LoRaWAN Payload Schema Toolkit

Multi-Tech Systems, Inc.
2205 Woodale Drive
Mounds View, MN 55112
United States

LoRaWAN Payload Schema Toolkit

Document Number: MT-DOC-0001

Multi-Tech LoRaWAN Solutions

Author:

Jason Reiss (Engineering)

Version: 1.0.0

Date: February 2026

Status: FINAL

Contents

Tables

Figures

Revision History

Version	Date	Author	Description
1.0.0	February 2026	Jason Reiss	Initial release

1 Introduction

1.1 Overview

The LoRaWAN Payload Schema Toolkit provides a declarative approach to defining payload structures for LoRaWAN devices. Instead of writing custom JavaScript decoders for each device, manufacturers and integrators define payload structures once in YAML format, then automatically generate decoders for multiple platforms.

1.1.1 Key Benefits

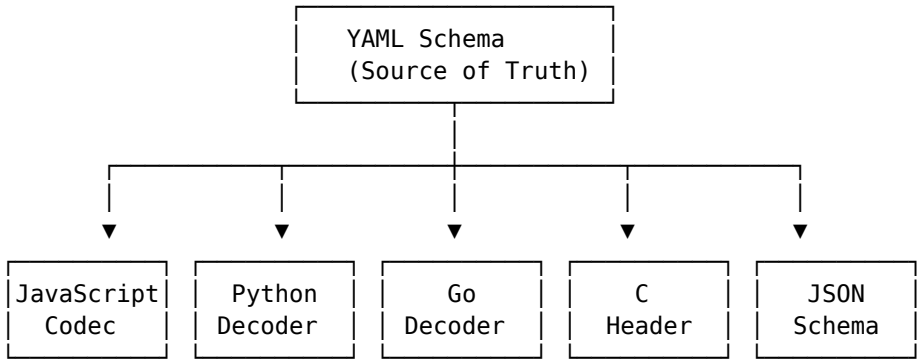
- **Security**: Declarative schemas eliminate code execution risks (no `eval()`)
- **Portability**: Same schema works across Python, JavaScript, Go, and C
- **Maintainability**: Human-readable YAML instead of complex JavaScript
- **Automation**: Generate TS013-compliant codecs, C headers, JSON schemas
- **Validation**: Built-in test vectors ensure correctness

1.1.2 What This Toolkit Provides

Component	Description
YAML Schema Language	Declarative payload definitions for binary-to-JSON decoding
Reference Interpreters	Python, JavaScript, Go, C implementations
Code Generators	Generate TS013-compliant codec code from schemas
Device Schemas	Ready-to-use schemas for Decentlab, Milesight, and more
Quality Tools	Validation, scoring, and cross-platform testing

1.2 Architecture

The toolkit follows a schema-centric architecture where the YAML definition serves as the single source of truth:



1.3 Performance and Security

1.3.1 Why Declarative Schemas?

Traditional LoRaWAN codecs use JavaScript with `eval()` or `new Function()`, creating security risks on shared infrastructure. Declarative schemas eliminate this attack surface entirely.

1.3.2 Performance Comparison

Implementation	Throughput	Security	Use Case
C Interpreter	32M msg/s	No eval	High-performance backends
Go Binary Schema	2.1M msg/s	No eval	Cloud platforms
JS Traditional	20M msg/s	eval risk	Legacy compatibility
Go YAML Schema	343K msg/s	No eval	Development
Python Schema	215K msg/s	No eval	Prototyping

1.3.3 Recommendations by Use Case

Scenario	Recommended	Why
Multi-tenant LNS	C Interpreter	32M msg/s, no code execution
Cloud platform	Go Binary Schema	2.1M msg/s, easy deployment
Edge gateway	C or QuickJS	Embedded-friendly
Development	Python/YAML	Human-readable, fast iteration

2 Quick Start

This section provides a rapid introduction to creating and using payload schemas.

