identification  
  message_id BIGINT NOT NULL,  
  
  -- Phone numbers  
  calling_number VARCHAR(255) NOT NULL,  
  called_number VARCHAR(255) NOT NULL,  
  
  -- SMSC routing  
  source_smsc VARCHAR(255),  
  dest_smsc VARCHAR(255),  
  
  -- Node information (for clustered deployments)  
  origin_node VARCHAR(255),  
  destination_node VARCHAR(255),  
  
  -- Timestamps  
  submission_time DATETIME NOT NULL,  
  delivery_time DATETIME,  
  expiry_time DATETIME,  
  
  -- Status and metadata  
  status VARCHAR(50) NOT NULL,  
  delivery_attempts INT DEFAULT 0,  
  message_parts INT,  
  deadletter BOOLEAN DEFAULT FALSE,  
  
  -- Optional message body (privacy controls)  
  message_body TEXT,  
  
  -- Audit timestamps  
  inserted_at DATETIME NOT NULL,  
  updated_at DATETIME NOT NULL,  
  
  -- Indexes  
  UNIQUE KEY cdrs_origin_node_message_id_index (origin_node,
```

```
message_id),  
  INDEX idx_cdrs_message_id (message_id),  
  INDEX idx_cdrs_calling_number (calling_number),  
  INDEX idx_cdrs_called_number (called_number),  
  INDEX idx_cdrs_status (status),  
  INDEX idx_cdrs_submission_time (submission_time),  
  INDEX idx_cdrs_dest_smsc (dest_smsc)  
);
```

The `(origin_node, message_id)` unique key gives each CDR a globally-unique identity across the cluster and makes CDR writes idempotent (the near-real-time write and the retention backstop can both attempt an insert; the second is a no-op via `ON CONFLICT DO NOTHING`).

PostgreSQL

```
CREATE TABLE cdrs (  
    id BIGSERIAL PRIMARY KEY,  
  
    -- Message identification  
    message_id BIGINT NOT NULL,  
  
    -- Phone numbers  
    calling_number VARCHAR(255) NOT NULL,  
    called_number VARCHAR(255) NOT NULL,  
  
    -- SMSC routing  
    source_smsc VARCHAR(255),  
    dest_smsc VARCHAR(255),  
  
    -- Node information (for clustered deployments)  
    origin_node VARCHAR(255),  
    destination_node VARCHAR(255),  
  
    -- Timestamps  
    submission_time TIMESTAMP NOT NULL,  
    delivery_time TIMESTAMP,  
    expiry_time TIMESTAMP,  
  
    -- Status and metadata  
    status VARCHAR(50) NOT NULL,  
    delivery_attempts INTEGER DEFAULT 0,  
    message_parts INTEGER,  
    deadletter BOOLEAN DEFAULT FALSE,  
  
    -- Optional message body (privacy controls)  
    message_body TEXT,  
  
    -- Audit timestamps  
    inserted_at TIMESTAMP NOT NULL,  
    updated_at TIMESTAMP NOT NULL  
);  
  
-- Indexes  
CREATE UNIQUE INDEX cdrs_origin_node_message_id_index ON  
cdrs(origin_node, message_id);  
CREATE INDEX idx_cdrs_message_id ON cdrs(message_id);
```

```
CREATE INDEX idx_cdrs_calling_number ON cdrs(calling_number);
CREATE INDEX idx_cdrs_called_number ON cdrs(called_number);
CREATE INDEX idx_cdrs_status ON cdrs(status);
CREATE INDEX idx_cdrs_submission_time ON cdrs(submission_time);
CREATE INDEX idx_cdrs_dest_smsc ON cdrs(dest_smsc);
```

Field Descriptions

Primary Key

Field	Type	Nullable	Description
<code>id</code>	BIGINT	NO	Auto-incrementing primary key for the CDR record

Message Identification

Field	Type	Nullable	Description
<code>message_id</code>	BIGINT	NO	Message identifier from the SMS-C message queue. Generated by a per-node counter (persisted in Mnesia, so it survives restarts), therefore only unique within a single node . For a globally-unique key across a multi-node cluster, use <code>(origin_node, message_id)</code> together — this pair is enforced by a unique index and is the recommended join/dedup key for billing systems.

Phone Numbers

Field	Type	Nullable	Description
<code>calling_number</code>	VARCHAR(255)	NO	Source MSISDN (mobile number) of the message sender. Typically in E.164 format (e.g., +15551234567).
<code>called_number</code>	VARCHAR(255)	NO	Destination MSISDN (mobile number) of the message recipient. Typically in E.164 format (e.g., +15551234567).

SMSC Routing

Field	Type	Nullable	Description
<code>source_smsc</code>	VARCHAR(255)	YES	Name or identifier of the source SMSC that submitted the message. NULL if submitted via API or other non-SMSC interface.
<code>dest_smsc</code>	VARCHAR(255)	YES	Name or identifier of the destination SMSC that delivered (or attempted to deliver) the message. NULL if message was never routed.

Node Information

For clustered deployments, tracks which nodes handled the message:

Field	Type	Nullable	Description
<code>origin_node</code>	VARCHAR(255)	YES	Erlang node name where message was originally received (e.g., <code>"sms@node1.example.com"</code>). Useful for troubleshooting and load distribution analysis.
<code>destination_node</code>	VARCHAR(255)	YES	Erlang node name where message was delivered from (if different from origin). NULL for single-node deployments or if message never delivered.

Timestamps

All timestamps are stored in UTC:

Field	Type	Nullable	Description
submission_time	DATETIME	NO	When the message was first submitted to the SMS-C. Used as the start time for billing calculations.
delivery_time	DATETIME	YES	When the message was successfully delivered. NULL if message expired, failed, or was rejected.
expiry_time	DATETIME	YES	When the message expired (became undeliverable). NULL if message was delivered or is still pending.

Delivery Duration Calculation:

```

TIMESTAMPDIFF(SECOND, submission_time, delivery_time) AS
delivery_duration_seconds

```

Status and Metadata

Field	Type	Nullable	Description
status	VARCHAR(50)	NO	Final message status. Valid values: delivered, expired, failed, rejected
delivery_attempts	INT	NO	Number of delivery attempts made before final status. Default: 0. Range: 0-255 typically.
message_parts	INT	YES	Number of SMS segments for concatenated messages. 1 for single-part messages, 2+ for multi-part. NULL if unknown.
deadletter	BOOLEAN	NO	Whether message was moved to dead letter queue. TRUE indicates message couldn't be delivered and exhausted all retries. Default: FALSE

Status Values:

Status	Description	Billable	Delivery Time
<code>delivered</code>	Successfully delivered to recipient	Yes	Set
<code>expired</code>	Exceeded validity period without delivery	Depends on billing policy	NULL
<code>failed</code>	Permanent delivery failure (invalid number, etc.)	Depends on billing policy	NULL
<code>rejected</code>	Rejected by routing rules or validation	No	NULL

Message Body

Field	Type	Nullable	Description
<code>message_body</code>	TEXT	YES	The actual SMS message content. Can be NULL if <code>delete_message_body_after_delivery</code> is enabled for privacy. Max length varies by database (typically 65,535 characters for TEXT type).

Privacy Modes:

- **Full retention:** Message body stored in CDR for compliance/archival
- **Privacy mode:** Message body set to NULL when `delete_message_body_after_delivery: true`
- **Compliance mode:** Body stored encrypted or hashed (requires custom implementation)

Audit Timestamps

Field	Type	Nullable	Description
<code>inserted_at</code>	DATETIME	NO	When the CDR record was first inserted into the database. Typically same as or shortly after <code>delivery_time/expiry_time</code> .
<code>updated_at</code>	DATETIME	NO	When the CDR record was last updated. Same as <code>inserted_at</code> if never updated.

SQL Examples

Basic Queries

Find all CDRs for a specific phone number:

```
SELECT * FROM cdrs
WHERE calling_number = '+15551234567'
      OR called_number = '+15551234567'
ORDER BY submission_time DESC
LIMIT 100;
```

Count messages by status:

```
SELECT status, COUNT(*) as count
FROM cdrs
GROUP BY status;
```

Average delivery time for delivered messages:

```
SELECT AVG(TIMESTAMPDIFF(SECOND, submission_time, delivery_time))
AS avg_delivery_seconds
FROM cdrs
WHERE status = 'delivered'
AND delivery_time IS NOT NULL;
```

Billing Queries

Daily message volume by destination SMSC:

```
SELECT
    DATE(submission_time) AS date,
    dest_smsc,
    COUNT(*) AS message_count,
    SUM(CASE WHEN status = 'delivered' THEN 1 ELSE 0 END) AS
delivered_count,
    SUM(message_parts) AS total_segments
FROM cdrs
WHERE submission_time >= DATE_SUB(NOW(), INTERVAL 30 DAY)
GROUP BY DATE(submission_time), dest_smsc
ORDER BY date DESC, message_count DESC;
```

Billable messages for a customer (by calling number prefix):

```
SELECT
    DATE(submission_time) AS date,
    COUNT(*) AS message_count,
    SUM(message_parts) AS total_segments,
    SUM(message_parts) * 0.01 AS total_cost
FROM cdrs
WHERE calling_number LIKE '+1555%'
AND status = 'delivered'
AND submission_time >= '2025-10-01'
AND submission_time < '2025-11-01'
GROUP BY DATE(submission_time);
```

Route performance analysis:

```

SELECT
    dest_smsc,
    COUNT(*) AS total_messages,
    SUM(CASE WHEN status = 'delivered' THEN 1 ELSE 0 END) AS
delivered,
    ROUND(100.0 * SUM(CASE WHEN status = 'delivered' THEN 1 ELSE 0
END) / COUNT(*), 2) AS delivery_rate_pct,
    AVG(delivery_attempts) AS avg_attempts,
    AVG(TIMESTAMPDIFF(SECOND, submission_time, delivery_time)) AS
avg_delivery_seconds
FROM cdrs
WHERE submission_time >= DATE_SUB(NOW(), INTERVAL 7 DAY)
    AND dest_smsc IS NOT NULL
GROUP BY dest_smsc
ORDER BY delivery_rate_pct DESC;

```

Analytics Queries

Messages by hour of day (traffic pattern):

```

SELECT
    HOUR(submission_time) AS hour,
    COUNT(*) AS message_count
FROM cdrs
WHERE submission_time >= DATE_SUB(NOW(), INTERVAL 7 DAY)
GROUP BY HOUR(submission_time)
ORDER BY hour;

```

Multi-part message analysis:

```
SELECT
    message_parts,
    COUNT(*) AS message_count,
    AVG(TIMESTAMPDIFF(SECOND, submission_time, delivery_time)) AS
avg_delivery_seconds
FROM cdrs
WHERE message_parts IS NOT NULL
    AND status = 'delivered'
GROUP BY message_parts
ORDER BY message_parts;
```

Failed message analysis:

```
SELECT
    called_number,
    COUNT(*) AS failure_count,
    AVG(delivery_attempts) AS avg_attempts,
    MAX(submission_time) AS last_failure
FROM cdrs
WHERE status IN ('failed', 'expired')
    AND submission_time >= DATE_SUB(NOW(), INTERVAL 7 DAY)
GROUP BY called_number
HAVING failure_count >= 5
ORDER BY failure_count DESC;
```

Compliance and Audit Queries

Find all messages between two parties in a time range:


```

SELECT
    submission_time,
    calling_number,
    called_number,
    status,
    message_body,
    delivery_time
FROM cdrs
WHERE (
    (calling_number = '+15551234567' AND called_number =
'+15559876543')
    OR
    (calling_number = '+15559876543' AND called_number =
'+15551234567')
)
AND submission_time >= '2025-10-01'
AND submission_time < '2025-11-01'
ORDER BY submission_time;

```

Retention policy enforcement (delete old CDRs):

```

-- Find records older than retention period (example: 2 years)
SELECT COUNT(*) FROM cdrs
WHERE submission_time < DATE_SUB(NOW(), INTERVAL 2 YEAR);

-- Delete old records (use with caution!)
DELETE FROM cdrs
WHERE submission_time < DATE_SUB(NOW(), INTERVAL 2 YEAR)
LIMIT 10000; -- Batch delete to avoid locking

```

Cluster Analysis

Message distribution across nodes:

```
SELECT
    origin_node,
    COUNT(*) AS message_count,
    SUM(CASE WHEN status = 'delivered' THEN 1 ELSE 0 END) AS
delivered_count
FROM cdrs
WHERE submission_time >= DATE_SUB(NOW(), INTERVAL 1 DAY)
GROUP BY origin_node;
```

Indexes

The following indexes are created to optimize common queries:

Index Name	Columns	Purpose
PRIMARY	id	Primary key, ensures unique record
cdrs_origin_node_message_id_index	origin_node, message_id	Unique. Globally-unique CDR identity across the cluster; makes writes idempotent
idx_cdrs_message_id	message_id	Lookup CDR by original message ID
idx_cdrs_calling_number	calling_number	Find messages from a specific sender
idx_cdrs_called_number	called_number	Find messages to a specific recipient
idx_cdrs_status	status	Filter by delivery status
idx_cdrs_submission_time	submission_time	Time-based queries, billing periods

Index Name	Columns	Purpose
idx_cdrs_dest_smsc	dest_smsc	Route performance analysis

Additional Index Recommendations

For high-volume deployments, consider these additional indexes:

Composite index for billing queries:

```
CREATE INDEX idx_cdrs_billing ON cdrs(calling_number, submission_time, status);
```

Composite index for route analysis:

```
CREATE INDEX idx_cdrs_route_perf ON cdrs(dest_smsc, submission_time, status);
```

Composite index for compliance searches:

```
CREATE INDEX idx_cdrs_party_time ON cdrs(calling_number, called_number, submission_time);
```

Full-text index for message body searches (MySQL):

```
ALTER TABLE cdrs ADD FULLTEXT INDEX idx_cdrs_message_body_ft (message_body);

-- Usage:
SELECT * FROM cdrs
WHERE MATCH(message_body) AGAINST('keyword' IN NATURAL LANGUAGE MODE);
```

Data Types by Database

Field type mappings across supported databases:

Field	MySQL/MariaDB	PostgreSQL	Notes
id	BIGINT AUTO_INCREMENT	BIGSERIAL	64-bit integer, auto-incrementing
message_id	BIGINT	BIGINT	64-bit integer
String fields	VARCHAR(255)	VARCHAR(255)	Variable-length string, max 255 chars
message_body	TEXT	TEXT	Large text, up to 65,535 bytes (MySQL), unlimited (PostgreSQL)
Timestamps	DATETIME	TIMESTAMP	UTC timestamps recommended
Integers	INT	INTEGER	32-bit signed integer
Booleans	BOOLEAN (TINYINT(1))	BOOLEAN	MySQL stores as 0/1

Privacy Considerations

The CDR table may contain sensitive personal information (phone numbers, message content). Consider these privacy measures:

1. Message Body Privacy

Configuration options in `config/runtime.exs`:

```
config :sms_c,  
  # Delete message body after successful delivery  
  delete_message_body_after_delivery: true,  
  
  # Hide message body in UI  
  hide_message_body_in_ui: true,  
  
  # Hide message body in exports  
  hide_message_body_in_export: true
```

2. Phone Number Masking

For analytics that don't require full numbers:

```
-- Mask last 4 digits of phone numbers  
SELECT  
  CONCAT(SUBSTRING(calling_number, 1, LENGTH(calling_number) - 4),  
  'XXXX') AS masked_calling,  
  CONCAT(SUBSTRING(called_number, 1, LENGTH(called_number) - 4),  
  'XXXX') AS masked_called,  
  COUNT(*) AS message_count  
FROM cdrs  
GROUP BY masked_calling, masked_called;
```

3. Database Encryption

Enable encryption at rest for the database server:

MySQL:

```
-- Enable table encryption  
ALTER TABLE cdrs ENCRYPTION='Y';
```

PostgreSQL: Use PostgreSQL transparent data encryption (TDE) or filesystem-level encryption.

4. Access Controls

Restrict CDR table access:

```
-- Create read-only billing user
CREATE USER 'billing_ro'@'%' IDENTIFIED BY 'secure_password';
GRANT SELECT ON sms_c.cdcs TO 'billing_ro'@'%';

-- Create limited analytics user (no message body access)
CREATE USER 'analytics'@'%' IDENTIFIED BY 'secure_password';
GRANT SELECT (id, message_id, calling_number, called_number,
source_smsc,
                dest_smsc, submission_time, delivery_time, status,
                delivery_attempts, message_parts)
ON sms_c.cdcs TO 'analytics'@'%';
```

Retention and Archival

Retention Policies

Define retention periods based on regulatory and business requirements:

Industry	Typical Retention	Regulatory Basis
Telecom (US)	18-24 months	FCC, state laws
Telecom (EU)	6 months - 2 years	GDPR, ePrivacy
Financial	5-7 years	SOX, SEC
Healthcare	6 years	HIPAA

Archival Strategy

1. Partition by Date (MySQL 8.0+, PostgreSQL 11+)

```
-- MySQL partitioning by month
ALTER TABLE cdrs PARTITION BY RANGE (TO_DAYS(submission_time)) (
  PARTITION p202510 VALUES LESS THAN (TO_DAYS('2025-11-01')),
  PARTITION p202511 VALUES LESS THAN (TO_DAYS('2025-12-01')),
  PARTITION p202512 VALUES LESS THAN (TO_DAYS('2026-01-01')),
  PARTITION p_future VALUES LESS THAN MAXVALUE
);

-- Drop old partition (fast archival)
ALTER TABLE cdrs DROP PARTITION p202510;
```

2. Archive to Cold Storage

```
-- Export old CDRs to archive table
CREATE TABLE cdrs_archive LIKE cdrs;

INSERT INTO cdrs_archive
SELECT * FROM cdrs
WHERE submission_time < DATE_SUB(NOW(), INTERVAL 2 YEAR);

-- Verify and delete from main table
DELETE FROM cdrs
WHERE submission_time < DATE_SUB(NOW(), INTERVAL 2 YEAR);
```

3. Automated Cleanup Script


```
#!/bin/bash
# cleanup_old_cdrs.sh - Run via cron

MYSQL_USER="cleanup_user"
MYSQL_PASS="secure_password"
MYSQL_DB="sms_c"
RETENTION_DAYS=730 # 2 years

# Archive old records
mysql -u"$MYSQL_USER" -p"$MYSQL_PASS" "$MYSQL_DB" <<EOF
INSERT INTO cdrs_archive
SELECT * FROM cdrs
WHERE submission_time < DATE_SUB(NOW(), INTERVAL $RETENTION_DAYS
DAY)
LIMIT 100000;

DELETE FROM cdrs
WHERE submission_time < DATE_SUB(NOW(), INTERVAL $RETENTION_DAYS
DAY)
LIMIT 100000;
EOF
```

Cron entry:

```
# Run daily at 2 AM
0 2 * * * /usr/local/bin/cleanup_old_cdrs.sh >>
/var/log/sms_c/cleanup.log 2>&1
```

Billing Integration

Rate Card Schema

Create a separate rates table for billing:

```
CREATE TABLE billing_rates (  
  id INT AUTO_INCREMENT PRIMARY KEY,  
  destination_prefix VARCHAR(20) NOT NULL,  
  description VARCHAR(255),  
  rate_per_message DECIMAL(10, 6) NOT NULL,  
  rate_per_segment DECIMAL(10, 6) NOT NULL,  
  currency VARCHAR(3) DEFAULT 'USD',  
  effective_date DATE NOT NULL,  
  expiry_date DATE,  
  INDEX idx_prefix (destination_prefix),  
  INDEX idx_dates (effective_date, expiry_date)  
);
```

-- Example rates

```
INSERT INTO billing_rates (destination_prefix, description,  
  rate_per_message, rate_per_segment, effective_date) VALUES  
  ('+1', 'United States/Canada', 0.0050, 0.0050, '2025-01-01'),  
  ('+44', 'United Kingdom', 0.0080, 0.0080, '2025-01-01'),  
  ('+61', 'Australia', 0.0100, 0.0100, '2025-01-01'),  
  ('+', 'International default', 0.0150, 0.0150, '2025-01-01');
```

Billing Query

Join CDRs with rates for invoicing:

```

SELECT
  DATE(c.submission_time) AS date,
  c.dest_smsc AS route,
  LEFT(c.called_number,
    CASE
      WHEN c.called_number LIKE '+1%' THEN 2
      WHEN c.called_number LIKE '+%' THEN
LENGTH(SUBSTRING_INDEX(c.called_number, '', 4))
      ELSE 0
    END
  ) AS destination_prefix,
  COUNT(*) AS message_count,
  SUM(c.message_parts) AS segment_count,
  COALESCE(r.rate_per_segment, 0.015) AS rate,
  SUM(c.message_parts) * COALESCE(r.rate_per_segment, 0.015) AS
total_cost
FROM cdrs c
LEFT JOIN billing_rates r ON c.called_number LIKE
CONCAT(r.destination_prefix, '%')
  AND c.submission_time >= r.effective_date
  AND (r.expiry_date IS NULL OR c.submission_time < r.expiry_date)
WHERE c.status = 'delivered'
  AND c.submission_time >= '2025-10-01'
  AND c.submission_time < '2025-11-01'
GROUP BY date, route, destination_prefix
ORDER BY date DESC, total_cost DESC;

```

Export for Billing Systems

Built-in CSV export (recommended):

The SMS-C can render CDRs to an RFC 4180 CSV file directly, without going through the SQL client. In a release, drive it from the remote console:

```

bin/omnimessage rpc
'SmsC.Messaging.CDR.export_csv_to_file("/tmp/cdrs-202607.csv",
since: ~U[2026-07-01 00:00:00Z], until: ~U[2026-08-01 00:00:00Z],
status: :delivered)'

```

Options: `:since`, `:until` (both `DateTime`), `:status` (an atom/string or a list, e.g. `[:delivered, :failed]`), and `:limit`. The returned value is `{:ok, {path, row_count}}`. The `message_body` column honours the `hide_message_body_in_export` setting — set it to `true` to emit `[REDACTED]` instead of message content. `SmsC.Messaging.CDR.export_csv/1` returns the same CSV as a string if you'd rather stream it elsewhere.

Direct SQL client export (alternative):

```
mysql -u billing_ro -p -D sms_c -e "
SELECT
  id,
  message_id,
  calling_number,
  called_number,
  dest_smsc,
  submission_time,
  delivery_time,
  status,
  message_parts
FROM cdrs
WHERE submission_time >= '2025-10-01'
      AND submission_time < '2025-11-01'
      AND status = 'delivered'
" --batch --silent | sed 's/\t/,/g' > billing_export_202510.csv
```

See Also

- [Configuration Guide](#) - Configure CDR export settings
- [Operations Guide](#) - CDR maintenance procedures
- [API Reference](#) - Query CDRs via REST API

SMS-C Configuration Reference

[← Back to Documentation Index](#)

Complete reference for all SMS-C configuration options with examples for common deployment scenarios.

Table of Contents

- [Configuration Files](#)
- [Database Configuration](#)
- [API Configuration](#)
- [Web UI Configuration](#)
- [Federation Configuration](#)
- [Message Queue Configuration](#)
- [Charging Configuration](#)
- [Diameter Sh \(HSS\) Configuration](#)
- [ENUM Configuration](#)
- [Number Translation Configuration](#)
- [Routing Configuration](#)
- [Performance Tuning Configuration](#)
- [Logging Configuration](#)
- [Common Configuration Scenarios](#)

Configuration Files

The SMS-C uses three main configuration files:

config/config.exs

Static configuration loaded at compile time. Contains:

- Application-wide defaults
- Logger configuration
- Development/test settings
- Performance tuning parameters

config/runtime.exs

Runtime configuration loaded at startup. Contains:

- Database connection settings
- Cluster configuration
- External service integration (OCS, ENUM)
- Initial routes and translation rules
- Environment-specific settings

config/prod.exs (optional)

Production-specific overrides.

Best Practice: Use environment variables in `runtime.exs` for sensitive values like passwords and API keys.

SQL CDR Storage Configuration

The SMS-C uses **Mnesia** for operational data (message queue, routing rules, number translations) and supports external **SQL databases** for long-term CDR (Call Detail Record) storage, billing, and analytics.

Supported SQL Databases

The system supports the following SQL databases for CDR export:

Database	Version	Adapter	Default Port	Best For
MySQL	8.0+	<code>Ecto.Adapters.MyXQL</code>	3306	General purpose, proven reliability
MariaDB	10.5+	<code>Ecto.Adapters.MyXQL</code>	3306	MySQL-compatible open source
PostgreSQL	13+	<code>Ecto.Adapters.Postgres</code>	5432	Advanced features, JSON support

Note: Mnesia is used automatically for operational data (message queue, routing, translations) and requires no configuration. The SQL database is **only** used for CDR export and long-term storage.

When CDRs Are Written

CDRs are written to the SQL database at two points:

1. **Near-real-time, on terminal state** — the moment a message reaches a final outcome (`delivered`, `expired`, `failed`, `rejected`, or `balance_rejected`) a CDR is written asynchronously. This is the primary path and keeps billing current without waiting for message retention.
2. **Retention-time backstop** — when the retention worker removes an expired message from Mnesia it exports a CDR first (recording `expired` for anything that never reached a terminal state). A message is only deleted once its CDR export succeeds, so nothing is lost if the CDR database is briefly unavailable — it is retried on the next sweep.

Both paths are idempotent: each CDR is keyed on the unique `(origin_node, message_id)` pair and writes use `ON CONFLICT DO NOTHING`, so a message covered by the near-real-time write is never duplicated by the backstop.

`message_id` is generated by a per-node counter (persisted in Mnesia, so it survives restarts) and is therefore only unique **within a single node**. When several SMS-C nodes write to a shared CDR database, `(origin_node, message_id)` together is the globally-unique key — see the [CDR Schema Reference](#). Set `cdr_enabled: false` to disable CDR writing entirely.

CSV / Text-File Export

CDRs can be exported from the SQL database to an RFC 4180 CSV file. In a release (where `mix` is unavailable) drive it from the remote console:

```
bin/omnimessage rpc
'SmsC.Messaging.CDR.export_csv_to_file("/tmp/cdrs-202607.csv",
since: ~U[2026-07-01 00:00:00Z], until: ~U[2026-08-01 00:00:00Z])'
```

Supported options: `:since`, `:until` (both `DateTime`), `:status` (an atom/string or list), and `:limit`. The `message_body` column honours the `hide_message_body_in_export` setting (emitted as `[REDACTED]` when enabled).

MySQL / MariaDB Configuration

```
# config/runtime.exs
config :sms_c, SmsC.Repo,
  adapter: Ecto.Adapters.MyXQL,
  username: System.get_env("DB_USERNAME") || "sms_user",
  password: System.get_env("DB_PASSWORD") || "secure_password",
  hostname: System.get_env("DB_HOSTNAME") || "localhost",
  port: String.to_integer(System.get_env("DB_PORT") || "3306"),
  database: System.get_env("DB_NAME") || "sms_c_prod",
  pool_size: String.to_integer(System.get_env("DB_POOL_SIZE") ||
"20")
```


PostgreSQL Configuration

```
# config/runtime.exs
config :sms_c, SmsC.Repo,
  adapter: Ecto.Adapters.Postgres,
  username: System.get_env("DB_USERNAME") || "sms_user",
  password: System.get_env("DB_PASSWORD") || "secure_password",
  hostname: System.get_env("DB_HOSTNAME") || "localhost",
  port: String.to_integer(System.get_env("DB_PORT") || "5432"),
  database: System.get_env("DB_NAME") || "sms_c_prod",
  pool_size: String.to_integer(System.get_env("DB_POOL_SIZE") ||
"20")
```

Choosing a SQL Database

MySQL/MariaDB - Recommended for most deployments:

- Excellent performance for CDR writes
- Proven reliability in telecom environments
- Wide tooling support for billing systems
- Easy replication setup

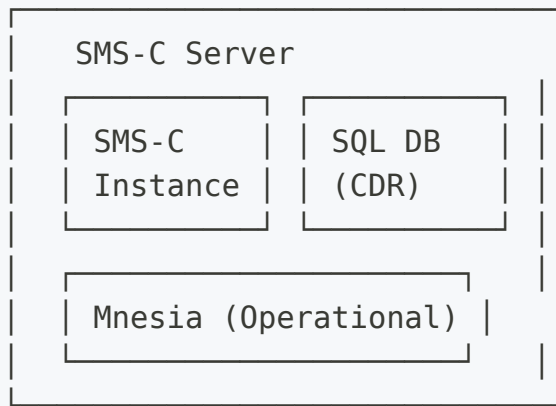
PostgreSQL - Consider if you need:

- Advanced JSON/JSONB features for analytics
- Complex queries on CDR data
- Existing PostgreSQL infrastructure
- PostGIS for geographic analysis

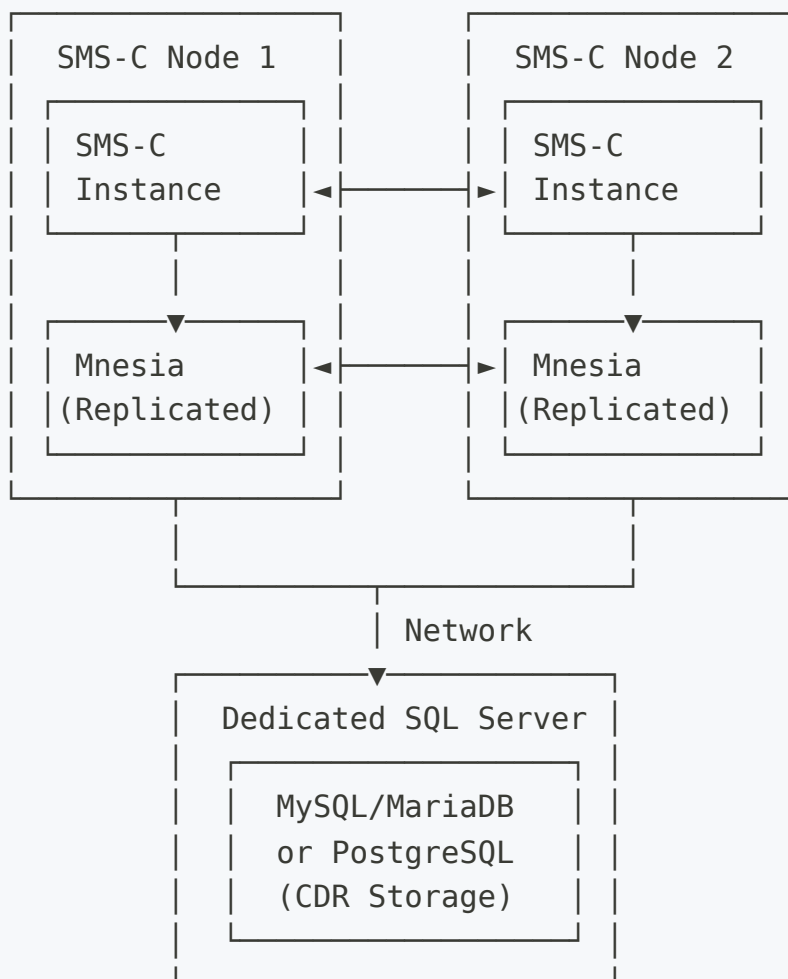
Deployment Topologies

Important: The SQL CDR database can run on a **separate server** from your SMS-C instance(s). This is the recommended approach for production deployments.

Single-Server Deployment (Development/Testing):



Distributed Deployment (Production - Recommended):



Benefits of Separate SQL Server:

- **Performance Isolation:** CDR writes don't impact message processing

- **Scalability:** Independently scale database and message processing
- **Reliability:** Database maintenance doesn't affect SMS-C uptime
- **Data Management:** Centralized CDR storage for multiple SMS-C instances
- **Backup Flexibility:** Independent backup schedules and retention policies

Pool Size Guidelines

Workload	Pool Size	Description
Development	5-10	Minimal concurrency
Low Volume (< 100 msg/sec)	10-15	Small deployments
Medium Volume (100-1000 msg/sec)	20-30	Typical production
High Volume (> 1000 msg/sec)	40-100	High-throughput scenarios

Calculation: `pool_size = (expected concurrent DB operations) * 1.5`

Database Connection Examples

Using Environment Variables (Recommended for Production):

```
# Set environment variables
export DB_USERNAME=sms_prod_user
export DB_PASSWORD=strong_password_here
export DB_HOSTNAME=db-primary.internal.example.com
export DB_PORT=3306
export DB_NAME=sms_c_production
export DB_POOL_SIZE=30
```

Direct Configuration (Development Only):

```
config :sms_c, SmsC.Repo,  
  username: "dev_user",  
  password: "dev_password",  
  hostname: "localhost",  
  database: "sms_c_dev",  
  pool_size: 5
```

Connection Pool Monitoring

Monitor pool usage via Prometheus metrics:

- `ecto_pools_queue_time` - Time waiting for connection
- `ecto_pools_query_time` - Query execution time
- `ecto_pools_connected_count` - Active connections

Alert if wait time exceeds 100ms consistently - indicates need for larger pool.

API Configuration

The REST API provides message submission and management capabilities.

Basic API Configuration

```
# config/runtime.exs  
config :api_ex,  
  port: String.to_integer(System.get_env("API_PORT") || "8443"),  
  listen_ip: System.get_env("API_LISTEN_IP") || "0.0.0.0",  
  enable_tls: System.get_env("API_ENABLE_TLS") != "false"
```

TLS/SSL Configuration

Production Setup with TLS (Recommended):

```
config :api_ex,  
  port: 8443,  
  listen_ip: "0.0.0.0",  
  enable_tls: true,  
  tls_cert_path: "/etc/sms_c/certs/server.crt",  
  tls_key_path: "/etc/sms_c/certs/server.key"
```

Development Setup without TLS:

```
config :api_ex,  
  port: 8080,  
  listen_ip: "127.0.0.1",  
  enable_tls: false
```

API Certificate Setup

Generate self-signed certificate for testing:

```
# Create certificate directory  
mkdir -p priv/cert  
  
# Generate private key  
openssl genrsa -out priv/cert/server.key 2048  
  
# Generate certificate signing request  
openssl req -new -key priv/cert/server.key -out  
priv/cert/server.csr \  
  -subj "/C=US/ST=State/L=City/O=Organization/CN=sms-  
api.example.com"  
  
# Generate self-signed certificate (valid 365 days)  
openssl x509 -req -days 365 -in priv/cert/server.csr \  
  -signkey priv/cert/server.key -out priv/cert/server.crt  
  
# Set permissions  
chmod 600 priv/cert/server.key  
chmod 644 priv/cert/server.crt
```

For production, use certificates from a trusted CA (Let's Encrypt, commercial CA, etc.).

API Access Control

IP Whitelisting (Application Firewall):

```
# Using iptables (Linux)
iptables -A INPUT -p tcp --dport 8443 -s 10.0.0.0/8 -j ACCEPT
iptables -A INPUT -p tcp --dport 8443 -j DROP

# Using firewalld (Red Hat/CentOS)
firewall-cmd --permanent --add-rich-rule='rule family="ipv4"
source address="10.0.0.0/8" port protocol="tcp" port="8443"
accept'
firewall-cmd --reload
```

API Key Authentication (Application Level):

Configure via custom plug in router - see Operations Guide for implementation details.

Web UI Configuration

The web interface provides route management, message browsing, and monitoring.

Basic Web UI Configuration

```
# config/runtime.exs
config :control_panel,
  port: String.to_integer(System.get_env("WEB_PORT") || "80"),
  hostname: System.get_env("WEB_HOSTNAME") || "localhost",
  enable_tls: System.get_env("WEB_ENABLE_TLS") == "true"
```

Production Web UI Setup

```
config :control_panel,  
  port: 443,  
  hostname: "sms-admin.example.com",  
  enable_tls: true,  
  tls_cert_path: "/etc/sms_c/certs/web.crt",  
  tls_key_path: "/etc/sms_c/certs/web.key"
```

Reverse Proxy Setup (Recommended)

Use Nginx or Apache as reverse proxy for additional security and features:

Nginx Configuration Example:

```

upstream sms_web {
    server 127.0.0.1:4000;
    keepalive 32;
}

server {
    listen 80;
    server_name sms-admin.example.com;
    return 301 https://$server_name$request_uri;
}

server {
    listen 443 ssl http2;
    server_name sms-admin.example.com;

    ssl_certificate /etc/letsencrypt/live/sms-
admin.example.com/fullchain.pem;
    ssl_certificate_key /etc/letsencrypt/live/sms-
admin.example.com/privkey.pem;
    ssl_protocols TLSv1.2 TLSv1.3;
    ssl_ciphers HIGH:!aNULL:!MD5;

    # Basic auth for additional security
    auth_basic "SMS-C Admin";
    auth_basic_user_file /etc/nginx/.htpasswd;

    location / {
        proxy_pass http://sms_web;
        proxy_http_version 1.1;
        proxy_set_header Upgrade $http_upgrade;
        proxy_set_header Connection "upgrade";
        proxy_set_header Host $host;
        proxy_set_header X-Real-IP $remote_addr;
        proxy_set_header X-Forwarded-For $proxy_add_x_forwarded_for;
        proxy_set_header X-Forwarded-Proto $scheme;
    }

    # WebSocket support for LiveView
    location /live {
        proxy_pass http://sms_web;
        proxy_http_version 1.1;
        proxy_set_header Upgrade $http_upgrade;
        proxy_set_header Connection "upgrade";
    }
}

```



```
    proxy_read_timeout 86400;
  }
}
```

Federation Configuration

OmniMessage uses HTTP-based federation for multi-controller deployments. Controllers discover each other via DNS SRV records or static peer lists, exchange health and frontend registries over HTTPS, and forward messages to remote controllers when needed.

See the [Geographic Federation Guide](#) for full architecture, deployment examples, and troubleshooting.

Quick Start

```
# config/runtime.exs
config :sms_c, :federation,
  enabled: true,
  dns_srv_domain: "_smc._tcp.smc.example.com"
```

Full Parameter Reference

Parameter	Type	Default	Description
<code>enabled</code>	Boolean	<code>false</code>	Master switch for federation.
<code>dns_srv_domain</code>	String	<code>""</code>	DNS SRV domain for peer discovery.
<code>dns_poll_interval_ms</code>	Integer	<code>30000</code>	DNS re-resolution interval.
<code>health_check_interval_ms</code>	Integer	<code>10000</code>	Peer health check interval.
<code>registry_sync_interval_ms</code>	Integer	<code>15000</code>	Frontend registry exchange interval.
<code>forward_retry_interval_ms</code>	Integer	<code>5000</code>	Queued message retry interval.
<code>forward_max_retries</code>	Integer	<code>50</code>	Max forward attempts before marking failed.
<code>http_timeout_ms</code>	Integer	<code>5000</code>	Timeout for all peer HTTP calls.
<code>api_port</code>	Integer	<code>8443</code>	API port for constructing peer URLs.
<code>static_peers</code>	List	<code>[]</code>	Static peer list (alternative to DNS SRV).

Network Requirements

Federation requires only port **8443** (HTTPS) between sites, compared to ports 4369 + 9100-9200 for Erlang clustering.

```
# Allow federation traffic from peer sites
iptables -A INPUT -p tcp -s 10.0.1.0/24 --dport 8443 -j ACCEPT
iptables -A INPUT -p tcp -s 10.0.2.0/24 --dport 8443 -j ACCEPT
```

Message Queue Configuration

Controls message retention and expiration behavior.

Message Expiration

```
# config/runtime.exs
config :sms_c,
  dead_letter_time_minutes: 1440 # 24 hours
```

Common Values:

- **60** - 1 hour (testing/development)
- **1440** - 24 hours (typical production)
- **4320** - 3 days (extended retention)
- **10080** - 7 days (maximum retention)

Messages older than this value become undeliverable and are marked for cleanup.

Delivery Retry Configuration

Retry behavior uses exponential backoff:

```
Retry Delay = 2^(attempt_count) minutes
```

Attempt	Delay
1	2 minutes
2	4 minutes
3	8 minutes
4	16 minutes
5	32 minutes
6	64 minutes
7	128 minutes
8	256 minutes

Maximum attempts before dead letter: Limited by `dead_letter_time_minutes`.

Cleanup Configuration

```
# config/config.exs
config :sms_c,
  cleanup_interval_minutes: 10,
  fingerprint_ttl_minutes: 5,
  event_ttl_days: 7
```

Cleanup Intervals:

- **cleanup_interval_minutes**: How often cleanup worker runs (default: 10)
- **fingerprint_ttl_minutes**: Duplicate detection window (default: 5)
- **event_ttl_days**: Event log retention (default: 7)

Charging Configuration

Online charging is performed over the **Diameter Ro** interface (Credit-Control Application, RFC 4006). For each charged message the SMS-C sends a Credit-Control-Request (Event-Request, `CC-Request-Type 4`) to the OCS for the message **sender**; the OCS authorizes or declines based on the subscriber's balance. This reuses the Diameter stack (see [Diameter Sh](#) below) — the `:ro` application must be registered.

Enable Charging

```
# config/runtime.exs
config :sms_c,
  default_charging_enabled: true

config :sms_c, :charging,
  # What to do when the sender is out of balance:
  #   :reject      - mark the message :balance_rejected and do not
  deliver it
  #   :auto_reply - additionally SMS out_of_balance_reply_text
  back to the sender
  out_of_balance_action: :reject,
  out_of_balance_reply_text: "Your message could not be sent due
  to insufficient balance.",
  # Fail mode when the OCS errors or the Diameter stack is not
  configured
  # (NOT a definitive out-of-balance answer):
  #   :allow - fail open, deliver the message (default)
  #   :reject - fail closed, treat as out of balance
  on_charging_error: :allow,
  # Rating profile selecting how the OCS rates the message. Sent
  as the CCR
  # Service-Context-Id.
  rating_profile: "sms@sms-c"
```

Charging also requires the Diameter stack enabled with the `:ro` application (Credit-Control is IETF base, vendor 0):

```

config :sms_c,
  diameter_enabled: true

config :diameter_ex,
  diameter: %{
    service_name: :omnimessage,
    host: "smsc01",
    realm: "epc.mnc005.mcc547.3gppnetwork.org",
    # ... see Diameter Sh section for the full stack/peers ...
    applications: [
      %{
        application_name: :ro,
        application_dictionary: :diameter_gen_3gpp_ro,
        vendor_specific_application_ids: [
          %{vendor_id: 0, auth_application_id: 4,
acct_application_id: nil}
        ]
      }
    ]
  }
}

```

Per-Route Charging

Each route's `charged` policy controls charging independently of the global default:

<code>charged</code>	Behaviour
<code>:yes</code>	Always balance-check via the OCS
<code>:no</code>	Never charge
<code>:default</code>	Use <code>default_charging_enabled</code>

Disable Charging

```
config :sms_c,  
  default_charging_enabled: false
```

When disabled (the default), `:default` routes route without any OCS balance check. Individual routes can still opt in with `charged: :yes`.

Defaults & Graceful Degradation

The `config :sms_c, :charging` block is optional — every key has a sensible default (`out_of_balance_action: :reject`, a built-in reply text, and `rating_profile: "sms@sms-c"`), and `default_charging_enabled` defaults to `false`.

By default charging **fails open** (`on_charging_error: :allow`), matching the previous HTTP charging path which logged failures but never blocked delivery:

- If a route requests charging but the Diameter stack is **not** enabled/configured, the message is **allowed** (logged as a warning).
- If the OCS is unreachable or returns a protocol error, the message is **allowed**.

Set `on_charging_error: :reject` to **fail closed** instead — OCS errors and an unconfigured stack are then treated as out of balance and the `out_of_balance_action` applies.

Either way, a **definitive out-of-balance answer** from the OCS always triggers the configured `out_of_balance_action`. Rejected messages are marked with status `:balance_rejected`.

Diameter Sh (HSS) Configuration

OmniMessage can query the HSS via the Diameter Sh interface to detect on-net subscribers who are temporarily offline, preventing messages from being routed out the default gateway. See the [HSS Subscriber Lookup Guide](#) for full operational documentation.