2.1 Prerequisites

```
# Clone the repository
git clone https://github.com/MultiTechSystems/lorawan-payload-schema.git
cd lorawan-payload-schema

# Create Python virtual environment
python3 -m venv .venv
source .venv/bin/activate

# Install dependencies
pip install -r requirements.txt
```

2.2 5-Minute Example

2.2.1 Step 1: Create Your Schema

Create `schemas/mycompany/temp-sensor.yaml`:

```
name: mycompany_temp_sensor
version: 1
description: "MyCompany Temperature Sensor v1.0"
endian: big

fields:
  - name: temperature
    type: s16
    div: 100
    unit: "°C"

  - name: humidity
    type: u8
    unit: "%"

  - name: battery
    type: u8
    div: 10
    unit: "V"

test_vectors:
  - name: "normal_reading"
    input: "09C44121"
    output:
      temperature: 25.0
      humidity: 65
      battery: 3.3
```

Payload breakdown:

- `09C4` = 2500 → 2500/100 = 25.0°C
- `41` = 65 → 65%

- 21 = 33 → 33/10 = 3.3V

2.2.2 Step 2: Validate Your Schema

```
python tools/validate_schema.py schemas/mycompany/temp-sensor.yaml -v
```

2.2.3 Step 3: Generate Codecs

JavaScript (for TTN/ChirpStack):

```
python tools/generate_ts013_codec.py schemas/mycompany/temp-sensor.yaml -o output/
```

C Header (for firmware):

```
python tools/generate-c.py schemas/mycompany/temp-sensor.yaml -o include/codec.h
```

2.2.4 Step 4: Check Quality Score

```
python tools/score_schema.py schemas/mycompany/temp-sensor.yaml
```

Quality tiers: Bronze (50-69%), Silver (70-84%), Gold (85-94%), Platinum (95-100%)

2.3 Common Patterns

2.3.1 Signed vs Unsigned

```yaml
# Unsigned (0 to 65535)
- name: distance
  type: u16
  unit: "mm"

# Signed (-32768 to 32767)
- name: temperature
  type: s16
  div: 100
  unit: "°C"
```

2.3.2 Enumerations

```yaml
- name: status
  type: u8
  lookup:
    0: "ok"
    1: "low_battery"
    2: "sensor_error"
    3: "tamper"
```

2.3.3 Optional Sensor Groups (Flagged)

```yaml
fields:
  - name: flags
    type: u8

  - flagged:
      field: flags
      groups:
        - bit: 0
```

```
      fields:
        - name: temperature
          type: s16
          div: 100
      - bit: 1
        fields:
        - name: humidity
          type: u8
```

2.3.4 Semantic Annotations

```
- name: temperature
  type: s16
  div: 100
  unit: "°C"
  ipso: {object: 3303, instance: 0, resource: 5700}
  senml: {name: "temp", unit: "Cel"}
```

3 Schema Language Reference

3.1 Document Structure

```yaml
name: string              # REQUIRED: unique identifier
version: integer          # REQUIRED: schema version
endian: big|little        # Default: big
description: string       # Optional
fields: [...]             # Field definitions (or use ports)
ports:                    # Port-based routing (or use fields)
  1: { fields: [...] }
  2: { fields: [...] }
definitions:              # Reusable field groups
  common_header: [...]
test_vectors: [...]       # Test cases
```

3.2 Field Types

3.2.1 Integer Types

Type	Bytes	Description
u8, u16, u32, u64	1,2,4,8	Unsigned integer
s8, s16, s32, s64	1,2,4,8	Signed integer (two's complement)

3.2.2 Floating Point Types

Type	Bytes	Description
`f16`, `f32`, `f64`	2,4,8	IEEE 754 float

3.2.3 Decimal Types

Type	Description
`udec`	Unsigned nibble-decimal (BCD-like)
`sdec`	Signed nibble-decimal

3.2.4 String/Byte Types

Type	Description
`ascii`	ASCII string (requires `length:`)
`hex`	Hex string output (requires `length:`)
`bytes`	Raw bytes (requires `length:`)
`base64`	Base64 encoded output (requires `length:`)

3.2.5 Special Types

Type	Description
`bool`	Boolean (0=false, nonzero=true)
`number`	Computed field (no wire bytes)
`string`	Literal string constant
`skip`	Skip bytes (padding)
`enum`	Enumerated values
`bitfield_string`	Bit flags as string

3.3 Bitfields

```yaml
type: u8[0:3]      # Bits 0-3 of byte (4 bits)
type: u16[8:15]    # High byte of u16
```

3.3.1 Endian Prefix

```yaml
type: le_u16       # Little-endian
type: be_u32       # Big-endian (explicit)
```

3.3.2 Byte Group

```yaml
- byte_group:
    bytes: 3
    fields:
      - name: value_a
        type: u8[0:3]
      - name: value_b
        type: u8[4:7]
```

3.4 Arithmetic Modifiers

Applied in YAML key order:

```yaml
- name: temperature
  type: s16
  div: 10           # Divide by 10
  add: -40          # Then subtract 40
  # Result: (raw / 10) - 40
```

Modifier	Effect
`add: n`	Add offset
`mult: n`	Multiply
`div: n`	Divide

3.5 Lookup Tables

```yaml
- name: status
  type: u8
  lookup: ["off", "on", "error", "unknown"]
```

3.6 Computed Fields

3.6.1 Polynomial (calibration curves)

```yaml
- name: raw_value
  type: u16
  div: 50

- name: calibrated
  type: number
  ref: $raw_value
  polynomial: [0.0000043, -0.00055, 0.0292, -0.053]  # ax³ + bx² + cx + d
```

3.6.2 Cross-Field Computation

```yaml
- name: ratio
  type: number
  compute:
    op: div          # div, mul, add, sub
    a: $field1
    b: $field2
```

3.6.3 Guard Conditions

```yaml
- name: safe_ratio
  type: number
  compute:
    op: div
    a: $numerator
    b: $denominator
  guard:
    when:
      - field: $denominator
        gt: 0
    else: 0
```

3.7 Transform Operations

```yaml
transform:
  - sqrt: true        # √x
  - abs: true         # |x|
  - pow: 2            # x²
  - floor: 0          # Clamp lower bound
  - ceiling: 100      # Clamp upper bound
  - clamp: [0, 100]   # Both bounds
  - log10: true       # Base-10 logarithm
  - log: true         # Natural logarithm
```

3.8 Conditional Parsing

3.8.1 Switch (by field value)

```yaml
- name: msg_type
  type: u8
```

```yaml
- switch:
    field: msg_type
    cases:
      1:
        - name: temperature
          type: s16
      2:
        - name: humidity
          type: u8
```

3.8.2 Flagged (bitmask presence)

```yaml
- name: flags
  type: u8

- flagged:
    field: flags
    groups:
      - bit: 0
        fields:
          - name: temperature
            type: s16
      - bit: 1
        fields:
          - name: humidity
            type: u8
```

3.9 TLV (Type-Length-Value)

```yaml
- tlv:
    tag_type: u8
    length_type: u8
    cases:
      0x01:
        - name: temperature
          type: s16
      0x02:
        - name: humidity
          type: u8
```

3.10 Repeat (Arrays)

```yaml
# Count-based
- name: readings
  type: repeat
  count: 4
  fields:
    - name: value
      type: u16

# Field-based count
- name: readings
  type: repeat
  count_field: num_readings
```

```
  fields:
    - name: value
      type: u16

# Until end of payload
- name: entries
  type: repeat
  until: end
  fields:
    - name: value
      type: u16
```

3.11 Nested Objects

```
- name: gps
  type: object
  fields:
    - name: latitude
      type: s32
      div: 10000000
    - name: longitude
      type: s32
      div: 10000000
```

3.12 Definitions (Reusable Groups)

```
definitions:
  header:
    - name: version
      type: u8
    - name: flags
      type: u8

fields:
  - use: header
  - name: payload
    type: bytes
    length: 10
```

3.13 Port-Based Routing

```
ports:
  1:
    description: "Sensor data"
    fields:
      - name: temperature
        type: s16
  2:
    description: "Status"
    fields:
      - name: battery
        type: u8
```

3.14 Test Vectors

```yaml
test_vectors:
  - name: basic_reading
    description: "Normal temperature reading"
    payload: "00 E7 32"
    expected:
      temperature: 23.1
      humidity: 50
```

3.15 Quick Reference

```
TYPES:         u8 u16 u32 u64 | s8 s16 s32 s64 | f16 f32 f64 | bool
               ascii hex bytes base64 | number string | skip enum
               udec sdec | bitfield_string

STRUCTURES:    object | repeat | byte_group | tlv

MODIFIERS:     add mult div | lookup | polynomial | compute | guard | transform

CONDITIONALS: switch (value match) | flagged (bitmask) | tlv (tag dispatch)

TRANSFORMS:    sqrt abs pow floor ceiling clamp log10 log

COMPUTE OPS:   add sub mul div

GUARD OPS:     gt gte lt lte eq ne

ENCODINGS:     sign_magnitude bcd gray

REFERENCES:    $field_name | use: definition_name
```

4 Code Generation

4.1 Overview

The toolkit generates codec implementations from YAML schemas for multiple platforms.