```

# Enable HSS dip in the routing path
config :sms_c,
  diameter_enabled: true

# Diameter stack configuration
config :diameter_ex,
  diameter: %{
    service_name: :omnimessage,
    listen_ip: "0.0.0.0",
    listen_port: 3868,
    decode_format: :map,
    host: "smsc01",
    realm: "epc.mnc005.mcc547.3gppnetwork.org",
    product_name: "OmniMessage",
    request_timeout: 5000,
    control_module: SmsC.Diameter.Control,
    processor_module: DiameterEx.Processor,
    vendor_id: 10415,
    supported_vendor_ids: [10415],
    applications: [
      %{
        application_name: :sh,
        application_dictionary: :diameter_gen_3gpp_sh,
        vendor_specific_application_ids: [
          %{vendor_id: 10415, auth_application_id: 16_777_217,
acct_application_id: nil}
        ]
      }
    ],
    peers: [
      %{
        host: "dra01.epc.mnc005.mcc547.3gppnetwork.org",
        ip: "10.0.0.1",
        port: 3868,
        realm: "epc.mnc005.mcc547.3gppnetwork.org",
        tls: false,
        transport: :diameter_tcp,
        initiate_connection: true
      }
    ]
  }
}

```


Parameter	Type	Required	Default	Description
<code>diameter_enabled</code>	Boolean	No	<code>false</code>	Enable HSS subscriber dip during route matching (for routes with <code>:on_net/</code> or <code>:off</code> criteria). When <code>false</code> , those criteria fail-clc
<code>hss_cache_ttl_seconds</code>	Integer	No	<code>7200</code>	TTL (seconds) cached HSS lookup results
<code>mock_sh</code>	Boolean	No	<code>false</code>	Use mock HSS responses for testing (no Diameter stack started)
<code>mock_sh_on_net_numbers</code>	List	No	<code>[]</code>	MSISDNs treat as on-net in m mode

For the full Diameter stack parameter reference (host, realm, peers, applications), see the [HSS Subscriber Lookup Guide](#).

ENUM Configuration

DNS-based E.164 number lookups for intelligent routing.

Disable ENUM (Default)

```
# config/runtime.exs
config :sms_c,
  enum_enabled: false
```

Enable ENUM with Default DNS

```
config :sms_c,
  enum_enabled: true,
  enum_domains: ["e164.arpa", "e164.org"],
  enum_dns_servers: [], # Use system default DNS
  enum_timeout: 5000 # 5 seconds
```

Enable ENUM with Custom DNS Servers

```
config :sms_c,
  enum_enabled: true,
  enum_domains: ["e164.internal.example.com", "e164.arpa"],
  enum_dns_servers: [
    {"10.0.1.53", 53}, # Internal DNS server
    {"8.8.8.8", 53},   # Google Public DNS (fallback)
    {"1.1.1.1", 53}    # Cloudflare DNS (fallback)
  ],
  enum_timeout: 3000 # 3 seconds (faster failover)
```

ENUM Domain Priority

Domains are queried in order until a successful lookup:

```
config :sms_c,
  enum_domains: [
    "e164.internal.example.com", # Try internal first
    "e164.carrier.net",          # Then carrier
    "e164.arpa"                  # Then public registry
  ]
```

ENUM Performance Tuning

For Low-Latency Networks:

```
enum_timeout: 2000 # 2 seconds
```

For High-Latency/Satellite Links:

```
enum_timeout: 10000 # 10 seconds
```

ENUM DNS Setup Example

Configure Private ENUM Zone (BIND9 format):

```
; Zone file for e164.internal.example.com
$ORIGIN e164.internal.example.com.
$TTL 300

; Number: +1-555-0100 becomes
0.0.1.0.5.5.5.1.e164.internal.example.com
0.0.1.0.5.5.5.1.e164.internal.example.com. IN NAPTR 100 10 "u"
"E2U+sip" "!^.*$!sip:15550100@voip-gateway.example.com!" .
0.0.1.0.5.5.5.1.e164.internal.example.com. IN NAPTR 100 20 "u"
"E2U+pstn" "!^.*$!pstn:gateway-a.example.com!" .

; Number: +1-555-0200
0.0.2.0.5.5.5.1.e164.internal.example.com. IN NAPTR 100 10 "u"
"E2U+sip" "!^.*$!sip:15550200@voip-gateway.example.com!" .
```

Test ENUM Resolution:

```
# Query ENUM domain
dig @10.0.1.53 NAPTR 0.0.1.0.5.5.5.1.e164.internal.example.com

# Expected output includes NAPTR records:
# 0.0.1.0.5.5.5.1.e164.internal.example.com. 300 IN NAPTR 100 10
"u" "E2U+sip" "!^.*$!sip:15550100@voip-gateway.example.com!" .
```

Number Translation Configuration

Regex-based number normalization applied before routing.

Disable Number Translation

```
# config/runtime.exs
config :sms_c,
  translation_rules: []
```

Basic Number Translation Examples

Add Country Code to Local Numbers:

```
config :sms_c,
  translation_rules: [
    %{
      calling_prefix: nil,
      called_prefix: "",
      source_smsc: nil,
      calling_match: "^(\\d{10})$",           # Match 10-digit
      numbers
      calling_replace: "+1\\1",               # Prepend +1
      called_match: "^(\\d{10})$",
      called_replace: "+1\\1",
      priority: 100,
      description: "Add +1 to 10-digit North American numbers",
      enabled: true
    }
  ]
```

Normalize International Format:

```
%{
  calling_prefix: nil,
  called_prefix: nil,
  source_smsc: nil,
  calling_match: "^00(\d+)$",          # Match 00 prefix
  calling_replace: "+\1",              # Replace with +
  called_match: "^00(\d+)$",
  called_replace: "+\1",
  priority: 10,
  description: "Convert 00 international prefix to +",
  enabled: true
}
```

Remove Formatting Characters:

```
%{
  calling_prefix: nil,
  called_prefix: nil,
  source_smsc: nil,
  calling_match: "^\\+?1?[\s\\-\\.\\(\\)]*(\\d{3})[\s\\-\\.\\(\\)]*(\\d{3})[\s\\-\\.\\(\\)]*(\\d{4})$",
  calling_replace: "+1\\1\\2\\3",
  called_match: "^\\+?1?[\s\\-\\.\\(\\)]*(\\d{3})[\s\\-\\.\\(\\)]*(\\d{3})[\s\\-\\.\\(\\)]*(\\d{4})$",
  called_replace: "+1\\1\\2\\3",
  priority: 50,
  description: "Normalize US phone number formatting",
  enabled: true
}
```

Carrier-Specific Translation

Route Code Stripping:

```
%{
  calling_prefix: nil,
  called_prefix: "101",                # Only for 101
prefix
  source_smsc: "carrier_a",            # Only from this
carrier
  calling_match: nil,                  # Don't change
calling
  calling_replace: nil,
  called_match: "^101(\d+)$",          # Strip 101 route
code
  called_replace: "\1",
  priority: 5,
  description: "Strip carrier route code from called number",
  enabled: true
}
```

Multi-Rule Translation

Rules are evaluated in priority order (lower number = higher priority):

```

config :sms_c,
  translation_rules: [
    # Priority 1: Most specific rules first
    %{
      calling_prefix: "1555",
      called_prefix: nil,
      source_smsc: nil,
      calling_match: "^(1555\d{7})$",
      calling_replace: "+\1",
      called_match: nil,
      called_replace: nil,
      priority: 1,
      description: "Premium number normalization",
      enabled: true
    },

    # Priority 50: General rules
    %{
      calling_prefix: nil,
      called_prefix: nil,
      source_smsc: nil,
      calling_match: "^(\\d{10})$",
      calling_replace: "+1\1",
      called_match: "^(\\d{10})$",
      called_replace: "+1\1",
      priority: 50,
      description: "General 10-digit normalization",
      enabled: true
    }
  ]
]

```

Routing Configuration

Initial routing rules loaded on first startup. See [SMS Routing Guide](#) for complete routing documentation.

Load Routes from Configuration

```
# config/runtime.exs
config :sms_c,
  sms_routes: [
    # Geographic routing example
    %{
      calling_regex: nil,
      called_regex: ~r/^\+1/,
      source_smsc: nil,
      dest_smsc: "north_america_gateway",
      source_type: nil,
      enum_domain: nil,
      enum_result_domain: nil,
      action: :route,
      auto_reply_message: nil,
      charged: :default,
      originating_network: :any,
      terminating_network: :any,
      weight: 100,
      priority: 50,
      description: "North America routing",
      enabled: true
    },

    # Load balanced routing example
    %{
      calling_regex: nil,
      called_regex: ~r/^\+44/,
      source_smsc: nil,
      dest_smsc: "uk_gateway_1",
      source_type: nil,
      enum_domain: nil,
      enum_result_domain: nil,
      action: :route,
      auto_reply_message: nil,
      charged: :default,
      originating_network: :any,
      terminating_network: :any,
      weight: 70,
      priority: 50,
      description: "UK primary gateway (70%)",
      enabled: true
    }
  ]
}
```



```

    },
    %{
      calling_regex: nil,
      called_regex: ~r/^\+44/,
      source_smsc: nil,
      dest_smsc: "uk_gateway_2",
      source_type: nil,
      enum_domain: nil,
      enum_result_domain: nil,
      action: :route,
      auto_reply_message: nil,
      charged: :default,
      originating_network: :any,
      terminating_network: :any,
      weight: 30,
      priority: 50,
      description: "UK backup gateway (30%)",
      enabled: true
    },

    # On-net only – restrict an SMPP bind to on-net destinations
    only
    %{
      calling_regex: nil,
      called_regex: nil,
      source_smsc: "carrier_smpp_bind",
      dest_smsc: "local_msc",
      source_type: :smpp,
      enum_domain: nil,
      enum_result_domain: nil,
      action: :route,
      auto_reply_message: nil,
      charged: :default,
      originating_network: :any,
      terminating_network: :on_net,
      weight: 100,
      priority: 50,
      description: "Carrier X – on-net termination only",
      enabled: true
    }
  ]

```

Skip Initial Route Loading

```
# Don't load routes from config (manage via Web UI only)
config :sms_c,
  sms_routes: []
```

Routes defined in configuration are ONLY loaded if the routing table is empty (first startup).

Performance Tuning Configuration

See [Performance Tuning Guide](#) for detailed optimization strategies.

Batch Insert Worker

```
# config/config.exs
config :sms_c,
  batch_insert_batch_size: 100,           # Messages per batch
  batch_insert_flush_interval_ms: 100    # Max wait time in ms
```

Performance Profiles:

Profile	Batch Size	Interval	Throughput	Latency
High Volume	200	200ms	~5,000 msg/sec	Up to 200ms
Balanced	100	100ms	~4,500 msg/sec	Up to 100ms
Low Latency	50	20ms	~3,000 msg/sec	Up to 20ms
Real-time	10	10ms	~1,500 msg/sec	Up to 10ms

Logging Configuration

Log Levels

```
# config/config.exs
config :logger, :console,
  level: :info, # :debug, :info, :warning, :error
  format: "$time $metadata[$level] $message\n",
  metadata: [:request_id, :message_id, :route_id]
```

Production Recommended: `:info` or `:warning` **Development Recommended:** `:debug`

Log Output Destinations

Console Only (Development):

```
config :logger,
  backends: [:console]
```

File Logger (Production):

```
config :logger,
  backends: [:console, {LoggerFileBackend, :file_log}]

config :logger, :file_log,
  path: "/var/log/sms_c/application.log",
  level: :info,
  format: "$time $metadata[$level] $message\n",
  metadata: [:request_id, :message_id]
```

Log Rotation

Using logrotate (Linux):

```
# /etc/logrotate.d/sms_c
/var/log/sms_c/*.log {
    daily
    rotate 30
    compress
    delaycompress
    notifempty
    create 0644 sms_user sms_group
    sharedscripts
    postrotate
        # Signal application to reopen log file
        systemctl reload sms_c
    endscript
}
```

Common Configuration Scenarios

High-Volume Aggregator

Optimize for maximum throughput (5,000+ messages/second):

```
# Database
config :sms_c, SmsC.Repo,
  pool_size: 50

# Batch worker
config :sms_c,
  batch_insert_batch_size: 200,
  batch_insert_flush_interval_ms: 200

# Message retention
config :sms_c,
  dead_letter_time_minutes: 1440 # 24 hours

# Charging (disabled for performance)
config :sms_c,
  default_charging_enabled: false

# Cleanup (extended intervals)
config :sms_c,
  cleanup_interval_minutes: 30
```

Enterprise Real-Time Messaging

Optimize for low latency (< 20ms):

```
# Database
config :sms_c, SmsC.Repo,
  pool_size: 20

# Batch worker (low latency)
config :sms_c,
  batch_insert_batch_size: 20,
  batch_insert_flush_interval_ms: 10

# Message retention
config :sms_c,
  dead_letter_time_minutes: 4320 # 3 days

# Charging (enabled) – balance checks over Diameter Ro
# (see the Charging Configuration section for the :charging +
# :diameter_ex blocks)
config :sms_c,
  default_charging_enabled: true
```

Multi-Tenant Service Provider

Separate configuration per tenant:

```
# Tenant 1 environment
export DB_NAME=sms_c_tenant1
export NODE_NAME=sms_tenant1@node1.example.com
# Per-tenant OCS rating is selected via the :charging
# rating_profile config

# Tenant 2 environment
export DB_NAME=sms_c_tenant2
export NODE_NAME=sms_tenant2@node1.example.com
```

Geographic Redundancy

Cluster across regions:

```
# US East cluster
config :sms_c,
  cluster_nodes: [
    : "sms@us-east-1a.example.com",
    : "sms@us-east-1b.example.com",
    : "sms@us-west-1a.example.com" # Cross-region for DR
  ],
  smsc_node_name: "us-east-1a"
```

Configuration Validation

Test configuration before deployment:

```
# Check configuration syntax
mix compile

# Validate database connection
mix ecto.create
mix ecto.migrate

# Check the Diameter Ro peer to the OCS (if charging enabled), in
a remote console:
#   :diameter.service_info(:omnimessage, :connections)

# Start application in interactive mode
iex -S mix phx.server
```

Environment Variables Reference

Common environment variables used in configuration:

Variable	Purpose	Example
DB_USERNAME	Database username	sms_prod_user
DB_PASSWORD	Database password	strong_password
DB_HOSTNAME	Database host	db.internal.example.com
DB_PORT	Database port	3306
DB_NAME	Database name	sms_c_production
DB_POOL_SIZE	Connection pool size	30
API_PORT	API listen port	8443
API_LISTEN_IP	API listen IP	0.0.0.0
WEB_PORT	Web UI port	443
NODE_NAME	Erlang node name	sms@node1.example.com
ERLANG_COOKIE	Cluster secret	shared_cookie_value

Configuration Best Practices

- 1. **Use Environment Variables** for sensitive values (passwords, API keys)
- 2. **Test Configuration Changes** in staging before production
- 3. **Document Custom Settings** in deployment notes
- 4. **Version Control Config Files** (excluding secrets)
- 5. **Monitor After Changes** for performance regressions
- 6. **Keep Backups** of working configurations
- 7. **Validate Before Restart** to avoid startup failures
- 8. **Use Consistent Naming** across environments
- 9. **Set Resource Limits** appropriate to hardware

10. **Review Periodically** to remove unused features

Troubleshooting Configuration Issues

Symptom	Likely Cause	Solution
Application won't start	Syntax error in config	Check logs, validate syntax
Database connection fails	Wrong credentials/host	Verify DB_* environment variables
API not accessible	Wrong port/IP binding	Check API_PORT and listen_ip
Cluster nodes won't connect	Cookie mismatch, firewall	Verify ERLANG_COOKIE, check ports 4369, 9100-9200
Charging failures	OCS/DRA unreachable	Check Diameter Ro peer (charging fails open)
ENUM lookups fail	DNS server unreachable	Test DNS connectivity, check timeout
Poor performance	Wrong batch settings	Review Performance Tuning Guide
Messages not routing	Routes not loaded	Check sms_routes config or Web UI

For additional help, see the [Troubleshooting Guide](#).

Message Storage Configuration (Mnesia)

Message Retention

Messages are stored in Mnesia for fast access with configurable automatic cleanup.

```
config :sms_c,  
  # How long to keep messages in Mnesia (hours)  
  message_retention_hours: 24,  
  
  # How often to check for old messages (minutes)  
  retention_check_interval_minutes: 60
```

Recommendations:

- **Production:** 24-72 hours (balance operational needs vs memory)
- **Development:** 4-8 hours (faster cleanup for testing)
- **High volume:** 12-24 hours (conserve memory)

Memory Impact:

- Average message: ~1KB
- 10,000 messages: ~10MB
- 100,000 messages: ~100MB

CDR (Call Detail Record) Export

When messages are delivered or expired, CDRs can be automatically written to your Ecto database for long-term storage and billing analytics.

```
config :sms_c,  
  # Enable/disable CDR writing  
  cdr_enabled: true
```

CDR Records Include:

- Message ID, calling/called numbers
- Source/destination SMSC
- Origin/destination node (for clusters)
- Submission, delivery, expiry timestamps
- Status, delivery attempts
- Optional message body (see privacy controls)

When to Disable:

- Testing environments where CDRs aren't needed
- Temporary troubleshooting to reduce database load

Privacy Controls

Configure message body visibility and retention for privacy compliance.

```
config :sms_c,  
  # Delete message body from Mnesia after successful delivery  
  delete_message_body_after_delivery: false,  
  
  # Hide message body in web UI  
  hide_message_body_in_ui: false,  
  
  # Hide message body in CSV exports  
  hide_message_body_in_export: false
```

Use Cases:

Configuration	Use Case
<code>delete_message_body_after_delivery: true</code>	Save Mnesia space, privacy compliance
<code>hide_message_body_in_ui: true</code>	Prevent operator viewing of message content
<code>hide_message_body_in_export: true</code>	Data export compliance, sanitized reports

Example Configurations:

Maximum Privacy (Compliance)

```
config :sms_c,
  delete_message_body_after_delivery: true,
  hide_message_body_in_ui: true,
  hide_message_body_in_export: true,
  cdr_enabled: true # Keep CDRs without bodies
```

Development (Full Visibility)

```
config :sms_c,
  delete_message_body_after_delivery: false,
  hide_message_body_in_ui: false,
  hide_message_body_in_export: false,
  cdr_enabled: true
```

Startup Logging

On application startup, configuration status is logged:

```
[info] Message storage: Mnesia (retention: 24h)
[info] CDR export: ENABLED
[info] Body deletion after delivery: DISABLED
[info] OCS charging: ENABLED (url: http://..., tenant: ...)
```

This provides immediate visibility into active features.

Geographic Federation

[← Back to Documentation Index](#) | [Configuration Reference](#) | [Operations Guide](#)

Overview

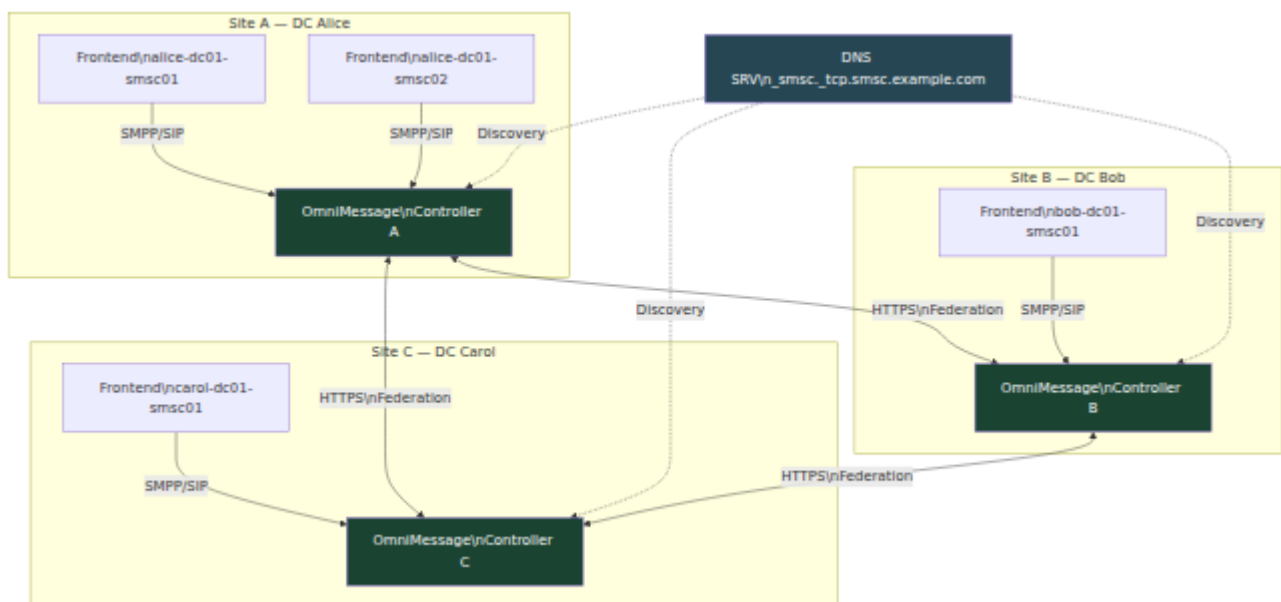
OmniMessage uses **HTTP-based federation** with a **pull model** for multi-controller deployments across data centres or regions. Each controller operates independently — managing its own message queue, routing table, and set of connected frontends. Controllers discover each other via DNS SRV records (or static configuration), exchange health and frontend registry information over HTTPS.

When routing determines a message belongs to a remote site, the message **stays on the origin controller's Mnesia**. A lightweight notification is sent to the destination controller, which triggers an immediate poll. The destination controller's FederationPoller periodically (and on notification) queries origin peers for messages destined to its local frontends, caches them, and makes them available to local frontends. After delivery, the destination controller reports delivery status back to the origin.

Key Characteristics

Aspect	Detail
Network	Works over any WAN, including unreliable links
Shared state	None — each controller is independent
Message ownership	Messages stay on the origin controller until delivered
Security	HTTPS with TLS
Ports	Single HTTPS port (8443)
Scale target	5-20 controllers
Partition handling	Poller retries — no split-brain

Architecture



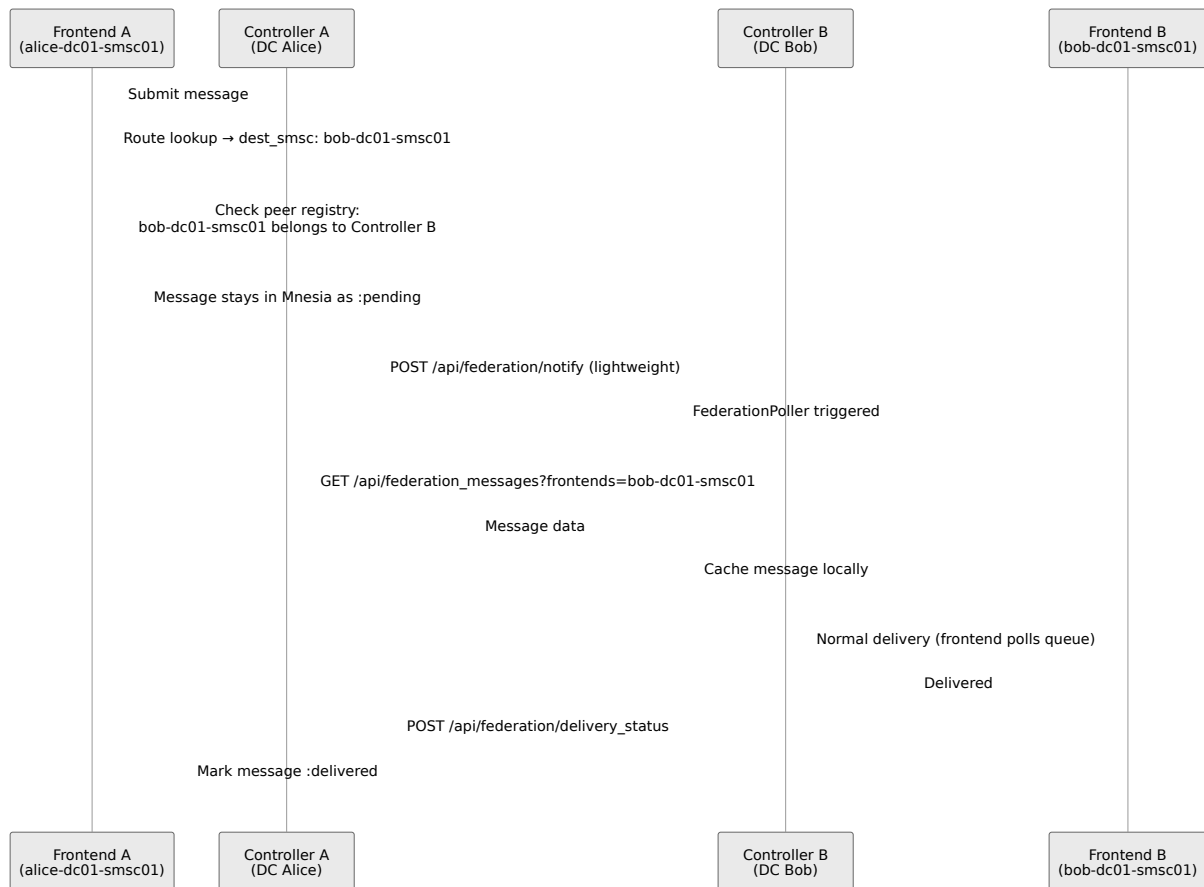
Key Design Decisions

- **Message flow (pull model):** Messages stay on the origin controller. The destination controller pulls them via periodic polling and on-demand notification. Frontends only communicate with their home controller.
- **Partition handling:** Poller retries automatically. Messages remain safe on the origin controller until successfully delivered.
- **Config sync:** None. Each controller manages its own routing table independently.
- **Peer mesh:** Full mesh. All controllers exchange health and frontend registries with all other controllers.
- **Discovery:** DNS SRV records (recommended) or static peer list (for environments without dynamic DNS).

Message Flow (Pull Model)

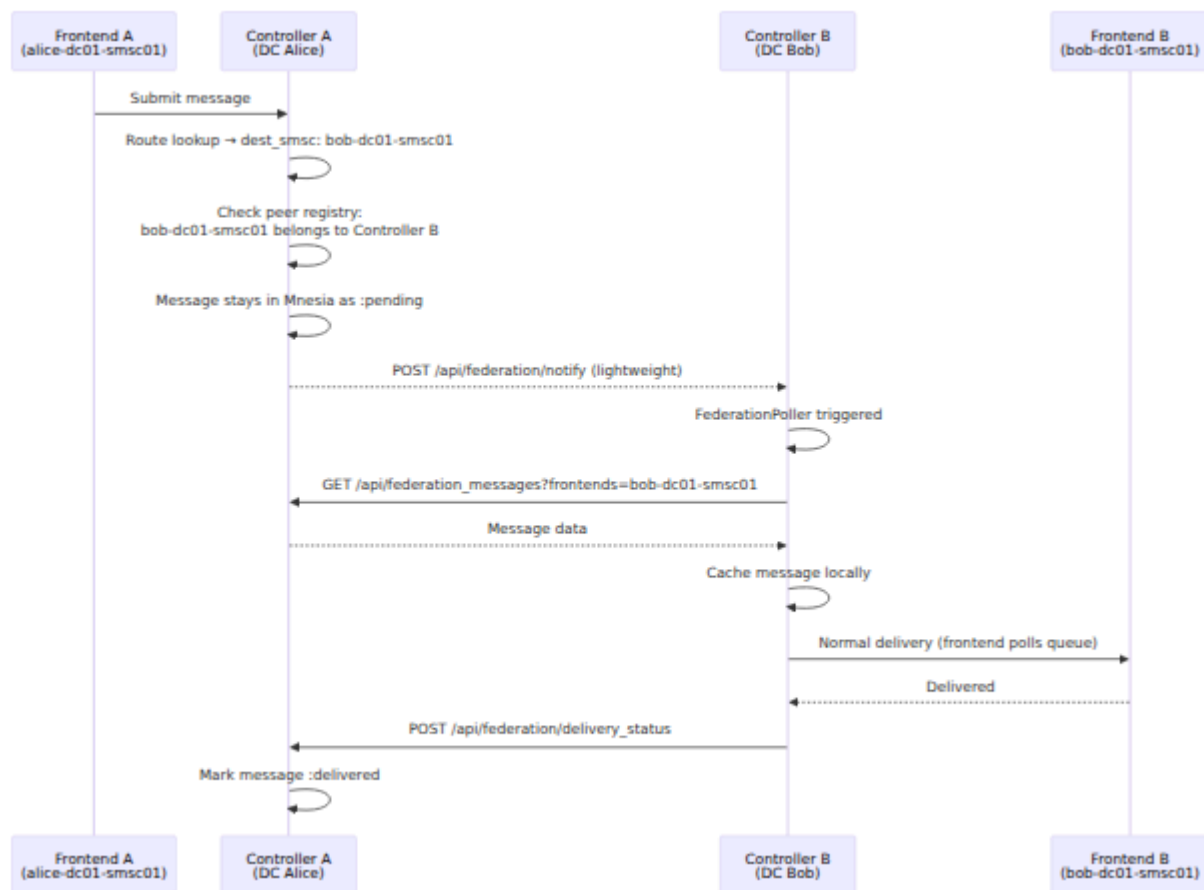
When a message arrives at a controller and routing determines the destination frontend is on a remote controller:

1. The message stays in the origin controller's Mnesia as `:pending`
2. A lightweight notification is sent to the destination controller (fire-and-forget)
3. The destination controller's FederationPoller fetches the message via GET `/api/federation_messages`
4. The message is cached locally and made available to local frontends
5. After delivery, the destination reports status back to the origin



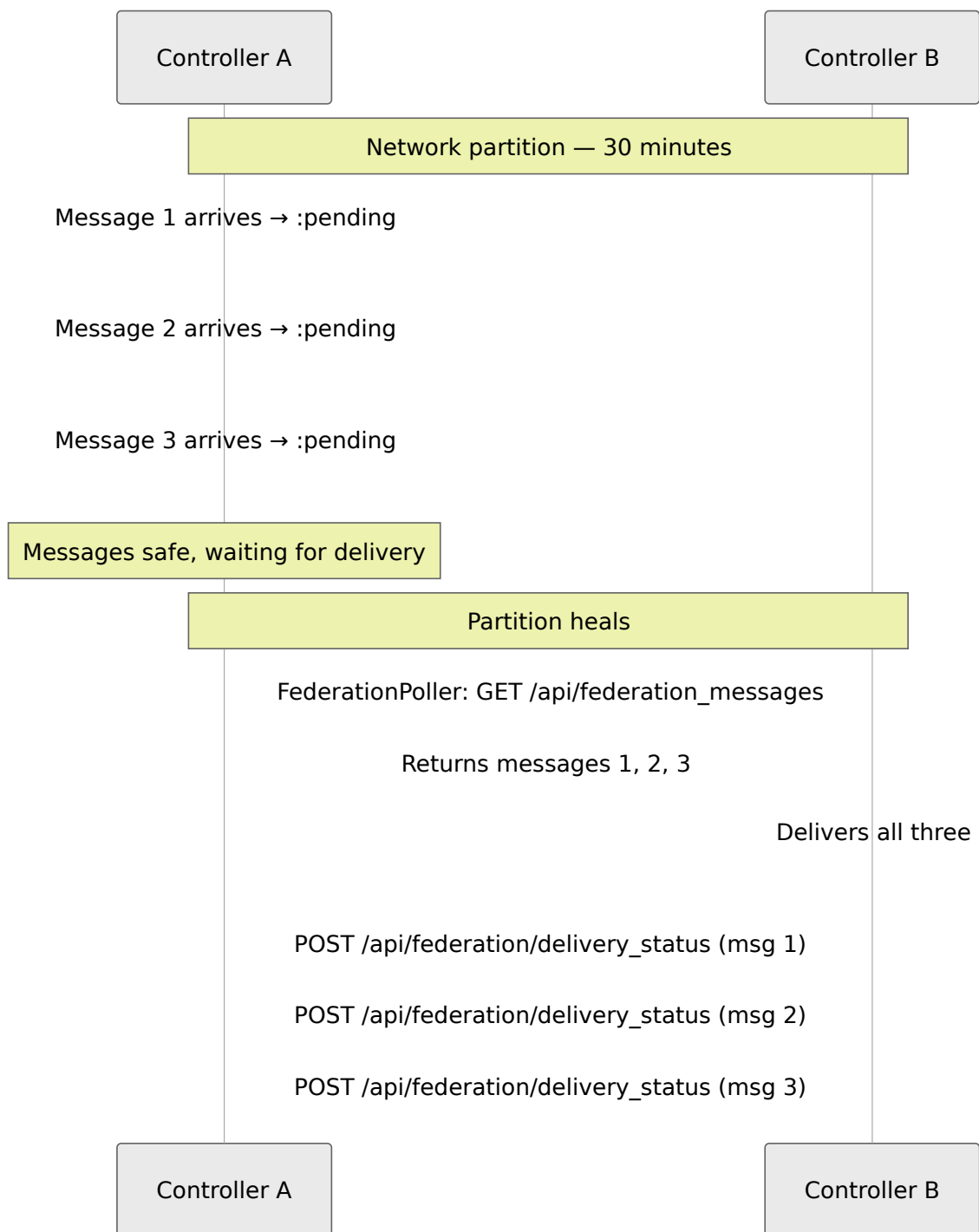
Peer Unreachable — Notification Fails

If the destination controller is down when the notification is sent, the notification silently fails. The message remains `:pending` on the origin. When the destination controller recovers, its FederationPoller resumes periodic polling and discovers the message during the next cycle (default: every 5 seconds). No messages are lost.



Partition Recovery — Multiple Queued Messages

During a prolonged partition, messages accumulate on the origin controller. When the link recovers, the FederationPoller catches up automatically — all pending messages are returned in a single query.



Three-Site Message Flow

In a multi-site mesh, each controller only polls for messages destined to its own frontends.



Peer Health and Registry Sync

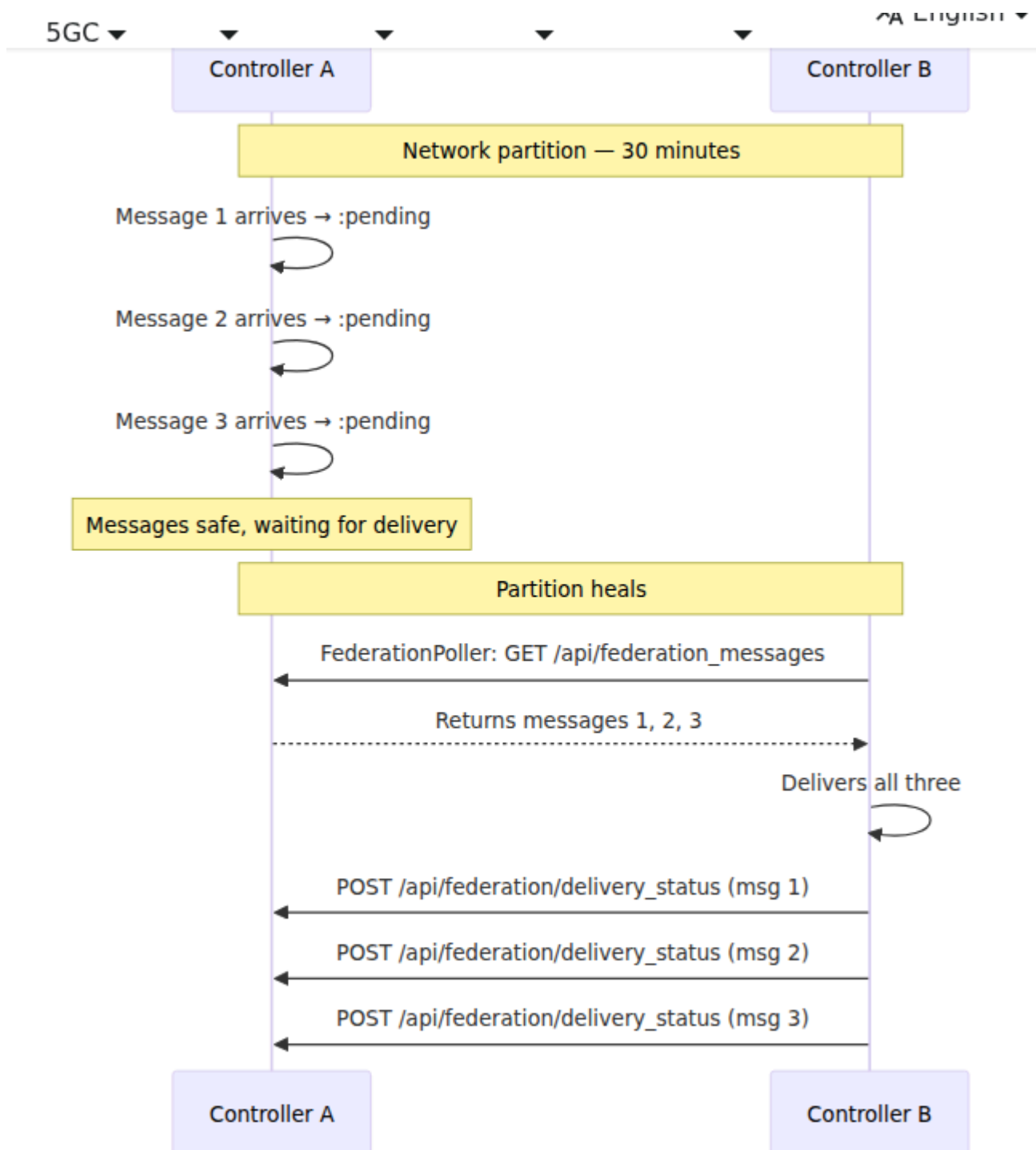
Federation controllers maintain awareness of each other through two periodic cycles:

Health Check Cycle

Every 10 seconds (configurable), each controller calls `POST /api/federation/health` on every known peer. The call is bidirectional — the caller sends its own status and receives the peer's status in the response. A peer is marked **unhealthy** after 3 consecutive failures.

Registry Sync Cycle

Every 15 seconds (configurable), each controller calls `POST /api/federation/registry` on every healthy peer. The call exchanges frontend lists — the caller sends its active frontends and receives the peer's active frontends. This is how controllers learn which peer owns which frontend.



Peer statuses and their effect on federation polling:

Status	Health Checks	FederationPoller	Registry Sync
Unknown	In progress	Not polled	Not attempted
Healthy	Passing	Active polling	Active
Degraded	1-2 failures	Not polled	Not attempted
Unhealthy	3+ failures	Not polled	Not attempted

Configuration

Federation is configured in `config/runtime.exs` under the `:federation` key. It is **disabled by default** and must be explicitly enabled.

Minimal Configuration (DNS SRV Discovery)

```
# config/runtime.exs
config :sms_c, :federation,
  enabled: true,
  dns_srv_domain: "_smc._tcp.smc.example.com"
```

Full Configuration Reference

```
# config/runtime.exs
config :sms_c, :federation,
  # Master switch – federation is completely inactive when false
  enabled: true,

  # DNS SRV domain for peer discovery (leave empty to use
  static_peers instead)
  dns_srv_domain: "_smc._tcp.smc.example.com",

  # How often to re-resolve DNS SRV records (milliseconds)
  dns_poll_interval_ms: 30_000,

  # How often to exchange health status with all known peers
  (milliseconds)
  health_check_interval_ms: 10_000,

  # How often to exchange frontend registries with healthy peers
  (milliseconds)
  registry_sync_interval_ms: 15_000,

  # How often the FederationPoller queries healthy peers for
  messages (milliseconds)
  poll_interval_ms: 5_000,

  # HTTP timeout for all peer-to-peer API calls (milliseconds)
  http_timeout_ms: 5_000,

  # API port used when constructing peer URLs from DNS SRV records
  api_port: 8443,

  # Static peer list – used when dns_srv_domain is empty
  static_peers: []
```

Federation Parameters

Parameter	Type	Required	Default	Description
<code>enabled</code>	Boolean	Yes	<code>false</code>	Master switch for federation. When <code>false</code> , federation services start but do not discover or contact peers.
<code>dns_srv_domain</code>	String	No	<code>""</code>	DNS SRV domain for automatic discovery. When empty, <code>static_peers</code> is used instead.
<code>dns_poll_interval_ms</code>	Integer	No	<code>30000</code>	Interval between DNS SRV re-resolution attempts. Lower values detect new peers faster but increase DNS load.
<code>health_check_interval_ms</code>	Integer	No	<code>10000</code>	Interval between health check rounds. Each round checks all known peers.

Parameter	Type	Required	Default	Description
<code>registry_sync_interval_ms</code>	Integer	No	<code>15000</code>	Interval between frontend registry sync rounds. Each round contains every healthy peer.
<code>poll_interval_ms</code>	Integer	No	<code>5000</code>	How often Federation queries healthy peers for messages. Notification trigger immediate polls.
<code>http_timeout_ms</code>	Integer	No	<code>5000</code>	Timeout for each individual HTTPS call to a peer. Applies to health checks, registry sync and message fetching.
<code>api_port</code>	Integer	No	<code>8443</code>	API port used when constructing peer URLs from DNS SRV records. Must match the

Parameter	Type	Required	Default	Description
				port configuration in <code>config</code> : <code>api_ex</code> .
<code>static_peers</code>	List	No	<code>[]</code>	List of peer configurations for environments without DNS SRV. Each is a map with <code>host</code> and <code>port</code> keys.

Static Peer Configuration

For environments where DNS SRV is not available, configure peers explicitly:

```
config :sms_c, :federation,  
  enabled: true,  
  dns_srv_domain: "",  
  static_peers: [  
    %{host: "10.0.1.2", port: 8443},  
    %{host: "10.0.2.2", port: 8443},  
    %{host: "10.0.3.2", port: 8443}  
  ]
```

Parameter	Type	Required	Default	Description
<code>host</code>	String	Yes	-	IP address or hostname of the remote controller.
<code>port</code>	Integer	No	<code>8443</code>	HTTPS API port of the remote controller.

DNS SRV Record Setup

DNS SRV records allow controllers to automatically discover each other. Each controller resolves the configured domain and uses the resulting host/port pairs as peers.

DNS Zone Configuration:

```
; Priority Weight Port Target
_smsg._tcp.smsg.example.com. 86400 IN SRV 10 100 8443 smsg-
alice.smsg.example.com.
_smsg._tcp.smsg.example.com. 86400 IN SRV 10 100 8443 smsg-
bob.smsg.example.com.
_smsg._tcp.smsg.example.com. 86400 IN SRV 10 100 8443 smsg-
carol.smsg.example.com.

; A records for the targets
smsg-alice.smsg.example.com.      86400 IN A 10.0.1.2
smsg-bob.smsg.example.com.       86400 IN A 10.0.2.2
smsg-carol.smsg.example.com.     86400 IN A 10.0.3.2
```

Adding a new site: Add a new SRV record to DNS. All existing controllers will discover the new peer within one `dns_poll_interval_ms` cycle (default 30 seconds).

Removing a site: Remove the SRV record from DNS. Existing controllers will remove the departed peer within one poll cycle.

Network Requirements

Federation requires only a single HTTPS port between sites.

Port	Protocol	Direction	Purpose
8443	TCP/TLS	Bidirectional	All federation traffic (health, registry, notify, messages, delivery status)
53	UDP/TCP	Outbound	DNS SRV resolution (if using DNS discovery)

Firewall rules (per-site):

```
# Allow federation traffic from peer controller IPs
iptables -A INPUT -p tcp -s 10.0.1.0/24 --dport 8443 -j ACCEPT
iptables -A INPUT -p tcp -s 10.0.2.0/24 --dport 8443 -j ACCEPT
iptables -A INPUT -p tcp -s 10.0.3.0/24 --dport 8443 -j ACCEPT
```

Federation API Endpoints

All federation endpoints are served on the standard API port (8443) under `/api/federation/`. These endpoints are called by peer controllers, not by external clients or frontends.

Endpoint	Method	Purpose
<code>/api/federation/identity</code>	GET	Returns this controller's node name and protocol version
<code>/api/federation/health</code>	POST	Bidirectional health exchange — caller sends status, callee responds with own
<code>/api/federation/registry</code>	POST	Bidirectional frontend registry exchange
<code>/api/federation/notify</code>	POST	Receive lightweight notification that a message is available for a local frontend
<code>/api/federation/delivery_status</code>	POST	Receive delivery confirmation from destination controller
<code>/api/federation_messages</code>	GET	Return pending messages for specified frontends (used by FederationPoller)

All requests and responses use JSON. Peer identification is carried in the `X-Federation-Node` header.

Deployment Examples

Example 1: Three-Site Deployment with DNS SRV

Three data centres, each with local frontends for their subscriber base. DNS SRV provides discovery.

```
# Site A – DC Alice (config/runtime.exs)
config :sms_c,
  smsc_node_name: "alice-dc01-smsc01"

config :sms_c, :federation,
  enabled: true,
  dns_srv_domain: "_smsc._tcp.smsc.example.com"
```

```
# Site B – DC Bob (config/runtime.exs)
config :sms_c,
  smsc_node_name: "bob-dc01-smsc01"

config :sms_c, :federation,
  enabled: true,
  dns_srv_domain: "_smsc._tcp.smsc.example.com"
```

```
# Site C – DC Carol (config/runtime.exs)
config :sms_c,
  smsc_node_name: "carol-dc01-smsc01"

config :sms_c, :federation,
  enabled: true,
  dns_srv_domain: "_smsc._tcp.smsc.example.com"
```

How it works: All three controllers resolve the same SRV domain and discover each other automatically. Each controller registers its local frontends. When a message arrives at DC Alice destined for a subscriber served by DC Bob, the routing table selects `bob-dc01-smsc01` as the destination. The federation layer detects this frontend belongs to Controller B and sends a notification. Controller

B's FederationPoller queries Controller A and caches the message for local delivery.

Example 2: Two-Site Active-Active with Static Peers

Two data centres connected by a dedicated link. No DNS SRV available.

```
# Primary DC (config/runtime.exs)
config :sms_c,
  smsc_node_name: "primary-smsc01"

config :sms_c, :federation,
  enabled: true,
  dns_srv_domain: "",
  static_peers: [
    %{host: "10.200.1.5", port: 8443}
  ]
```

```
# Secondary DC (config/runtime.exs)
config :sms_c,
  smsc_node_name: "secondary-smsc01"

config :sms_c, :federation,
  enabled: true,
  dns_srv_domain: "",
  static_peers: [
    %{host: "10.100.1.5", port: 8443}
  ]
```

How it works: Each controller is configured with the other's IP address. Health checks and registry syncs proceed over the dedicated link. Messages stay on whichever controller received them. Each controller's FederationPoller queries the peer for messages destined to local frontends.

Example 3: Aggressive Polling for Low-Latency Links

For sites connected by reliable, low-latency links where fast federation is important:

```
config :sms_c, :federation,  
  enabled: true,  
  dns_srv_domain: "_smc.tcp.cluster.internal",  
  health_check_interval_ms: 5_000,  
  registry_sync_interval_ms: 5_000,  
  poll_interval_ms: 1_000,  
  http_timeout_ms: 2_000
```

Use case: Campus or metro-area deployments where sites are connected by < 10 ms RTT links and fast failover is more important than minimising control traffic.

Metrics

Federation exposes the following telemetry events that can be observed via Prometheus on port 9568.

Metric: `sms_c_federation_message_received_count` **Type:** Counter

Description: Number of federation notifications received from peers **Labels:**

- `origin_node` - Peer that sent the notification

Metric: `sms_c_federation_health_check_count` **Type:** Counter **Description:** Health check results per peer **Labels:**

- `peer` - Peer identifier
- `result` - `ok` or `failed`

Example queries:


```
# Notification rate per minute
rate(sms_c_federation_message_received_count[5m]) * 60

# Health check failures by peer
sum by (peer)
(rate(sms_c_federation_health_check_count{result="failed"}[5m]))
```

Key Indicators to Monitor

What to Watch	Where	Alert Threshold
Cached federated messages	<code>GET /api/federation/status</code> or Web UI	Growing count (peer not delivering)
Peer health status	<code>GET /api/federation/identity</code> on each peer	Any peer unhealthy > 5 minutes
Health check latency	Application logs	> 2 seconds (link degradation)

Troubleshooting

Peer Not Discovered

Symptoms: Controller logs show no peers discovered; `cluster_status` returns empty.

Possible causes:

- DNS SRV domain is incorrect or unresolvable
- DNS server is unreachable from the controller host
- SRV records have not propagated yet
- Federation is not enabled (`enabled: false`)

Resolution:

1. Verify DNS SRV resolution from the controller host: `dig SRV _smc._tcp.smc.example.com`
2. Check that `enabled: true` is set in the federation config
3. Check application logs for "Federation DNS discovery" messages at startup
4. If using static peers, verify the `static_peers` list contains correct host/port entries

Peer Marked Unhealthy

Symptoms: Messages to a specific site are not being polled. Logs show "Health check failed" messages.

Possible causes:

- Remote controller is down
- Firewall blocking port 8443 between sites
- TLS certificate issues
- Network partition between sites

Resolution:

1. Verify the remote controller is running: `curl -k https://<peer-host>:8443/api/federation/identity`
2. Check firewall rules allow traffic on port 8443 from the local controller's IP
3. Review application logs on both sides for TLS or connection errors
4. Check network connectivity between sites

Messages Not Being Delivered to Remote Site

Symptoms: Messages remain `:pending` on the origin controller. Cached federated message count is 0 on the destination.

Possible causes:

- Destination controller is unhealthy (FederationPoller only queries healthy peers)
- FederationPoller is not running
- Frontend registry not synced (destination doesn't know which frontends are local)

Resolution:

1. Check peer status on the destination controller
2. Verify the FederationPoller is in the supervision tree (check application logs for "Federation poller started")
3. Verify frontend registries are synced (GET /api/federation/status — check frontends lists)
4. Manually test the federation_messages endpoint: `curl -k 'https://<origin>:8443/api/federation_messages?frontends=<frontend_name>&include_unrouted=false'`

HSS Subscriber Lookup (Diameter Sh)

[← Back to Documentation Index](#) | [SMS Routing Guide](#)

Overview

OmniMessage can query the Home Subscriber Server (HSS) via the Diameter Sh interface to determine whether an MSISDN belongs to the network (on-net) or not (off-net). This answer is used as a **route match criterion**: routes can require the originating or terminating number to be on-net or off-net before they apply.

The dip is **lazy** — it happens *during route-table evaluation*, only when a route being evaluated declares an `originating_network` or `terminating_network` requirement of `:on_net` or `:off_net`. Routes that leave both at `:any` (the default) never trigger a dip. There is no global HSS lookup performed before routing.

OmniMessage sends a **User-Data-Request (UDR)** for the number. If the HSS confirms the subscriber exists (result code 2001) the number is **on-net**; if the HSS returns *Unknown User* (result code 5001) the number is **off-net**. See [SMS Routing Guide → On-Net / Off-Net Matching](#) for how these criteria combine with the rest of a route.

On-net vs off-net — HSS or ENUM? The HSS (Sh) answers "is this number a subscriber in *our* HSS?" with an explicit on/off-net result. ENUM (DNS NAPTR) answers "which carrier domain does this number resolve to?" and you derive on/off-net by matching your own domain. Use Sh when your HSS is authoritative; use ENUM for portability/inter-carrier routing. See [SMS Routing Guide → ENUM Lookup](#).

Relevant Specifications

Specification	Title
3GPP TS 29.329	Sh interface based on Diameter
3GPP TS 29.328	IMS Sh interface — signalling flows and message contents
RFC 6733	Diameter Base Protocol

Where the HSS Dip Fits

OmniMessage evaluates a message in this order:

Message received after
number translation



OmniCore
5GC ▼

OmniCall ▼

OmniRAN ▼

OmniCharge ▼

Platform ▼

🌐 English ▼

Yes

Route directly
to\serving frontend

No

Evaluate routing table

Route
has\non_net/off_net\ncriterion?

Yes

HSS dip via
HssCache\ncached per-
MSISDN

HSS result

2001 — on-net

Criterion satisfied
if\nroute wants :on_net

5001 — off-net

Criterion satisfied
if\nroute wants :off_net

Error / timeout

Criterion fails —\nroute
does not match

No

Match on other criteria
only

Any route matched?

Yes



No



Execute action of
selected route

No route found —
message held

Location-Based Routing (checked first)

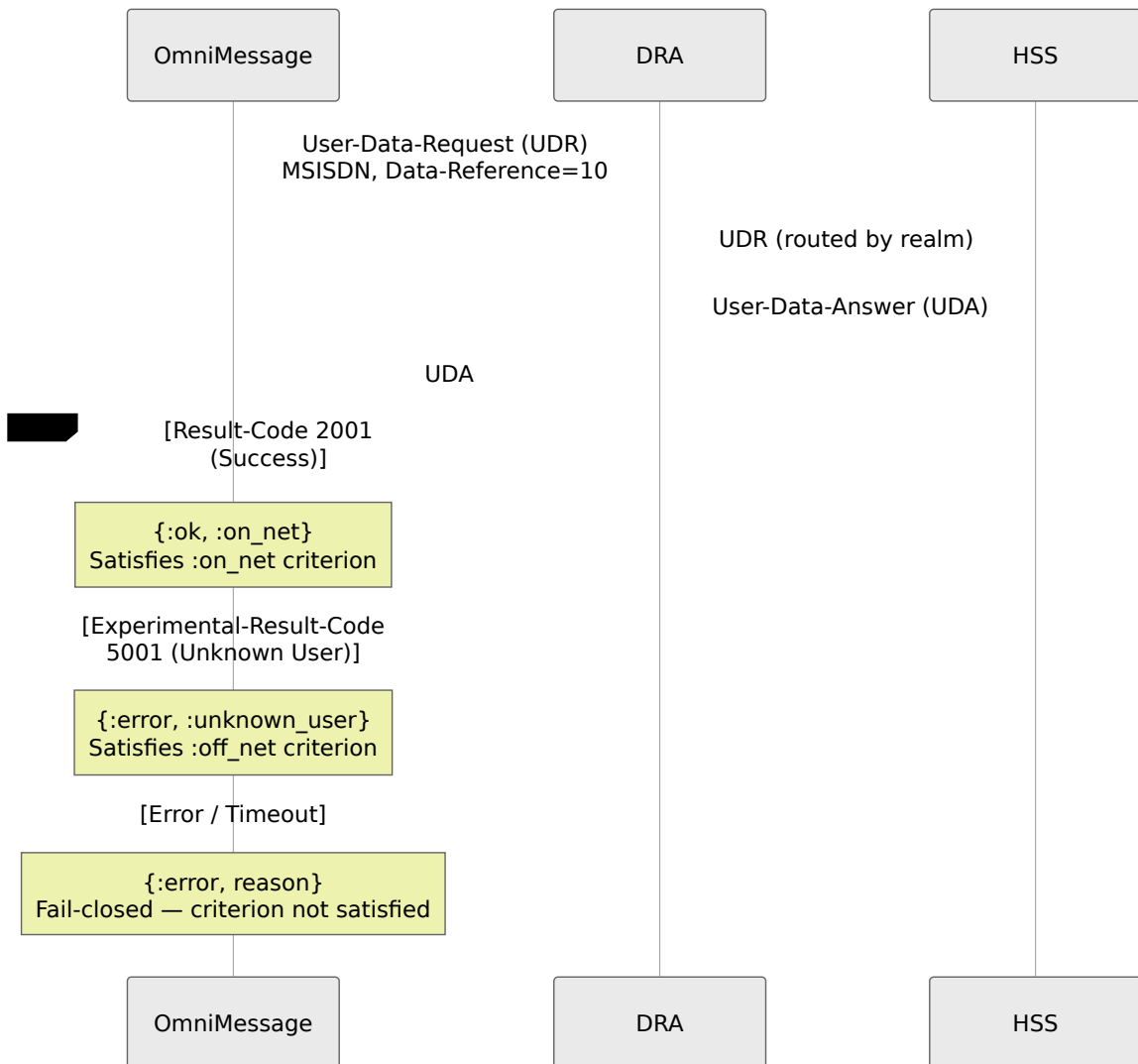
If the destination MSISDN is found in the registrations/locations table, the message is routed directly to the frontend serving that subscriber. No HSS lookup or routing-table evaluation occurs. (See [Location Expiry](#).)

Routing Table + Lazy HSS Dip

If there is no active registration, the routing table is evaluated. For each route that declares `originating_network/terminating_network` of `:on_net` or `:off_net`, an HSS dip is performed (via the cache) as part of matching that route. A route only matches if its network criterion is satisfied along with all its other criteria. If Diameter is disabled or the HSS does not give a definitive answer, the criterion fails (fail-closed) and the route does not match.

Holding a message for **store-and-forward** is a separate concept — it is triggered by the `:saf` route *action*, not by the HSS dip. See [Store-and-Forward Delivery](#).

Diameter Sh Message Flow



UDR Parameters

AVP	Value	Description
Session-Id	Auto-generated	Unique session identifier
Auth-Session-State	1 (NO_STATE_MAINTAINED)	No session state required for queries
Data-Reference	10	Request all user data
Origin-Host	Configured host + realm	OmniMessage's Diameter identity
Origin-Realm	Configured realm	OmniMessage's Diameter realm
Destination-Realm	Configured realm	HSS realm (routed via DRA)
User-Identity	TBCD-encoded MSISDN	The subscriber's number
Server-Name	"OmniMessage"	Application server identity
Vendor-Specific-Application-Id	Auth: 16777217, Vendor: 10415	Sh application identifier per 3GPP TS 29.329

UDA Result Codes

`SmsC.Diameter.Sh.subscriber_exists?/1` maps the UDA to one of three results:

Code	Type	Result	Meaning for route matching
2001	Result-Code	<code>{:ok, :on_net}</code>	On-net — satisfies <code>:on_net</code> , fails <code>:off_net</code>
5001	Experimental-Result-Code	<code>{:error, :unknown_user}</code>	Off-net — satisfies <code>:off_net</code> , fails <code>:on_net</code>
Other / timeout / no peers	—	<code>{:error, reason}</code>	Fail-closed — satisfies neither; route does not match

Configuration

Enabling HSS Lookup

HSS lookup is controlled by two configuration blocks in `config/runtime.exs`:

1. The `diameter_enabled` flag under `:sms_c` — controls whether HSS dips are performed during route matching. When `false`, any route with an `:on_net`/`:off_net` criterion fails to match (fail-closed).
2. The `:diameter_ex` configuration — configures the Diameter stack itself (identity, peers, applications)

Minimal Configuration

```
# Enable HSS dips during route matching
config :sms_c,
  diameter_enabled: true

# Diameter stack configuration
config :diameter_ex,
  diameter: %{
    service_name: :omnimessage,
    listen_ip: "0.0.0.0",
    listen_port: 3868,
    decode_format: :map,
    host: "smsc01",
    realm: "epc.mnc005.mcc547.3gppnetwork.org",
    product_name: "OmniMessage",
    request_timeout: 5000,
    peer_selection_algorithm: :random,
    allow_undefined_peers_to_connect: true,
    log_unauthorized_peer_connection_attempts: true,
    control_module: SmsC.Diameter.Control,
    processor_module: DiameterEx.Processor,
    auth_application_ids: [],
    acct_application_ids: [],
    vendor_id: 10415,
    supported_vendor_ids: [10415],
    applications: [
      %{
        application_name: :sh,
        application_dictionary: :diameter_gen_3gpp_sh,
        vendor_specific_application_ids: [
          %{
            vendor_id: 10415,
            auth_application_id: 16_777_217,
            acct_application_id: nil
          }
        ]
      }
    ],
    peers: [
      %{
        port: 3868,
        host: "dra01.epc.mnc005.mcc547.3gppnetwork.org",
```

```
    ip: "10.0.0.1",  
    realm: "epc.mnc005.mcc547.3gppnetwork.org",  
    tls: false,  
    transport: :diameter_tcp,  
    initiate_connection: true  
  }  
]  
}
```

OmniMessage Parameters

Parameter	Type	Required	Default	Description
<code>diameter_enabled</code>	Boolean	No	<code>false</code>	Enable HSS discovery during route matching. When <code>false</code> , routes <code>:on_net/:off</code> criteria fail-close.
<code>hss_cache_ttl_seconds</code>	Integer	No	<code>7200</code>	TTL (seconds) cached HSS lookup results in <code>runtime.exclude</code> (not exposed to the operator Y schema).
<code>mock_sh</code>	Boolean	No	<code>false</code>	Use mock HSS responses instead of real Diameter queries. For testing only.
<code>mock_sh_on_net_numbers</code>	List	No	<code>[]</code>	List of MSISDNs treated as on-net when <code>mock_sh</code> is <code>true</code> .

Diameter Stack Parameters

Parameter	Type	Required	Default
service_name	Atom	Yes	-
listen_ip	String	No	"0.0.0.0"
listen_port	Integer	No	3868
decode_format	Atom	Yes	-
host	String	Yes	-
realm	String	Yes	-
product_name	String	No	-
request_timeout	Integer	No	500
peer_selection_algorithm	Atom	No	:random

Parameter	Type	Required	Default
<code>allow_undefined_peers_to_connect</code>	Boolean	No	<code>true</code>
<code>log_unauthorized_peer_connection_attempts</code>	Boolean	No	<code>true</code>
<code>control_module</code>	Module	Yes	-
<code>processor_module</code>	Module	Yes	-
<code>auth_application_ids</code>	List	No	<code>[]</code>
<code>acct_application_ids</code>	List	No	<code>[]</code>
<code>vendor_id</code>	Integer	Yes	-
<code>supported_vendor_ids</code>	List	Yes	-

Application Parameters

The `applications` list configures which Diameter applications are enabled. For HSS lookup, only the Sh application is required.

Parameter	Type	Required	Default	
<code>application_name</code>	Atom	Yes	-	Application name. Must be a valid lookup.
<code>application_dictionary</code>	Atom	Yes	-	Erlang dictionary. Must be a valid dictionary. <code>:diameter</code>
<code>vendor_specific_application_ids</code>	List	Yes	-	Vendor specific application IDs. See below.

Vendor-Specific Application ID (Sh):

Parameter	Type	Value	Description
<code>vendor_id</code>	Integer	<code>10415</code>	3GPP vendor ID
<code>auth_application_id</code>	Integer	<code>16_777_217</code>	Sh application ID per 3GPP TS 29.329
<code>acct_application_id</code>	-	<code>nil</code>	Not used for Sh

Peer Parameters

The `peers` list defines the Diameter peers (typically DRAs or the HSS directly) that OmniMessage connects to.

Parameter	Type	Required	Default	Description
<code>host</code>	String	Yes	-	Peer's Diameter Host-Identifier (FQDN). Must match exactly with the peer's configured identity.
<code>ip</code>	String	Yes	-	Peer's IP address for TCP connection.
<code>port</code>	Integer	No	3868	Peer's Diameter port.
<code>realm</code>	String	Yes	-	Peer's Diameter Realm.
<code>tls</code>	Boolean	No	<code>false</code>	Enable TLS for this peer connection.
<code>transport</code>	Atom	No	<code>:diameter_tcp</code>	Transport protocol: <code>:diameter_tls</code> or <code>:diameter_udp</code>
<code>initiate_connection</code>	Boolean	No	<code>true</code>	If <code>true</code> , OmniMessage initiates the connection to the peer. If <code>false</code> , OmniMessage

Parameter	Type	Required	Default	Description
				waits for the peer to connect

Multiple Peers

Configure multiple peers for redundancy. OmniMessage selects among connected peers using the `peer_selection_algorithm`:

```
peers: [
  %{
    port: 3868,
    host: "dra01.epc.mnc005.mcc547.3gppnetwork.org",
    ip: "10.0.0.1",
    realm: "epc.mnc005.mcc547.3gppnetwork.org",
    tls: false,
    transport: :diameter_tcp,
    initiate_connection: true
  },
  %{
    port: 3868,
    host: "dra02.epc.mnc005.mcc547.3gppnetwork.org",
    ip: "10.0.0.2",
    realm: "epc.mnc005.mcc547.3gppnetwork.org",
    tls: false,
    transport: :diameter_tcp,
    initiate_connection: true
  }
]
```

Mock Mode

For testing environments without a live HSS, enable mock mode:

```
config :sms_c,  
  diameter_enabled: true,  
  mock_sh: true,  
  mock_sh_on_net_numbers: [  
    "68987654321",  
    "68912345678"  
  ]
```

In mock mode, any MSISDN in the `mock_sh_on_net_numbers` list resolves to `{:ok, :on_net}` (equivalent to result code 2001). All other MSISDNs resolve to `{:error, :unknown_user}` (equivalent to result code 5001). No Diameter stack is started and no network connections are made.

HSS Result Cache

HSS lookups are cached per-MSISDN by `SmsC.Diameter.HssCache` to avoid redundant Diameter round-trips:

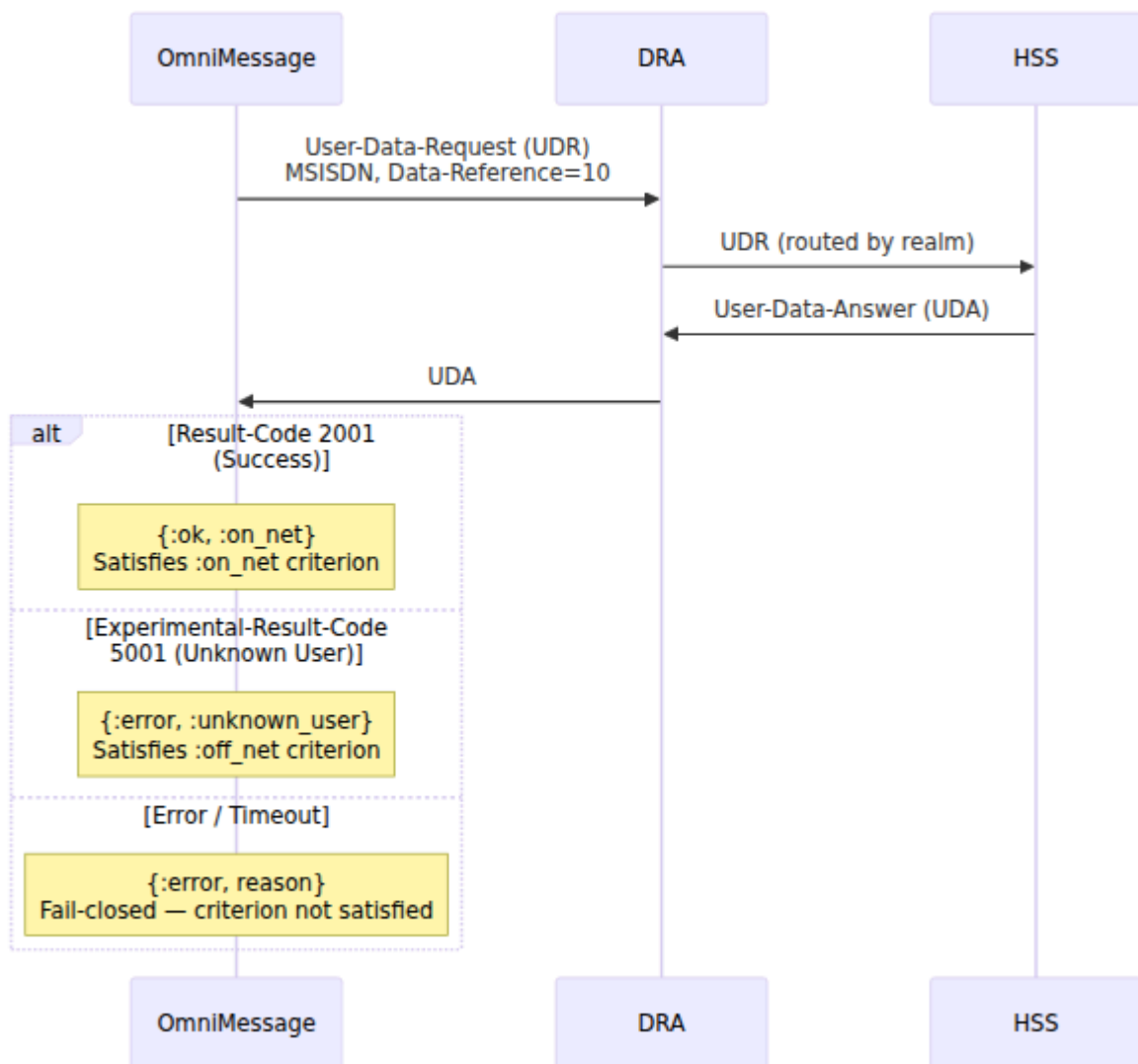
- **TTL:** `hss_cache_ttl_seconds` (default 7200 = 2 hours)
- **Shared:** multiple routes needing the same MSISDN's status in one evaluation share a single dip
- **Forced refresh:** `HssCache.lookup!/1` bypasses the cache (used when a message is explicitly re-routed)
- **Predicates:** `HssCache.on_net?/1` is true only for `{:ok, :on_net}`; `off_net?/1` only for `{:error, :unknown_user}`; every other result (including `nil` when Diameter is disabled) is treated as neither — which is what produces fail-closed matching.

Store-and-Forward Delivery

Store-and-forward is driven by the `:saf` route **action**, not by the HSS dip. A route with `action: :saf` holds the message for later delivery. To hold *only* on-net subscribers, combine `:saf` with `terminating_network: :on_net` and provide an off-net fallback route (see [SMS Routing Guide](#) → `saf`).

When a message is held for SAF, it remains in the Mnesia message queue with `dest_smsc` unset. When the subscriber registers and a location update arrives, the delivery loop picks up pending messages for that MSISDN and routes them to the serving frontend.

Messages in the queue are subject to the standard retention policy. If the subscriber does not register before the message expires, the message is moved to the dead letter queue per the configured `dead_letter_time_minutes`.



Operational Notes

Startup Logging

OmniMessage logs the Diameter Sh status on startup:

- **Enabled:** `Diameter Sh (HSS): ENABLED (host: smsc01, realm: epc.mnc005.mcc547.3gppnetwork.org, peers: 2)`
- **Disabled:** `Diameter Sh (HSS): DISABLED`

Fail-Closed Behaviour

If an HSS lookup does not return a definitive answer — timeout, no connected peers, unhandled response, or an unexpected result code —

`subscriber_exists?/1` returns `{:error, reason}`. For a route requiring `:on_net` or `:off_net`, that criterion is **not** satisfied, so the route **does not match** and evaluation continues with the next route.

This is a **fail-closed** design: without a definitive answer, OmniMessage will not treat a number as on-net or off-net, so an on-net-only route won't fire and an off-net-only gateway route won't fire either. Provide a catch-all fallback route (`terminating_network: :any`) if you want messages to route during HSS outages.

When **Diameter is explicitly disabled** (`diameter_enabled: false`), `HssCache.lookup/1` returns `nil` and the same fail-closed matching applies to any route with an `:on_net`/`:off_net` criterion. Routes that leave both network fields at `:any` are unaffected.

Observability

HSS dips are observable through:

- **Per-message route logs** — each route's `route_evaluated` event includes the network check, e.g. `term_network: ✓ (terminating requires on_net, +15551234 is on-net)` or `✗ (... is off-net)`. Visible in the control panel message activity log and the Events API.

- **Diameter Sh telemetry** — request, response (labelled by result), duration, and error metrics are emitted by `SmsC.Telemetry` (see [METRICS.md](#)).
- **Application logs** — `HSS dip: <msisdn> is ON-NET (2001) / is NOT in HSS (5001)` lines from `SmsC.Diameter.Sh`.

Location Expiry

Location-based routing checks the `expires` timestamp on subscriber registrations. If a subscriber's registration has expired — for example, they entered airplane mode — the location is treated as inactive and evaluation proceeds to the routing table (including any lazy HSS dip a matching route requires).

This prevents messages from being sent to a frontend that can no longer reach the subscriber, avoiding unnecessary delivery failures and retry cycles.

Route-Level On-Net / Off-Net Enforcement

Routes enforce on-net/off-net requirements through two fields — **not** boolean flags and **not** a route action:

Field	Values	Checks	Behaviour
<code>originating_network</code>	<code>:any</code> <code>:on_net</code> <code>:off_net</code>	Calling number	Route matches only if the source MSISDN's HSS status matches (fail-closed)
<code>terminating_network</code>	<code>:any</code> <code>:on_net</code> <code>:off_net</code>	Called number	Route matches only if the destination MSISDN's HSS status matches (fail-closed)

Default for both is `:any` (no HSS check). A value of `:on_net` requires result code 2001; `:off_net` requires 5001; any indefinite answer fails the criterion.

Configuration Example

```
config :sms_c, :sms_routes, [  
  # On-net destinations only – deliver to local gateway  
  %{  
    calling_regex: nil,  
    called_regex: ~r/^1907/,  
    source_smsc: nil,  
    dest_smsc: "local-gateway",  
    source_type: nil,  
    action: :route,  
    charged: :default,  
    weight: 100,  
    priority: 10,  
    description: "Local gateway – on-net destinations only",  
    originating_network: :any,  
    terminating_network: :on_net,  
    enabled: true  
  },  
  
  # Off-net fallback – catch-all for everything the on-net route  
  # didn't take  
  %{  
    calling_regex: nil,  
    called_regex: nil,  
    source_smsc: nil,  
    dest_smsc: "external-gateway",  
    source_type: :ims,  
    action: :route,  
    charged: :default,  
    weight: 100,  
    priority: 255,  
    description: "External gateway – fallback",  
    originating_network: :any,  
    terminating_network: :any,  
    enabled: true  
  }  
]  
1
```

How it works: for a message to an unregistered subscriber:

1. **Location check** — no active registration found
2. **Route evaluation** — routes are checked in priority order
3. The P10 "local-gateway" route has `terminating_network: :on_net`, so an HSS dip is performed on the called number. If the HSS returns 2001 (on-net), the route matches and the message is delivered to `local-gateway`. If it returns 5001 (off-net) or fails (fail-closed), the route does not match.
4. The P255 "external-gateway" route has `terminating_network: :any`, so it needs no HSS answer and matches as the fallback.

Use case: keep on-net traffic on local infrastructure while off-net (or HSS-uncertain) traffic reaches an external carrier. To *hold* on-net subscribers for store-and-forward instead of delivering to a gateway, set the P10 route's `action` to `:saf`.

Troubleshooting

On-Net Subscribers Routed to the Fallback Gateway

Symptoms: Messages to on-net subscribers take the off-net/fallback route instead of the on-net route.

Possible causes:

- `diameter_enabled` is `false` — the `:on_net` criterion fails-closed, so the on-net route never matches
- No Diameter peers are connected (check application logs for peer connection status)
- HSS is returning errors or timing out (check the Sh error telemetry and "HSS dip" log lines)
- Number translation is producing a different number format than the HSS expects

Resolution:

1. Verify `diameter_enabled: true` in `runtime.exs`
2. Check logs for Diameter peer connection status on startup (`Diameter Sh (HSS): ENABLED (... peers: N)`)
3. Look for `HSS dip:` log lines and Sh error metrics
4. Confirm the number format after translation matches what the HSS expects (E.164 without `+`)

Messages Held But Never Delivered

Symptoms: Messages are correctly held (`:saf`) for on-net subscribers but never delivered, even after the subscriber registers.

Possible causes:

- Location updates are not reaching OmniMessage
- The MSISDN in the location registration does not match the queued message's `destination_msisdn`
- Messages have expired and been moved to the dead letter queue

Resolution:

1. Verify location/registration updates are being received (check the Registrations page in the Control Panel)
2. Confirm the MSISDN format is consistent between the message queue and registration table
3. Check `dead_letter_time_minutes` — increase if subscribers may be offline for extended periods

Diameter Peer Not Connecting

Symptoms: Logs show no Diameter peer connections on startup.

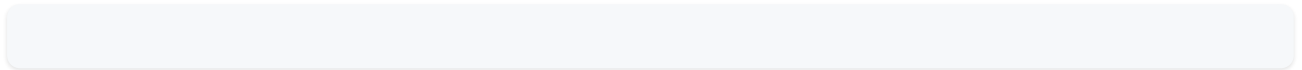
Possible causes:

- Incorrect peer IP or port in configuration
- Firewall blocking port 3868

- Peer host identity mismatch (the configured `host` must exactly match the peer's Diameter identity)
- Realm mismatch between OmniMessage and the peer

Resolution:

1. Verify peer IP address and port are correct
2. Confirm firewall rules allow TCP traffic on the configured port
3. Check that the peer's `host` value exactly matches what the peer advertises in its capability exchange
4. Ensure `realm` matches between OmniMessage and peer configurations



SMS-C Prometheus Metrics Documentation

[← Back to Documentation Index](#)

Overview

This document describes all Prometheus metrics exposed by the SMS-C system. These metrics are designed for operations staff to monitor system health, performance, and troubleshoot issues.

Accessing Metrics

The Prometheus metrics endpoint is available at:

```
http://localhost:9568/metrics
```

This endpoint exposes metrics in Prometheus text format that can be scraped by a Prometheus server. The metrics are updated in real-time as the system processes messages.

Metric Naming Convention

All metrics follow the pattern: `sms_c.<category>.<metric_name>.<type>`

Categories:

- `license` - License status metrics
- `message` - Message processing metrics
- `routing` - Routing decision metrics
- `enum` - ENUM/NAPTR lookup metrics

- `delivery` - Message delivery metrics
- `queue` - Queue management metrics
- `charging` - Billing/charging metrics
- `mnesia` - Database metrics
- `frontend` - Frontend connection metrics
- `location` - Location/registration metrics
- `phoenix.endpoint` - HTTP API request metrics
- `vm` - Erlang VM system metrics

License Metrics

`sms_c_license_status`

Type: Gauge

Description: Current license status of the OmniMessage SMS-C system.

Values:

- `1` - Valid license
- `0` - Invalid/expired license

Labels: None

Product Name: `omnimessage`

Use Case: Monitor license validity to ensure system is operating with a valid license. When invalid, messages are still received but routed to destination "NOLICENCE" instead of normal routing.

Behavior When License Invalid:

- Inbound messages are **accepted** and stored
- Message destination (`dest_smsc`) is **automatically set to "NOLICENCE"**
- Normal routing is **bypassed**
- UI and monitoring remain **accessible**

- Database and all services remain **operational**

Alerting:

```
- alert: SMS_C_License_Invalid
  expr: sms_c_license_status == 0
  for: 1m
  labels:
    severity: critical
  annotations:
    summary: "SMS-C license invalid or expired"
    description: "License status is invalid - messages being
routed to NOLICENCE"
```

Example Prometheus Queries:

```
# Check if license is valid
sms_c_license_status == 1

# Alert on invalid license
sms_c_license_status == 0

# Count messages routed to NOLICENCE (indicates license issue)
sms_c_routing_route_matched_count{dest_smsc="NOLICENCE"}
```

Message Processing Metrics

sms_c_message_received_count

Type: Counter

Description: Total number of messages received by the SMS-C from all sources.

Labels:

- `source_smsc`: Name of the source SMSC that sent the message

- `source_type`: Type of source connection (ims, circuit_switched, smpp)
- `message_type`: Type of message (sms, mms)

Use Case: Monitor incoming message volume by source and type. Use to detect traffic patterns, identify busy periods, and spot anomalies in message flow.

Alerting: Set alerts for sudden drops (potential source connectivity issues) or spikes (potential attack/spam).

sms_c_message_validated_count

Type: Counter

Description: Total number of message validations performed.

Labels:

- `valid`: Whether validation passed (true or false)

Use Case: Track validation success/failure rates. High failure rates may indicate malformed messages or integration issues.

Alerting: Alert when validation failure rate exceeds threshold (e.g., > 5% failures).

sms_c_message_processing_stop_duration

Type: Histogram

Description: Time taken to process a message from receipt to completion (includes validation, routing, and queueing).

Unit: Milliseconds

Buckets: 10, 50, 100, 250, 500, 1000, 2500, 5000 ms

Labels:

- `success`: Whether processing succeeded (true or false)

Use Case: Monitor end-to-end message processing performance. Identify slowdowns in the processing pipeline.

Alerting: Alert when p95 or p99 latency exceeds SLA thresholds.

Routing Metrics

`sms_c_routing_route_matched_count`

Type: Counter

Description: Total number of times a specific route was matched and selected for message routing.

Labels:

- `route_id`: Unique identifier of the matched route
- `dest_smsc`: Destination SMSC selected by the route
- `priority`: Priority value of the matched route

Use Case: Understand which routes are being used most frequently. Identify underutilized or overloaded routes. Useful for capacity planning and route optimization.

Alerting: Alert if high-priority routes are rarely matched (may indicate routing misconfiguration).

`sms_c_routing_failed_count`

Type: Counter

Description: Total number of routing failures where no suitable route could be found.

Labels:

- `reason`: Failure reason (no_route_found, validation_failed, etc.)

Use Case: Track routing failures to identify configuration gaps or unexpected traffic patterns.

Alerting: Alert on any routing failures as they indicate messages cannot be delivered.

sms_c_routing_action_count

Type: Counter

Description: Total number of special routing actions taken.

Labels:

- `action`: Type of action (drop, auto_reply, forward)
- `route_id`: Route that triggered the action

Use Case: Monitor drop rules (anti-spam), auto-reply usage, and forwarding patterns.

Alerting: Alert on unexpected spikes in drop actions (may indicate spam attack).

sms_c_routing_stop_duration

Type: Histogram

Description: Time taken to evaluate all routes and select the best match.

Unit: Milliseconds

Buckets: 1, 5, 10, 25, 50, 100, 250, 500 ms

Labels:

- `dest_smsc`: Selected destination SMSC

Use Case: Monitor routing engine performance. Slow routing indicates too many routes or complex matching logic.

Alerting: Alert when routing takes consistently longer than expected (e.g., p95 > 50ms).

ENUM/NAPTR Lookup Metrics

`sms_c_enum_cache_hit_count`

Type: Counter

Description: Total number of ENUM lookups served from cache (did not require DNS query).

Labels:

- `domain`: ENUM domain queried

Use Case: Monitor cache effectiveness. High cache hit rates reduce DNS load and improve performance.

Alerting: Alert if cache hit rate drops below threshold (may indicate cache issues or unusual traffic).

`sms_c_enum_cache_miss_count`

Type: Counter

Description: Total number of ENUM lookups that required a DNS query (not in cache).

Labels:

- `domain`: ENUM domain queried

Use Case: Track cache misses to understand cache effectiveness. Use with hit count to calculate hit rate.

Calculation: `cache_hit_rate = hits / (hits + misses)`

`sms_c_enum_cache_size_size`

Type: Gauge

Description: Current number of entries in the ENUM cache.

Use Case: Monitor cache size to ensure it's not growing unbounded. Help tune cache TTL settings.

Alerting: Alert if cache size exceeds expected bounds (may indicate memory leak).

`sms_c_enum_lookup_stop_duration`

Type: Histogram

Description: Time taken to complete an ENUM lookup (including DNS query if not cached).

Unit: Milliseconds

Buckets: 10, 50, 100, 250, 500, 1000, 2500, 5000 ms

Labels:

- `domain`: ENUM domain queried
- `success`: Whether lookup succeeded (true or false)
- `cache_hit`: Whether result was served from cache (true or false)

Use Case: Monitor ENUM lookup performance. Identify slow DNS servers or network issues.

Alerting: Alert when p95 lookup time exceeds timeout threshold.

sms_c_enum_naptr_records_record_count

Type: Histogram

Description: Number of NAPTR records returned by a successful ENUM lookup.

Buckets: 0, 1, 2, 3, 5, 10

Labels:

- `domain`: ENUM domain queried

Use Case: Understand ENUM record distribution. Most lookups should return 1-3 records.

Alerting: Alert if frequently returning 0 records (DNS configuration issue).

Delivery Metrics

sms_c_delivery_queued_count

Type: Counter

Description: Total number of messages queued for delivery to a destination SMSC.

Labels:

- `dest_smsc`: Destination SMSC name

Use Case: Monitor message flow to each destination. Useful for capacity planning.

Alerting: Compare with delivery success/failure counts to detect accumulation.

sms_c_delivery_attempted_count

Type: Counter

Description: Total number of delivery attempts made (includes retries).

Labels:

- `dest_smsc`: Destination SMSC name

Use Case: Track delivery attempt volume. High attempt count relative to queued count indicates retry behavior.

sms_c_delivery_succeeded_count

Type: Counter

Description: Total number of messages successfully delivered to destination SMSC.

Labels:

- `dest_smsc`: Destination SMSC name

Use Case: Track successful deliveries per destination. Primary success metric.

Alerting: Alert if success rate drops below SLA threshold.

Calculation: `success_rate = succeeded / queued`

sms_c_delivery_failed_count

Type: Counter

Description: Total number of messages that failed delivery after all retry attempts.

Labels:

- `dest_smsc`: Destination SMSC name
- `reason`: Failure reason

Use Case: Track delivery failures to identify problematic destinations or failure patterns.

Alerting: Alert on elevated failure rates or specific failure reasons.

sms_c_delivery_dead_letter_count

Type: Counter

Description: Total number of messages moved to dead letter queue (undeliverable).

Labels:

- `reason`: Reason for dead letter (e.g., `max_retries_exceeded`, `expired`)

Use Case: Monitor undeliverable messages requiring manual intervention.

Alerting: Alert on any dead letter events as they represent complete delivery failure.

sms_c_delivery_succeeded_attempt_count

Type: Histogram

Description: Number of delivery attempts required before successful delivery.

Buckets: 1, 2, 3, 5, 10

Labels:

- `dest_smsc`: Destination SMSC name

Use Case: Understand retry behavior. Most deliveries should succeed on first attempt.

Alerting: Alert if average attempt count exceeds 2 (indicates destination reliability issues).

sms_c_delivery_failed_attempt_count

Type: Histogram

Description: Number of delivery attempts made before final failure.

Buckets: 1, 2, 3, 5, 10

Labels:

- `dest_smsc`: Destination SMSC name

Use Case: Understand how many retries occur before giving up.

sms_c_delivery_time_delta_delta_ms

Type: Histogram

Description: Time from message submission to delivery confirmation. This metric captures the complete end-to-end latency from when a message enters the SMS-C to when delivery is confirmed, with detailed labels for analysis by source, destination, and retry behavior.

Unit: Milliseconds

Buckets: 50, 100, 250, 500, 1000, 2000, 5000, 10000, 20000, 60000, 600000 ms (50ms to 10 minutes)

Labels:

- `source_smsc`: Source SMSC that submitted the message
- `dest_smsc`: Destination SMSC that delivered the message
- `delivery_attempts`: Number of delivery attempts required (0 = first attempt success)

Use Case: Analyze end-to-end delivery performance by source-destination pair. Identify slow paths, problematic SMSC combinations, and correlate latency with retry behavior. Essential for SLA monitoring and capacity planning.

Alerting:

- Alert when p95 exceeds SLA threshold for specific source/dest combinations
- Alert when messages consistently require multiple attempts

Example Prometheus Queries:

```

# P95 delivery time by source and destination SMSC
histogram_quantile(0.95,
  sum by (source_smsc, dest_smsc, le) (
    rate(sms_c_delivery_time_delta_delta_ms_bucket[5m])
  )
)

# P99 delivery time overall
histogram_quantile(0.99,
  sum by (le) (
    rate(sms_c_delivery_time_delta_delta_ms_bucket[5m])
  )
)

# Average delivery time by source SMSC
sum by (source_smsc) (rate(sms_c_delivery_time_delta_delta_ms_sum[5m]
/
sum by (source_smsc)
(rate(sms_c_delivery_time_delta_delta_ms_count[5m])))

# Delivery time for messages requiring retries vs first-attempt success
histogram_quantile(0.95,
  sum by (le)
(rate(sms_c_delivery_time_delta_delta_ms_bucket{delivery_attempts="0"
[5m])))
)
histogram_quantile(0.95,
  sum by (le)
(rate(sms_c_delivery_time_delta_delta_ms_bucket{delivery_attempts!="0"
[5m])))
)