4.2 JavaScript (TS013)

Generate TS013-compliant JavaScript codecs for The Things Network, ChirpStack, and Helium:

```
python tools/generate_ts013_codec.py schema.yaml > codec.js
```

4.2.1 Generated Interface

```
function decodeUplink(input) {
  return {
    data: { /* decoded fields */ },
    warnings: [],
    errors: []
  };
}

function encodeDownlink(input) {
  return {
    bytes: [ /* encoded payload */ ],
    fPort: 1,
    warnings: [],
    errors: []
  };
}
```

4.3 C Code Generation

Generate header-only C codecs for embedded systems:

```
python tools/generate-c.py schema.yaml -o codec.h
```

4.3.1 Generated Structure

For a schema with temperature, humidity, and battery fields:

```
typedef struct {
    s2_t temperature;
    u1_t humidity;
    u2_t battery_mv;
} env_sensor_t;

static inline int decode_env_sensor(const u1_t* buf, size_t len,
                                    env_sensor_t* out);
static inline int encode_env_sensor(const env_sensor_t* in,
                                    u1_t* buf, size_t max_len);
```

4.3.2 Device Usage

Decoding (Downlink Reception):

```c
#include "codec.h"

void handle_downlink(const uint8_t* payload, size_t len) {
    env_sensor_t config;

    int consumed = decode_env_sensor(payload, len, &config);
    if (consumed < 0) {
        return;  // Error
    }

    // Apply multipliers (noted in generated code comments)
    float temperature = config.temperature * 0.01f;
    apply_config(temperature, config.humidity);
}
```

Encoding (Uplink Transmission):

```c
#include "codec.h"

void send_uplink(void) {
    env_sensor_t data;
    uint8_t payload[16];

    // Fill struct (reverse the multipliers)
    data.temperature = (int16_t)(read_temp_sensor() / 0.01f);
    data.humidity = read_humidity();
    data.battery_mv = read_battery_mv();

    int len = encode_env_sensor(&data, payload, sizeof(payload));
    if (len > 0) {
        lorawan_send(FPORT, payload, len);
    }
}
```

4.3.3 Return Values

Value	Meaning
> 0	Success: bytes consumed (decode) or written (encode)
-1	Invalid parameters (NULL pointer)
-2	Buffer too short

4.3.4 Multipliers

Multipliers are NOT applied automatically. They appear as comments:

```c
out->temperature = read_u2_le(buf + pos);
/* Note: apply mult 0.01 in application */
```

4.4 Generated vs Runtime Interpreter

Aspect	Generated	Runtime Interpreter
Schema changes	Requires regenerate + recompile	Load new binary schema
Code size	Smaller per-schema	Fixed ~2KB + schema
Performance	Fastest (200M msg/s)	Fast (32M msg/s)
Flexibility	Fixed at compile time	Dynamic
Use case	Production devices	Development, OTA updates

4.5 Dependencies

Generated C headers require `rt.h` which provides:

- Type aliases: `u1_t`, `u2_t`, `u4_t`, `s1_t`, `s2_t`, `s4_t`
- Read helpers: `read_u2_le()`, `read_u2_be()`, etc.
- Write helpers: `write_u2_le()`, `write_u2_be()`, etc.

5 Output Formats

The decoder can output data in multiple formats for different platforms.

5.1 Raw Format (Default)

Simple flat dictionary:

```
{
  "temperature": 23.45,
  "humidity": 65.0,
  "pressure": 1013.2,
  "battery": 3.3
}
```

Use case: Direct application use, custom backends.

5.2 IPSO Smart Objects Format

OMA LwM2M standard object IDs:

```
{
  "3303": { "value": 23.45, "unit": "°C" },
  "3304": { "value": 65.0, "unit": "%RH" },
  "3315": { "value": 1013.2, "unit": "hPa" },
  "3316": { "value": 3.3, "unit": "V" }
}
```

IPSO Object Reference:

Object ID	Name
3303	Temperature Sensor
3304	Humidity Sensor
3315	Barometer
3316	Voltage
3325	CO2 Sensor
3330	Distance
3301	Illuminance

Use case: LwM2M platforms (Leshan, Wakaama).

5.3 SenML Format (RFC 8428)

IETF Sensor Measurement Lists standard:

```
[
  { "n": "temperature", "v": 23.45, "u": "°C" },
  { "n": "humidity", "v": 65.0, "u": "%RH" },
  { "n": "pressure", "v": 1013.2, "u": "hPa" },
  { "n": "battery", "v": 3.3, "u": "V" }
]
```

SenML Fields:

- n - name
- v - value (numeric)

- vs - value (string)
- vb - value (boolean)
- u - unit
- t - time (optional)

Use case: CoAP integration, CBOR encoding.