# Slowest source-destination pairs (by p95)
topk(10,
  histogram_quantile(0.95,
    sum by (source_smsc, dest_smsc, le) (
      rate(sms_c_delivery_time_delta_delta_ms_bucket[1h])
    )
  )
)

# Percentage of messages delivered within 2 seconds
sum(rate(sms_c_delivery_time_delta_delta_ms_bucket{le="2000"}[5m])) /

```



```
sum(rate(sms_c_delivery_time_delta_delta_ms_count[5m])) * 100

# Messages delivered within 2 seconds by destination
sum by (dest_smsc)
(rate(sms_c_delivery_time_delta_delta_ms_bucket{le="2000"}[5m])) /
sum by (dest_smsc) (rate(sms_c_delivery_time_delta_delta_ms_count[5m]
* 100
```

sms_c_delivery_time_by_smsc_delta_ms

Type: Histogram

Description: Time from message submission to delivery confirmation, labelled by source and destination SMSC only. This is a lower-cardinality companion to `sms_c_delivery_time_delta_delta_ms` — it omits the `delivery_attempts` label so average delivery time per source/destination pair can be read directly without aggregating across attempt counts. Emitted from the same `[ :sms_c, :delivery, :time_delta]` event.

Unit: Milliseconds

Buckets: 50, 100, 250, 500, 1000, 2000, 5000, 10000, 20000, 60000, 600000 ms (50ms to 10 minutes)

Labels:

- `source_smsc`: Source SMSC that submitted the message
- `dest_smsc`: Destination SMSC that delivered the message

Use Case: Dashboard the average end-to-end delivery time per source/destination pair. Prefer this metric over `sms_c_delivery_time_delta_delta_ms` for average/SLA panels where the attempt-count breakdown is not needed; use the `_delta` variant when you need to slice by retry behaviour.

Example Prometheus Queries:

```
# Average delivery time (ms) by source/destination SMSC
sum by (source_smsc, dest_smsc)
(rate(sms_c_delivery_time_by_smsc_delta_ms_sum[5m])) /
sum by (source_smsc, dest_smsc)
(rate(sms_c_delivery_time_by_smsc_delta_ms_count[5m]))

# Overall average delivery time
sum(rate(sms_c_delivery_time_by_smsc_delta_ms_sum[5m])) /
sum(rate(sms_c_delivery_time_by_smsc_delta_ms_count[5m]))

# P95 delivery time by source/destination SMSC
histogram_quantile(0.95,
  sum by (source_smsc, dest_smsc, le) (
    rate(sms_c_delivery_time_by_smsc_delta_ms_bucket[5m])
  )
)
```

Queue Metrics

sms_c_queue_size_size

Type: Gauge

Description: Current total number of messages in queue (all states combined).

Labels:

- `queue_type`: Type of queue (message_queue, dead_letter)

Use Case: Monitor queue depth to detect backlogs or processing issues.

Alerting: Alert when queue size exceeds capacity thresholds.

sms_c_queue_size_pending

Type: Gauge

Description: Current number of messages pending delivery (not yet attempted).

Labels:

- `queue_type`: Type of queue

Use Case: Monitor pending message count. High pending counts indicate processing delays.

Alerting: Alert when pending count exceeds threshold for extended period.

sms_c_queue_size_failed

Type: Gauge

Description: Current number of messages in failed state (awaiting retry).

Labels:

- `queue_type`: Type of queue

Use Case: Monitor failed message accumulation. Indicates delivery issues.

Alerting: Alert on elevated failed count as it impacts delivery rates.

sms_c_queue_size_delivered

Type: Gauge

Description: Current number of delivered messages awaiting cleanup/removal from queue.

Labels:

- `queue_type`: Type of queue

Use Case: Monitor cleanup lag. High counts indicate cleanup process is falling behind.

Alerting: Alert if delivered messages accumulate significantly.

sms_c_queue_oldest_message_age_seconds

Type: Gauge

Description: Age (in seconds) of the oldest message currently in pending state.

Labels:

- `queue_type`: Type of queue

Use Case: Detect message aging and processing stalls. Critical for SLA monitoring.

Alerting: Alert when oldest message age exceeds SLA threshold (e.g., > 300 seconds).

Charging Metrics

Charging is performed over the **Diameter Ro** interface (Credit-Control). The `account` label carries the charged subscriber's MSISDN (the message sender). A `charging_failed` event is emitted both for a definitive out-of-balance decline (`reason: :no_credit`) and for an OCS error (transport/protocol). Only a `:no_credit` decline rejects a message (status `:balance_rejected`); OCS errors fail open and the message is delivered.

sms_c_charging_requested_count

Type: Counter

Description: Total number of charging requests (Diameter Ro CCR Event-Requests) sent to the OCS.

Labels:

- `account`: Charged subscriber MSISDN (message sender)

Use Case: Track charging volume per account. Useful for billing reconciliation.

sms_c_charging_succeeded_count

Type: Counter

Description: Total number of successful charging operations.

Labels:

- `account`: Account identifier charged

Use Case: Monitor charging success rate per account.

Calculation: `success_rate = succeeded / requested`

sms_c_charging_failed_count

Type: Counter

Description: Total number of failed charging operations.

Labels:

- `account`: Account identifier
- `reason`: Failure reason

Use Case: Identify charging failures that may impact revenue or require account intervention.

Alerting: Alert on elevated charging failure rates.

sms_c_charging_succeeded_duration

Type: Histogram

Description: Time taken to complete a successful charging request.

Unit: Milliseconds

Buckets: 10, 50, 100, 250, 500, 1000, 2500, 5000 ms

Labels:

- `account`: Account identifier

Use Case: Monitor billing system performance. Slow charging can delay message delivery.

Alerting: Alert when p95 charging time exceeds threshold.

System Health Metrics

sms_c_mnesia_table_size_record_count

Type: Gauge

Description: Current number of records in each Mnesia database table. Polled every 10 seconds.

Labels:

- `table`: Table name

Tracked Tables:

- `sms_route` - SMS routing rules
- `message_store` - Message queue (pending, delivered, failed messages)

- `location_store` - Subscriber location/registration data
- `frontend_store` - Frontend/SMSC registrations
- `translation_rule` - Number translation rules
- `cell_tower_store` - Cell tower location data
- `message_events` - Message event logs

Use Case: Monitor database growth. Detect unexpected data accumulation. Capacity planning.

Alerting: Alert on unexpected table growth rates or when approaching capacity limits.

Example Prometheus Queries:

```
# All table sizes
sms_c_mnesia_table_size_record_count

# Message queue size
sms_c_mnesia_table_size_record_count{table="message_store"}

# Active subscriber registrations
sms_c_mnesia_table_size_record_count{table="location_store"}

# Routing rules count
sms_c_mnesia_table_size_record_count{table="sms_route"}
```

sms_c_frontend_status_count

Type: Gauge

Description: Number of frontends in each connection status.

Labels:

- `frontend_name`: Frontend identifier
- `status`: Connection status (connected, disconnected)

Use Case: Monitor frontend connectivity. Detect connection failures.

Alerting: Alert when expected frontends disconnect.

sms_c_location_registered_count

Type: Counter

Description: Total number of location/subscriber registrations received by the system.

Labels:

- `location`: Frontend/SMSC name where subscriber is registered
- `ims_capable`: Whether the subscriber supports IMS (true/false)

Use Case: Monitor subscriber registration activity. Track IMS vs non-IMS subscribers. Detect registration storms or failures.

Alerting: Set alerts for:

- Registration rate drops (may indicate network issues)
- Unusual spikes in registrations
- High ratio of non-IMS registrations (legacy device influx)

Example Query:

```
# Registration rate per minute
rate(sms_c_location_registered_count[1m])

# IMS vs non-IMS registration ratio
sum(rate(sms_c_location_registered_count{ims_capable="true"}[5m]))
/
sum(rate(sms_c_location_registered_count[5m]))
```

sms_c_location_active_registrations_count

Type: Gauge

Description: Current number of active subscriber registrations, grouped by location (frontend/SMSC) and IMS capability. This metric is polled every 10 seconds and reflects the current state of the location store.

Labels:

- `location`: Frontend/SMSC name where subscribers are registered
- `ims_capable`: Whether the registrations are IMS capable (true/false)

Use Case: Monitor current subscriber distribution across frontends. Track IMS adoption. Identify load imbalances between frontends. Capacity planning for each SMSC instance.

Alerting: Set alerts for:

- Sudden drops in registrations for a location (may indicate frontend issues)
- Imbalanced distribution across frontends
- Total registrations approaching capacity limits

Example Prometheus Queries:

```
# Total active registrations
sum(sms_c_location_active_registrations_count)

# Active registrations by location
sum by (location) (sms_c_location_active_registrations_count)

# Active IMS-capable registrations by location
sum by (location)
(sms_c_location_active_registrations_count{ims_capable="true"})

# IMS adoption rate per location
sum by (location)
(sms_c_location_active_registrations_count{ims_capable="true"}) /
sum by (location) (sms_c_location_active_registrations_count) *
100

# Total IMS vs non-IMS registrations
sum by (ims_capable) (sms_c_location_active_registrations_count)

# Locations with most registrations
topk(10, sum by (location)
(sms_c_location_active_registrations_count))

# Registration distribution as percentage of total
sum by (location) (sms_c_location_active_registrations_count) /
sum(sms_c_location_active_registrations_count) * 100
```

HTTP API Request Metrics

phoenix_endpoint_stop_duration

Type: Distribution (Histogram)

Description: HTTP request processing duration in milliseconds, from request start to response completion.

Labels:

- `route`: API endpoint route (e.g., `/api/messages`, `/api/frontends`)

Buckets: 10ms, 50ms, 100ms, 250ms, 500ms, 1s, 2.5s, 5s

Use Case: Monitor API performance. Identify slow endpoints. Track response time SLAs.

Alerting: Set alerts for:

- P95 latency > 500ms for critical endpoints
- P99 latency > 1s for any endpoint
- Increasing latency trends

Example Query:

```
# P95 response time by endpoint
histogram_quantile(0.95,
  rate(phoenix_endpoint_stop_duration_bucket[5m]))

# Requests slower than 1 second
sum(rate(phoenix_endpoint_stop_duration_bucket{le="1000"}[5m]))
```

phoenix_endpoint_stop_count

Type: Counter

Description: Total number of HTTP requests completed, categorized by route and HTTP status code.

Labels:

- `route`: API endpoint route
- `status`: HTTP status code (200, 201, 400, 404, 500, etc.)

Use Case: Monitor API request volume and success rates. Track error rates by endpoint.

Alerting: Set alerts for:

- Error rate > 5% for any endpoint
- 5xx errors on critical endpoints
- Sudden drops in request volume

Example Query:

```
# Request rate per endpoint
sum by (route) (rate(phoenix_endpoint_stop_count[5m]))

# Error rate by endpoint
sum by (route) (rate(phoenix_endpoint_stop_count{status=~"5.."}
[5m])) /
sum by (route) (rate(phoenix_endpoint_stop_count[5m]))

# Success rate
sum(rate(phoenix_endpoint_stop_count{status=~"2.."}[5m])) /
sum(rate(phoenix_endpoint_stop_count[5m]))
```

phoenix_router_dispatch_exception_count

Type: Counter

Description: Total number of exceptions/errors raised during HTTP request processing.

Labels:

- `route`: API endpoint route where exception occurred
- `kind`: Type of exception (error, exit, throw)

Use Case: Track application errors. Identify problematic endpoints. Monitor system stability.

Alerting: Set alerts for any non-zero value on critical endpoints.

Example Query:

```
# Exception rate by endpoint
rate(phoenix_router_dispatch_exception_count[5m])

# Total exceptions in last hour
increase(phoenix_router_dispatch_exception_count[1h])
```

Erlang VM Metrics

vm_memory_total

Type: Gauge

Description: Total memory allocated by the Erlang VM in bytes.

Use Case: Monitor overall memory usage. Detect memory leaks. Plan capacity.

Alerting: Alert when memory usage > 80% of available system memory.

vm_memory_processes

Type: Gauge

Description: Memory used by Erlang processes in bytes.

Use Case: Track process memory consumption. Most common source of memory growth.

Alerting: Alert on sustained high growth rate.

vm_total_run_queue_lengths_total

Type: Gauge

Description: Total number of processes waiting to be scheduled across all CPU schedulers.

Use Case: Measure system load. High values indicate CPU saturation.

Alerting: Alert when consistently $> 10 \times$ number of CPU cores.

vm_system_counts_process_count

Type: Gauge

Description: Current number of processes running in the VM.

Use Case: Monitor process creation patterns. Detect process leaks.

Alerting: Alert when approaching process limit (default 262,144).

Metric Collection and Polling

The system automatically collects the following metrics every 10 seconds:

- Queue sizes and ages
- Mnesia table sizes
- ENUM cache statistics

All other metrics are event-driven and emitted when the corresponding action occurs.

Common Monitoring Patterns

Delivery Success Rate by Destination

Track the success rate of message delivery for each destination SMSC:

Formula:
$$\frac{(\text{sms_c_delivery_succeeded_count})}{(\text{sms_c_delivery_queued_count})}$$

Interpretation: Should be > 95% for healthy destinations. Lower rates indicate delivery issues.

End-to-End Message Latency

Monitor total time from message receipt to delivery:

Metrics:

- `sms_c_message_processing_stop_duration` (processing)
- `sms_c_delivery_time_delta_delta_ms` (delivery)

Interpretation: Sum represents total user-facing latency.

ENUM Cache Effectiveness

Measure how well the ENUM cache is performing:

Formula:
$$\frac{(\text{sms_c_enum_cache_hit_count})}{(\text{sms_c_enum_cache_hit_count} + \text{sms_c_enum_cache_miss_count})}$$

Interpretation: Should be > 80% after warm-up. Lower rates may indicate short TTL or high traffic variance.

Route Utilization

Identify which routes handle the most traffic:

Metric: `sms_c_routing_route_matched_count` grouped by `route_id`

Interpretation: Use to identify hot routes for optimization and capacity planning.

Queue Backlog Trend

Monitor if message queue is growing (backlog) or shrinking (catching up):

Metrics:

- `sms_c_queue_size_pending` (current pending)
- `sms_c_queue_oldest_message_age_seconds` (age trending)

Interpretation: Growing pending count + increasing age = backlog forming.

Retry Rate

Understand how often delivery retries are required:

Metric: `sms_c_delivery_succeeded_attempt_count` histogram percentiles

Interpretation: If $p95 > 1$, most messages require retries. Indicates destination reliability issues.

Recommended Alerts

Alert	Condition	Severity
High Routing Failure Rate	<code>routing_failed_count</code> increase	Critical
Queue Backlog	<code>queue_size_pending</code> > threshold	Warning
Old Messages in Queue	<code>queue_oldest_message_age_seconds</code> > 300	Critical
Delivery Failure Spike	<code>delivery_failed_count</code> spike	High
Dead Letter Events	<code>delivery_dead_letter_count</code> > 0	High
ENUM Lookup Timeouts	<code>enum_lookup_stop_duration</code> p95 > 5000ms	Warning
Low Cache Hit Rate	ENUM cache hit rate < 0.7	Warning
Frontend Disconnected	<code>frontend_status_count{status="disconnected"}</code> > 0	High
Charging Failures	<code>charging_failed_count</code> > threshold	High

Alert	Condition	Severity
Slow Message Processing	<code>message_processing_stop_duration</code> p95 > 1000ms	Warning

Dashboard Recommendations

Operations Dashboard

Purpose: Real-time system health monitoring

Panels:

1. Message throughput (received/processed/delivered per minute)
 2. Queue sizes (pending, failed, delivered)
 3. Delivery success rate by destination
 4. p95 processing and delivery latency
 5. Active frontends status
 6. Current alerts
-

Performance Dashboard

Purpose: System performance analysis

Panels:

1. Message processing duration histogram
2. Routing duration histogram
3. ENUM lookup duration histogram
4. Charging duration histogram
5. Delivery attempts distribution
6. Cache hit rates

Business Dashboard

Purpose: Traffic and usage analysis

Panels:

1. Messages by source SMSC
 2. Messages by destination SMSC
 3. Route utilization heatmap
 4. Auto-reply and drop action counts
 5. ENUM usage statistics
 6. Charging volume by account
-

Metric Retention

Recommended Prometheus retention settings:

- **Raw metrics:** 15 days
- **5-minute aggregates:** 90 days
- **1-hour aggregates:** 2 years

This provides detailed recent history while maintaining long-term trends for capacity planning.

Troubleshooting with Metrics

Scenario: Messages Not Being Delivered

Investigation Steps:

1. Check `sms_c_message_received_count` - Are messages being received?
2. Check `sms_c_routing_failed_count` - Are they being routed?

3. Check `sms_c_delivery_queued_count` - Are they being queued?
 4. Check `sms_c_delivery_failed_count` - Are delivery attempts failing?
 5. Check `dest_smsc` labels to identify problematic destination
-

Scenario: Slow Message Processing

Investigation Steps:

1. Check `sms_c_message_processing_stop_duration` histogram - Overall processing time
 2. Check `sms_c_routing_stop_duration` - Is routing slow?
 3. Check `sms_c_enum_lookup_stop_duration` - Are ENUM lookups slow?
 4. Check `sms_c_charging_succeeded_duration` - Is charging slow?
 5. Identify bottleneck and investigate specific component
-

Scenario: Growing Message Queue

Investigation Steps:

1. Check `sms_c_queue_size_pending` trend - Is it growing?
 2. Check `sms_c_delivery_attempted_count` - Are delivery attempts happening?
 3. Check `sms_c_delivery_failed_count` - Are they failing?
 4. Check `sms_c_delivery_time_delta_delta_ms` - Is delivery taking too long?
 5. Check `dest_smsc` labels to identify slow destinations
-

Prometheus Query Examples

Message Throughput

Messages Received Per Second (5-minute average):

```
rate(sms_c_message_received_count[5m])
```

Messages Received Per Minute (1-hour average):

```
rate(sms_c_message_received_count[1h]) * 60
```

Total Messages Today:

```
increase(sms_c_message_received_count[24h])
```

Messages by Source Type:

```
sum by (source_type) (rate(sms_c_message_received_count[5m]))
```

Messages by Source SMSC:

```
sum by (source_smsc) (rate(sms_c_message_received_count[5m]))
```

Delivery Performance

Delivery Success Rate (Percentage):

```
(rate(sms_c_delivery_succeeded_count[5m]) /  
rate(sms_c_delivery_queued_count[5m])) * 100
```

Delivery Failure Rate (Percentage):

```
(rate(sms_c_delivery_failed_count[5m]) /  
rate(sms_c_delivery_queued_count[5m])) * 100
```

Average Delivery Attempts (p95):

```
histogram_quantile(0.95,  
sms_c_delivery_succeeded_attempt_count_bucket)
```

Delivery Success by Destination:

```
sum by (dest_smsc) (rate(sms_c_delivery_succeeded_count[5m]))
```

Delivery Failure Reasons:

```
sum by (reason) (rate(sms_c_delivery_failed_count[5m]))
```

Time to Delivery (p95):

```
histogram_quantile(0.95,  
sms_c_delivery_time_delta_delta_ms_bucket)
```

Time to Delivery (p99):

```
histogram_quantile(0.99,  
sms_c_delivery_time_delta_delta_ms_bucket)
```

Queue Metrics

Current Pending Messages:

```
sms_c_queue_size_pending
```

Failed Messages Awaiting Retry:

```
sms_c_queue_size_failed
```

Oldest Message Age (Minutes):

```
sms_c_queue_oldest_message_age_seconds / 60
```

Queue Growth Rate (Messages/Hour):

```
rate(sms_c_queue_size_size[1h]) * 3600
```

Messages Entering Queue:

```
rate(sms_c_delivery_queued_count[5m])
```

Messages Leaving Queue:

```
rate(sms_c_delivery_succeeded_count[5m]) +  
rate(sms_c_delivery_failed_count[5m])
```

Queue Backlog (Entering - Leaving):

```
rate(sms_c_delivery_queued_count[5m]) -  
(rate(sms_c_delivery_succeeded_count[5m]) +  
rate(sms_c_delivery_failed_count[5m]))
```

Routing Performance

Routing Success Rate:

```
(1 - (rate(sms_c_routing_failed_count[5m]) /  
(rate(sms_c_routing_route_matched_count[5m]) +  
rate(sms_c_routing_failed_count[5m])))) * 100
```

Most Used Routes:

```
topk(10, sum by (route_id, dest_smsc)  
(rate(sms_c_routing_route_matched_count[1h])))
```

Routing Latency (p50, p95, p99):

```
histogram_quantile(0.50, sms_c_routing_stop_duration_bucket)
histogram_quantile(0.95, sms_c_routing_stop_duration_bucket)
histogram_quantile(0.99, sms_c_routing_stop_duration_bucket)
```

Routing Failures Per Minute:

```
rate(sms_c_routing_failed_count[5m]) * 60
```

Drop Actions Per Hour:

```
increase(sms_c_routing_action_count{action="drop"}[1h])
```

Auto-Reply Actions Per Hour:

```
increase(sms_c_routing_action_count{action="auto_reply"}[1h])
```

ENUM Performance

ENUM Cache Hit Rate:

```
rate(sms_c_enum_cache_hit_count[5m]) /
(rate(sms_c_enum_cache_hit_count[5m]) +
rate(sms_c_enum_cache_miss_count[5m]))
```

ENUM Cache Hit Percentage:

```
(rate(sms_c_enum_cache_hit_count[5m]) /
(rate(sms_c_enum_cache_hit_count[5m]) +
rate(sms_c_enum_cache_miss_count[5m]))) * 100
```

ENUM Lookup Latency (p95):


```
histogram_quantile(0.95, sms_c_enum_lookup_stop_duration_bucket)
```

ENUM Lookups Per Second (Cached vs Uncached):

```
# Cached (fast)
rate(sms_c_enum_cache_hit_count[5m])

# Uncached (requires DNS query)
rate(sms_c_enum_cache_miss_count[5m])
```

Average NAPTR Records Returned:

```
rate(sms_c_enum_naptr_records_record_count_sum[5m]) /
rate(sms_c_enum_naptr_records_record_count_count[5m])
```

ENUM Cache Size:

```
sms_c_enum_cache_size_size
```

Processing Performance

Message Processing Latency (p95):

```
histogram_quantile(0.95,
sms_c_message_processing_stop_duration_bucket)
```

Message Processing Latency (p99):

```
histogram_quantile(0.99,
sms_c_message_processing_stop_duration_bucket)
```

Processing Failures:

```
rate(sms_c_message_processing_stop_duration_count{success="false"}[5m])
```

Validation Failure Rate:

```
rate(sms_c_message_validated_count{valid="false"}[5m]) /  
rate(sms_c_message_validated_count[5m])
```

Charging Metrics

Charging Success Rate:

```
rate(sms_c_charging_succeeded_count[5m]) /  
rate(sms_c_charging_requested_count[5m])
```

Charging Failures Per Minute:

```
rate(sms_c_charging_failed_count[5m]) * 60
```

Charging Latency (p95):

```
histogram_quantile(0.95, sms_c_charging_succeeded_duration_bucket)
```

Charging Volume by Account:

```
sum by (account) (rate(sms_c_charging_requested_count[1h]))
```

Frontend Health

Active Frontends:

```
sum(sms_c_frontend_status_count{status="connected"})
```

Disconnected Frontends:

```
sum(sms_c_frontend_status_count{status="disconnected"})
```

Frontends by Name:

```
sum by (frontend_name)  
(sms_c_frontend_status_count{status="connected"})
```

System Health

Mnesia Table Sizes:

```
sms_c_mnesia_table_size_record_count
```

Route Count:

```
sms_c_mnesia_table_size_record_count{table="sms_route"}
```

Translation Rule Count:

```
sms_c_mnesia_table_size_record_count{table="translation_rule"}
```

Grafana Dashboard Examples

Dashboard 1: Real-Time Operations

Purpose: Monitor current system activity and health.

Panels:

1. Message Throughput (Graph)

- Query: `rate(sms_c_message_received_count[5m])`
- Query: `rate(sms_c_delivery_succeeded_count[5m])`
- Unit: messages/second
- Legend: `{{source_type}}`

2. Delivery Success Rate (Gauge)

- Query: `(rate(sms_c_delivery_succeeded_count[5m]) / rate(sms_c_delivery_queued_count[5m])) * 100`
- Unit: percent (0-100)
- Thresholds:
 - Red: < 90
 - Yellow: 90-95
 - Green: > 95

3. Queue Depth (Graph)

- Query: `sms_c_queue_size_pending`
- Query: `sms_c_queue_size_failed`
- Unit: messages
- Legend: `{{queue_type}}`

4. Oldest Message Age (Stat)

- Query: `sms_c_queue_oldest_message_age_seconds / 60`
- Unit: minutes
- Thresholds:
 - Green: < 5
 - Yellow: 5-10
 - Red: > 10

5. Active Frontends (Stat)

- Query: `sum(sms_c_frontend_status_count{status="connected"})`
- Unit: count
- Color: Blue

6. Routing Failures (Graph)

- Query: `rate(sms_c_routing_failed_count[5m]) * 60`
- Unit: failures/minute
- Alert threshold: `> 0`

Dashboard 2: Performance Analysis

Purpose: Analyze system performance and identify bottlenecks.

Panels:

1. End-to-End Latency (Graph)

- Query: `histogram_quantile(0.50, sms_c_message_processing_stop_duration_bucket)` (p50)
- Query: `histogram_quantile(0.95, sms_c_message_processing_stop_duration_bucket)` (p95)
- Query: `histogram_quantile(0.99, sms_c_message_processing_stop_duration_bucket)` (p99)
- Unit: milliseconds
- Legend: Percentile

2. Component Latencies (Bar Gauge)

- Routing: `histogram_quantile(0.95, sms_c_routing_stop_duration_bucket)`
- ENUM: `histogram_quantile(0.95, sms_c_enum_lookup_stop_duration_bucket)`
- Charging: `histogram_quantile(0.95, sms_c_charging_succeeded_duration_bucket)`
- Delivery: `histogram_quantile(0.95, sms_c_delivery_time_delta_delta_ms_bucket)`
- Unit: milliseconds
- Horizontal bars

3. Delivery Attempts Distribution (Heatmap)

- Query: `sms_c_delivery_succeeded_attempt_count_bucket`

- Shows how many attempts are typically needed
- Color scale: Blue (1 attempt) to Red (many attempts)

4. ENUM Cache Performance (Graph)

- Hit Rate: $\frac{\text{rate}(\text{sms_c_enum_cache_hit_count}[5\text{m}])}{(\text{rate}(\text{sms_c_enum_cache_hit_count}[5\text{m}]) + \text{rate}(\text{sms_c_enum_cache_miss_count}[5\text{m}]))}$
- Cache Size: `sms_c_enum_cache_size_size`
- Dual Y-axis (rate vs size)

5. Processing Success Rate (Gauge)

- Query: $\frac{(\text{rate}(\text{sms_c_message_processing_stop_duration_count}\{\text{success}=\text{"true"}\}[5\text{m}])}{\text{rate}(\text{sms_c_message_processing_stop_duration_count}[5\text{m}]))} * 100$
- Unit: percent
- Thresholds:
 - Red: < 95
 - Yellow: 95-99
 - Green: > 99

Dashboard 3: Traffic Analysis

Purpose: Analyze message traffic patterns and routing distribution.

Panels:

1. Messages by Source Type (Pie Chart)

- Query: `sum by (source_type)`
`(increase(sms_c_message_received_count[1h]))`
- Shows distribution: IMS vs CS vs SMPP

2. Messages by Source SMSC (Bar Chart)

- Query: `sum by (source_smsc)`
`(rate(sms_c_message_received_count[1h]))`

- Top 10 sources
- Horizontal bars

3. Route Utilization (Table)

- Columns:
 - Route ID
 - Destination SMSC
 - Messages (1h): `sum by (route_id, dest_smsc) (increase(sms_c_routing_route_matched_count[1h]))`
 - Priority
 - Success Rate
- Sorted by message count

4. Delivery by Destination (Graph)

- Query: `sum by (dest_smsc) (rate(sms_c_delivery_succeeded_count[5m]))`
- Unit: messages/second
- Stacked area chart
- Legend: `{{dest_smsc}}`

5. Drop/Auto-Reply Actions (Stat)

- Dropped: `increase(sms_c_routing_action_count{action="drop"}[1h])`
- Auto-Replied: `increase(sms_c_routing_action_count{action="auto_reply"}[1h])`
- Side by side stats

6. Hourly Traffic Pattern (Graph)

- Query: `rate(sms_c_message_received_count[1h]) * 3600`
- Time range: Last 7 days
- Shows daily patterns

Dashboard 4: Capacity & Resources

Purpose: Monitor resource usage and capacity limits.

Panels:

1. Queue Capacity (Graph)

- Current: `sms_c_queue_size_size`
- Capacity line: Fixed value based on system limits
- Shows utilization trend

2. Database Table Growth (Graph)

- Messages:
`sms_c_mnesia_table_size_record_count{table="sms_route"}`
- Translations:
`sms_c_mnesia_table_size_record_count{table="translation_rule"}`
- Trend over last 30 days

3. Message Backlog Trend (Graph)

- Query: `rate(sms_c_delivery_queued_count[5m]) - (rate(sms_c_delivery_succeeded_count[5m]) + rate(sms_c_delivery_failed_count[5m]))`
- Positive = backlog growing
- Negative = catching up

4. Peak Traffic (Stat)

- Query: `max_over_time(rate(sms_c_message_received_count[5m])[24h:])`
- Shows highest 5m rate in last 24h
- Unit: messages/second

5. Capacity Utilization (Gauge)

- Query: `(rate(sms_c_message_received_count[5m]) / MAX_CAPACITY) * 100`

- Replace MAX_CAPACITY with your system limit
- Unit: percent
- Thresholds:
 - Green: < 70
 - Yellow: 70-85
 - Red: > 85

Dashboard 5: SLA Compliance

Purpose: Track SLA metrics and compliance.

Panels:

1. SLA Compliance (Gauge)

- Delivery Success:
$$\frac{\text{rate}(\text{sms_c_delivery_succeeded_count}[1h])}{\text{rate}(\text{sms_c_delivery_queued_count}[1h])} * 100$$
- Target line at 99%
- Thresholds:
 - Red: < 95
 - Yellow: 95-99
 - Green: >= 99

2. Messages Delivered Within SLA (Stat)

- Query:
$$\frac{\text{count}(\text{sms_c_delivery_time_delta_delta_ms_bucket}\{\text{le}="5000"\})}{\text{count}(\text{sms_c_delivery_time_delta_delta_ms_bucket})}$$
- Shows percentage delivered within 5 seconds
- Unit: percent

3. SLA Violations (Counter)

- Messages exceeding 5 minutes:
$$\text{increase}(\text{sms_c_queue_oldest_message_age_seconds}\{\} > 300)[24h:]$$
- Should be 0

4. Uptime (Stat)

- Query: `up{job="sms-c"}`
- Binary: 1 = up, 0 = down
- Shows current status

5. Daily Success Rate Trend (Graph)

- Query: `avg_over_time((rate(sms_c_delivery_succeeded_count[1h]) / rate(sms_c_delivery_queued_count[1h]))[24h:1h])`
- Time range: Last 30 days
- SLA line at 99%

Alert Rule Examples

Critical Alerts

Routing Failures:

```
alert: RoutingFailuresDetected
expr: increase(sms_c_routing_failed_count[5m]) > 0
for: 2m
labels:
  severity: critical
annotations:
  summary: "{{ $value }}" routing failures in last 5 minutes"
  description: "Messages cannot be routed. Check routing configuration."
```

Queue Backlog:

```
alert: MessageQueueBacklog
expr: sms_c_queue_size_pending > 10000
for: 5m
labels:
  severity: critical
annotations:
  summary: "Message queue has {{ $value }}" pending messages"
  description: "Queue is backing up. Check delivery performance."
```

Old Messages in Queue:

```
alert: OldMessagesInQueue
expr: sms_c_queue_oldest_message_age_seconds > 300
for: 2m
labels:
  severity: critical
annotations:
  summary: "Oldest message is {{ $value }} seconds old"
  description: "Messages not being delivered. Check frontends."
```

All Frontends Disconnected:

```
alert: NoActiveFrontends
expr: sum(sms_c_frontend_status_count{status="connected"}) == 0
for: 1m
labels:
  severity: critical
annotations:
  summary: "No frontends connected"
  description: "No delivery path available. Check frontend connectivity."
```

Dead Letter Queue Growing:

```
alert: DeadLetterMessagesIncreasing
expr: rate(sms_c_delivery_dead_letter_count[10m]) > 0
for: 5m
labels:
  severity: critical
annotations:
  summary: "{{ $value }} messages moved to dead letter queue"
  description: "Messages are becoming undeliverable. Investigate failures."
```

Warning Alerts

Low Delivery Success Rate:

```
alert: LowDeliverySuccessRate
expr: (rate(sms_c_delivery_succeeded_count[10m]) /
rate(sms_c_delivery_queued_count[10m])) < 0.95
for: 10m
labels:
  severity: warning
annotations:
  summary: "Delivery success rate is {{ $value |
humanizePercentage }}"
  description: "Success rate below 95%. Investigate delivery
failures."
```

High Retry Rate:

```
alert: HighDeliveryRetryRate
expr: histogram_quantile(0.95,
sms_c_delivery_succeeded_attempt_count_bucket) > 2
for: 15m
labels:
  severity: warning
annotations:
  summary: "95th percentile delivery attempts: {{ $value }}"
  description: "Messages requiring multiple attempts. Check
destination reliability."
```

Slow Message Processing:

```
alert: SlowMessageProcessing
expr: histogram_quantile(0.95,
sms_c_message_processing_stop_duration_bucket) > 1000
for: 10m
labels:
  severity: warning
annotations:
  summary: "95th percentile processing time: {{ $value }}ms"
  description: "Message processing is slow. Check system
resources."
```

ENUM Lookups Failing:

```
alert: HighEnumFailureRate
expr: rate(sms_c_enum_lookup_stop_duration_count{success="false"}
[10m]) > 0.1
for: 10m
labels:
  severity: warning
annotations:
  summary: "ENUM lookup failure rate: {{ $value }}"
  description: "DNS lookups failing. Check DNS servers."
```

Low ENUM Cache Hit Rate:

```
alert: LowEnumCacheHitRate
expr: rate(sms_c_enum_cache_hit_count[10m]) /
(rate(sms_c_enum_cache_hit_count[10m]) +
rate(sms_c_enum_cache_miss_count[10m])) < 0.70
for: 30m
labels:
  severity: warning
annotations:
  summary: "ENUM cache hit rate: {{ $value | humanizePercentage
}}"
  description: "Low cache efficiency. May indicate unique number
traffic."
```

Charging Failures:

```
alert: ChargingFailuresDetected
expr: rate(sms_c_charging_failed_count[10m]) > 0.05
for: 10m
labels:
  severity: warning
annotations:
  summary: "Charging failure rate: {{ $value }}"
  description: "Charging system errors. Check OCS connectivity."
```

Additional Notes

- All duration metrics use nanosecond precision internally but are converted to milliseconds for reporting
- Counter metrics are cumulative and should be used with `rate()` or `increase()` functions in Prometheus queries
- Gauge metrics represent instantaneous values at collection time
- Histogram metrics provide percentile calculations (p50, p95, p99) and can be used to create heatmaps
- All metrics include default labels added by Prometheus (instance, job, etc.)
- When creating dashboards, use appropriate time ranges: 5m for real-time, 1h for trends, 24h+ for capacity planning
- Set up recording rules in Prometheus for frequently-used complex queries to improve dashboard performance
- Use variable templating in Grafana for dynamic dashboards (select `dest_smsc`, `source_smsc`, etc.)

MMS Centre (MMSc)

[← Back to Documentation Index](#) | [Configuration Reference](#) | [API Reference](#)

Overview

OmniMessage includes an **MMS Centre (MMSc)** that handles the receipt, storage, and delivery notification of multimedia messages. The MMSc supports the two primary MMS inter-working interfaces defined by 3GPP TS 23.140: **MM1** for direct handset communication and **MM4** for inter-carrier relay.

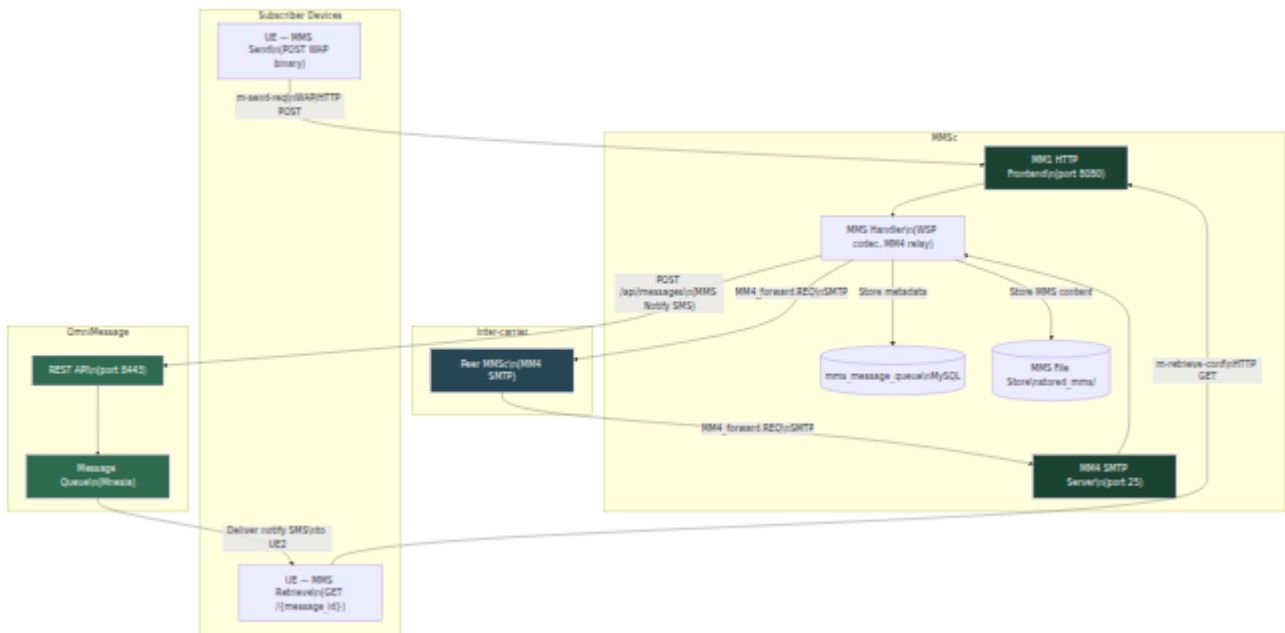
When a subscriber sends or receives an MMS, the MMSc stores the message content locally and injects an **MMS notification SMS** into the OmniMessage message queue via the standard REST API. The receiving handset retrieves the MMS by making an HTTP GET directly to the MMSc.

Relevant Specifications

Specification	Title
3GPP TS 23.140	Multimedia Messaging Service (MMS) — functional description
3GPP TS 26.140	MMS — media formats and codecs
OMA MMS Encapsulation Protocol	MM1 WAP binary encoding specification

Architecture

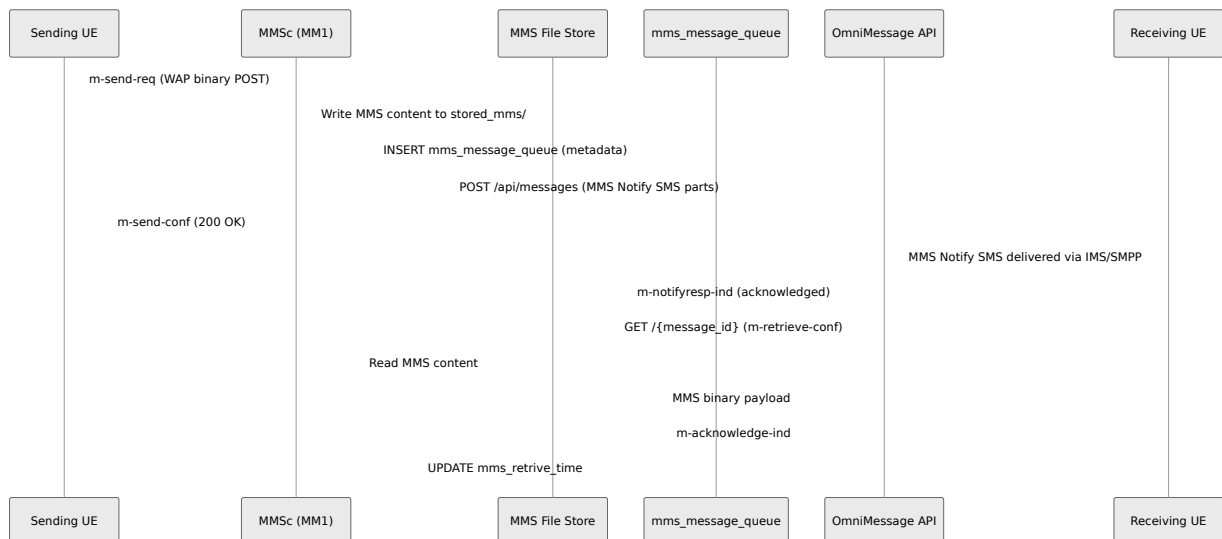
The MMSc sits alongside OmniMessage and interfaces with it exclusively through the REST API. No direct database access to the message store occurs.



MMS Delivery Flow

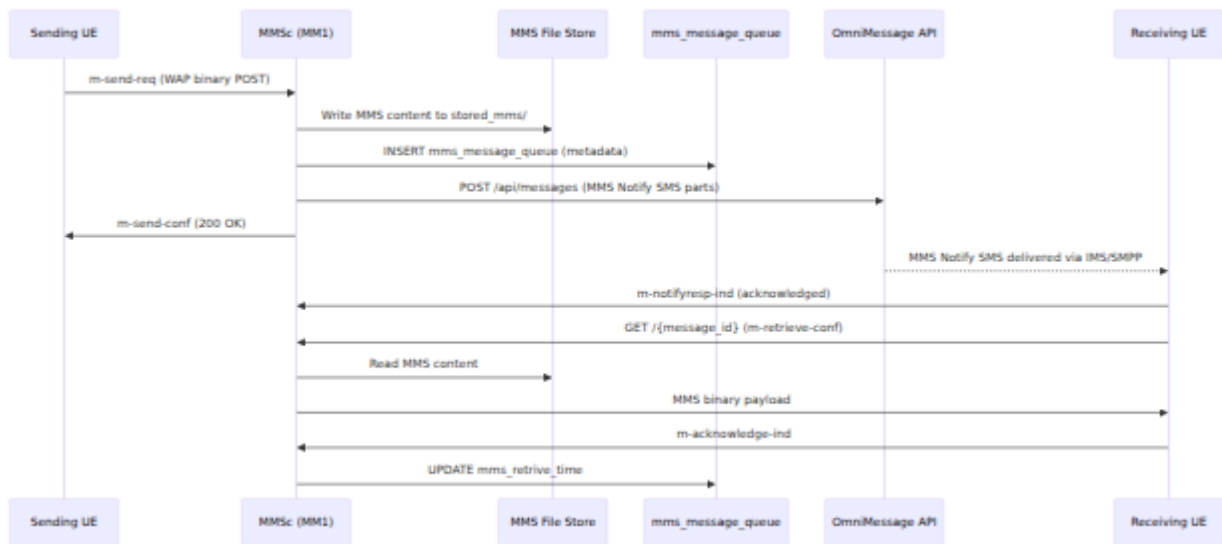
On-Net Delivery (MM1)

A subscriber sends an MMS to another subscriber on the same network.



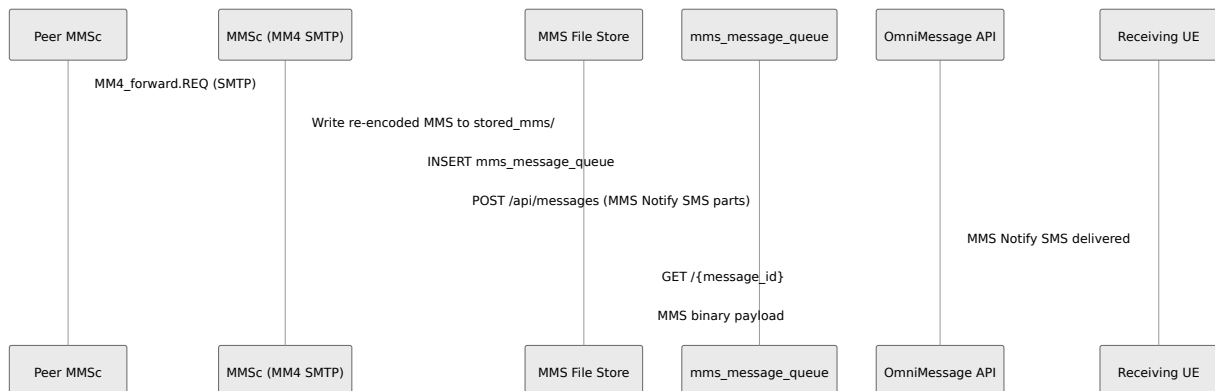
Off-Net Delivery (MM4)

A subscriber sends an MMS to a subscriber on a different carrier.



Inbound MM4 (from Peer Carrier)

A subscriber on another network sends an MMS to an on-net subscriber.



MMS Notification SMS

The MMS Notify SMS is a binary SMS (DCS `0x44`, 8-bit data) containing a **WSP-encoded** `m-notification-ind` PDU. It is split into 64-byte segments, each wrapped with a concatenated SMS UDH, and submitted to OmniMessage as individual messages via `POST /api/messages`.

The notification payload encodes the MMS content location as an HTTP URL pointing to the MMSc:

```
http://<mmsc.mms_domain>/<mms_message_id>.00
```

The receiving handset uses this URL to retrieve the MMS directly from the MMSc over MM1.

On/Off-Net Determination

The MMSc uses **ENUM DNS (NAPTR lookups)** to determine whether a destination subscriber is on-net or off-net before deciding whether to send an MMS Notify SMS or relay via MM4.

Result	Action
NAPTR record matches a configured <code>enum_routing</code> entry with <code>on_net: true</code>	MMS stored locally; notify SMS injected into OmniMessage queue
NAPTR record matches an entry with <code>on_net: false</code>	MMS stored locally; forwarded to peer via MM4 SMTP relay
No NAPTR record or ENUM timeout	Treated as off-net; MM4 relay attempted

Configuration

The MMSc is configured via `MMSc/config.yaml`.

Configuration Example

```
log_prefix: 'MMSc'

smcsc_api:
  url: 'https://10.4.10.215:8443'

mmsc:
  host: 0.0.0.0
  port: 8080
  location: 'mmsc01.mnc001.mcc001.3gppnetwork.org'
  mms_domain: "mms.example.com"
  remote_mmsc_ip:
    - 10.101.2.150
  mm4_peers:
    sinch:
      - mm4_destination_host: mm4.sinch.com
        mm4_destination_port: 25
        mms_domain: mms1.mmsnni.carrier.net

database:
  mysql:
    server: 127.0.0.1
    username: mmsc
    password: changeme
    database: mmsc
    ping_before_query: true

prometheus:
  listen_port: 8083

number_routing:
  enum_domain: e164.arpa
  enum_timeout_seconds: 1
  enum_servers:
    - 10.4.10.80
  enum_routing:
    - domain: 'mobile.example.local'
      operator: 'ExampleMNO'
      on_net: true
      next_hop: 'internal'
  destination_rewriting:
    - regex_match: "^1(\\d{10}$)"
```

```
regex_replace: "\1"  
description: Strip leading 1 from 11-digit NANP number  
default_route_mm4: 'sinch'
```

`smc_api` Parameters

Parameter	Type	Required	Default	Description
<code>url</code>	String	Yes	—	Base URL of the OmniMessage REST API. Used to submit MMS Notify SMS messages.
<code>api_key</code>	String	No	—	Bearer token for API authentication, if required. Omit if the API has no authentication configured.
<code>hss_url</code>	String	No	—	Base URL of an HSS/PCRF API for resolving client IP addresses to MSISDNs on the MM1 interface. When omitted, IP-to-MSISDN resolution is unavailable and requests from non-localhost IPs will be rejected.

mmsc Parameters

Parameter	Type	Required	Default	Description
host	String	No	0.0.0.0	Interface address to bind the MM1 HTTP server.
port	Integer	No	8080	TCP port for the MM1 HTTP server.
location	String	Yes	—	Node identity string for this MMSc, used as the source_mmsc recorded against delivered messages.
mms_domain	String	Yes	—	Domain name of this MMSc. Encoded into the MMS content location URL sent in Notify SMS messages (e.g. mms.example.com).
remote_mmsc_ip	List	No	[]	IP addresses of peer MMScs. Requests from these IPs on the retrieval endpoint are not recursively forwarded, preventing loops. Also used as fallback sources on a local cache-miss.

Parameter	Type	Required	Default	Description
<code>mm4_peers</code>	Map	No	—	Named groups of MM4 relay targets. Each key is a peer group name; each entry is a list of SMTP endpoint definitions. See MM4 Peer Parameters below.

MM4 Peer Parameters

Each entry under `mmsc.mm4_peers.<name>` is a list. When multiple entries are present, one is selected at random per relay attempt.

Parameter	Type	Required	Description
<code>mm4_destination_host</code>	String	Yes	Hostname or IP of the peer carrier's MM4 SMTP server.
<code>mm4_destination_port</code>	Integer	Yes	SMTP port on the peer MMSc. Typically 25.
<code>mms_domain</code>	String	Yes	MMS domain of the peer carrier. Used in SMTP <code>To</code> headers and message addressing.

`database.mysql` Parameters

Parameter	Type	Required	Default	Description
<code>server</code>	String	Yes	—	MySQL server hostname or IP.
<code>username</code>	String	Yes	—	Database username.
<code>password</code>	String	Yes	—	Database password.
<code>database</code>	String	Yes	—	Database name. Must match the schema created from <code>MMSc/schema.sql</code> .
<code>ping_before_query</code>	Boolean	No	<code>false</code>	Reconnect to MySQL before each query. Recommended for long-running processes to recover from server-side connection timeouts.

`number_routing` Parameters

Parameter	Type	Required	Default	Description
<code>enum_domain</code>	String	Yes	—	Root DNS zone for ENUM NAPTR lookups (e.g. <code>e164.arpa</code>).
<code>enum_timeout_seconds</code>	Integer	No	<code>1</code>	DNS query timeout in seconds. Set to <code>0</code> to disable ENUM and treat all destinations as off-net.
<code>enum_servers</code>	List	No	System DNS	DNS resolver addresses to use for ENUM queries.
<code>enum_routing</code>	List	No	—	Ordered list of NAPTR domain-to-route mappings. See ENUM Routing Parameters below.
<code>destination_rewriting</code>	List	No	—	Ordered list of regex rewrite rules applied to destination MSISDNs before ENUM lookup and MM4 relay.

Parameter	Type	Required	Default	Description
				First matching rule wins.
<code>default_route_mm4</code>	String	Yes	—	Key of the MM4 peer group (from <code>mmsc.mm4_peers</code>) to use when no other route applies.

ENUM Routing Parameters

Parameter	Type	Description
<code>domain</code>	String	NAPTR replacement domain string to match against (substring match).
<code>operator</code>	String	Human-readable operator name for logging.
<code>on_net</code>	Boolean	<code>true</code> — subscriber is on-net; inject notify SMS. <code>false</code> — relay via MM4.
<code>next_hop</code>	String	Optional. Informational label recorded in logs; not used for routing.

`prometheus` Parameters

Parameter	Type	Required	Default	Description
<code>listen_port</code>	Integer	No	<code>8083</code>	Port on which Prometheus metrics are exposed. Choose a port that does not conflict with the OmniMessage core metrics port (default 9568).

Database Schema

The MMSc maintains its own MySQL database for MMS metadata and delivery state. The schema is defined in `MMSc/schema.sql` and consists of a single table.

mms_message_queue

Column	Type	Description
id	INT UNSIGNED	Auto-increment primary key.
unique_id	VARCHAR(64)	MMSc-assigned message identifier. Also used as the retrieval URL path segment.
send_time	DATETIME	Timestamp when the MMS was received by this MMSc.
expires	DATETIME	Message expiry time, after which it will not be delivered.
source_msisdn	VARCHAR(32)	Sending subscriber's MSISDN.
destination_msisdn_unique	VARCHAR(32)	Primary recipient MSISDN.
mms_destination_msisdn_list	TEXT	Comma-separated list of all recipient MSISDNs (for group MMS).
mms_subject	VARCHAR(255)	MMS subject line.
mms_file_path	VARCHAR(512)	Path to the stored MMS content file (stored_mms/<id>.mm)

Column	Type	Description
<code>source_mmsc</code>	VARCHAR(128)	Identity of the MMSc or MM4 peer that originated the message.
<code>dest_mmsc</code>	VARCHAR(128)	Set to <code>MM4_<peer></code> after outbound MM4 relay, to the retrieving MMS location after local delivery.
<code>mms_notify_time</code>	DATETIME	Timestamp when the MMS Notify SMS was submitted to OmniMessage.
<code>mms_retrieve_time</code>	DATETIME	Timestamp when the subscriber retrieved the MMS.

Processes

The MMSc runs as two independent processes.

MM1 HTTP Frontend

Handles WAP MMS submissions (`m-send-req`) and retrievals (`GET /<id>`) from handsets. Run via gunicorn:

```
gunicorn --bind 0.0.0.0:8080 MMSc:app
```

The MM1 frontend should sit behind a reverse proxy (nginx, HAProxy) for TLS termination. Handsets typically connect to the MMSc APN using the carrier's MMS proxy settings.

MM4 SMTP Server

Listens for inbound `MM4_forward.REQ` messages from peer carrier MMScs. Run directly:

```
python smtp_server.py
```

Port 25 requires either root privileges or `CAP_NET_BIND_SERVICE`. Outbound MM4 relay uses Python's built-in `smtpplib` and does not require a separate MTA.

Metrics

The MMSc exposes Prometheus metrics on the port configured under `prometheus.listen_port`.

MMS Operations

Metric: `prom_mms_operations_count_total` **Type:** Counter **Description:** Count of MMS operations processed. **Labels:**

- `method` — Operation type: `HTTP_POST`, `m-send-req`, `m-notifyresp-ind`, `m-acknowledge-ind`, `InsertMMS`, `GetMMS`, `UpdateMMS`, `MMS_Get`, `MMS_Get_Fail`, `SMTP_Start`, `SMTP_Complete`

Example queries:

```
# MMS submit rate
rate(prom_mms_operations_count_total{method="m-send-req"}[5m])

# MM4 inbound completion rate
rate(prom_mms_operations_count_total{method="SMTP_Complete"}[5m])
```

API Response Times

Metric: `prom_mms_api_response_time` **Type:** Histogram **Description:** MM1 HTTP handler latency in seconds. **Labels:**

- `result` — HTTP status code: 200, 404, 500
- `function` — Handler: m-send-req, m-notifyresp-ind, m-acknowledge-ind, GET MMS, GET MMS Remote, unknown

Example queries:

```
# 95th percentile retrieval latency
histogram_quantile(0.95,
  rate(prom_mms_api_response_time_bucket{function="GET MMS"}[5m]))
```

MM4 Relay Times

Metric: `prom_mm4_response_time` **Type:** Histogram **Description:** MM4 SMTP relay latency in seconds. **Labels:**

- `result` — 200 (success) or 000 (failure)
- `direction` — in (inbound from peer) or out (outbound to peer)
- `peer` — MM4 peer group name

Example queries:

```
# MM4 outbound error rate
sum(rate(prom_mm4_response_time_count{direction="out",result="000"}[5m]))
  / sum(rate(prom_mm4_response_time_count{direction="out"}[5m]))
```

DNS Response Times

Metric: `prom_dns_response_time` **Type:** Histogram **Description:** ENUM DNS query latency in seconds. A spike here will delay MMS delivery decisions.

SMS-C Number Translation Guide

[← Back to Documentation Index](#)

Overview

The SMS-C Number Translation system provides flexible, regex-based transformation of phone numbers before routing. Translation rules can normalize numbers, add international prefixes, format numbers for specific gateways, and chain multiple transformations together. Rules are stored in Mnesia for persistence and can be modified at runtime without service interruption.

Key Features

- **Prefix-based matching:** Match numbers by prefix before applying transformations
- **Regex-based transformation:** Powerful pattern matching and replacement with capture groups
- **Source SMSC filtering:** Apply different translations based on message origin
- **Priority-based evaluation:** Control rule order with configurable priorities (1-255)
- **Rule chaining:** Continue processing through multiple rules with loop prevention
- **Separate calling/called transforms:** Independent transformation for originating and destination numbers
- **Configuration file loading:** Load initial rules from `runtime.exs` on first startup
- **Runtime configuration:** Add, modify, or disable rules without restarting
- **Web UI:** Full CRUD interface for rule management

- **Simulation tool:** Test translation logic with step-by-step evaluation
- **Backup/Restore:** Export and import translation configurations
- **Pre-routing integration:** Translations applied before routing for consistent number formats

Architecture

Data Model

Each translation rule contains the following fields:

Field	Type	Description	Required
<code>rule_id</code>	integer	Auto-incrementing unique identifier	Yes (auto)
<code>calling_prefix</code>	string/nil	Prefix match for calling number (nil = wildcard)	No
<code>called_prefix</code>	string/nil	Prefix match for called number (nil = wildcard)	No
<code>source_smsc</code>	string/nil	Source SMSC name (nil = wildcard)	No
<code>calling_match</code>	string/nil	Regex pattern to match calling number. Also acts as the gate for the source TON/NPI override	No
<code>calling_replace</code>	string/nil	Replacement pattern for calling number	No
<code>called_match</code>	string/nil	Regex pattern to match called number	No
<code>called_replace</code>	string/nil	Replacement pattern for called number	No
<code>source_ton</code>	string/nil	Force the source address Type of Number. Blank/absent = leave inferred	No
<code>source_npi</code>	string/nil	Force the source address Numbering Plan Indicator. Blank/absent = leave inferred	No

Field	Type	Description	Required
priority	integer	Rule priority (1-255, lower = higher priority)	Yes
description	string	Human-readable description	No
enabled	boolean	Enable/disable rule	Yes
continue	boolean	Continue evaluating rules after match (default: false)	No

Note: Rules are evaluated in priority order (lowest number first). Only enabled rules are evaluated.

See [Forcing Type of Number \(TON\) and Numbering Plan Indicator \(NPI\)](#) for full documentation of the `source_ton` and `source_npi` fields.

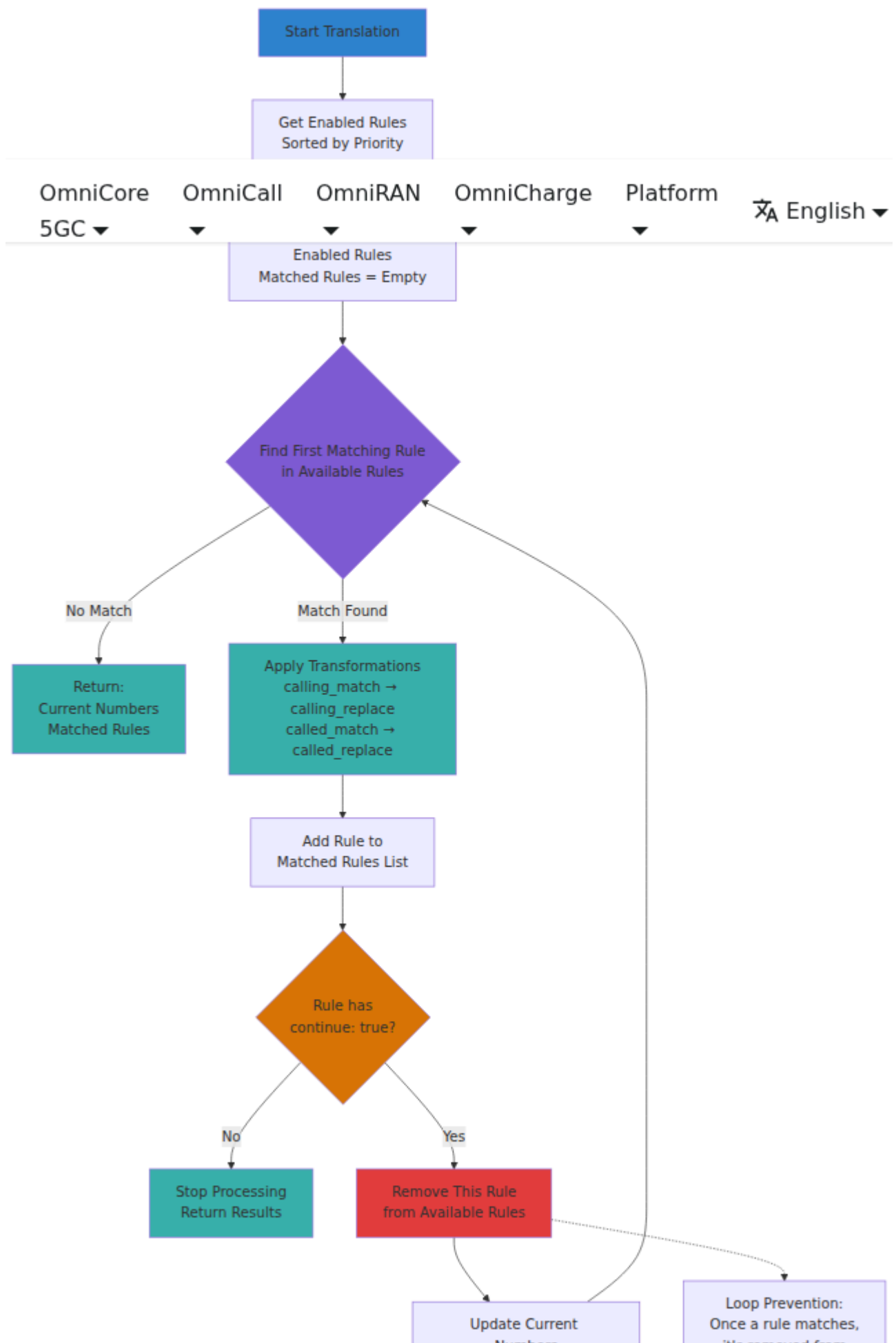
Translation Algorithm

When translating numbers, the system:

1. **Retrieves enabled rules** sorted by priority (lowest first)
2. **Evaluates rules sequentially** against message parameters:
 - Match `calling_prefix` (if specified)
 - Match `called_prefix` (if specified)
 - Match `source_smsc` (if specified)
3. **Applies first matching rule:**
 - Transform calling number using `calling_match` and `calling_replace`
 - Transform called number using `called_match` and `called_replace`
4. **Checks continue flag:**
 - If `continue: false` → Stop processing, return result
 - If `continue: true` → Remove matched rule from available rules, continue with step 2 using **transformed numbers**
5. **Returns final numbers** and list of all applied rules

Rule Chaining with Loop Prevention

The `continue` flag enables powerful rule chaining while preventing infinite loops:



numbers
to Transformed Values

it's removed from
Available Rules
Can't match again

Wildcards

- `nil` or empty values act as wildcards that match any value
- A rule with no matching criteria is a catch-all rule
- A rule with no transformation patterns (`nil` match/replace) passes numbers through unchanged

Example: Rule Chaining Scenario

Parse error on line 20: ...] style R1 fill:#38B2AC style R -----^
Expecting 'SOLID_OPEN_ARROW', 'DOTTED_OPEN_ARROW', 'SOLID_ARROW',
'BIDIRECTIONAL_SOLID_ARROW', 'DOTTED_ARROW',
'BIDIRECTIONAL_DOTTED_ARROW', 'SOLID_CROSS', 'DOTTED_CROSS',
'SOLID_POINT', 'DOTTED_POINT', got 'TXT'

Try again

Forcing Type of Number (TON) and Numbering Plan Indicator (NPI)

In addition to regex-based number rewriting, a translation rule can **force** the Type of Number (TON) and Numbering Plan Indicator (NPI) of the source (originating) address. This is useful when a sender's number string alone does not convey the correct address semantics — for example a short code that must be signalled as `unknown/unknown` rather than treated as a national or international number.

These overrides are independent of number rewriting: a rule can force TON/NPI **without** changing the number at all.

The Override Fields

Field	Applies To	Effect
<code>source_ton</code>	Source (originating) address	Forces the Type of Number on the built SMS-DELIVER originating address (the TPDU)
<code>source_npi</code>	Source (originating) address	Forces the Numbering Plan Indicator on the built SMS-DELIVER originating address (the TPDU)

Blank or absent = no override. When a field is left blank (or the rule is absent entirely), the address value is **inferred** instead — see [Inference Fallback](#) below.

Note: `source_ton`/`source_npi` are baked directly into the originating address encoded in the **TPDU** (the SMS-DELIVER that the IMS/SS7 frontend transmits). The value lives in the encoded TPDU — nothing extra is stored on the message record. There is no destination TON/NPI override: the SMS-DELIVER TPDU carries no destination address, and frontends derive destination addressing themselves.

Allowed Values

Type of Number (TON):

Value	Meaning
unknown	Type of number unknown
international	International number (full E.164 with country code)
national	National (significant) number
network_specific	Network-specific number
subscriber	Subscriber number
alphanumeric	Alphanumeric address (e.g. a brand name sender ID)
abbreviated	Abbreviated number

Numbering Plan Indicator (NPI):

Value	Meaning
unknown	Numbering plan unknown
isdn	ISDN/telephony numbering plan (E.164)
data	Data numbering plan (X.121)
telex	Telex numbering plan
land_mobile	Land mobile numbering plan (E.212)
national	National numbering plan
private	Private numbering plan
ermes	ERMES numbering plan

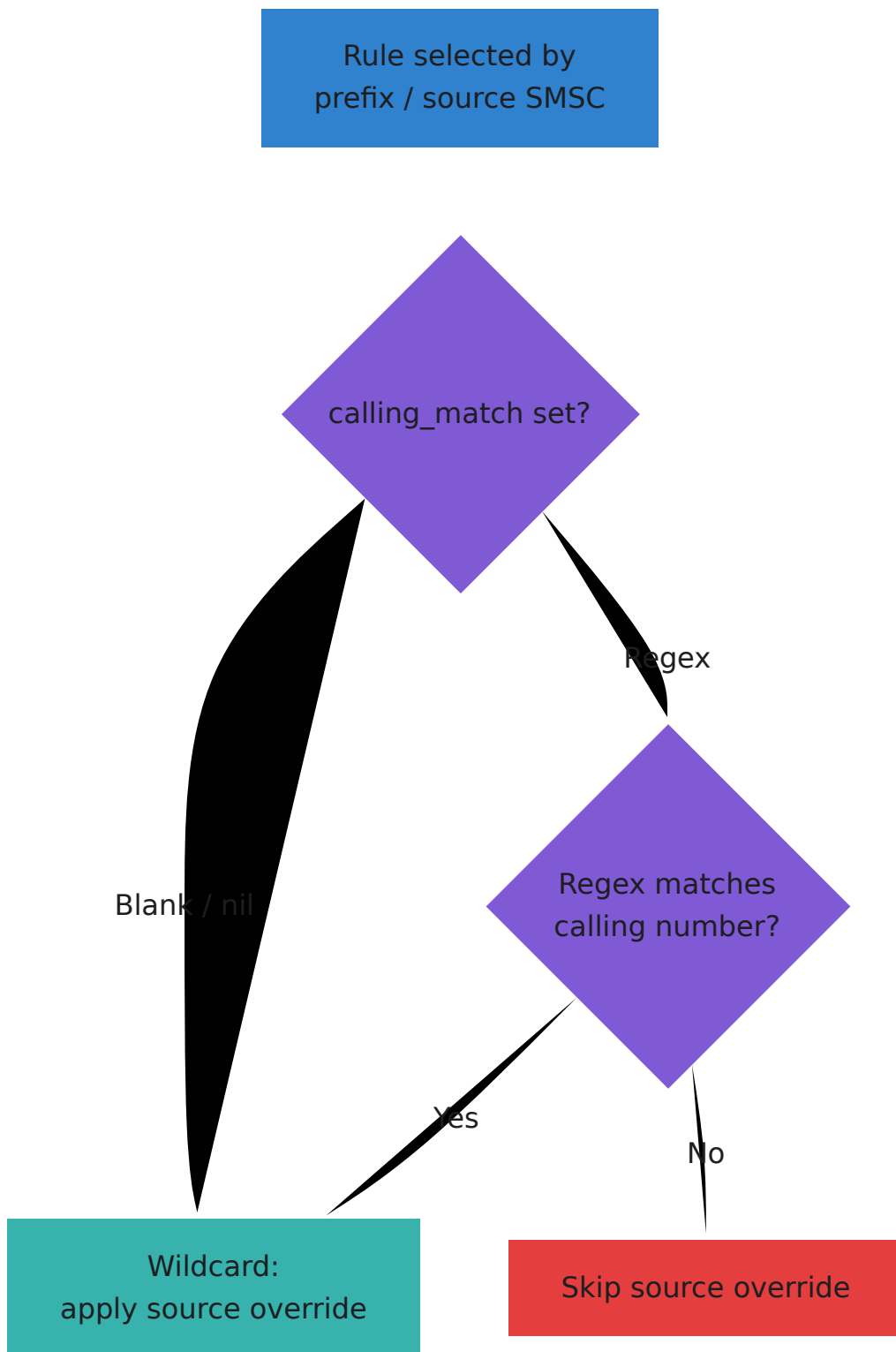
Any value outside these sets is rejected when the rule is created or updated.

Match Gating: Scoping an Override to a Sender Pattern

The override is **gated** by the rule's **Calling Number Match** (`calling_match`) regex, which lets a single rule target a subset of senders by pattern:

- The source override (`source_ton/source_npi`) is applied only when the rule's `calling_match` regex matches the calling number.
- A **blank match is a wildcard** — the override applies whenever the rule is otherwise selected (by prefix and/or source SMSC).

A rule may carry a `calling_match` with **no** corresponding `calling_replace`. In that configuration the match acts purely as a **gate** for the override and does **not** rewrite the number. This is the recommended way to apply a TON/NPI override to only those numbers matching a pattern.



When `continue: true` chains multiple rules, TON/NPI overrides accumulate with **last-write-wins** semantics: a later rule that sets a non-blank value overwrites an earlier one; a blank value never clears a previously-set override.

Inference Fallback

When **no** rule forces an override, the source address values are **inferred** from the number string at TPDU-build time:

Input	Inferred TON	Inferred NPI
Numeric, ≥ 5 digits	international	isdn
Numeric, < 5 digits	national	isdn
Non-numeric (alphanumeric sender, e.g. Acme)	alphanumeric	unknown

A forced override always takes precedence over the inferred value. Forcing only the TON (leaving NPI blank) overrides the TON while leaving the NPI inferred, and vice versa.

Setting TON/NPI Overrides

These fields are settable through three interfaces:

- **Web UI** — the Number Translation rule editor exposes Source TON and Source NPI dropdowns. Leaving a dropdown unset means "no override".
- **REST API** — the `/api/translations` endpoints accept `source_ton` and `source_npi` as string values (see [Examples](#) below).
- **Configuration** — the `config :sms_c, :translation_rules` list accepts the same fields as string values, loaded on first startup.

Worked Example: Short-Code Senders

A sender that is a 3–6 digit short code should be signalled as TON/NPI `unknown/unknown` **without** rewriting the number. The rule uses `calling_match` purely as a gate (no `calling_replace`):

Configuration (`config/runtime.exs`):

```

config :sms_c, :translation_rules, [
  %{
    calling_match: "^\\d{3,6}$",
    calling_replace: nil,
    source_ton: "unknown",
    source_npi: "unknown",
    priority: 5,
    continue: true,
    description: "Short codes (3-6 digits) => unknown/unknown
TON/NPI",
    enabled: true
  }
]

```

How it works: The rule is evaluated for every message (no `calling_prefix`, `called_prefix`, or `source_smsc` restricts it). The `calling_match` regex `^\\d{3,6}$` gates the source override, so `source_ton/source_npi` are forced to `unknown/unknown` **only** when the calling number is a bare 3-to-6-digit string. Because `calling_replace` is `nil`, the number itself is left unchanged. `continue: true` allows later rules to run as well.

Use case: Application-originated traffic from short codes that would otherwise be mis-inferred as `national` (4-digit) or `international` (5-6-digit) numbers.

Equivalent REST API request — `POST /api/translations:`

```

{
  "calling_match": "^\\d{3,6}$",
  "calling_replace": null,
  "source_ton": "unknown",
  "source_npi": "unknown",
  "priority": 5,
  "continue": true,
  "description": "optimERA short codes (3-6 digits) =>
unknown/unknown TON/NPI",
  "enabled": true
}

```

The API returns the created rule (including its assigned `rule_id`) with the TON/NPI values serialised as strings.

Worked Example: Force National for a Known Range

Force the source address of a known local number range to `national/isdn`, without rewriting the number:

Configuration (`config/runtime.exs`):

```
%{
  calling_prefix: "07",
  calling_match: "^07\d{8}$",
  calling_replace: nil,
  source_ton: "national",
  source_npi: "isdn",
  priority: 20,
  continue: false,
  description: "Local 07xxxxxxxx numbers => national/isdn",
  enabled: true
}
```

How it works: `calling_prefix: "07"` narrows the rule to numbers starting with `07`, and the `calling_match` regex gates the override to exactly a `07` followed by 8 digits. When matched, the source address is signalled as `national/isdn` rather than the inferred `international/isdn` (which would apply because the number has ≥ 5 digits). The number is not rewritten.

Use case: A gateway or peer that requires nationally-formatted local numbers to be flagged as `national` TON even though they are stored in their raw dialled form.

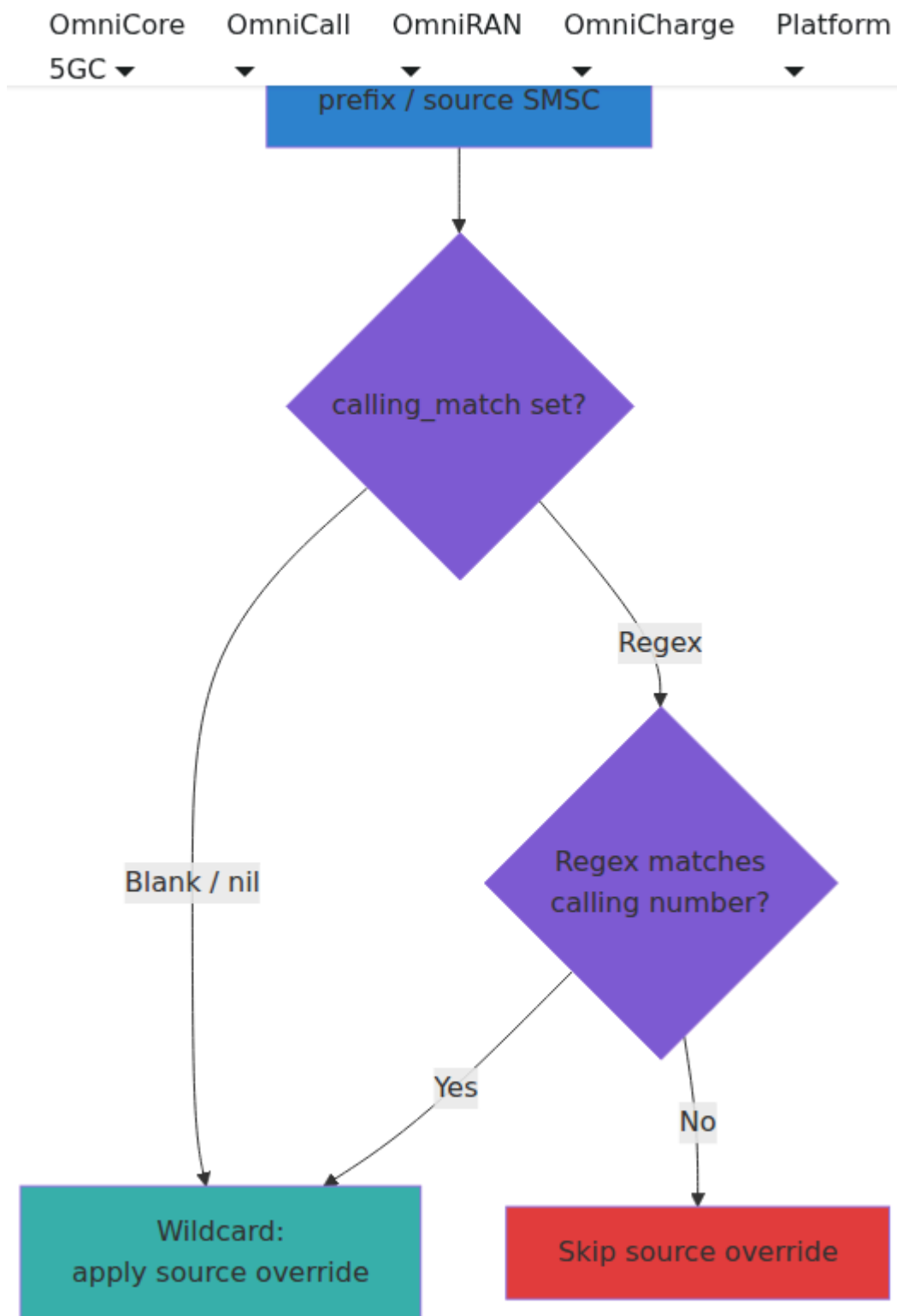
Configuration

Loading Rules from Configuration File

Translation rules can be defined in `config/runtime.exs` and will be automatically loaded on first startup.

Important: Rules from configuration are only loaded when the translation table is **empty** (first startup). This preserves rules added via the Web UI during runtime and prevents duplicates on restarts.

Configuration Loading Flow



Example Configuration

```
# config/runtime.exs
config :sms_c, :translation_rules, [
  # Add +1 to 10-digit US numbers
  %{
    calling_prefix: nil,
    called_prefix: nil,
    source_smsc: "us_domestic_smsc",
    calling_match: "^(\\d{10})$",
    calling_replace: "+1\\1",
    called_match: "^(\\d{10})$",
    called_replace: "+1\\1",
    priority: 10,
    description: "Add +1 to 10-digit US numbers from domestic
SMSC",
    enabled: true,
    continue: false
  },

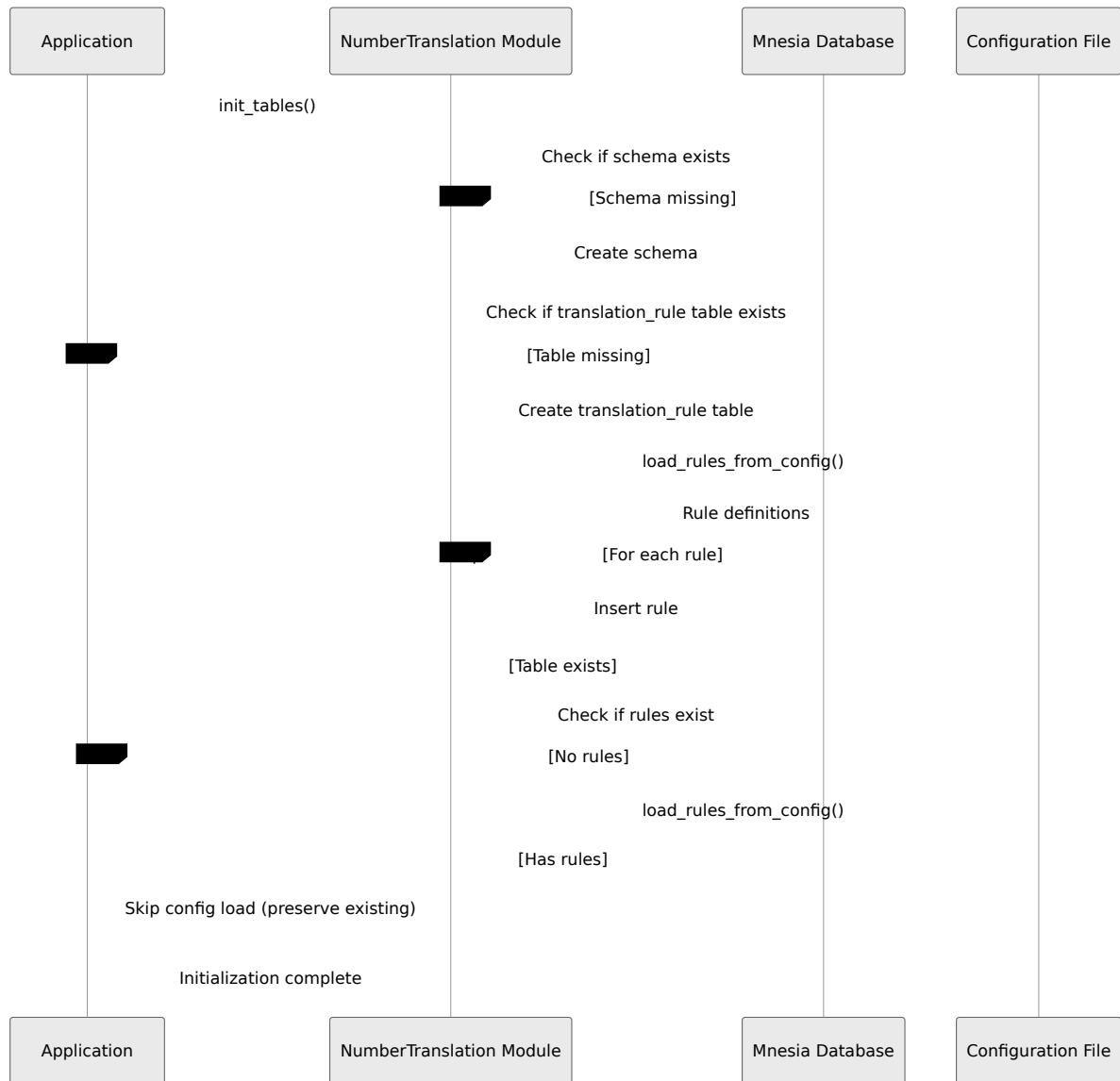
  # Strip leading zeros from international format
  %{
    calling_prefix: "00",
    called_prefix: nil,
    source_smsc: nil,
    calling_match: "^00(.+)$",
    calling_replace: "+\\1",
    called_match: nil,
    called_replace: nil,
    priority: 5,
    description: "Convert 00 international prefix to +",
    enabled: true,
    continue: true # Continue to apply more formatting
  },

  # Force short-code senders to unknown/unknown TON/NPI (gate
only, no rewrite)
  %{
    calling_match: "^\\d{3,6}$",
    calling_replace: nil,
    source_ton: "unknown",
    source_npi: "unknown",
    priority: 6,
    description: "Short codes (3-6 digits) => unknown/unknown
TON/NPI",
```

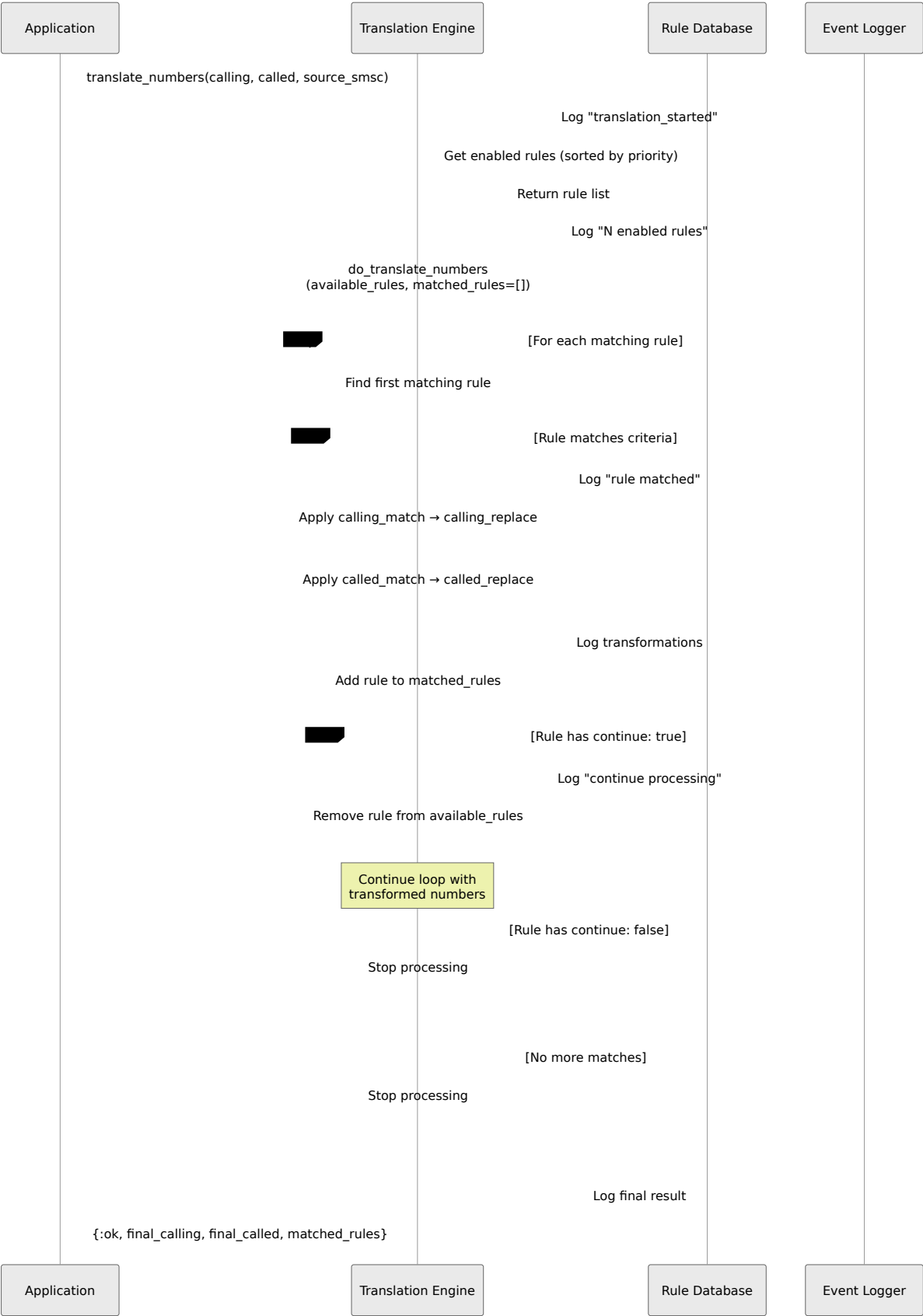
```
    enabled: true,  
    continue: true  
  },  
  