5.4 TTN Normalized Format

The Things Network v3 payload format:

```
{
  "decoded_payload": {
    "temperature": 23.45,
    "humidity": 65.0
  },
  "normalized_payload": [
    {
      "measurement": {
        "temperature": { "value": 23.45, "unit": "°C" }
      }
    },
    {
      "measurement": {
        "humidity": { "value": 65.0, "unit": "%RH" }
      }
    }
  ]
}
```

Use case: TTN Console, TTN integrations.

5.5 Format Comparison

Format	Structure	Size	Interoperability	Best For
Raw	Flat dict	Smallest	Application-specific	Custom backends
IPSO	Object-based	Medium	High (OMA/LwM2M)	LwM2M platforms
SenML	Record array	Medium	High (IETF/CoAP)	RFC-compliant
TTN	Normalized	Largest	TTN ecosystem	TTN integrations

5.6 Schema Definition

To enable semantic output formats, add `semantic` annotations:

```
fields:
  - name: temperature
    type: s16
    mult: 0.01
    unit: "°C"
    semantic:
      ipso: 3303
      senml: "urn:dev:ow:temp"
```

5.7 API Usage

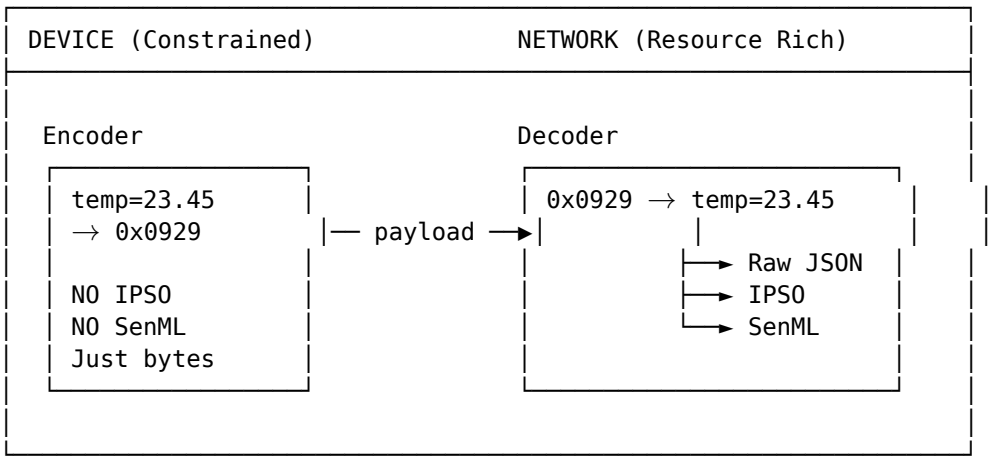
```
from schema_interpreter import SchemaInterpreter

interpreter = SchemaInterpreter(schema)
result = interpreter.decode(payload)

# Raw format (default)
raw = result.data

# IPSO format
ipso = interpreter.get_semantic_output(result.data, 'ipso')

# SenML format
senml = interpreter.get_semantic_output(result.data, 'senml')
```

6 Bidirectional Codec

The codec is symmetric - devices and networks use the same encode/decode functions for opposite directions.

6.1 Device vs Network Responsibilities

Devices do NOT need semantic format knowledge (IPSO, SenML, TTN). The device codec only handles byte-level encoding/decoding. Semantic transformation happens on the network side.



6.2 Link Directions

Direction	Device Role	Network Role	Content
Uplink	Encode	Decode	Sensor readings, status
Downlink	Decode	Encode	Configuration, commands

6.3 Schema Types

A device typically has two schemas:

6.3.1 Uplink Schema (Sensor Data)

```yaml
name: sensor_uplink
version: 1
endian: big
fields:
  - name: temperature
    type: s16
    mult: 0.01
  - name: humidity
    type: u8
    mult: 0.5
  - name: battery
    type: u16
```

6.3.2 Downlink Schema (Commands)

```yaml
name: device_config
version: 1
endian: big
fields:
  - name: command
    type: u8
    lookup:
      0x01: set_interval
      0x02: reboot
      0x03: set_threshold
  - name: parameter
    type: u16
```

6.4 Binary Schema Benefits

With binary schemas, the codec is separated from schema data:

```
┌─────────────────────────────────────────────────────┐
│                   DEVICE FIRMWARE                   │
├─────────────────────────────────────────────────────┤
│  Generic Codec Library (~2KB)                       │
│  ├── schema_encode()                                │
│  ├── schema_decode()                                │
│  └── schema_load_binary()                           │
├─────────────────────────────────────────────────────┤
│  Uplink Schema (binary, ~30 bytes)   ← Flash/EEPROM │
│  Downlink Schema (binary, ~20 bytes) ← Flash/EEPROM │
└─────────────────────────────────────────────────────┘
```

Advantages:

- Same firmware binary across product variants
- Schema updates without reflashing
- OTA schema updates possible
- Schemas can be embedded in QR codes for provisioning

6.5 C API Reference

6.5.1 Decoding (Bytes → Values)

```c
#include "schema_interpreter.h"

// Load schema from binary
schema_t schema;
schema_load_binary(&schema, binary_data, binary_len);

// Decode payload
decode_result_t result;
int rc = schema_decode(&schema, payload, payload_len, &result);

// Access values
double temp = result_get_double(&result, "temperature", 0.0);
int64_t status = result_get_int(&result, "status", 0);
```

6.5.2 Encoding (Values → Bytes)

```
// Prepare input values
encode_inputs_t inputs;
encode_inputs_init(&inputs);
encode_inputs_add_double(&inputs, "temperature", 23.45);
encode_inputs_add_double(&inputs, "humidity", 65.0);

// Encode to payload
encode_result_t result;
int rc = schema_encode(&schema, &inputs, &result);

// Send result.data (result.len bytes)
lora_send(result.data, result.len);
```

6.6 Modifier Handling

Modifiers are applied in opposite directions:

Direction	Operation	Modifier Order
Decode	bytes → value	raw × mult ÷ div + add
Encode	value → bytes	(value - add) × div ÷ mult

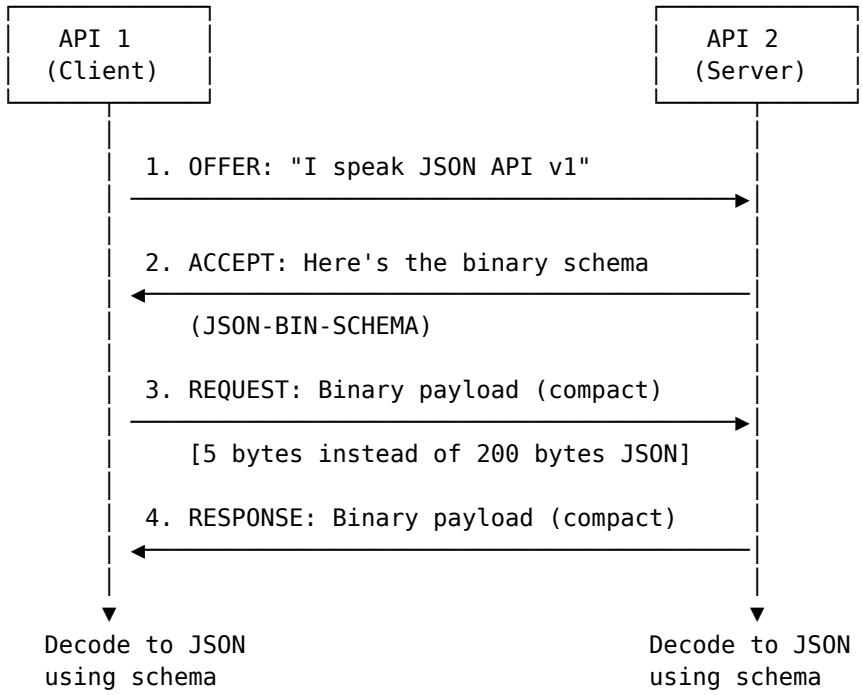
Example with `mult: 0.01`:

- Decode: raw 2345 → `23.45`
- Encode: value `23.45` → raw 2345

7 Binary API Negotiation

The binary schema enables symmetrical compression between API endpoints. Instead of transmitting verbose JSON, endpoints negotiate a schema and exchange compact binary.