  # Format UK numbers for specific gateway  
  %{  
    calling_prefix: "+44",  
    called_prefix: "+44",  
    source_smsc: nil,  
    calling_match: "^\\+44(.*)$",  
    calling_replace: "0044\\1",  
    called_match: "^\\+44(.*)$",  
    called_replace: "0044\\1",  
    priority: 20,  
    description: "Format UK numbers for legacy gateway",  
    enabled: true,  
    continue: false  
  }  
]  
]
```


Getting Started

Initialization Flow



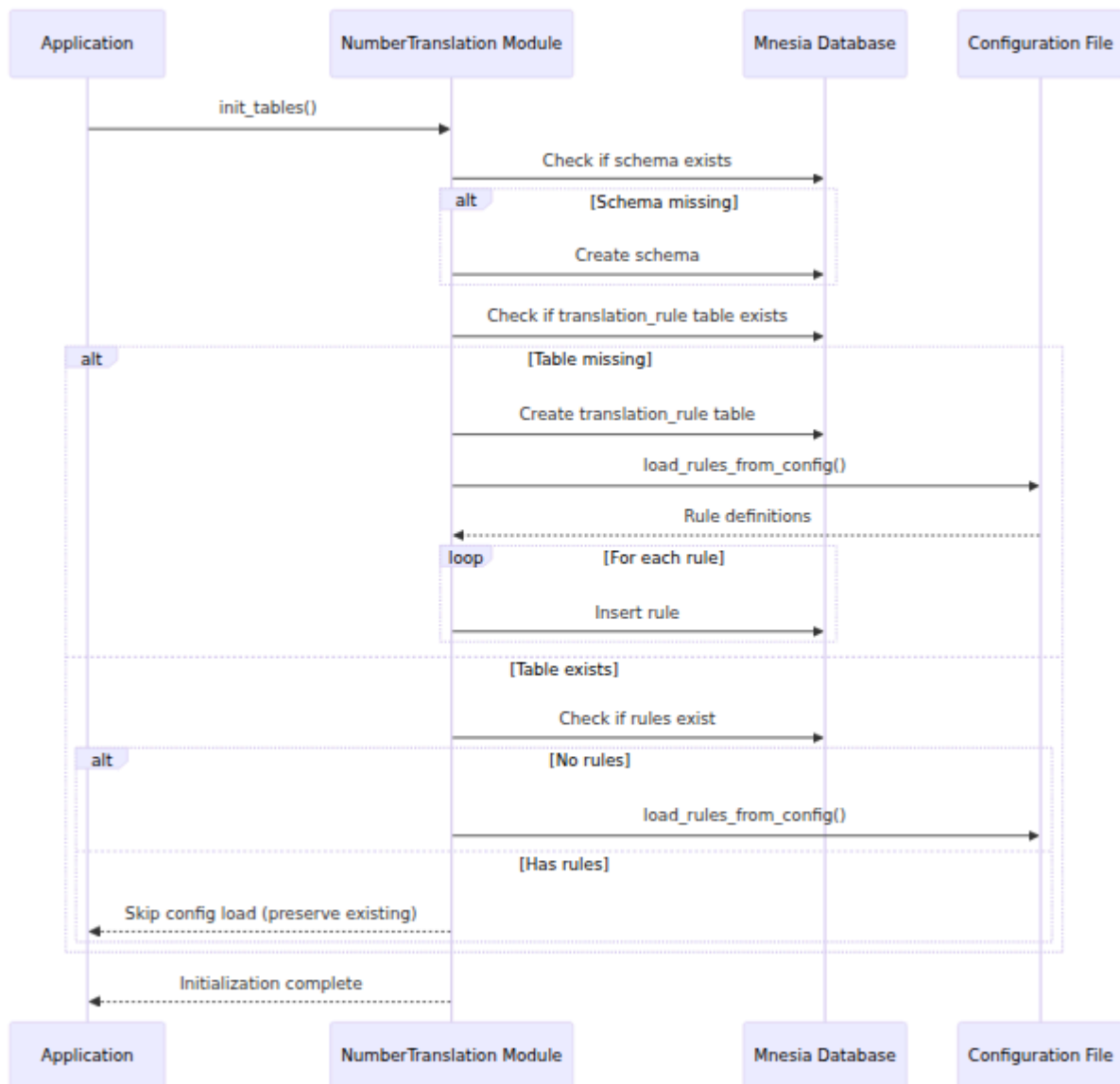
Message Translation Flow



Common Use Cases

International Number Normalization

Normalize various international formats to E.164:



Gateway-Specific Formatting

Chain rules to format numbers for specific gateway requirements:

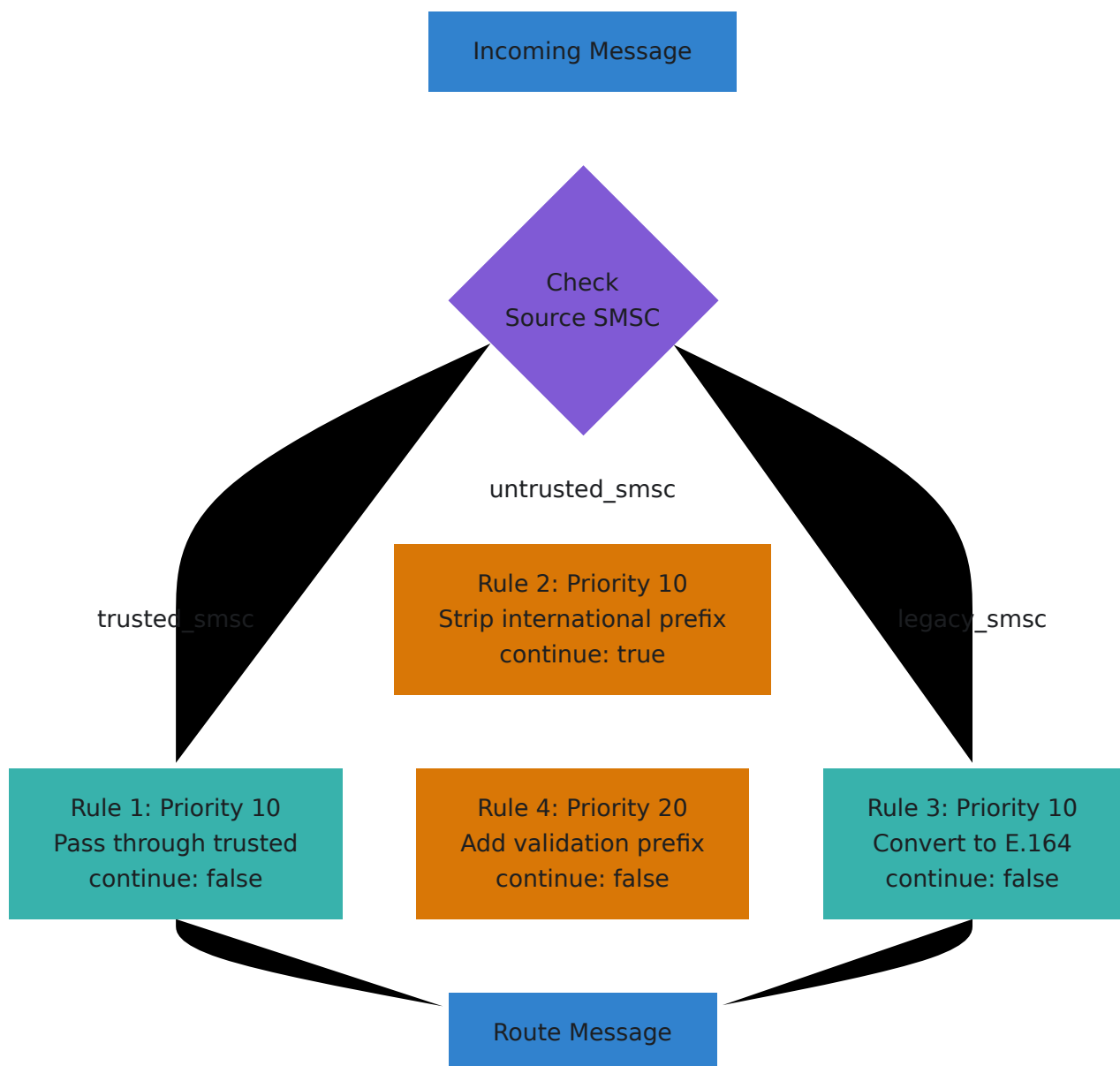
Parse error on line 2: ...rt TD I[Input: "5551234567"] --> S1[-----^
Expecting 'SQE', 'DOUBLECIRCLEEND', 'PE', '-)', 'STADIUMEND',

'SUBROUTINEEND', 'PIPE', 'CYLINDEREND', 'DIAMOND_STOP', 'TAGEND',
'TRAPEND', 'INVTRAPEND', 'UNICODE_TEXT', 'TEXT', 'TAGSTART', got 'STR'

Try again

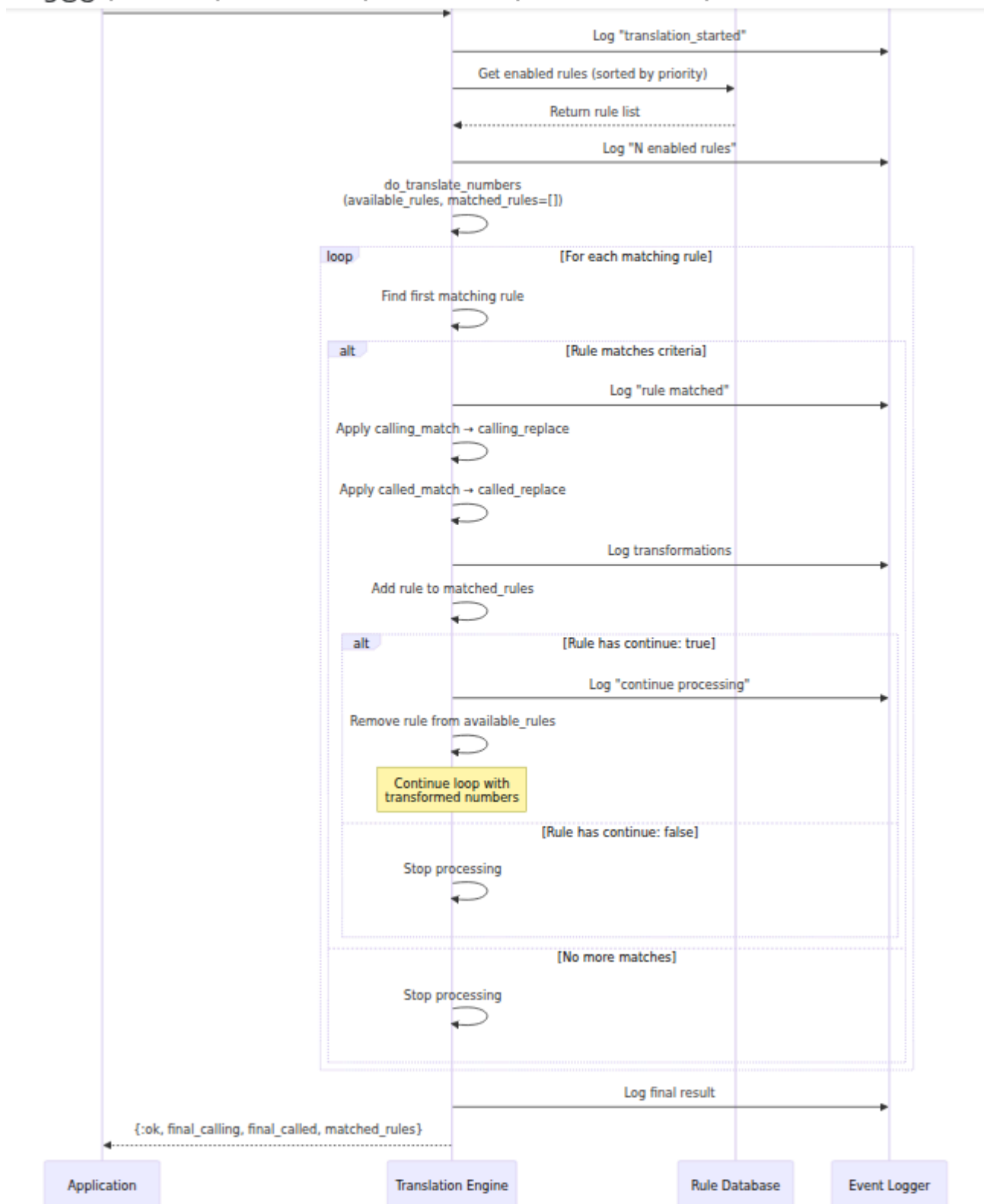
SMSC-Specific Translations

Apply different translations based on message source:



Prefix-Based Routing Preparation

Normalize numbers before routing to ensure consistent prefix matching:



Number Portability Handling

Handle ported numbers that require prefix changes:

Parse error on line 18: ... style Input fill:#3182CE style R -----^
Expecting 'SOLID_OPEN_ARROW', 'DOTTED_OPEN_ARROW', 'SOLID_ARROW',
'BIDIRECTIONAL_SOLID_ARROW', 'DOTTED_ARROW',
'BIDIRECTIONAL_DOTTED_ARROW', 'SOLID_CROSS', 'DOTTED_CROSS',
'SOLID_POINT', 'DOTTED_POINT', got 'TXT'

Try again

Web Interface

Translation Rule Management UI

Access the rule management interface at `/number_translation` (via navigation menu):

Features:

- View all rules in a sortable table by priority
- Add new rules with form validation
- Edit existing rules
- Enable/disable rules without deleting
- Delete rules with confirmation
- Visual indicator for rules with `continue: true`
- Import/Export rules as JSON

Adding a Rule:

1. Fill in matching criteria (optional):
 - Calling prefix (e.g., "+1", "44")
 - Called prefix (e.g., "+639", "1555")
 - Source SMSC (leave empty for any)
2. Define transformations (optional):
 - Calling number regex match and replace
 - Called number regex match and replace
3. Set the source address override (optional):

- **Source TON / Source NPI** dropdowns — force the originating address Type of Number / Numbering Plan Indicator (baked into the TPDU). Gated by the Calling Number Match regex.
 - Leave a dropdown unset to leave the value inferred. See [Forcing Type of Number \(TON\) and Numbering Plan Indicator \(NPI\)](#).
4. Set priority (1-255, lower = higher priority)
 5. Set status:
 - **Enabled**: Rule is active
 - **Continue Processing**: Continue evaluating more rules after this one
 6. Add description
 7. Click "Add Rule" or "Update Rule"

Continue Processing Toggle:

- **Stop** (default): Stop processing after this rule matches
- **Continue**: Apply this rule and continue evaluating remaining rules
- Rules with continue enabled show a blue "↓ Continue" badge in the table

Editing a Rule:

1. Click "Edit" next to the rule
2. Modify fields as needed
3. Click "Update Rule"

Rule Table Indicators:

- **Enabled/Disabled** badge shows rule status
- **↓ Continue** badge shows rules that will continue processing
- **Priority** badge shows evaluation order
- Regex patterns displayed in monospace font for clarity

Translation Simulator

Access the simulator at `/translation_simulator` (via navigation menu):

Features:

- Test translation logic with actual numbers
- **Step-by-Step Transformation** showing each rule applied
- See before/after values for each transformation
- View which rules matched and why
- Load example scenarios for quick testing
- View test history (last 10 tests)

Using the Simulator:

1. Enter test parameters:
 - Calling number (from)
 - Called number (to)
 - Source SMSC (optional)
2. Click "Test Translation"
3. View comprehensive results:
 - **Translation Result:** Final numbers after all transformations
 - **Rules Applied:** Count and list of all rules that matched
 - **Step-by-Step Transformations:** Detailed view of each rule:
 - Step number and rule information
 - Rule description
 - Before → After for both calling and called numbers
 - "↓ Continue" indicator for rules that continued processing
 - Transformations highlighted in green
 - Unchanged values marked as "passed through"
4. Load pre-configured examples using the example buttons
5. Review test history to compare different scenarios

Example Output:

Translation Result

Calling Number: 5551234567 → +1-555-123-4567

Called Number: 9078720155 → +1-907-872-0155

✓ Translated by 3 rule(s)

Step-by-Step Transformations

Step 1
Rule #1 (Priority 10) ↓ Continue
Add country code to 10-digit numbers
Called: 9078720155 → +19078720155

Step 2
Rule #2 (Priority 20) ↓ Continue
Format area code with dashes
Called: +19078720155 → +1-907-8720155

Step 3
Rule #3 (Priority 30)
Final formatting for gateway
Called: +1-907-8720155 → +1-907-872-0155

API Reference

Core Operations Overview



Translation Parameters

translate_numbers accepts the following parameters:

- `calling_number` (optional): Originating phone number
- `called_number` (optional): Destination phone number
- `source_smsc` (optional): Source SMSC identifier
- `message_id` (optional): For event logging

Returns:

- `{:ok, translated_calling, translated_called, [rules_applied]}` - Always successful
- Returns original numbers if no rules match
- Returns list of all rules that were applied (in order)

```
# Example usage
{:ok, new_calling, new_called, rules} =
  NumberTranslation.translate_numbers(
    calling_number: "5551234567",
    called_number: "9078720155",
    source_smsc: "domestic_gateway",
    message_id: "msg_123"
  )

# Check if any translation occurred
if rules != [] do
  Logger.info("Applied #{length(rules)} translation rules")
  Enum.each(rules, fn rule ->
    Logger.info("  - Rule ##{rule.rule_id}: #{rule.description}")
  end)
end
```

Rule Management Operations

```
# Add a new rule
{:ok, rule} = NumberTranslation.add_rule(%{
  calling_prefix: nil,
  called_prefix: nil,
  source_smsc: "gateway1",
  calling_match: "^(\\d{10})$",
  calling_replace: "+1\\1",
  called_match: "^(\\d{10})$",
  called_replace: "+1\\1",
  priority: 10,
  description: "Add +1 to 10-digit numbers",
  enabled: true,
  continue: false
})

# Update a rule
{:ok, updated_rule} = NumberTranslation.update_rule(rule_id, %{
  enabled: false,
  description: "Disabled for testing"
})

# Delete a rule
:ok = NumberTranslation.delete_rule(rule_id)

# Get a specific rule
rule = NumberTranslation.get_rule(rule_id)

# List all rules
all_rules = NumberTranslation.list_rules()

# List only enabled rules (sorted by priority)
enabled_rules = NumberTranslation.list_enabled_rules()
```

Import/Export Operations

```
# Export all rules
backup = NumberTranslation.export_rules()
# Returns: %{
#   version: "1.0",
#   exported_at: ~U[2024-01-15 10:30:00Z],
#   count: 5,
#   rules: [...]
# }

# Save to JSON file
json = Jason.encode!(backup, pretty: true)
File.write!("translation_rules_backup.json", json)

# Import rules (merge with existing)
{:ok, %{imported: 3, failed: 0}} =
  NumberTranslation.import_rules(backup, mode: :merge)

# Import rules (replace all existing)
{:ok, %{imported: 5, failed: 0}} =
  NumberTranslation.import_rules(backup, mode: :replace)
```

Best Practices

Rule Design

1. Keep priorities organized:

- **1-10**: Critical normalization rules (add country codes, fix formats)
- **11-50**: Gateway-specific formatting
- **51-100**: Optional transformations
- **101+**: Catch-all or debugging rules

2. Use continue strategically:

- Enable `continue: true` for normalization rules that prepare numbers for further processing

- Disable `continue: false` for final formatting rules
- Avoid long chains (3-4 rules maximum) to maintain performance

3. Document your rules:

- Always add clear descriptions
- Include examples in the description (e.g., "5551234567 → +15551234567")
- Document the purpose and expected input/output

4. Test regex patterns:

- Test patterns with the simulator before deploying
- Use capture groups (`\1`, `\2`) for flexible transformations
- Escape special regex characters (dots, parentheses, etc.)

Performance

1. Minimize rule count:

- Combine similar rules where possible
- Use prefix matching to reduce regex evaluations
- Remove or disable unused rules

2. Optimize regex patterns:

- Use prefix matching first (faster than regex)
- Keep regex patterns simple
- Avoid backtracking-heavy patterns

3. Limit rule chaining:

- Long chains (5+ rules) can impact performance
- Consider combining multiple steps into one rule if possible
- Monitor translation latency with Telemetry metrics

Operations

1. Test before deploy:

- Use the simulator with real-world examples
- Test edge cases (empty numbers, special characters)
- Verify continue flag behavior

2. Backup regularly:

- Export rules before making major changes
- Version control your exports
- Test imports in non-production first

3. Monitor translations:

- Enable message_id logging for debugging
- Check event logs for translation decisions
- Monitor which rules are being applied

4. Gradual rollout:

- Add new rules as disabled first
- Test with simulator
- Enable and monitor
- Adjust as needed

Regex Tips

1. Common patterns:

- 10-digit US number: `^(\d{10})$`
- International format: `^\+(\d+)$`
- Remove leading zeros: `^0+(.+)$`
- Add dashes: `^(\d{3})(\d{3})(\d{4})$` → `\1-\2-\3`

2. Capture groups:

- Use parentheses to capture: `^(\d{3})(\d{7})$`
- Reference in replace: `+1\1\2`
- Multiple captures: `^\+(\d{1,3})(\d+)$` → `00\1\2`

3. Escape special characters:

- Literal dot: `\.`
- Literal plus: `\+`
- Literal parenthesis: `\(` or `\)`

Troubleshooting

Rule Not Matching

Symptom: Expected rule doesn't match, numbers pass through unchanged

Possible causes:

- Prefix doesn't match (check for exact prefix match)
- Source SMSC doesn't match
- Regex pattern doesn't match input format
- Rule is disabled
- Higher priority rule matched first (with `continue: false`)

Solutions:

1. Use simulator to see which rules are evaluated
2. Check rule status (enabled/disabled)
3. Verify prefix matching (case-sensitive)
4. Test regex pattern separately
5. Check priority order

Wrong Transformation Applied

Symptom: Number transformed but result is incorrect

Possible causes:

- Regex pattern matches but replace pattern is wrong
- Multiple rules applying in unexpected order
- Capture group references incorrect (\1, \2, etc.)

Solutions:

1. Use simulator to see step-by-step transformations
2. Verify regex pattern captures correct groups
3. Check replace pattern syntax
4. Test regex in online regex tester
5. Review rule priority and continue flags

Infinite Loop / Performance Degradation

Symptom: Translation takes very long or appears to hang

Note: This should not happen due to loop prevention, but if it does:

Possible causes:

- Bug in loop prevention logic
- Extremely long regex evaluation
- Very long rule chain

Solutions:

1. Check application logs for errors
2. Review rules with continue: true
3. Simplify regex patterns
4. Reduce number of chained rules
5. Report bug if loop prevention failed

Unexpected Rule Chaining

Symptom: More rules applied than expected

Possible causes:

- Rules have continue: true when they shouldn't
- Priority ordering allows multiple matches
- Transformed number matches additional rules

Solutions:

1. Use simulator to see exact rule chain
2. Review continue flags on all rules
3. Adjust priorities to control order
4. Set continue: false on final rule

Translation Not Applied Before Routing

Symptom: Router sees untranslated numbers

Possible causes:

- Translation not integrated in message flow
- Translation happening after routing
- Application code bypassing translation

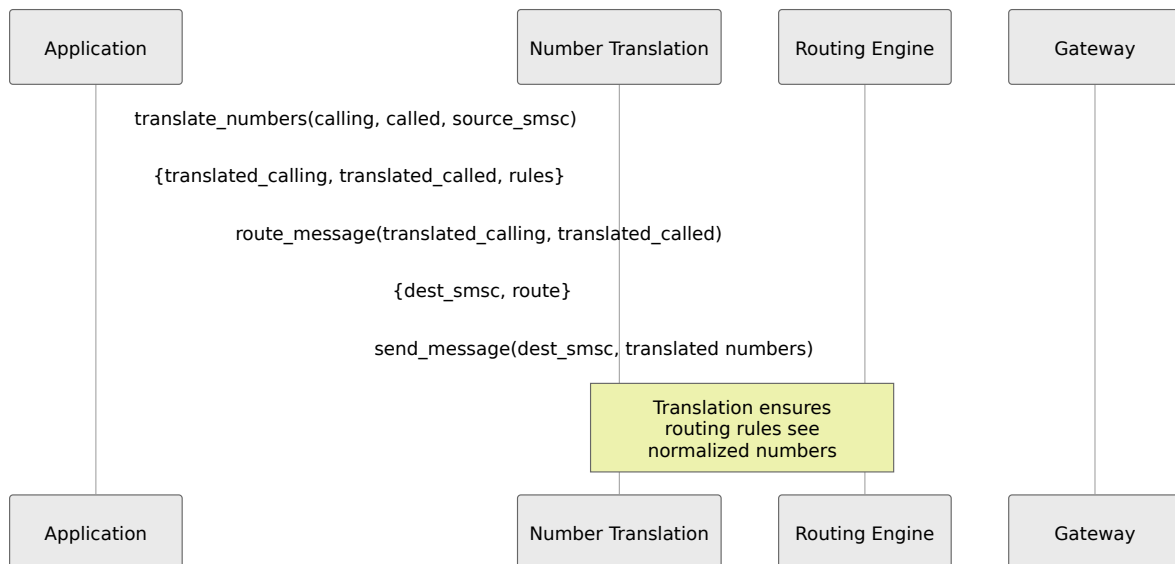
Solutions:

1. Verify application integration: translation should be called before routing
2. Check message processing pipeline
3. Review event logs for translation events
4. Ensure `translate_numbers` is called in correct order

Advanced Topics

Integration with Routing

Translation happens **before** routing to ensure consistent number formats:



Event Logging

Translation decisions are logged via the EventLogger:

- `translation_started`: Translation begins
- `translation_candidates`: Number of enabled rules
- `translation_matched`: Rule matched and applied
- `translation_calling`: Calling number transformed
- `translation_called`: Called number transformed
- `translation_continue`: Rule has `continue=true`, continuing evaluation
- `translation_none`: No rules matched

Enable logging by passing `message_id` to `translate_numbers/1`.

Telemetry Metrics

Monitor translation performance with Telemetry:

```
:telemetry.attach(  
  "number-translation-handler",  
  [:sms_c, :number_translation, :translate, :stop],  
  fn _event_name, measurements, metadata, _config ->  
    # measurements: %{duration: microseconds}  
    # metadata: %{rules_applied: count, ...}  
  end,  
  nil  
)
```

Key metrics to monitor:

- Translation duration (p50, p95, p99)
- Rules applied per message
- Rules matched vs not matched
- Continue flag usage

Clustering

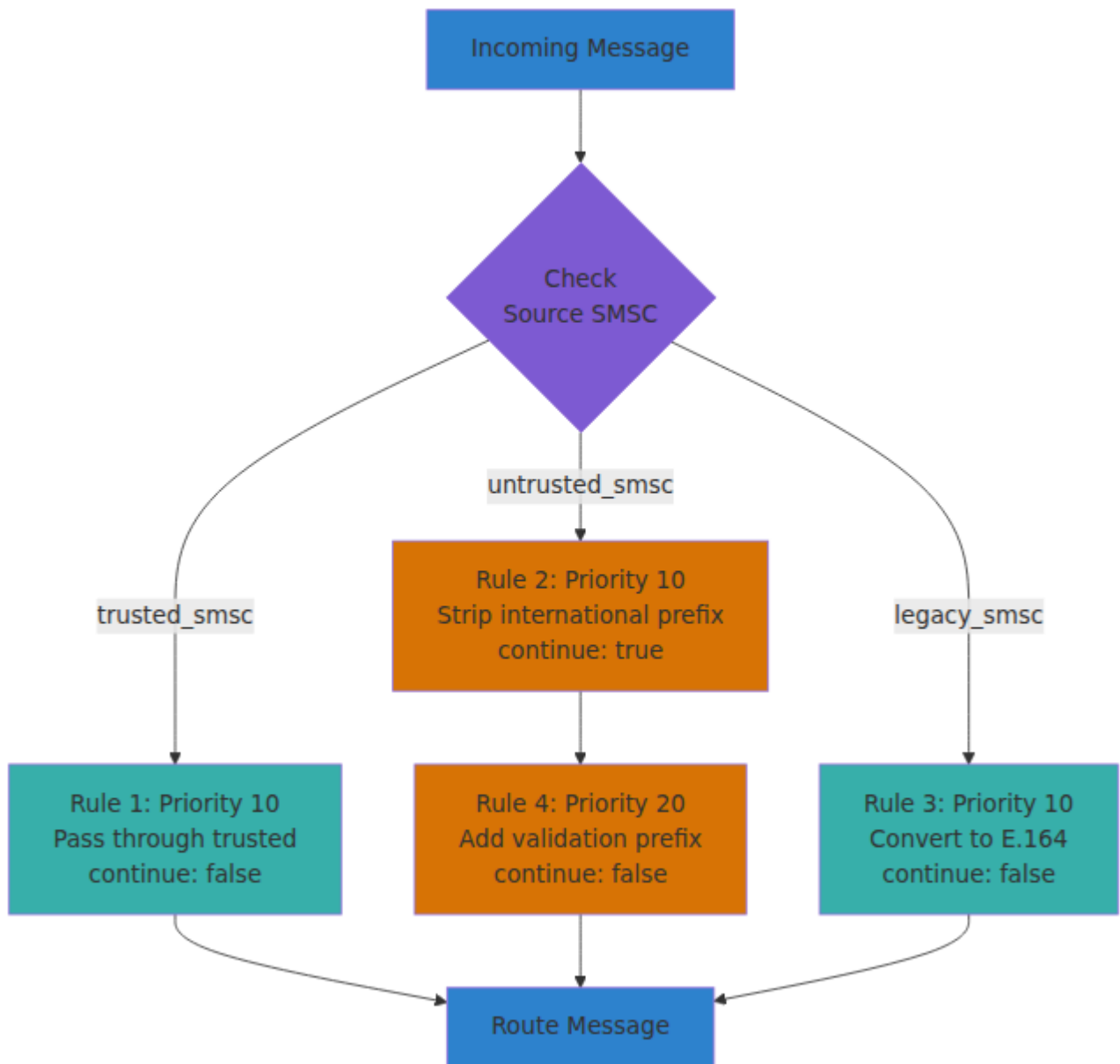
Mnesia tables are automatically distributed across clustered nodes. Translation rules are replicated for high availability.

Parse error on line 25: ... style New fill:#3182CE style P -----^
Expecting 'SOLID_OPEN_ARROW', 'DOTTED_OPEN_ARROW', 'SOLID_ARROW',
'BIDIRECTIONAL_SOLID_ARROW', 'DOTTED_ARROW',
'BIDIRECTIONAL_DOTTED_ARROW', 'SOLID_CROSS', 'DOTTED_CROSS',
'SOLID_POINT', 'DOTTED_POINT', got 'TXT'

Try again

Migration Strategies

When deploying new translation rules:



Examples

Example 1: US Number Normalization

Requirement: Convert various US number formats to E.164 (+1XXXXXXXXXX)

```
# Rule 1: 10-digit numbers (highest priority)
%{
  calling_match: "^(\\d{10})$",
  calling_replace: "+1\\1",
  called_match: "^(\\d{10})$",
  called_replace: "+1\\1",
  priority: 5,
  description: "Add +1 to bare 10-digit numbers",
  enabled: true,
  continue: false
}

# Rule 2: 1 + 10 digits (medium priority)
%{
  calling_match: "^1(\\d{10})$",
  calling_replace: "+1\\1",
  called_match: "^1(\\d{10})$",
  called_replace: "+1\\1",
  priority: 10,
  description: "Convert 1XXXXXXXXXX to +1XXXXXXXXXX",
  enabled: true,
  continue: false
}

# Test cases:
# "5551234567" → "+15551234567" (Rule 1)
# "15551234567" → "+15551234567" (Rule 2)
# "+15551234567" → "+15551234567" (No match, pass through)
```

Example 2: International Prefix Conversion with Chaining

Requirement: Convert 00 prefix to +, then format for gateway

```

# Rule 1: Convert 00 to + (continues to next rule)
%{
    calling_match: "^00(.+)$",
    calling_replace: "+\1",
    called_match: "^00(.+)$",
    called_replace: "+\1",
    priority: 5,
    description: "Convert 00 international prefix to +",
    enabled: true,
    continue: true # Continue to format
}

# Rule 2: Format for gateway (stops processing)
%{
    calling_match: "^\\+(\\d+)$",
    calling_replace: "00\1",
    called_match: "^\\+(\\d+)$",
    called_replace: "00\1",
    priority: 10,
    description: "Format + numbers as 00 for gateway",
    enabled: true,
    continue: false # Stop after this
}

# Test case:
# Step 1: "00441234567890" → "+441234567890" (Rule 1, continue)
# Step 2: "+441234567890" → "00441234567890" (Rule 2, stop)
# Result: "00441234567890"
# Rules applied: [Rule 1, Rule 2]

```

Example 3: SMSC-Specific Handling

Requirement: Apply different rules based on source SMSC

```
# Rule 1: Trusted SMSC - pass through (priority 5)
%{
  source_smsc: "trusted_gateway",
  calling_match: nil, # No transformation
  calling_replace: nil,
  called_match: nil,
  called_replace: nil,
  priority: 5,
  description: "Pass through numbers from trusted gateway",
  enabled: true,
  continue: false
}

# Rule 2: Untrusted SMSC - normalize (priority 10)
%{
  source_smsc: "untrusted_gateway",
  calling_match: "^(.*)$",
  calling_replace: "+VALIDATE\1",
  called_match: "^(.*)$",
  called_replace: "+VALIDATE\1",
  priority: 10,
  description: "Add validation prefix for untrusted source",
  enabled: true,
  continue: false
}

# Rule 3: Catch-all for other SMSCs (priority 100)
%{
  source_smsc: nil, # Wildcard
  calling_match: "^(\\d{10})$",
  calling_replace: "+1\\1",
  called_match: "^(\\d{10})$",
  called_replace: "+1\\1",
  priority: 100,
  description: "Default: Add +1 to 10-digit numbers",
  enabled: true,
  continue: false
}
```


Example 4: Multi-Step Formatting Chain

Requirement: Normalize → Add country code → Format with dashes

```
# Rule 1: Strip leading zeros (continue)
%{
  calling_match: "^0+(.)$",
  calling_replace: "\1",
  called_match: "^0+(.)$",
  called_replace: "\1",
  priority: 5,
  description: "Strip leading zeros",
  enabled: true,
  continue: true
}

# Rule 2: Add country code if missing (continue)
%{
  calling_match: "^(\\d{10})$",
  calling_replace: "+1\\1",
  called_match: "^(\\d{10})$",
  called_replace: "+1\\1",
  priority: 10,
  description: "Add +1 to 10-digit numbers",
  enabled: true,
  continue: true
}

# Rule 3: Format with dashes (stop)
%{
  calling_match: "^\\+1(\\d{3})(\\d{3})(\\d{4})$",
  calling_replace: "+1-\\1-\\2-\\3",
  called_match: "^\\+1(\\d{3})(\\d{3})(\\d{4})$",
  called_replace: "+1-\\1-\\2-\\3",
  priority: 15,
  description: "Format as +1-XXX-XXX-XXXX",
  enabled: true,
  continue: false
}

# Test case:
# Input: "005551234567"
# Step 1: "005551234567" → "5551234567" (Rule 1, continue)
# Step 2: "5551234567" → "+15551234567" (Rule 2, continue)
# Step 3: "+15551234567" → "+1-555-123-4567" (Rule 3, stop)
```

```
# Result: "+1-555-123-4567"  
# Rules applied: [Rule 1, Rule 2, Rule 3]
```

Support

For issues or questions:

- Check the test suite at `test/sms_c/messaging/number_translation_test.exe` for examples
- Use the simulator to debug translation logic
- Review event logs for translation decisions
- Check Mnesia table contents: `:mnesia.table_info(:translation_rule, :size)`
- Monitor Telemetry metrics for performance issues

SMS-C Operations Guide

[← Back to Documentation Index](#)

Daily operational procedures, monitoring, and maintenance tasks for SMS-C operations teams.

Table of Contents

- [Daily Operations](#)
- [Monitoring](#)
- [Message Tracking](#)
- [Route Management](#)
- [Frontend Management](#)
- [Number Translation Management](#)
- [System Maintenance](#)
- [Backup and Recovery](#)
- [Capacity Planning](#)
- [Incident Response](#)

Daily Operations

Morning Health Check

Perform these checks at the start of each day:

1. Check System Status

```
# API health check
curl https://api.example.com:8443/api/status

# Expected response:
# {"status":"ok","application":"OmniMessage","timestamp":"2025-10-30T08:00:00Z"}
```

2. Review Prometheus Metrics

Access Prometheus dashboard and check:

- Message throughput (last 24 hours)
- Routing failure rate (should be < 1%)
- Queue backlog (should be < 1000 pending)
- Delivery success rate (should be > 95%)
- Frontend connection status (all expected frontends active)

3. Check Message Queue

Access Web UI: https://sms-admin.example.com/message_queue

Review:

- Total pending messages (should be low)
- Oldest message age (should be < 5 minutes)
- Messages with high delivery attempts (investigate if > 3)
- Dead letter messages (investigate any present)

4. Review Frontend Status

Access Web UI: https://sms-admin.example.com/frontend_status

Verify:

- All expected frontends are active
- No unexpired disconnections
- No frontend errors in last 24 hours

5. Check Application Logs

Access Web UI: `https://sms-admin.example.com/logs` or check log files

Look for:

- Error-level messages
- Routing failures
- Charging failures
- Database connection issues
- Cluster node problems

Message Volume Monitoring

Check Hourly Message Counts:

Use Prometheus query:

```
# Messages received per hour
increase(sms_c_message_received_count[1h])

# Messages delivered per hour
increase(sms_c_delivery_succeeded_count[1h])

# Calculate delivery rate
rate(sms_c_delivery_succeeded_count[1h]) /
rate(sms_c_message_received_count[1h])
```

Expected Patterns:

- Business hours: Higher volume
- Nights/weekends: Lower volume
- Delivery rate: Should be > 95%

Alert Conditions:

- Sudden drop in messages (> 50% decrease)
- Sudden spike in messages (> 200% increase)

- Delivery rate drop below 90%

Monitoring

Key Metrics to Watch

Message Processing Metrics

Message Received Count (`sms_c_message_received_count`):

- **What:** Total messages entering system
- **Alert:** Sudden drop or spike
- **Query:** `rate(sms_c_message_received_count[5m])`

Message Processing Duration (`sms_c_message_processing_stop_duration`):

- **What:** End-to-end processing time
- **Alert:** p95 > 1000ms
- **Query:** `histogram_quantile(0.95, sms_c_message_processing_stop_duration)`

Routing Metrics

Routing Failures (`sms_c_routing_failed_count`):

- **What:** Messages that couldn't be routed
- **Alert:** Any failures (> 0)
- **Query:** `increase(sms_c_routing_failed_count[5m])`

Route Matched (`sms_c_routing_route_matched_count`):

- **What:** Which routes are being used
- **Alert:** High-priority routes not matching
- **Query:** `sms_c_routing_route_matched_count`

Delivery Metrics

Delivery Success Rate:

- **What:** Percentage of successful deliveries
- **Alert:** Rate < 95%
- **Query:** `rate(sms_c_delivery_succeeded_count[5m]) / rate(sms_c_delivery_queued_count[5m])`

Delivery Attempts (`sms_c_delivery_succeeded_attempt_count`):

- **What:** Retries needed for delivery
- **Alert:** p95 > 2 (too many retries)
- **Query:** `histogram_quantile(0.95, sms_c_delivery_succeeded_attempt_count)`

Queue Metrics

Queue Size (`sms_c_queue_size_size`):

- **What:** Total messages in queue
- **Alert:** Size > 10,000
- **Query:** `sms_c_queue_size_size`

Oldest Message Age (`sms_c_queue_oldest_message_age_seconds`):

- **What:** Age of oldest pending message
- **Alert:** Age > 300 seconds
- **Query:** `sms_c_queue_oldest_message_age_seconds`

Dashboard Setup

Operational Dashboard Panels:

1. Message Throughput (Graph)

- Messages received (5-minute rate)
- Messages delivered (5-minute rate)
- Time range: Last 24 hours

2. Queue Status (Single Stats)

- Current pending messages
- Oldest message age
- Failed message count

3. **Delivery Performance** (Graph)

- Success rate over time
- Failure rate over time
- Time range: Last 24 hours

4. **Routing Status** (Table)

- Route ID
- Match count (last hour)
- Destination SMSC
- Priority

5. **Frontend Status** (Table)

- Frontend name
- Status (active/expired)
- Last seen
- Message count (last hour)

6. **System Health** (Single Stats)

- API response time (p95)
- Database query time (p95)
- ENUM lookup time (p95)

Alert Configuration

Critical Alerts (Immediate Response Required):

```
# No route found - messages cannot be delivered
- alert: RoutingFailures
  expr: increase(sms_c_routing_failed_count[5m]) > 0
  severity: critical
  description: "{{ $value }}" messages failed routing in last 5
minutes"

# Queue building up - processing falling behind
- alert: QueueBacklog
  expr: sms_c_queue_size_pending > 10000
  severity: critical
  description: "Queue has {{ $value }}" pending messages"

# Messages aging - delivery stuck
- alert: OldMessagesInQueue
  expr: sms_c_queue_oldest_message_age_seconds > 300
  severity: critical
  description: "Oldest message is {{ $value }}" seconds old"

# Frontend disconnected - no delivery path
- alert: FrontendDisconnected
  expr: sms_c_frontend_status_count{status="disconnected"} > 0
  severity: critical
  description: "{{ $value }}" frontends disconnected"
```

Warning Alerts (Investigation Needed):

```

# Delivery success rate dropping
- alert: LowDeliveryRate
  expr: rate(sms_c_delivery_succeeded_count[10m]) /
rate(sms_c_delivery_queued_count[10m]) < 0.90
  severity: warning
  description: "Delivery success rate is {{ $value }}"

# Too many delivery retries
- alert: HighRetryRate
  expr: histogram_quantile(0.95,
sms_c_delivery_succeeded_attempt_count) > 2
  severity: warning
  description: "95th percentile delivery attempts: {{ $value }}"

# ENUM lookups slow or failing
- alert: SlowEnumLookups
  expr: histogram_quantile(0.95, sms_c_enum_lookup_stop_duration)
> 5000
  severity: warning
  description: "ENUM lookups taking > 5 seconds"

# Low ENUM cache hit rate
- alert: LowEnumCacheHitRate
  expr: rate(sms_c_enum_cache_hit_count[10m]) /
(rate(sms_c_enum_cache_hit_count[10m]) +
rate(sms_c_enum_cache_miss_count[10m])) < 0.70
  severity: warning
  description: "ENUM cache hit rate: {{ $value }}"

```

Message Tracking

Find Specific Message

By Message ID:

1. **Web UI:** Navigate to `/message_queue`
2. Enter message ID in search box
3. View full details and event history

Via API:

```
curl https://api.example.com:8443/api/messages/12345
```

By Phone Number:

1. **Web UI:** Navigate to `/message_queue`
2. Enter phone number in search box
3. View all messages for that number

Track Message Lifecycle

View Event History:

1. **Web UI:** Click on message in queue, view "Events" section
2. **API:** `GET /api/events/12345`

Common Event Sequence:

1. `message_inserted` - Message created
↓
2. `number_translated` - Numbers normalized (if configured)
↓
3. `message_routed` - Routing decision made
↓
4. `charging_attempted` - Charging check (if enabled)
↓
5. `message_delivered` - Successfully delivered

Failed Delivery Sequence:

```
1. message_inserted
  ↓
2. message_routed
  ↓
3. delivery_attempt_1 - First attempt failed
  ↓
4. delivery_attempt_2 - Second attempt failed (2min delay)
  ↓
5. delivery_attempt_3 - Third attempt failed (4min delay)
  ↓
6. message_dead_letter - Exceeded retry limit
```

Check Delivery Status

Awaiting Delivery:

- Status: "queued" (ready to send now) or "backoff" (waiting out a retry)
- deliver_after: nil/past for `queued`, a future timestamp for `backoff`
- delivery_attempts: 0 for a fresh `queued` message, higher for `backoff`

Delivered Messages:

- Status: "delivered" (delivery confirmed) or "sent" (submitted, no DLR yet)
- deliver_time: Timestamp of delivery
- dest_smsc: Frontend that delivered

Undelivered / Failed Messages:

- Status: "backoff" with high delivery_attempts — still retrying (capped at a 30-minute interval, until the message's `expires` is reached)
- Status: "expired", "dropped", "balance_rejected", or "auto_replied" — terminal
- Check event log for failure reasons

Location-Based Message Routing

The SMS-C supports location-based message retrieval, allowing frontends to automatically receive messages destined for subscribers registered at their

location.

How It Works:

When a frontend queries for pending messages using `get_messages_for_smsc(smsc_name)`, the system returns messages in two ways:

1. **Explicit Routing** - Messages where `dest_smsc` explicitly matches the frontend name
2. **Location-Based Routing** - Messages where:
 - `dest_smsc` is `null` (not explicitly routed)
 - `destination_msisdn` has an active location record
 - The location's `location` field matches the frontend name
 - The location has not expired

Example Scenario:

A subscriber with MSISDN `+447700900123` registers at frontend `uk_gateway`:

```
# Subscriber registers (creates location record)
POST /api/locations
{
  "msisdn": "+447700900123",
  "imsi": "234150123456789",
  "location": "uk_gateway",
  "expires": "2025-11-01T12:00:00Z"
}
```

When a message arrives for this subscriber without explicit routing:

```
# Message submitted without dest_smsc
POST /api/messages
{
  "source_msisdn": "+15551234567",
  "destination_msisdn": "+447700900123",
  "message_body": "Hello",
  "source_smsc": "api"
  # Note: dest_smsc is null
}
```

The `uk_gateway` frontend will automatically receive this message when it polls:

```
# Frontend polls for messages
GET /api/messages/queue?smsc=uk_gateway

# Returns the message even though dest_smsc is null
# because the destination subscriber is registered at uk_gateway
```

Location Requirements:

For location-based routing to work:

- The `locations` table must have an entry for the `destination_msisdn`
- The `location` field must match the querying SMSC name
- The `expires` timestamp must be in the future

Monitoring Location-Based Routing:

Check location records:

```
# Via API
GET /api/locations/{msisdn}

# Check if location is expired
# expires field should be > current time
```

Common Issues:

- **Message not delivered:** Check if location has expired
- **Wrong frontend:** Verify `location` field matches expected frontend name
- **Location not found:** Subscriber may need to re-register

Manual Interventions

Retry Failed Message:

```
# Reset delivery_attempts and deliver_after
curl -X PATCH https://api.example.com:8443/api/messages/12345 \
  -H "Content-Type: application/json" \
  -d '{
    "delivery_attempts": 0,
    "deliver_after": "2025-10-30T12:00:00Z"
  }'
```

Change Destination:

```
# Route to different SMSC
curl -X PATCH https://api.example.com:8443/api/messages/12345 \
  -H "Content-Type: application/json" \
  -d '{
    "dest_smsc": "backup_gateway"
  }'
```

Delete Stuck Message:

```
curl -X DELETE https://api.example.com:8443/api/messages/12345
```

Route Management

View Current Routes

Web UI: Navigate to `/sms_routing`

Via API:

```
# List all routes  
curl https://api.example.com:8443/api/routes
```

Check Route Usage:

Prometheus query:

```
# Messages routed by each route (last hour)  
increase(sms_c_routing_route_matched_count[1h])
```

Add New Route

Web UI:

1. Navigate to `/sms_routing`
2. Click "Add New Route"
3. Fill in fields:
 - **Calling Prefix:** Source number prefix (optional)
 - **Called Prefix:** Destination number prefix (required for geographic routing)
 - **Source SMSC:** Source system filter (optional)
 - **Dest SMSC:** Destination gateway (required unless auto-reply/drop)
 - **On-Net Only:** Restrict this route to on-net destinations only (requires Diameter/HSS)
 - **Priority:** Route priority (1-255, lower = higher priority)
 - **Weight:** Load balancing weight (1-100)
 - **Description:** Human-readable description
 - **Enabled:** Check to activate immediately
4. Click "Save Route"

Example: Geographic Route:

- Called Prefix: `+44`

- Dest SMSC: uk_gateway
- Priority: 50
- Weight: 100
- Description: "UK routing"

Example: Load Balanced Route:

Create two routes with same criteria but different weights:

Route 1:

- Called Prefix: +44
- Dest SMSC: uk_primary
- Priority: 50
- Weight: 70
- Description: "UK primary (70%)"

Route 2:

- Called Prefix: +44
- Dest SMSC: uk_backup
- Priority: 50
- Weight: 30
- Description: "UK backup (30%)"

Example: On-Net Only Route:

Restrict an external carrier or internal application to only send to on-net subscribers. Messages to off-net destinations are dropped.

- Source SMSC: carrier_smpp_bind
- Dest SMSC: local_msc
- On-Net Only: checked
- Priority: 50
- Weight: 100
- Description: "Carrier X — on-net only"

This requires Diameter/HSS to be enabled. If the HSS is unavailable, messages are dropped (fail-closed).

Test Routes

Routing Simulator:

1. Navigate to `/simulator`
2. Enter test parameters:
 - Calling Number: `+15551234567`
 - Called Number: `+447700900000`
 - Source SMSC: (optional)
 - Source Type: (optional)
3. Click "Simulate Routing"
4. Review results:
 - **Selected Route:** Which route was chosen
 - **All Matches:** Which routes matched criteria
 - **Evaluation:** Why each route matched or didn't match

Test Before Production:

- Test all new routes in simulator
- Verify correct route is selected
- Check priority ordering
- Validate weight distribution

Modify Existing Route

Web UI:

1. Navigate to `/sms_routing`
2. Find route in list
3. Click "Edit"
4. Modify fields
5. Click "Save Route"

Common Modifications:

- **Disable Route:** Uncheck "Enabled" (temporary removal)
- **Adjust Weight:** Change load balance distribution
- **Change Priority:** Reorder route evaluation
- **Update Destination:** Switch to different SMSC

Delete Route

Web UI:

1. Navigate to `/sms_routing`
2. Find route in list
3. Click "Delete"
4. Confirm deletion

Warning: Deleting routes is permanent. Consider disabling instead.

Export/Import Routes

Export Routes (Backup):

1. Navigate to `/sms_routing`
2. Click "Export Routes"
3. Save JSON file

Import Routes:

1. Navigate to `/sms_routing`
2. Click "Import Routes"
3. Select JSON file
4. Choose import mode:
 - **Merge:** Add to existing routes
 - **Replace:** Delete all and import

Use Cases:

- Backup before major changes

- Copy routes between environments
- Disaster recovery
- Configuration versioning

Frontend Management

Monitor Frontend Connections

Web UI: Navigate to `/frontend_status`

Check:

- All expected frontends are "active"
- Last seen times are recent (< 90 seconds)
- No unexpected expired frontends

Via API:

```
# Get active frontends
curl https://api.example.com:8443/api/frontends/active

# Get statistics
curl https://api.example.com:8443/api/frontends/stats
```

Investigate Disconnections

Frontend Expired:

1. Check frontend logs for errors
2. Verify network connectivity to SMS-C
3. Confirm frontend is running
4. Check frontend registration logic (should re-register every 60s)

Registration Not Showing:

1. Verify frontend is calling POST `/api/frontends/register`

2. Check API logs for registration errors
3. Verify JSON payload format
4. Test registration manually with curl

Example Manual Registration:

```
curl -X POST https://api.example.com:8443/api/frontends/register \
-H "Content-Type: application/json" \
-d '{
  "frontend_name": "test_gateway",
  "frontend_type": "smpp",
  "ip_address": "10.0.1.50",
  "hostname": "gateway.example.com"
}'
```

View Frontend History

Web UI:

1. Navigate to `/frontend_status`
2. Find frontend in list
3. Click "History"
4. Review past registrations

Via API:

```
curl https://api.example.com:8443/api/frontends/history/uk_gateway
```

Use Cases:

- Investigate connection reliability
- Track frontend uptime patterns
- Identify configuration changes

Number Translation Management

Number translation rules are managed via `config/runtime.exs`. Changes require application restart.

View Active Translation Rules

Check configuration file:

```
cat config/runtime.exs | grep -A 20 "translation_rules:"
```

Common Translation Tasks

Add Country Code to Local Numbers:

Edit `config/runtime.exs`:

```
%{
  calling_prefix: nil,
  called_prefix: nil,
  source_smsc: nil,
  calling_match: "^(\\d{10})$",
  calling_replace: "+1\\1",
  called_match: "^(\\d{10})$",
  called_replace: "+1\\1",
  priority: 100,
  description: "Add +1 to 10-digit US numbers",
  enabled: true
}
```

Normalize International Format:

```
%{
  calling_prefix: nil,
  called_prefix: nil,
  source_smsc: nil,
  calling_match: "^00(\d+)$",
  calling_replace: "+\1",
  called_match: "^00(\d+)$",
  called_replace: "+\1",
  priority: 10,
  description: "Convert 00 prefix to +",
  enabled: true
}
```

Carrier-Specific Code Stripping:

```
%{
  calling_prefix: nil,
  called_prefix: "101",
  source_smsc: "carrier_a",
  calling_match: nil,
  calling_replace: nil,
  called_match: "^101(\d+)$",
  called_replace: "\1",
  priority: 5,
  description: "Strip carrier code from carrier A",
  enabled: true
}
```

Test Translation Rules

After configuration changes:

1. Restart application to load new rules
2. Submit test message with source/destination that should match
3. Check event log for `number_translated` event
4. Verify numbers were transformed correctly

Disable Translation Rule

Set `enabled: false` in rule:

```
%{  
  ...  
  enabled: false  
}
```

Restart application.

System Maintenance

Database Maintenance

Check Database Size:

Use your database management tools to monitor CDR storage size:

- **MySQL/MariaDB:** Query `information_schema.tables` for database size
- **PostgreSQL:** Use `pg_database_size()` function or `\l+` command in psql

Cleanup Old CDR Records:

CDR records should be archived and purged periodically based on your retention policy:

- Configure automatic archiving based on business requirements (typically 30-90 days in operational database)
- Archive older records to data warehouse or cold storage
- Delete archived records from operational database in batches to avoid lock contention

Optimize Tables:

Periodically optimize database tables to maintain performance:

- **MySQL/MariaDB:** Run `OPTIMIZE TABLE` command during low-traffic periods
- **PostgreSQL:** Run `VACUUM ANALYZE` regularly (or enable autovacuum)

Run Weekly during low-traffic period to maintain optimal performance.

Mnesia Database Maintenance

Check Mnesia Table Size:

```
# In IEx console
:mnesia.table_info(:sms_route, :size)
:mnesia.table_info(:translation_rule, :size)
```

Backup Mnesia Tables:

```
# Export routes (Web UI)
# Navigate to /sms_routing
# Click "Export Routes"

# Or via Mnesia backup
:mnesia.backup("/var/backups/sms_c/mnesia_backup.bup")
```

Restore Mnesia:

```
# Via Web UI import
# Or restore backup:
:mnesia.restore("/var/backups/sms_c/mnesia_backup.bup", [])
```

Log Rotation

Configure logrotate for application logs:

```
# /etc/logrotate.d/sms_c
/var/log/sms_c/*.log {
    daily
    rotate 30
    compress
    delaycompress
    notifempty
    create 0644 sms_user sms_group
    sharedscripts
    postrotate
        systemctl reload sms_c || true
    endscript
}
```

Restart Application

Graceful Restart (zero downtime in cluster):

```
# Restart one node at a time
systemctl restart sms_c

# Wait for node to join cluster
# Repeat for each node
```

Emergency Restart (all nodes):

```
systemctl restart sms_c
```

After Restart:

- Verify all frontends reconnect
- Check Prometheus for metric continuity
- Monitor logs for errors
- Verify message processing resumes

Backup and Recovery

What to Backup

1. Configuration Files:

- `config/runtime.exs`
- `config/config.exs`
- `config/prod.exs` (if exists)

2. Routing Tables (Mnesia):

- Export via Web UI
- Or Mnesia backup command

3. SQL CDR Database:

- Daily full backup
- Transaction log backups (continuous)

4. TLS Certificates:

- `priv/cert/*.crt`
- `priv/cert/*.key`

Backup Procedures

Daily Configuration Backup:

```
#!/bin/bash
# /opt/sms_c/scripts/backup_config.sh

BACKUP_DIR="/var/backups/sms_c/$(date +%Y%m%d)"
mkdir -p $BACKUP_DIR

# Backup configuration
cp -r /opt/sms_c/config $BACKUP_DIR/

# Backup certificates
cp -r /opt/sms_c/priv/cert $BACKUP_DIR/

# Set permissions
chmod 600 $BACKUP_DIR/cert/*

echo "Configuration backup completed: $BACKUP_DIR"
```

Database Backup:

```
#!/bin/bash
# /opt/sms_c/scripts/backup_database.sh

BACKUP_DIR="/var/backups/sms_c/database"
DATE=$(date +%Y%m%d_%H%M%S)

mkdir -p $BACKUP_DIR

# Backup SQL CDR database
# MySQL/MariaDB: Use mysqldump with --single-transaction for consistency
# PostgreSQL: Use pg_dump -F c for custom format

# Example structure (adapt to your database):
# - Use appropriate backup tool (mysqldump, pg_dump)
# - Enable transaction-safe backups for consistency
# - Compress output to save space
# - Configure retention period (e.g., 30 days)

# Remove old backups
find $BACKUP_DIR -name "sms_c_*.gz" -mtime +30 -delete

echo "Database backup completed: sms_c_${DATE}"
```

Routing Table Backup:

```
#!/bin/bash
# /opt/sms_c/scripts/backup_routes.sh

BACKUP_DIR="/var/backups/sms_c/routes"
DATE=$(date +%Y%m%d)

mkdir -p $BACKUP_DIR

# Export via API
curl https://api.example.com:8443/api/routes/export \
  > $BACKUP_DIR/routes_${DATE}.json

echo "Routes backup completed: routes_${DATE}.json"
```

Schedule Backups (crontab):

```
# Daily at 2 AM
0 2 * * * /opt/sms_c/scripts/backup_config.sh
0 2 * * * /opt/sms_c/scripts/backup_database.sh
0 2 * * * /opt/sms_c/scripts/backup_routes.sh
```

Recovery Procedures

Restore Configuration:

```
# Stop application
systemctl stop sms_c

# Restore config files
cp -r /var/backups/sms_c/20251030/config/* /opt/sms_c/config/

# Restore certificates
cp -r /var/backups/sms_c/20251030/cert/* /opt/sms_c/priv/cert/

# Start application
systemctl start sms_c
```

Restore SQL CDR Database:

Use appropriate restore tools for your database:

- **MySQL/MariaDB:** Decompress and pipe to mysql client
- **PostgreSQL:** Use pg_restore with custom format dumps

Important: Stop the SMS-C application before restoring database to prevent data conflicts.

Restore Routing Tables:

1. Navigate to Web UI `/sms_routing`
2. Click "Import Routes"
3. Select backup JSON file
4. Choose "Replace" mode
5. Confirm import

Capacity Planning

Monitor Growth Trends

Message Volume Trend:

Prometheus query (30-day average):

```
avg_over_time(sms_c_message_received_count[30d])
```

Database Growth Rate:

```
-- Monthly data growth
SELECT
  DATE_FORMAT(inserted_at, '%Y-%m') AS month,
  COUNT(*) AS message_count,
  ROUND(SUM(LENGTH(message_body)) / 1024 / 1024, 2) AS data_mb
FROM message_queues
GROUP BY month
ORDER BY month DESC
LIMIT 12;
```

Capacity Indicators

CPU Usage:

- **Normal:** < 50% average
- **High:** > 70% sustained
- **Critical:** > 90%

Memory Usage:

- **Normal:** < 70% of available
- **High:** > 80%
- **Critical:** > 90%

Disk Usage:

- **Normal:** < 60% full
- **High:** > 75%
- **Critical:** > 85%

Queue Depth:

- **Normal:** < 1000 pending
- **High:** > 5000 pending
- **Critical:** > 10,000 pending

Scaling Recommendations

When to Scale Vertically (Upgrade Resources):

- CPU consistently > 70%
- Memory consistently > 80%
- Single-node bottleneck

When to Scale Horizontally (Add Nodes):

- CPU > 50% on all nodes
- Message volume > 5,000 msg/sec
- Geographic distribution needed
- High availability required

Database Scaling:

- Read replicas for reporting queries
- Connection pooling optimization
- Index optimization
- Partition large tables by date

Incident Response

Severity Levels

Critical (Immediate Response):

- No messages being delivered
- All frontends disconnected
- Database unavailable
- API completely down

High (Response within 1 hour):

- Delivery success rate < 80%
- Multiple frontends disconnected
- Routing failures > 10%
- Queue backlog growing

Medium (Response within 4 hours):

- Single frontend disconnected
- Delivery success rate 80-95%
- Slow message processing
- ENUM lookups failing

Low (Response within 24 hours):

- Minor performance degradation
- Single route issue
- Non-critical warning alerts

Incident Checklist

1. Assess Severity:

- Check Prometheus alerts
- Review dashboard metrics

- Check message queue status
- Verify frontend connections

2. Gather Information:

- Recent configuration changes?
- Recent deployments?
- External dependencies status (OCS, DNS)?
- Error messages in logs?

3. Immediate Actions:

- Stop ongoing changes
- Roll back recent deployments if suspected cause
- Enable verbose logging if needed
- Notify stakeholders

4. Investigation:

- Review application logs
- Check system resource usage
- Examine database performance
- Test external dependencies

5. Resolution:

- Apply fix
- Test in simulator
- Deploy to production
- Monitor for improvement

6. Post-Incident:

- Document root cause
- Update monitoring/alerts
- Implement preventive measures
- Update runbooks

Common Incidents

High Queue Backlog:

1. Check delivery success rate
2. Verify frontends are connected and polling
3. Check database performance
4. Review Prometheus for bottlenecks
5. Consider increasing batch size/interval

Routing Failures:

1. Review routing configuration
2. Test in routing simulator
3. Check for missing routes
4. Verify catch-all route exists
5. Check event logs for failure reasons

Frontend Disconnections:

1. Check frontend system status
2. Verify network connectivity
3. Review frontend logs
4. Test manual API registration
5. Check firewall rules

Slow Message Processing:

1. Check database query performance
2. Review batch worker configuration
3. Verify adequate resources (CPU/Memory)
4. Check for ENUM lookup delays
5. Review charging system performance

For detailed troubleshooting procedures, see the [Troubleshooting Guide](#).

Performance Tuning Guide

[← Back to Documentation Index](#)

This guide explains how to optimize SMS-C performance for different workload scenarios.

Performance Overview

SMS-C delivers **1,750 messages/second** throughput using Mnesia for in-memory message storage with automatic SQL database archiving for CDR retention.

Key Performance Metrics

Measured on Intel i7-8650U @ 1.90GHz (8 cores):

Operation	Throughput	Latency (avg)	Improvement
Message Insert (with routing)	1,750 msg/sec	0.58ms	21x faster than SQL
Message Insert (simple)	1,750 msg/sec	0.57ms	21x faster than SQL
Get Messages for SMSC	800 msg/sec	1.25ms	In-memory query
Memory per Insert	62 KB	-	50% reduction

Capacity: ~150 million messages per day on single node

Table of Contents

- [Message Storage Architecture](#)
- [Mnesia Optimization](#)
- [CDR Archiving Configuration](#)
- [Query Optimization](#)
- [Benchmarking](#)

Message Storage Architecture

SMS-C uses a dual-storage architecture for optimal performance:

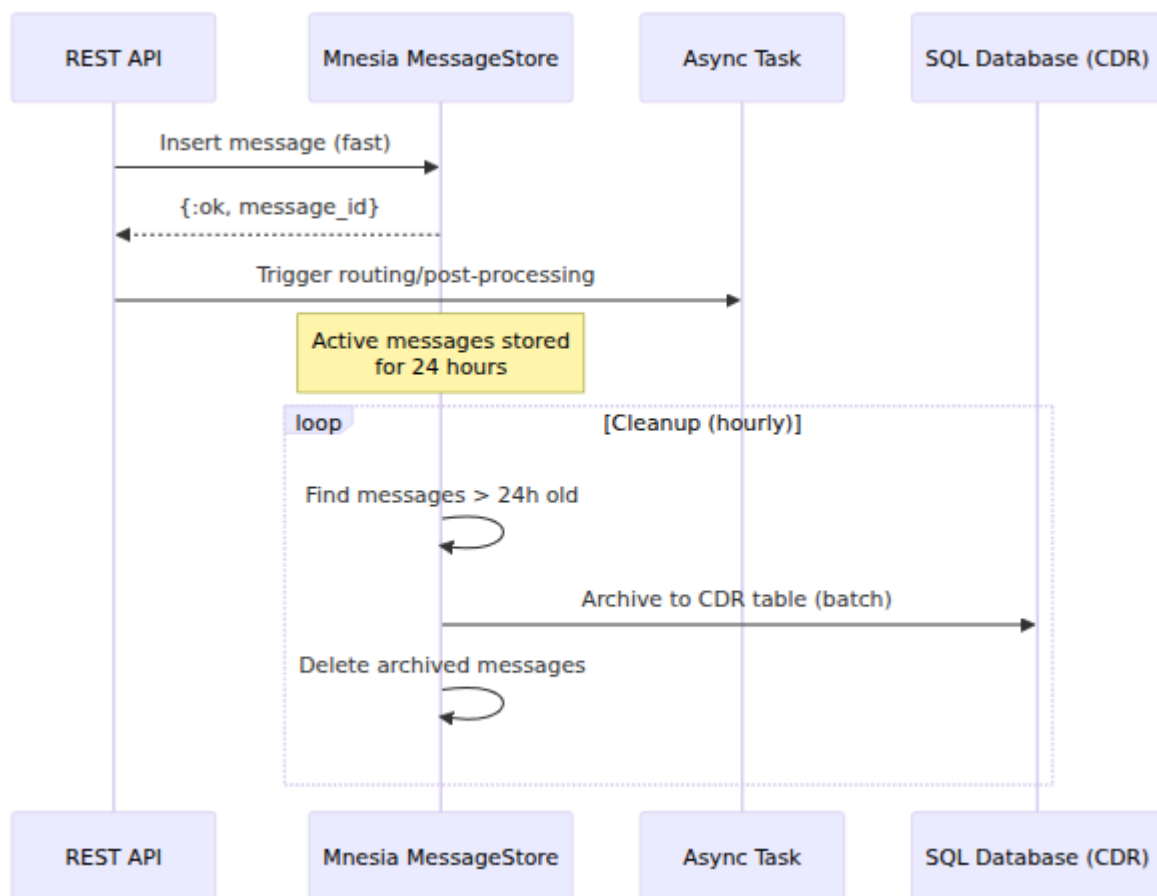
Active Message Store (Mnesia)

- **Purpose:** Ultra-fast message insertion, routing, and delivery
- **Storage:** In-memory with disk persistence (`disc_copies`)
- **Performance:** 1,750 msg/sec insert throughput, 0.58ms latency
- **Retention:** Configurable (default: 24 hours)
- **Clustering:** Supports distributed Mnesia for horizontal scaling

CDR Archive (SQL Database)

- **Purpose:** Long-term message history and reporting
- **Storage:** SQL database (MySQL/MariaDB or PostgreSQL) for durable archival
- **Performance:** Batched writes to minimize database load
- **Retention:** Permanent (or per data retention policy)
- **Queries:** Analytics, reporting, compliance

Data Flow



Mnesia Optimization

Message Retention Configuration

```
# config/runtime.exs
config :sms_c,
  message_retention_hours: 24 # Default: 24 hours
```

Tuning Guidelines:

- **High volume (>1M msg/day):** 12-24 hours retention
 - Minimizes Mnesia table size
 - Faster queries
 - More frequent archiving to MySQL

- **Medium volume (100K-1M msg/day):** 24-48 hours retention
 - Good balance for most deployments
 - Adequate buffer for retry logic
- **Low volume (<100K msg/day):** 48-168 hours retention
 - Longer message history in fast storage
 - Less frequent archiving

Mnesia Table Indices

MessageStore automatically creates indices on:

- `status` - For filtering pending/delivered messages
- `dest_smsc` - For SMSC-specific queries
- `expires` - For expiration handling
- `destination_msisdn` - For subscriber queries
- `source_msisdn` - For subscriber queries

Mnesia Disc Persistence

Messages are stored as `disc_copies` providing:

- ☐ In-memory performance
- ☐ Automatic disk persistence
- ☐ Crash recovery
- ☐ No data loss on restart

CDR Archiving Configuration

The `BatchInsertWorker` handles CDR archiving to MySQL using batched writes:


```
# config/runtime.exs
config :sms_c,
  batch_insert_batch_size: 100,          # CDR batch size
  batch_insert_flush_interval_ms: 100    # Auto-flush interval
```

CDR Tuning Guidelines

High Volume Archiving

```
batch_insert_batch_size: 200
batch_insert_flush_interval_ms: 200
```

- Larger batches reduce MySQL load
- Higher latency for CDR writes (acceptable for archiving)

Balanced (Recommended)

```
batch_insert_batch_size: 100
batch_insert_flush_interval_ms: 100
```

- Good balance for most deployments
- CDRs written within 100ms

Real-time CDR Requirements

```
batch_insert_batch_size: 20
batch_insert_flush_interval_ms: 20
```

- Faster CDR writes for compliance
- More MySQL write operations

Query Optimization

Using Mnesia Indices Effectively

Queries that use indexed fields are fastest:

```
# Fast queries (use indices)
MessageStore.list(status: :pending)
MessageStore.list(dest_smsc: "gateway-1")
Messaging.get_messages_for_smsc("gateway-1")

# Slower queries (full table scan)
MessageStore.list(limit: :infinity) # Returns all messages
```

MySQL Connection Pool

For CDR queries and archiving, configure MySQL connection pool:

```
# config/runtime.exs
config :sms_c, SmsC.Repo,
  pool_size: 10 # Increase for heavy CDR reporting
```

Guidelines:

- Standard deployment: `pool_size: 10`
- Heavy CDR reporting: `pool_size: 20-30`
- Archiving only: `pool_size: 5`

Benchmarking

Running Benchmarks

The project includes Benchee-based benchmarks for performance testing:

```
# Raw SMS API benchmark (compares sync vs async)
mix run benchmarks/raw_sms_bench.exs

# General message API benchmark
mix run benchmarks/message_api_bench.exs
```

Interpreting Results

Example output:

Name		ips	average
deviation	median	99th %	
submit_message_raw_async (batch)		4.65 K	0.22 ms
±41.72%	0.184 ms	0.55 ms	
submit_message_raw (sync)		0.0696 K	14.36 ms
±33.42%	12.57 ms	33.71 ms	

Key metrics:

- **ips:** Iterations per second (higher is better)
- **average:** Average execution time (lower is better)
- **median:** Middle value, more representative than average for skewed distributions
- **99th %:** 99th percentile latency (important for SLA compliance)

Performance Baseline

Expected performance on modern hardware (Intel i7-8650U, 8 cores):

Metric	insert_message (Mnesia)	Previous (MySQL)
Throughput (with routing)	1,750 msg/sec	83 msg/sec
Throughput (simple)	1,750 msg/sec	89 msg/sec
Response Time (avg)	0.58ms	16ms
Response Time (p99)	<5ms	30ms
Memory per operation	62 KB	121 KB
Performance Gain	21x faster	-

Key Improvements:

- ☐ Removed duplicate number translation calls
- ☐ Async post-processing (routing, charging, events)
- ☐ Mnesia in-memory storage vs MySQL disk I/O
- ☐ 50% memory reduction

Monitoring

Runtime Statistics

Check batch worker statistics:

```
SmsC.Messaging.BatchInsertWorker.stats()
```

Returns:

```
%{
  total_enqueued: 10000,
  total_flushed: 9900,
  total_batches: 99,
  current_queue_size: 100,
  flush_errors: 0,
  last_flush_at: ~U[2025-10-22 12:34:56Z],
  last_flush_count: 100,
  last_flush_duration_ms: 45
}
```

Key Metrics to Monitor

1. **Queue Size:** `current_queue_size` - Should be below `batch_size` most of the time
2. **Flush Duration:** `last_flush_duration_ms` - Should be < 100ms for `batch_size=100`
3. **Flush Errors:** `flush_errors` - Should be 0 or very low
4. **Throughput:** `total_flushed / uptime` - Should match expected load

Alerts

Set up monitoring alerts for:

- Queue size consistently at max (indicates backpressure)
- Flush duration increasing (database performance degradation)
- Flush errors > 0 (database connectivity issues)
- Throughput below expected (performance degradation)

Troubleshooting

Symptom: Low Throughput

Possible causes:

1. Database connection pool exhausted: Increase `pool_size`

2. Slow database: Check query performance, add indexes
3. Network latency: Optimize network path to database
4. Batch size too small: Increase `batch_insert_batch_size`

Symptom: High Latency

Possible causes:

1. Flush interval too high: Reduce `batch_insert_flush_interval_ms`
2. Batch size too high: Reduce `batch_insert_batch_size`
3. Database slow writes: Check disk I/O, optimize tables
4. Using async API when you need sync: Switch to synchronous endpoint

Symptom: Memory Issues

Possible causes:

1. Queue backing up: Messages accumulating faster than flushing
2. Batch size too large: Reduce `batch_insert_batch_size`
3. Flush failures: Check `flush_errors` in stats
4. Need to restart worker: `Supervisor.terminate_child/2` and restart

Best Practices

1. **Start with defaults** (100/100ms) and tune based on observed behavior
2. **Monitor in production** for at least 1 week before optimizing
3. **Test configuration changes** in staging with production-like load
4. **Use benchmarks** to validate configuration changes
5. **Document your tuning** decisions for future reference
6. **Set up alerts** before optimizing to catch regressions
7. **Consider time zones** - peak load varies by region

Example Configurations

Configuration: High-Volume Aggregator

```
# config/prod.exs
config :sms_c,
  batch_insert_batch_size: 200,
  batch_insert_flush_interval_ms: 200

config :sms_c, SmsC.Repo,
  pool_size: 50
```

Configuration: Enterprise Real-Time Messaging

```
# config/prod.exs
config :sms_c,
  batch_insert_batch_size: 20,
  batch_insert_flush_interval_ms: 10

config :sms_c, SmsC.Repo,
  pool_size: 20
```

Configuration: Development/Testing

```
# config/dev.exs
config :sms_c,
  batch_insert_batch_size: 10,
  batch_insert_flush_interval_ms: 50

config :sms_c, SmsC.Repo,
  pool_size: 5
```

Further Reading

- [Ecto Performance Guide](#)

- [Benchee Documentation](#)
- [Phoenix Under Pressure](#)

SMS Routing

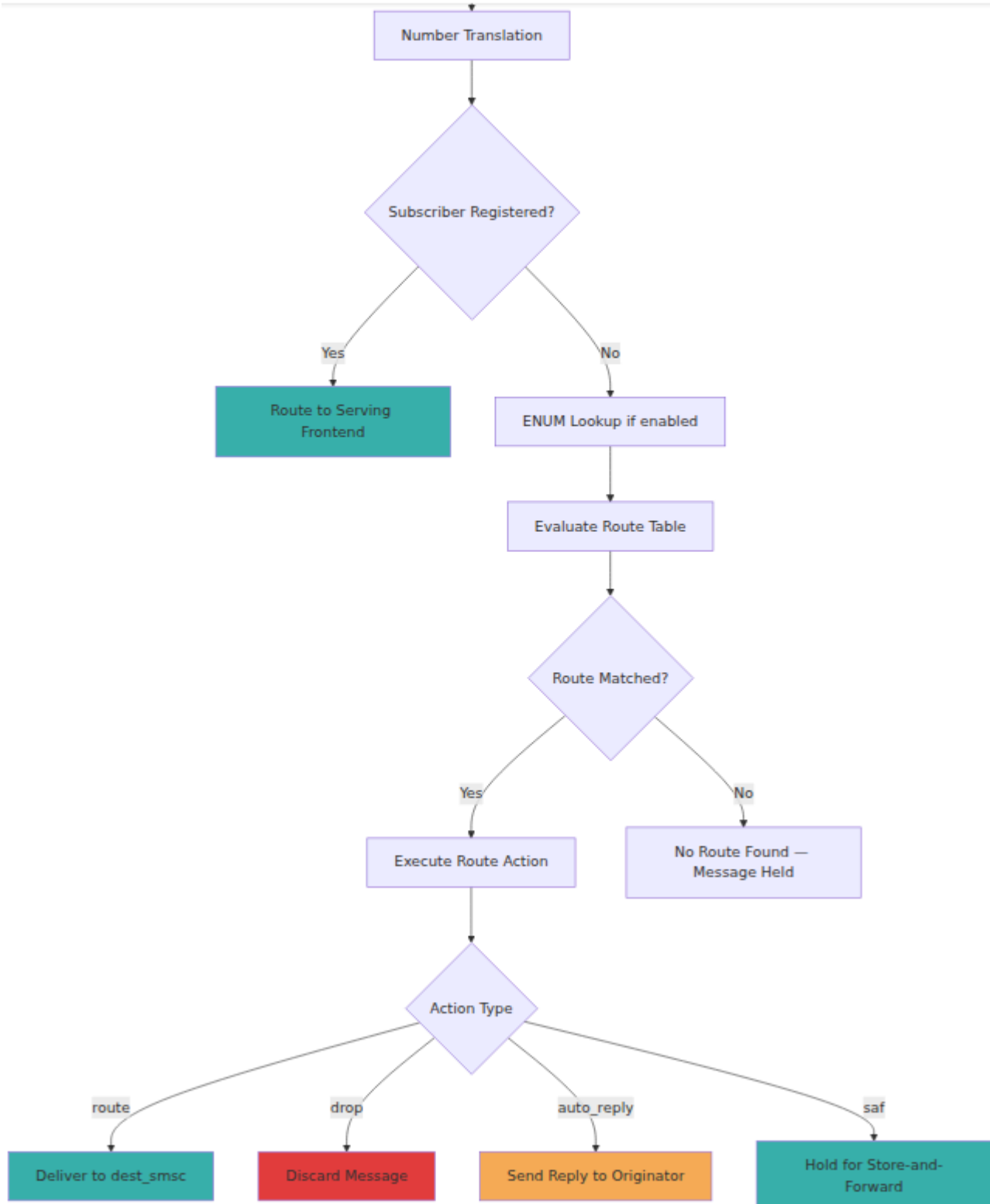
[← Back to Documentation Index](#)

Overview

The SMS-C routing engine determines where each message goes after it enters the system. Routes are stored in Mnesia, can be modified at runtime via API or control panel, and support regex matching, priority ordering, weighted load balancing, ENUM (DNS NAPTR) result matching, and HSS-based on-net/off-net matching.

Message Processing Pipeline

Every message follows this pipeline from submission to routing:



On-net/off-net status is **not** a route action. It is a *match criterion* (`originating_network` / `terminating_network`) evaluated while the route table is filtered — see [On-Net / Off-Net Matching](#).

Priority Order

1. **Location-based routing** — if the destination subscriber has an active registration, route directly to the serving frontend. All route rules are skipped.
2. **Route table evaluation** — all enabled routes are evaluated against the message. Each route is logged with pass/fail details per match criterion (including any HSS on-net/off-net check).
3. **Route action execution** — the selected route's action determines what happens to the message.

Route Data Model

Field	Type	Description
<code>route_id</code>	Integer	Auto-generated unique identifier
<code>calling_regex</code>	String	Regex for source number (<code>nil</code> = wildcard)
<code>called_regex</code>	String	Regex for destination number (<code>nil</code> = wildcard)
<code>source_smsc</code>	String	Source SMSC name (<code>nil</code> = wildcard)
<code>dest_smsc</code>	String	Destination SMSC. Required for <code>route</code> ; optional for <code>saf</code> ; unused by <code>drop</code> / <code>auto_reply</code>
<code>source_type</code>	String	Filter by source type: <code>ims</code> , <code>circuit_switched</code> , <code>smpp</code> , or <code>nil</code>
<code>enum_domain</code>	String	ENUM domain label for the route (<code>nil</code> = none). Stored on the route and adds specificity, but the actual DNS query uses the global <code>enum_domains</code> config — see ENUM Lookup
<code>enum_result_domain</code>	String	Match criterion: route matches only if an ENUM NAPTR result resolves to this domain (<code>nil</code> = wildcard)
<code>action</code>	String	Route action: <code>route</code> , <code>drop</code> , <code>auto_reply</code> , <code>saf</code>
<code>auto_reply_message</code>	String	Reply text (required when action is <code>auto_reply</code>)

Field	Type	Description
charged	String	Charging: yes, no, default
weight	Integer	Load balancing weight (1-100)
priority	Integer	Priority (1-255, lower = higher priority)
description	String	Human-readable label
enabled	Boolean	Active/inactive toggle
originating_network	Atom	Match criterion on source MSISDN HSS status: :any :on_net :off_net (default :any)
terminating_network	Atom	Match criterion on destination MSISDN HSS status: :any :on_net :off_net (default :any)

Route Actions

The `action` field has exactly four valid values: `:route`, `:drop`, `:auto_reply`, `:saf`.

route (Default)

Delivers the message to `dest_smsc`. This is the standard routing action.

```
{
  "action": "route",
  "called_regex": "^61",
  "dest_smsc": "australia-gw",
  "priority": 100,
  "description": "Route Australian numbers to AU gateway"
}
```

drop

Discards the message. No `dest_smsc` required.

```
{
  "action": "drop",
  "called_regex": "^900",
  "priority": 5,
  "description": "Block premium-rate numbers"
}
```

auto_reply

Sends `auto_reply_message` back to the originator. The original message is marked as auto-replied.

```
{
  "action": "auto_reply",
  "called_regex": "^911$",
  "auto_reply_message": "Emergency: SMS to 911 is not supported.
Please dial 911.",
  "priority": 1,
  "description": "911 auto-response"
}
```

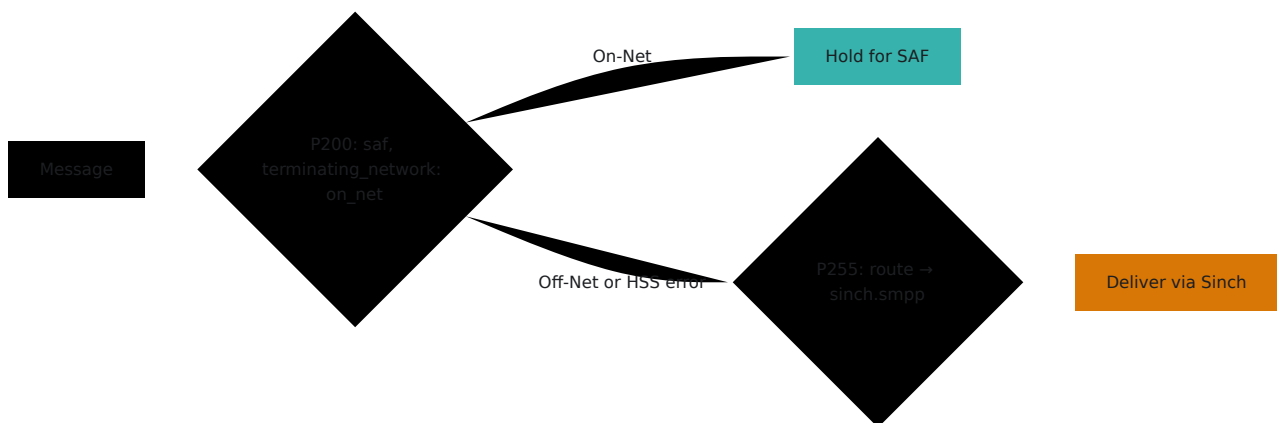
saf

Store-and-forward. The message is held and delivered later, when the destination subscriber registers. `dest_smsc` is optional (if set, it is used when the held message is eventually delivered).

`saf` itself is **unconditional** — it does not perform an HSS dip. To hold *only* on-net subscribers for store-and-forward, combine `saf` with a `terminating_network: :on_net` match criterion (see below), and provide an off-net fallback route.

```
{
  "action": "saf",
  "terminating_network": "on_net",
  "priority": 200,
  "description": "Hold on-net subscribers for store-and-forward"
}
```

Typical Setup: On-Net Hold + Off-Net Fallback



The P200 route only *matches* when the destination is confirmed on-net, so on-net subscribers are held for SAF. Off-net numbers (and HSS errors — fail-closed) fail that route and fall through to the P255 fallback. This ensures on-net subscribers never leak to the off-net gateway.

ENUM Lookup

ENUM lets routing decisions depend on a DNS E.164 NAPTR lookup of the destination number — typically for number-portability or inter-carrier routing, where DNS (not your HSS) is the authority on which carrier a number belongs to.

How it works

1. **Enabled globally.** ENUM only runs when `enum_enabled: true` and `enum_domains` is non-empty. It is a single lookup performed **once per message, before the route table is evaluated** — not per route.
2. **Query.** The called number is converted to reverse ENUM format (`+15551234` → `4.3.2.1.5.5.5.1.e164.arpa`) and a NAPTR query is issued.

Each configured domain in `enum_domains` is tried **in order**, stopping at the first that returns records.

3. **Result domain extraction.** Each returned NAPTR record is parsed and reduced to a `result_domain` — the last two labels of the `replacement` field, or of the host in a `sip:user@host` rewrite (`sip:+15551234@ims.example.com` → `example.com`).
4. **Matching.** A route with `enum_result_domain` set matches only if **any** returned NAPTR record's `result_domain` equals that string exactly. `enum_result_domain: nil` is a wildcard (always matches, even when no ENUM lookup ran).
5. **Caching.** Results are cached in ETS for 15 minutes per `(number, domain)`. Failures are cached too, to avoid hammering DNS on repeated misses.

Deriving on-net/off-net from ENUM

ENUM does not return a boolean on/off-net status — it returns a domain. You express "on-net" by pointing a route's `enum_result_domain` at your own IMS/carrier domain, and giving everything else a fallback route. Contrast this with HSS matching (below), which returns an explicit on-net/off-net answer.

What ENUM does *not* do

- The raw NAPTR result (`enum_result`) is used **only** as a route match key. It is **not** used to rewrite the destination URI or pick a next hop — where a matched message goes is entirely `route.dest_smsc`. The live delivery pipeline discards the raw ENUM result after routing.
- `enum_domain` on a route is **not** used to scope the DNS query (the query always uses the global `enum_domains` list) and is **not** a match criterion. It only contributes to **specificity scoring**.

Configuration

ENUM DNS behaviour (`enum_enabled`, `enum_domains`, `enum_dns_servers`, `enum_timeout`) is configured globally — see [CONFIGURATION.md](#) → [ENUM Configuration](#).

On-Net / Off-Net Matching

Whether a destination (or originator) is on-net is answered by a **Diameter Sh**
dip to the HSS, surfaced on routes as two match criteria. These are **not**
boolean flags and **not** a route action — they are enum-valued criteria
evaluated alongside the regex/SMSC/type checks. A route only matches if its
network criteria are satisfied.

`originating_network` / `terminating_network`

Each takes one of three values:

Value	Meaning
<code>:any</code>	No HSS check; matches on-net and off-net alike (default)
<code>:on_net</code>	Route matches only if the MSISDN is confirmed on-net (HSS returns success, result code 2001)
<code>:off_net</code>	Route matches only if the MSISDN is confirmed off-net (HSS returns <i>Unknown User</i> , result code 5001)

- `originating_network` checks the **calling** number; `terminating_network` checks the **called** number.
- **Use cases:** restrict a route to deliver only to on-net subscribers (`terminating_network: :on_net`); restrict an SMPP bind or external carrier to accept only on-net originators (`originating_network: :on_net`); send an off-net-only fallback via an external gateway (`terminating_network: :off_net`).

Fail-closed behaviour

Any value other than `:any` requires a definitive HSS answer. If the answer is not definitive, the route **does not match** (and the engine tries the next route):

- Diameter disabled → route does not match

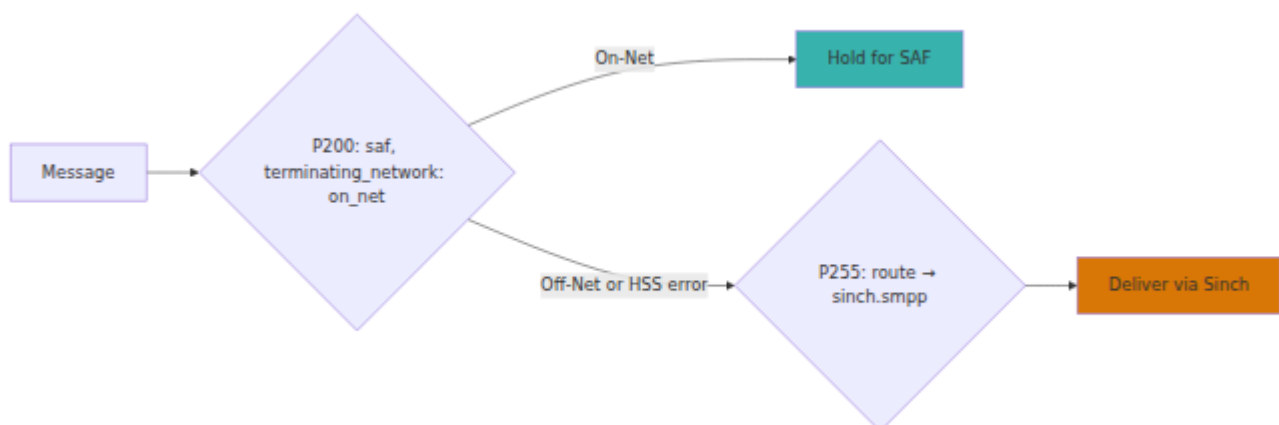
- HSS unreachable / timeout / transport error → route does not match
- HSS returns an error other than *Unknown User* → route does not match

Note that even `:off_net` fails closed: it matches only on an explicit *Unknown User* (5001), not on a transport failure. This prevents leaking traffic to an off-net gateway when the HSS is simply down.

HSS Result Cache

HSS lookups are expensive (a Diameter round-trip). The system caches results per-MSISDN to avoid redundant dips.

- **Cache TTL:** 2 hours (configurable via `hss_cache_ttl_seconds`)
- **Lazy dip:** the HSS is queried only when a route being evaluated has `originating_network` or `terminating_network` set to `:on_net` or `:off_net`. Routes that leave both at `:any` never trigger a dip.
- **Shared across routes:** if multiple routes need the same MSISDN's status, only one HSS dip occurs
- **Forced re-route:** `HssCache.lookup!/1` bypasses the cache (fresh dip) when a message is explicitly re-routed



Route Evaluation Logging

Every route evaluated for a message is logged as a `route_evaluated` event on that message. The log includes:

- Route ID, action, and priority
- Pass/fail status for each match criterion (including `enum_result` and the `orig_network/term_network` HSS checks)
- Detail of what was compared (regex pattern vs number, SMSC name vs source, required network vs actual HSS status, etc.)

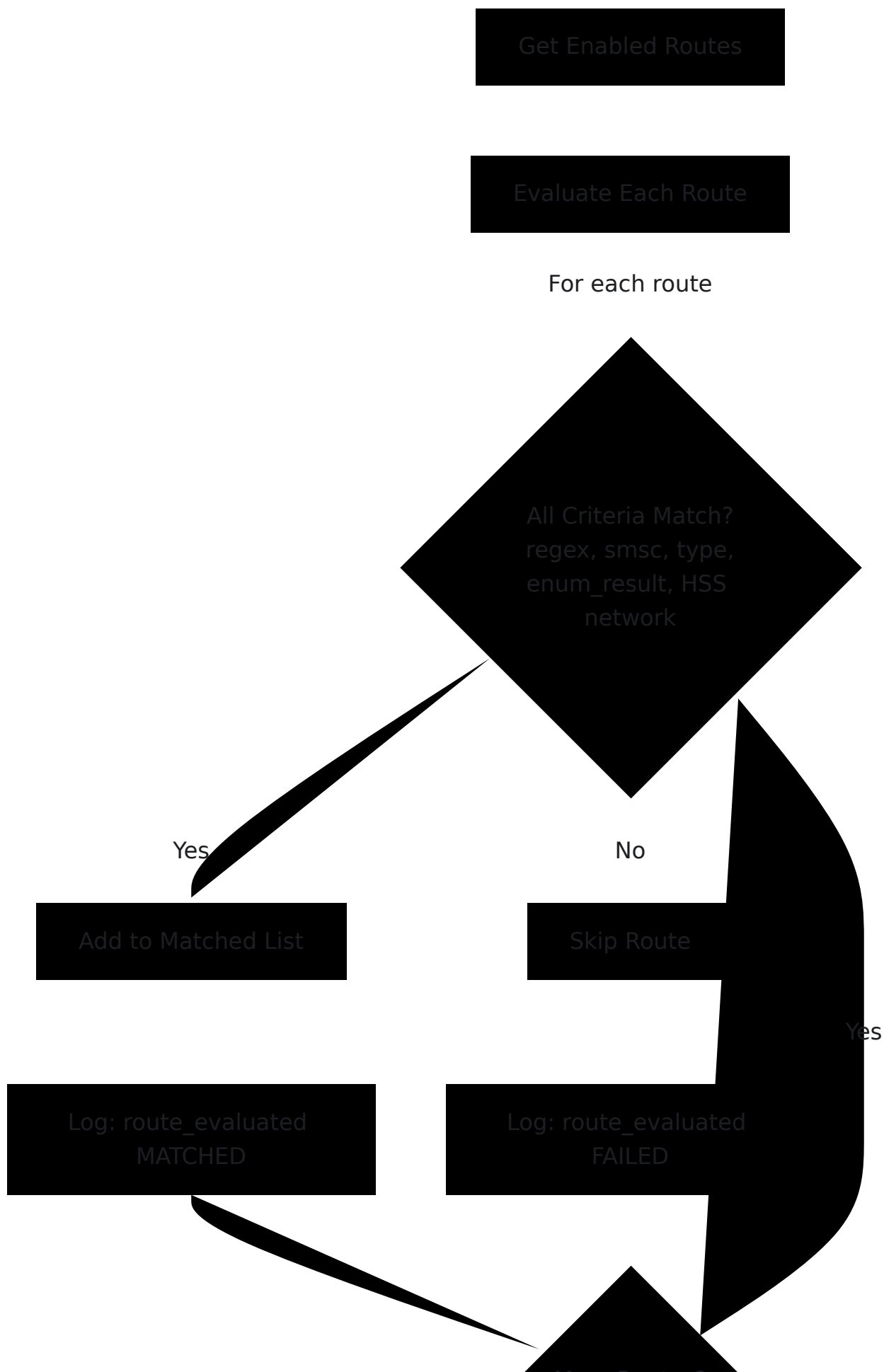
Example event log entries:

```
Route #4 [saf] MATCHED: P200 "Hold on-net" |
  calling_regex: ✓ (wildcard), called_regex: ✓ (wildcard),
  source_smsc: ✓ (wildcard), source_type: ✓ (wildcard),
  enum_result: ✓ (wildcard),
  term_network: ✓ (terminating requires on_net, +15551234 is on-
net)
```

```
Route #3 [route] FAILED: P255 "Sinch fallback" → sinch.smpp |
  calling_regex: ✓ (wildcard), called_regex: ✓ (wildcard),
  source_smsc: ✓ (wildcard), source_type: ✗ (ims vs
circuit_switched),
  enum_result: ✓ (wildcard), term_network: ✓ (any)
```

These events are visible in the control panel's message activity log and via the Events API.

Route Evaluation Algorithm





The HSS on-net/off-net check is part of **criteria matching** (a route that fails it simply does not match) — there is no separate post-selection HSS step.

Specificity Scoring

When multiple routes match, specificity determines which is preferred within the same priority level:

Criterion	Points
<code>called_regex</code> pattern length	length × 100
<code>calling_regex</code> pattern length	length × 50
<code>source_smsc</code> specified	+25
<code>enum_result_domain</code> specified	+15
<code>source_type</code> specified	+10
<code>enum_domain</code> specified	+5

Longer, more specific patterns always win over wildcards.

Configuration

Loading Routes from Config

Routes defined in `config/runtime.exs` under `:sms_routes` are loaded on first startup when the routing table is empty. Subsequent restarts preserve routes added via API.

```
config :sms_c, :sms_routes, [
  %{
    called_regex: ~r/^68987/,
    dest_smsc: "pacific-gw",
    action: :route,
    priority: 100,
    description: "Pacific Islands routing"
  },
  %{
    action: :saf,
    terminating_network: :on_net,
    priority: 200,
    description: "Hold on-net subscribers for store-and-forward"
  },
  %{
    action: :auto_reply,
    called_regex: ~r/^911$/,
    auto_reply_message: "Emergency: Please dial 911.",
    priority: 1,
    description: "911 auto-response"
  }
]
```


Route Configuration Parameters

Parameter	Type	Required	Default	Description
<code>action</code>	Atom/String	No	<code>:route</code>	Route action <code>:route</code> , <code>:call</code> , <code>:drop</code> , <code>:auto_reply</code>
<code>calling_regex</code>	Regex	No	<code>nil</code>	Regex pattern for source number matches all
<code>called_regex</code>	Regex	No	<code>nil</code>	Regex pattern for destination <code>nil</code> match
<code>source_smsc</code>	String	No	<code>nil</code>	Filter by source SMSC name matches all
<code>dest_smsc</code>	String	Conditional	<code>nil</code>	Required for Optional for Not used by <code>drop</code> / <code>auto_reply</code>
<code>source_type</code>	Atom	No	<code>nil</code>	Filter by source type: <code>:ims</code> , <code>:circuit_switched</code> , <code>:smpp</code>
<code>enum_domain</code>	String	No	<code>nil</code>	ENUM domain for the route specificity; scope the call
<code>enum_result_domain</code>	String	No	<code>nil</code>	Match only ENUM NAP

Parameter	Type	Required	Default	Description
				resolves to domain. <code>nil</code> matches all
<code>auto_reply_message</code>	String	Conditional	<code>nil</code>	Required when action is <code>:auto_reply</code>
<code>charged</code>	Atom	No	<code>:default</code>	Charging point <code>:yes</code> , <code>:no</code> , <code>:default</code>
<code>weight</code>	Integer	No	<code>100</code>	Load balance weight (1-255)
<code>priority</code>	Integer	No	<code>100</code>	Evaluation (1-255, low to high)
<code>description</code>	String	No	<code>" "</code>	Human-readable description
<code>enabled</code>	Boolean	No	<code>true</code>	Whether the service is active
<code>originating_network</code>	Atom	No	<code>:any</code>	Source MSI status requirement <code>:any</code> , <code>:on_net</code> , <code>:off_net</code>
<code>terminating_network</code>	Atom	No	<code>:any</code>	Destination HSS status requirement <code>:on_net</code> , <code>:off_net</code>

HSS Cache Configuration

Parameter	Type	Default	Description
<code>hss_cache_ttl_seconds</code>	Integer	<code>7200</code>	Time-to-live for cached HSS lookup results (seconds). Set in <code>runtime.exs</code>

API

Create Route

POST /api/routes

```
{
  "action": "saf",
  "terminating_network": "on_net",
  "priority": 200,
  "description": "Hold on-net subscribers for store-and-forward"
}
```

List Routes

GET /api/routes

Update Route

PUT /api/routes/:id

Delete Route

```
DELETE /api/routes/:id
```

Import/Export

```
POST /api/routes/import  
GET /api/routes/export
```

Import supports `merge` (add to existing) and `replace` (clear and replace all) modes.

Troubleshooting

No Route Found

Symptoms: Message stays pending with `dest_smsc: null`, event log shows `sms_routing_failed`

Possible causes:

- No enabled routes match the message criteria
- Route regex patterns don't match the number format (e.g. missing country code)
- Every candidate route has a `terminating_network/originating_network` requirement the HSS did not satisfy (off-net, or an HSS error under fail-closed), and no `:any` fallback route exists

Resolution:

1. Check the message's event log — every route evaluated is logged with pass/fail per criterion (including the HSS network check)
2. Use the routing simulator in the control panel to test with the same numbers

3. Ensure a catch-all fallback route (`originating_network: :any`, `terminating_network: :any`) exists at a high priority number (e.g. 255)

On-Net Subscriber Routed Off-Net

Symptoms: Messages to on-net subscribers are delivered via an external gateway instead of being held/delivered locally

Possible causes:

- No route with `terminating_network: :on_net` configured to capture on-net destinations
- The on-net route has a higher priority *number* (lower precedence) than the off-net fallback route
- HSS lookup failing (check `hss_dip_error` / Sh telemetry) — under fail-closed the on-net route does not match and traffic falls through to the fallback
- Diameter not enabled in configuration

Resolution:

1. Add a route with `terminating_network: :on_net` at a lower priority number than the off-net fallback
2. Verify Diameter is enabled and HSS peers are connected via the control panel
3. Check the HSS cache status and recent dip results in the event log

HSS Cache Stale

Symptoms: Subscriber ported or provisioned but still routing based on old on-net/off-net status

Resolution:

1. Re-route the message via the API — forced re-routing bypasses the HSS cache
2. Wait for cache TTL expiry (default 2 hours)
3. Reduce `hss_cache_ttl_seconds` if faster updates are needed

SMS-C Troubleshooting Guide

[← Back to Documentation Index](#)

Comprehensive guide for diagnosing and resolving common SMS-C issues.

Table of Contents

- [Diagnostic Tools](#)
- [Message Delivery Issues](#)
- [Routing Problems](#)
- [Performance Issues](#)
- [Database Problems](#)
- [Frontend Connection Issues](#)
- [Charging/Billing Issues](#)
- [ENUM Lookup Problems](#)
- [Cluster Issues](#)
- [API Problems](#)
- [Web UI Issues](#)
- [System Resource Issues](#)

Diagnostic Tools

Quick Health Check

```
# 1. Check API status
curl https://api.example.com:8443/api/status

# 2. Check Prometheus metrics endpoint
curl https://api.example.com:9568/metrics | grep sms_c

# 3. Check application logs
tail -f /var/log/sms_c/application.log

# 4. Check process status
systemctl status sms_c

# 5. Check SQL CDR database connectivity (MySQL/MariaDB)
mysql -u sms_user -p -h db.example.com -e "SELECT 1"

# For PostgreSQL:
# psql -U sms_user -h db.example.com -d sms_c_prod -c "SELECT 1"
```

Log Analysis

View Recent Errors:

```
# Last 100 error-level log entries
tail -1000 /var/log/sms_c/application.log | grep "\[error\]"

# Search for specific error patterns
grep "routing_failed" /var/log/sms_c/application.log

# Find SQL database errors
grep -i "database\|sql\|ecto" /var/log/sms_c/application.log |
grep error
```

Monitor Logs in Real-Time:

```
# Follow logs with filter
tail -f /var/log/sms_c/application.log | grep -E "
(error|warning|critical)"
```

Metric Queries

Check Message Processing Rate:

```
# Messages per second
rate(sms_c_message_received_count[5m])

# Delivery success rate
rate(sms_c_delivery_succeeded_count[5m]) /
rate(sms_c_delivery_queued_count[5m])
```

Check Queue Status:

```
# Current queue depth
sms_c_queue_size_pending

# Oldest message age (seconds)
sms_c_queue_oldest_message_age_seconds
```

Check System Performance:

```
# Message processing latency (p95)
histogram_quantile(0.95,
sms_c_message_processing_stop_duration_bucket)

# Routing latency (p95)
histogram_quantile(0.95, sms_c_routing_stop_duration_bucket)
```


Message Delivery Issues

Messages Not Being Delivered

Symptoms:

- Messages stuck in "queued"/"backoff" status
- High undelivered message count
- No delivery notifications

Diagnostic Steps:

1. Check Frontend Connections:

```
curl https://api.example.com:8443/api/frontends/active
```

Expected: List of active frontends
Problem: Empty list or missing expected frontends

2. Check Message Queue:

Access Web UI: `/message_queue`

- Filter by status: "queued" or "backoff"
- Check `dest_smsc` value
- Verify `deliver_after` is not in future

3. Check Routing:

Access Web UI: `/simulator`

- Test with actual message parameters
- Verify route matches and destination is correct

4. Check Frontend Polling:

Review frontend system logs:

- Is frontend querying `/api/messages`?

- Is frontend sending `smsc` header correctly?

Solutions:

No Frontends Connected:

```
# Check frontend system status
systemctl status frontend_service

# Verify frontend can reach API
curl -k https://api.example.com:8443/api/status

# Manually register frontend
curl -X POST https://api.example.com:8443/api/frontends/register \
  -H "Content-Type: application/json" \
  -d '{
    "frontend_name": "test_gateway",
    "frontend_type": "smp",
    "ip_address": "10.0.1.50"
  }'
```

Messages Routed to Wrong SMSC:

- Review routing configuration
- Check route priorities
- Test in routing simulator
- Verify frontend name matches `dest_smsc` in messages

Messages Scheduled for Future:

- Check `deliver_after` timestamp
- Reset if needed:

```
curl -X PATCH https://api.example.com:8443/api/messages/12345 \
  -H "Content-Type: application/json" \
  -d '{"deliver_after": "2025-10-30T12:00:00Z"}'
```

Messages Failing with Retries

Symptoms:

- `delivery_attempts` counter increasing
- Messages with high attempt count (> 3)
- Exponential backoff delays

Diagnostic Steps:

1. Check Event Log:

```
curl https://api.example.com:8443/api/events/12345
```

Look for:

- Delivery failure events
- Error descriptions
- Retry timestamps

2. Check Frontend Logs:

- Why is frontend failing to deliver?
- Network errors?
- Protocol errors?
- Downstream system unavailable?

Solutions:

Temporary Network Issues:

- Wait for retry (automatic)
- Monitor for successful delivery

Persistent Failures:

```
# Route to alternate gateway
curl -X PATCH https://api.example.com:8443/api/messages/12345 \
  -H "Content-Type: application/json" \
  -d '{"dest_smsc": "backup_gateway"}'

# Reset retry counter
curl -X PATCH https://api.example.com:8443/api/messages/12345 \
  -H "Content-Type: application/json" \
  -d '{"delivery_attempts": 0, "deliver_after": "2025-10-30T12:00:00Z"}'
```

Invalid Destination Number:

- Verify number format
- Check number translation rules
- Delete message if truly invalid

Dead Letter Messages

Symptoms:

- `deadletter: true` in message
- Messages past expiration time
- Status still "queued"/"backoff" (failed deliveries are not dead-lettered; they stop being served once past `expires`)

Diagnostic Steps:

1. Find Dead Letter Messages:

Access Web UI: `/message_queue`

- Filter by expired status
- Check expiration timestamps

2. Check Why Expired:

- Review event log
- Check delivery attempt history

- Verify routing was successful

Solutions:

Extend Expiration:

```
# Add 24 hours to expiration
curl -X PATCH https://api.example.com:8443/api/messages/12345 \
  -H "Content-Type: application/json" \
  -d '{"expires": "2025-10-31T12:00:00Z", "deadletter": false}'
```

Routing Problems

No Route Found

Symptoms:

- Error: `no_route_found`
- `sms_c_routing_failed_count` metric increasing
- Event log shows "routing_failed"

Diagnostic Steps:

1. Check Routes Exist:

Access Web UI: `/sms_routing`

- Verify routes are configured
- Check at least one route is enabled

2. Test Routing:

Access Web UI: `/simulator`

- Enter message parameters (calling number, called number, source SMSC)
- Review evaluation results
- Check why routes didn't match

3. Check Route Criteria:

- Prefix matches required?
- Source SMSC filter too restrictive?
- All routes disabled?

Solutions:

No Routes Configured:

Add catch-all route:

```
Calling Prefix: (empty)
Called Prefix: (empty)
Source SMSC: (empty)
Dest SMSC: default_gateway
Priority: 255
Weight: 100
Enabled: ✓
Description: Catch-all default route
```

Routes Too Specific:

Add broader route:

```
Called Prefix: +
Dest SMSC: international_gateway
Priority: 200
Weight: 100
Enabled: ✓
Description: International catch-all
```

All Routes Disabled:

- Enable appropriate routes via Web UI
- Check configuration didn't accidentally disable routes

Wrong Route Selected

Symptoms:

- Messages routed to unexpected destination
- Wrong gateway receiving traffic
- Load balance not distributing as expected

Diagnostic Steps:

1. Use Routing Simulator:

Access Web UI: `/simulator`

- Test with actual message parameters
- Review "All Matches" section
- Check priority and specificity scores

2. Check Route Priorities:

- Lower number = higher priority
- Routes evaluated in priority order
- Within same priority, weights apply

3. Check Route Specificity:

Specificity scoring:

- Longer called prefix: +100 points per character
- Longer calling prefix: +50 points per character
- Source SMSC specified: +25 points
- Source type specified: +10 points
- ENUM domain specified: +15 points

Solutions:

Adjust Priorities:

Make specific route higher priority:

Premium Route:

Called Prefix: +1555

Priority: 10 (high priority)

General Route:

Called Prefix: +1

Priority: 50 (lower priority)

Adjust Weights:

Change load balance distribution:

Primary (70%):

Weight: 70

Backup (30%):

Weight: 30

Add More Specific Route:

Override general route for specific case:

Specific Route:

Called Prefix: +15551234

Dest SMSC: dedicated_gateway

Priority: 1

General Route:

Called Prefix: +1

Dest SMSC: general_gateway

Priority: 50

Auto-Reply Not Working

Symptoms:

- Auto-reply route configured but not triggering
- No reply messages being sent

- Event log missing auto-reply event

Diagnostic Steps:

1. Check Route Configuration:

- `action: :auto_reply`
- `auto_reply_message` contains text
- Route is enabled
- Route matches message criteria

2. Test in Simulator:

- Verify route is selected
- Check for "auto_reply" indication

3. Check Event Log:

```
curl https://api.example.com:8443/api/events/12345 | grep  
auto_reply
```

Solutions:

Route Not Matching:

- Broaden criteria (remove filters)
- Check priority (should be higher than normal routes)
- Verify enabled status

Message Not Set:

Edit route, add message:

```
Auto-Reply: ✓  
Auto-Reply Message: "Thank you for your message. We will respond  
soon."
```

Wrong Priority:

Auto-reply routes should have high priority (low number):

Auto-Reply Route:

Priority: 10

Normal Route:

Priority: 50

Performance Issues

High Message Processing Latency

Symptoms:

- `sms_c_message_processing_stop_duration` p95 > 1000ms
- Slow API responses
- Queue building up

Diagnostic Steps:

1. Check Component Latencies:

```
# Routing latency
histogram_quantile(0.95, sms_c_routing_stop_duration_bucket)

# ENUM lookup latency
histogram_quantile(0.95, sms_c_enum_lookup_stop_duration_bucket)

# Charging latency
histogram_quantile(0.95, sms_c_charging_succeeded_duration_bucket)

# Delivery latency
histogram_quantile(0.95, sms_c_delivery_succeeded_duration_bucket)
```

2. Check System Resources:

```
# CPU usage
top -b -n 1 | grep sms_c
```

```
# Memory usage
ps aux | grep beam.smp
```

****Solutions**:**

****Routing Slow**** (Many routes):

- Reduce number of enabled routes
- Combine similar routes
- Optimize route criteria

****ENUM Lookups Slow**:**

- Check DNS server latency
- Increase timeout
- Use faster/closer DNS servers
- Disable ENUM if not needed

****Charging Slow**:**

- Check OCS performance
- Increase OCS timeout
- Disable charging if not needed
- Use async charging

****Database Slow**:**

- Increase connection pool size
- Add indexes
- Optimize queries
- Upgrade database resources

****Configuration Changes**:**

```
```elixir
config/config.exs
Increase batch size for throughput
config :sms_c,
 batch_insert_batch_size: 200,
 batch_insert_flush_interval_ms: 200

Increase database pool
```

```
config :sms_c, SmsC.Repo,
 pool_size: 50
```

## Low Message Throughput

### Symptoms:

- Processing < 100 msg/sec
- Using async API but still slow
- High API response times

### Diagnostic Steps:

#### 1. Check Batch Worker:

```
In production console (iex)
SmsC.Messaging.BatchInsertWorker.stats()
```

Look for:

- `current_queue_size` near max
- `flush_errors` > 0
- `last_flush_duration_ms` very high

#### 2. Check Bottlenecks:

```
Database query time
ecto_pools_query_time

Connection pool queue time
ecto_pools_queue_time
```

### Solutions:

#### Database Bottleneck:

Increase pool size:

```
config :sms_c, SmsC.Repo,
 pool_size: 50 # Increase from 20
```

### Batch Configuration:

Tune for throughput:

```
config :sms_c,
 batch_insert_batch_size: 200, # Larger batches
 batch_insert_flush_interval_ms: 200 # Longer interval
```

### Use Async Endpoint:

```
High throughput: use /create_async
curl -X POST
https://api.example.com:8443/api/messages/create_async

NOT: /api/messages (synchronous)
```

## Queue Backlog Growing

### Symptoms:

- `sms_c_queue_size_pending` increasing
- Oldest message age increasing
- Processing can't keep up with incoming rate

### Diagnostic Steps:

#### 1. Check Incoming vs Delivery Rate:

```
Incoming rate
rate(sms_c_message_received_count[5m])

Delivery rate
rate(sms_c_delivery_succeeded_count[5m])
```

## 2. Check Frontend Capacity:

- Are frontends polling frequently enough?
- Are frontends processing messages fast enough?
- Any frontend errors?

## 3. Check Delivery Success Rate:

```
rate(sms_c_delivery_succeeded_count[5m]) /
rate(sms_c_delivery_attempted_count[5m])
```

### Solutions:

#### Frontends Not Polling:

- Check frontend connectivity
- Verify polling interval (should be 5-10 seconds)
- Restart frontend services

#### Frontends Too Slow:

- Add more frontend instances
- Optimize frontend processing
- Increase frontend concurrency

#### High Retry Rate:

- Investigate delivery failures
- Fix downstream issues
- Route to alternate gateways

#### Temporary Spike:

- Wait for queue to drain
- Monitor until normal
- Consider capacity upgrades if recurring

# Database Problems

## Connection Failures

### Symptoms:

- Error: "unable to connect to database"
- API returning 500 errors
- Application won't start

### Diagnostic Steps:

#### 1. Check SQL CDR Database Status:

```
MySQL/MariaDB
systemctl status mysql

PostgreSQL
systemctl status postgresql

Test connectivity (MySQL/MariaDB)
mysql -u sms_user -p -h db.example.com -e "SELECT 1"

Test connectivity (PostgreSQL)
psql -U sms_user -h db.example.com -d sms_c_prod -c "SELECT 1"
```

#### 2. Check Network:

```
Ping database host
ping db.example.com

Check port connectivity (MySQL/MariaDB: 3306, PostgreSQL: 5432)
telnet db.example.com 3306
or
telnet db.example.com 5432
```

#### 3. Check Credentials:

```
Verify environment variables
echo $DB_USERNAME
echo $DB_HOSTNAME
echo $DB_PORT

Try manual connection with same credentials (MySQL/MariaDB)
mysql -u $DB_USERNAME -p$DB_PASSWORD -h $DB_HOSTNAME

For PostgreSQL:
psql -U $DB_USERNAME -h $DB_HOSTNAME -d sms_c_prod
```

## **Solutions:**

### **Database Down:**

```
Start database (MySQL/MariaDB)
systemctl start mysql

Start database (PostgreSQL)
systemctl start postgresql
```

### **Wrong Credentials:**

Update configuration:

```
export DB_USERNAME=correct_user
export DB_PASSWORD=correct_password

Restart application
systemctl restart sms_c
```

### **Network Issue:**

- Check firewall rules
- Verify security groups (cloud)
- Check VPN/network connectivity

### **Connection Pool Exhausted:**



Increase pool size:

```
config :sms_c, SmsC.Repo,
 pool_size: 50 # Increase from current value
```

## Slow Queries

### Symptoms:

- Database query time high
- API responses slow
- Connection pool queue building up

### Diagnostic Steps:

#### 1. Check Slow Query Log:

```
-- MySQL/MariaDB: Enable slow query log
SET GLOBAL slow_query_log = 'ON';
SET GLOBAL long_query_time = 1; -- Log queries > 1 second

-- View slow queries (MySQL/MariaDB)
SELECT * FROM mysql.slow_log ORDER BY query_time DESC LIMIT 10;

-- PostgreSQL: Enable slow query log in postgresql.conf
-- log_min_duration_statement = 1000 # milliseconds
-- Then check PostgreSQL logs
```

#### 2. Check Missing Indexes:

```
-- Check table indexes
SHOW INDEX FROM message_queues;

-- Expected indexes:
-- - source_smsc
-- - dest_smsc
-- - send_time
-- - inserted_at
```

### 3. Check Table Stats:

```
-- Table sizes (MySQL/MariaDB)
SELECT
 table_name,
 table_rows,
 ROUND(data_length / 1024 / 1024, 2) AS data_mb,
 ROUND(index_length / 1024 / 1024, 2) AS index_mb
FROM information_schema.tables
WHERE table_schema = 'sms_c_prod';

-- Table sizes (PostgreSQL)
-- SELECT schemaname, tablename,
--
pg_size_pretty(pg_total_relation_size(schemaname||'.'||tablename))
AS size
-- FROM pg_tables WHERE schemaname = 'public';
```

### Solutions:

#### Missing Indexes:

```
CREATE INDEX idx_message_queues_source_smsc ON
message_queues(source_smsc);
CREATE INDEX idx_message_queues_dest_smsc ON
message_queues(dest_smsc);
CREATE INDEX idx_message_queues_send_time ON
message_queues(send_time);
CREATE INDEX idx_message_queues_status ON message_queues(status);
```

#### Table Fragmentation:

```
-- MySQL/MariaDB
OPTIMIZE TABLE message_queues;
OPTIMIZE TABLE frontend_registrations;

-- PostgreSQL
-- VACUUM ANALYZE message_queues;
-- VACUUM ANALYZE frontend_registrations;
```

## Too Much Data:

Clean up old records:

```
-- Delete delivered messages older than 30 days
DELETE FROM message_queues
WHERE status = 'delivered'
AND deliver_time < DATE_SUB(NOW(), INTERVAL 30 DAY)
LIMIT 10000;
```

## Disk Space Full

### Symptoms:

- Error: "Disk full"
- Cannot write to database
- Application crashes

### Diagnostic Steps:

#### 1. Check Disk Usage:

```
df -h

Check SQL database directory (MySQL/MariaDB)
du -sh /var/lib/mysql

Check SQL database directory (PostgreSQL)
du -sh /var/lib/postgresql
```

#### 2. Find Large Files:

```
Find largest files (MySQL/MariaDB)
find /var/lib/mysql -type f -exec du -h {} + | sort -rh
| head -20

Find largest files (PostgreSQL)
find /var/lib/postgresql -type f -exec du -h {} + | sort
-rh | head -20

Check log files
du -sh /var/log/sms_c/*
```

## Solutions:

### Clean Old Data:

```
-- Delete old messages
DELETE FROM message_queues
WHERE inserted_at < DATE_SUB(NOW(), INTERVAL 90 DAY)
LIMIT 100000;
```

### Rotate Logs:

```
Force logrotate
logrotate -f /etc/logrotate.d/sms_c

Clear old log files
find /var/log/sms_c -name "*.log.*" -mtime +30 -delete
```

### Expand Disk:

- Resize volume (cloud)
- Add new disk and extend volume
- Move data to larger disk

# Frontend Connection Issues

## Frontend Not Showing as Active

### Symptoms:

- Frontend status shows "expired"
- Frontend not in active list
- Messages not being delivered to frontend

### Diagnostic Steps:

#### 1. Check Registration:

```
curl https://api.example.com:8443/api/frontends/active | grep frontend_name
```

#### 2. Check Frontend Logs:

- Is frontend calling `/api/frontends/register`?
- Any API errors?
- Registration frequency (should be every 60s)

#### 3. Check API Logs:

```
grep "frontend.*register" /var/log/sms_c/application.log | tail -20
```

### Solutions:

#### Frontend Not Registering:

Test manual registration:

```
curl -X POST https://api.example.com:8443/api/frontends/register \
-H "Content-Type: application/json" \
-d '#123;
 "frontend_name": "uk_gateway",
 "frontend_type": "smpp",
 "ip_address": "10.0.1.50"
}'
```

If successful, problem is in frontend code/configuration.

### Registration Timing Out:

Frontends expire after 90 seconds. Ensure registration every 60 seconds:

```
Frontend should call register every 60 seconds
while True:
 register_with_smsc()
 time.sleep(60)
```

### Network Issues:

- Check firewall between frontend and API
- Verify DNS resolution
- Test with curl from frontend server

## Frontend Repeatedly Connecting/Disconnecting

### Symptoms:

- Frontend status flipping between active/expired
- High registration count in history
- Unstable connection

### Diagnostic Steps:

#### 1. Check Frontend Health:

- Is frontend process stable?
- Any crashes or restarts?
- Resource issues (CPU/memory)?

## 2. Check Network Stability:

```
Check packet loss
ping -c 100 api.example.com

Check connection resets
netstat -s | grep -i reset
```

## 3. Check Registration Timing:

- Too frequent? (every few seconds)
- Too infrequent? ( > 90 seconds)

### Solutions:

#### Frontend Unstable:

- Fix frontend application issues
- Increase frontend resources
- Check frontend logs for errors

#### Network Issues:

- Check for intermittent connectivity
- Review firewall logs
- Check load balancer health checks

#### Wrong Registration Interval:

Correct interval:

```
REGISTRATION_INTERVAL = 60 # seconds
```

# Charging/Billing Issues

## Charging Failures

Charging is performed over **Diameter Ro** (Credit-Control). Note charging **fails open**: an OCS error or an unconfigured Diameter stack logs a failure but still delivers the message. Only a definitive out-of-balance answer rejects a message (status `:balance_rejected`).

### Symptoms:

- `sms_c_charging_failed_count` increasing
- Event log shows "charging\_failed" or "balance\_rejected"
- Messages marked with status `:balance_rejected` (out of balance), or log warnings about the Diameter stack not being enabled

### Diagnostic Steps:

1. **Check the Diameter Ro peer state** (CER/CEA established to the DRA/OCS):

```
In a remote console
:diameter.service_info(:omnimessage, :connections)
```

2. **Confirm the stack is enabled and the `:ro` application is registered:**

```
grep -E "diameter_enabled|application_name: :ro"
config/runtime.exs
```

A log line `Charging requested but the Diameter Ro stack is not enabled/configured` means charging is being skipped (messages allowed) — enable the stack.

3. **Check the rating profile** matches the rating context configured on the OCS:



```
grep rating_profile config/runtime.exs
```

## Solutions:

### OCS / DRA Unreachable:

- Verify IP reachability and that the DRA has this node provisioned as a Diameter peer (Origin-Host/Realm).
- Confirm the CCR `Service-Context-Id` (`rating_profile`) matches the OCS's SMS rating profile.

### Out-of-balance handling (`config :sms_c, :charging`):

```
config :sms_c, :charging,
 out_of_balance_action: :reject, # or :auto_reply
 out_of_balance_reply_text: "...",
 rating_profile: "sms@sms-c"
```

### Disable Charging Temporarily:

```
config :sms_c,
 default_charging_enabled: false
```

Restart application.

## Account Issues:

- Check account exists in OCS
- Verify account has balance
- Check rating plans are configured

## Charging Too Slow

### Symptoms:

- `sms_c_charging_succeeded_duration` p95 > 500ms

- Message processing slow when charging enabled
- Fast when charging disabled

## Diagnostic Steps:

### 1. Check Charging Latency:

```
histogram_quantile(0.95, sms_c_charging_succeeded_duration_bucket)
```

2. **Check OCS / DRA performance** (Diameter Ro round-trip). High `sms_c_charging_succeeded_duration` usually points at OCS rating latency or a slow/flapping DRA peer.

### 3. Check Network Latency to the DRA:

```
Ping the DRA / OCS host
ping -c 10 dra01.example.com
```

## Solutions:

### OCS Slow:

- Optimize OCS configuration
- Add OCS resources
- Use faster rating engine

### Network Latency:

- Deploy OCS closer to SMS-C
- Use direct network path
- Avoid VPN/tunnels if possible

### Timeout Too Low:

Increase timeout:

```
config :sms_c,
 ocs_timeout: 5000 # 5 seconds
```

# ENUM Lookup Problems

## ENUM Lookups Failing

### Symptoms:

- `sms_c_enum_lookup_stop_duration` showing failures
- Event log shows ENUM errors
- Routes with `enum_result_domain` not matching

### Diagnostic Steps:

#### 1. Check ENUM Configuration:

```
grep -A 10 "enum_" config/runtime.exs
```

#### 2. Test DNS Connectivity:

```
Test DNS server
dig @8.8.8.8 e164.arpa

Test ENUM query
For +15551234567:
dig @8.8.8.8 NAPTR 7.6.5.4.3.2.1.5.5.5.1.e164.arpa
```

#### 3. Check DNS Server:

```
Is custom DNS reachable?
ping 10.0.1.53

Test port
nc -zv 10.0.1.53 53
```

## Solutions:

### DNS Server Unreachable:

Use alternate DNS:

```
config :sms_c,
 enum_dns_servers: [
 {"8.8.8.8", 53}, # Google Public DNS
 {"1.1.1.1", 53} # Cloudflare DNS
]
```

### ENUM Domain Wrong:

Update domain:

```
config :sms_c,
 enum_domains: ["e164.arpa"] # Use standard domain
```

### Timeout Too Short:

Increase timeout:

```
config :sms_c,
 enum_timeout: 10000 # 10 seconds
```

### Disable ENUM (if not needed):

```
config :sms_c,
 enum_enabled: false
```

## ENUM Cache Issues

### Symptoms:

- Low cache hit rate (< 70%)

- Cache size growing unbounded
- Memory usage high

## **Diagnostic Steps:**

### **1. Check Cache Stats:**

```
Cache hit rate
rate(sms_c_enum_cache_hit_count[5m]) /
(rate(sms_c_enum_cache_hit_count[5m]) +
rate(sms_c_enum_cache_miss_count[5m]))

Cache size
sms_c_enum_cache_size_size
```

### **2. Check Traffic Pattern:**

- Are numbers repeating?
- Cache TTL appropriate?

## **Solutions:**

### **Low Hit Rate** (Expected):

- Traffic to unique numbers (normal)
- Monitor but don't alarm if < 70%

### **Cache Growing:**

Clear cache via NAPTR Test page or restart application.

### **High Memory Usage:**

- Expected with large cache
- Monitor overall system memory
- Consider TTL adjustment

# Cluster Issues

## Node Can't Join Cluster

### Symptoms:

- Single node running
- Cluster queries returning only local results
- Erlang distribution errors

### Diagnostic Steps:

#### 1. Check Node Names:

```
In IEx console
Node.self()
Expected: :sms@node1.example.com

Node.list()
Expected: List of other nodes
```

#### 2. Check Erlang Cookie:

```
Check cookie file
cat ~/.erlang.cookie

Verify same on all nodes
```

#### 3. Check Network:

```
Can nodes reach each other?
ping node2.example.com

Check ports
nc -zv node2.example.com 4369
nc -zv node2.example.com 9100-9200
```

## Solutions:

### Cookie Mismatch:

Set same cookie on all nodes:

```
export ERLANG_COOKIE=same_secret_value_here

Or update ~/.erlang.cookie
echo "same_secret_value_here" > ~/.erlang.cookie
chmod 400 ~/.erlang.cookie
```

### Firewall Blocking:

Open required ports:

```
EPMD
iptables -A INPUT -p tcp --dport 4369 -j ACCEPT

Erlang distribution
iptables -A INPUT -p tcp --dport 9100:9200 -j ACCEPT
```

### DNS Issues:

Use IP addresses instead of hostnames:

```
config :sms_c,
 cluster_nodes: [
 : "sms@10.0.1.10",
 : "sms@10.0.1.11"
]
```

## Cluster Split Brain

### Symptoms:

- Nodes running but disconnected
- Different data on different nodes

- Mnesia inconsistencies

## Diagnostic Steps:

### 1. Check Node Connectivity:

```
On each node (IEx)
Node.list()
```

### 2. Check Mnesia:

```
:mnesia.system_info(:running_db_nodes)
```

## Solutions:

### Reconnect Nodes:

```
Stop all nodes
systemctl stop sms_c

Start one node first
systemctl start sms_c # On node1

Wait for it to fully start, then start others
systemctl start sms_c # On node2
systemctl start sms_c # On node3
```

### Mnesia Inconsistency:

- Export routes from correct node
- Stop all nodes
- Delete Mnesia directory
- Start nodes
- Import routes



# API Problems

## API Not Responding

### Symptoms:

- Connection timeout
- Connection refused
- No response

### Diagnostic Steps:

#### 1. Check API Process:

```
Is application running?
systemctl status sms_c

Check listening ports
netstat -tlnp | grep 8443
```

#### 2. Check Firewall:

```
Check iptables
iptables -L -n | grep 8443

Test local connectivity
curl -k https://localhost:8443/api/status
```

#### 3. Check TLS Configuration:

```
Check certificate exists
ls -l priv/cert/server.crt priv/cert/server.key

Check certificate validity
openssl x509 -in priv/cert/server.crt -noout -dates
```

### Solutions:

## Application Not Running:

```
systemctl start sms_c
```

## Firewall Blocking:

```
Allow API port
iptables -A INPUT -p tcp --dport 8443 -j ACCEPT
```

## Certificate Issues:

Generate new certificate (see Configuration Guide).

## Wrong Port:

Check configuration:

```
grep "port:" config/runtime.exs
```

# API Returning 500 Errors

## Symptoms:

- Internal Server Error
- 500 status code
- Error in logs

## Diagnostic Steps:

### 1. Check Application Logs:

```
tail -100 /var/log/sms_c/application.log | grep "\[error\]"
```

### 2. Check Database:

```
mysql -u sms_user -p -e "SELECT 1"
```

### 3. Check Resources:

```
Memory
free -h

CPU
top -b -n 1

Disk
df -h
```

### Solutions:

#### Database Unavailable:

- Start database
- Fix connection configuration

#### Out of Memory:

- Restart application
- Increase system memory
- Check for memory leaks

#### Application Error:

- Check specific error in logs
- Fix configuration issue
- Restart application

# Web UI Issues

## Can't Access Web UI

### Symptoms:

- Connection timeout
- 404 Not Found
- Page won't load

### Diagnostic Steps:

#### 1. Check Application Status:

```
systemctl status sms_c
```

#### 2. Check Port:

```
netstat -tlnp | grep 80
```

#### 3. Check URL:

- Correct hostname?
- Correct port?
- HTTP vs HTTPS?

### Solutions:

#### Wrong Port:

Check configuration:

```
grep "control_panel" config/runtime.exs
```

Access on correct port (default: 80 or 4000).

## Application Not Running:

```
systemctl start sms_c
```

## Firewall:

```
iptables -A INPUT -p tcp --dport 80 -j ACCEPT
```

# LiveView Not Updating

## Symptoms:

- Page loads but doesn't update
- Data is stale
- WebSocket errors in browser console

## Diagnostic Steps:

### 1. Check Browser Console:

- Open Developer Tools (F12)
- Look for WebSocket errors
- Check network tab for failed requests

### 2. Check Proxy Configuration:

If using reverse proxy, ensure WebSocket support:

```
location /live {
 proxy_http_version 1.1;
 proxy_set_header Upgrade $http_upgrade;
 proxy_set_header Connection "upgrade";
};
```

## Solutions:

### WebSocket Blocked:

- Configure proxy for WebSocket
- Check firewall
- Check browser extensions

### **Refresh Page:**

- Hard refresh (Ctrl+F5)
- Clear browser cache

# **System Resource Issues**

## **High CPU Usage**

### **Symptoms:**

- CPU consistently > 80%
- System slow
- Application unresponsive

### **Diagnostic Steps:**

#### **1. Check Process:**

```
top -b -n 1 | grep beam.smp
```

#### **2. Check Metrics:**

```
Message processing rate
rate(sms_c_message_received_count[5m])

Routing operations
rate(sms_c_routing_route_matched_count[5m])
```

### **Solutions:**

### **High Traffic:**

- Scale horizontally (add nodes)
- Scale vertically (add CPU)

### **Inefficient Routing:**

- Reduce number of routes
- Optimize route criteria

### **Too Many ENUM Lookups:**

- Check cache hit rate
- Consider disabling if not needed

## **High Memory Usage**

### **Symptoms:**

- Memory usage > 90%
- Application crashes
- Out of memory errors

### **Diagnostic Steps:**

#### **1. Check Memory:**

```
free -h

ps aux | grep beam.smp
```

#### **2. Check Cache Sizes:**

```
sms_c_enum_cache_size_size
```

### **Solutions:**

#### **ENUM Cache Too Large:**

- Clear cache

- Reduce TTL
- Disable ENUM if not needed

### **Batch Queue Growing:**

```
Check worker stats (IEx)
SmsC.Messaging.BatchInsertWorker.stats()
```

If queue is large, flush manually or restart.

### **Add Memory:**

- Scale vertically
- Add swap (temporary)

### **Memory Leak:**

- Restart application
- Report issue for investigation

---

For additional assistance, consult:

- [Operations Guide](#) - Daily procedures
- [Configuration Guide](#) - Configuration options
- [Metrics Guide](#) - Monitoring setup
- Application logs - `/var/log/sms_c/application.log`