7.1 Concept



7.2 Symmetrical Compression

The schema defines bidirectional transformation:

Direction	Input	Output	Size
Encode	JSON object	Binary bytes	10-50x smaller
Decode	Binary bytes	JSON object	Original structure

Example:

JSON (87 bytes):

```
{"temperature":23.45,"humidity":65,"pressure":1013.2,"battery":3.3}
```

Binary (8 bytes):

```
09 29 41 27 B4 00 21
```

Schema (shared once):

```
fields:
  - name: temperature
    type: s16
    div: 100
  - name: humidity
```

```
    type: u8
  - name: pressure
    type: u16
    div: 10
  - name: battery
    type: u8
    div: 10
```

7.3 Negotiation Protocol

7.3.1 Step 1: Capability Advertisement

Client advertises supported API:

```
POST /api/sensor/data
Content-Type: application/json
Accept: application/vnd.payload-schema+binary
X-Schema-Version: 1

{"temperature":23.45,"humidity":65}
```

7.3.2 Step 2: Schema Response

Server responds with binary schema:

```
HTTP/1.1 200 OK
Content-Type: application/vnd.payload-schema
X-Schema-ID: sensor-v1
X-Schema-Hash: a3f2b1c4

[Binary schema bytes]
```

7.3.3 Step 3: Binary Communication

Subsequent requests use binary:

```
POST /api/sensor/data
Content-Type: application/vnd.payload-schema+binary
X-Schema-ID: sensor-v1

[8 bytes binary payload]
```

Server decodes using cached schema, responds in binary.

7.4 Schema Distribution Methods

Method	Use Case	Bandwidth
Inline	First request	Schema sent with response
Reference	Cached	Schema ID only, both sides have it
OTA	IoT devices	Schema pushed via LoRaWAN downlink
QR Code	Provisioning	Schema embedded in device QR

7.5 Implementation

7.5.1 Client Side (Python)

```python
class BinaryAPIClient:
    def __init__(self, base_url):
        self.base_url = base_url
        self.schemas = {}  # schema_id -> Schema

    def negotiate(self, endpoint, sample_json):
        """Negotiate binary schema for endpoint"""
        resp = requests.post(
            f"{self.base_url}{endpoint}",
            json=sample_json,
            headers={"Accept": "application/vnd.payload-schema"}
        )
        schema_id = resp.headers["X-Schema-ID"]
        self.schemas[endpoint] = {
            "id": schema_id,
            "schema": Schema.from_binary(resp.content)
        }
        return schema_id

    def send(self, endpoint, data):
        """Send data using negotiated binary schema"""
        schema = self.schemas[endpoint]["schema"]
        binary = schema.encode(data)

        return requests.post(
            f"{self.base_url}{endpoint}",
            data=binary,
            headers={
                "Content-Type": "application/vnd.payload-schema+binary",
                "X-Schema-ID": self.schemas[endpoint]["id"]
            }
        )
```

7.5.2 Server Side (Go)

```go
func HandleSensorData(w http.ResponseWriter, r *http.Request) {
    schemaID := r.Header.Get("X-Schema-ID")

    if schemaID == "" {
        // First request - negotiate schema
        schema := GetSchemaForEndpoint("/api/sensor/data")
        w.Header().Set("X-Schema-ID", schema.ID)
        w.Header().Set("Content-Type", "application/vnd.payload-schema")
        w.Write(schema.ToBinary())
        return
    }

    // Subsequent requests - decode binary
    schema := GetSchemaByID(schemaID)
    body, _ := io.ReadAll(r.Body)
```

```
    data := schema.Decode(body)

    // Process data...
    result := ProcessSensorData(data)

    // Respond in binary
    w.Header().Set("Content-Type", "application/vnd.payload-schema+binary")
    w.Write(schema.Encode(result))
}
```

7.6 Bandwidth Savings

Scenario	JSON	Binary	Savings
Temperature reading	28 bytes	2 bytes	93%
Sensor bundle (5 values)	120 bytes	8 bytes	93%
GPS + sensors	250 bytes	18 bytes	93%
Config command	80 bytes	4 bytes	95%

Over 1000 messages:

- JSON: 120 KB
- Binary: 8 KB
- Schema overhead: 50 bytes (one-time)

7.7 Security Considerations

7.7.1 Schema Integrity

```
# Schema includes hash for verification
schema:
  name: sensor_v1
  hash: sha256:a3f2b1c4...
  fields: [...]
```

Clients verify schema hash before use.

7.7.2 Version Mismatch

```
HTTP/1.1 409 Conflict
X-Schema-ID: sensor-v2
X-Schema-Required: true

Schema version mismatch. Please renegotiate.
```

Client must re-negotiate when server schema changes.

7.8 Comparison with Alternatives

Protocol	Schema	Compression	Human Readable	LoRaWAN Ready
Payload Schema	YAML/Binary	90%+	Yes (YAML)	Yes
Protocol Buffers	.proto	80%	No	Partial
MessagePack	None	30%	No	Yes

Protocol	Schema	Compression	Human Readable	LoRaWAN Ready
CBOR	None	40%	No	Yes
JSON	None	0%	Yes	No

Payload Schema advantages:

- Human-readable schema definition
- Designed for LoRaWAN constraints
- Bidirectional (encode + decode)
- Semantic annotations (IPSO, SenML)

8 TTN Codec Conversion

Guide for converting The Things Network JavaScript codecs to Payload Schema format.

8.1 Benefits of Conversion

Before (JS)	After (YAML Schema)
200+ lines of JavaScript	30-50 lines of YAML
Platform-specific (TTN only)	Generate for any platform
Manual testing	Automatic test vector validation
No type information	Full type metadata
Hard to review	Human-verifiable structure

8.2 Field Type Reference

JS Pattern	Schema Type
`bytes[i]`	`u8`
`(bytes[i] << 8) | bytes[i+1]`	`u16`
`bytes[i] | (bytes[i+1] << 8)`	`u16 + endian: little`
`bytes.readInt16BE(i)`	`s16`
`bytes[i] & 0x0F`	`u8[0:3]`
`(bytes[i] >> 4) & 0x0F`	`u8[4:7]`

8.3 Modifier Mapping

JS Pattern	Schema Modifier
`raw / 100`	`div: 100`
`raw * 0.01`	`mult: 0.01`
`raw - 40`	`add: -40`
`(raw - 400) / 10`	`add: -400 then div: 10`

YAML key order determines modifier application order.

8.4 Common Conversion Patterns

8.4.1 Pattern 1: Simple Sensor

Original JS:

```
function decodeUplink(input) {
  var data = {};
  data.temperature = ((input.bytes[0] << 8) | input.bytes[1]) / 100;
  data.humidity = input.bytes[2];
  return { data: data };
}
```

Schema YAML:

```yaml
name: simple_sensor
endian: big
fields:
  - name: temperature
    type: u16
    div: 100
    unit: "°C"
  - name: humidity
    type: u8
    unit: "%"
```

8.4.2 Pattern 2: Signed Temperature

Original JS:

```javascript
var raw = (input.bytes[0] << 8) | input.bytes[1];
if (raw > 32767) raw -= 65536;
data.temperature = raw / 10;
```

Schema YAML:

```yaml
- name: temperature
  type: s16    # Signed type handles sign extension
  div: 10
  unit: "°C"
```

8.4.3 Pattern 3: Bitfield Extraction

Original JS:

```javascript
data.battery_low = (input.bytes[0] >> 7) & 1;
data.motion = (input.bytes[0] >> 6) & 1;
data.tamper = (input.bytes[0] >> 5) & 1;
```

Schema YAML:

```yaml
- byte_group:
    - name: battery_low
      type: u8[7:7]
    - name: motion
      type: u8[6:6]
    - name: tamper
      type: u8[5:5]
```

8.4.4 Pattern 4: TLV (Tag-Length-Value)

Original JS:

```javascript
var i = 0;
while (i < input.bytes.length) {
  var tag = input.bytes[i++];
  var len = input.bytes[i++];
  // Process based on tag...
}
```

Schema YAML:

```yaml
- name: records
  type: tlv
  tag_size: 1
  length_size: 1
  tags:
    0x01:
      name: temperature
      type: s16
      div: 10
```

8.4.5 Pattern 5: Formula to Polynomial

Original JS:

```javascript
// Topp equation for soil moisture
var e = raw / 50;
data.vwc = 4.3e-6 * Math.pow(e, 3) - 5.5e-4 * Math.pow(e, 2) + 0.0292 * e - 0.053;
```

Schema YAML:

```yaml
- name: dielectric
  type: u16
  div: 50

- name: vwc
  type: number
  ref: $dielectric
  polynomial: [0.0000043, -0.00055, 0.0292, -0.053]
  unit: "m³/m³"
```

8.5 Quality Tiers

Tier	Score	Requirements
Platinum	95-100%	Full coverage, cross-validation
Gold	85-94%	Strong coverage
Silver	70-84%	Good coverage
Bronze	50-69%	Basic validation

8.6 Verification Checklist

After conversion:

- ☐ Schema validates: `python tools/validate_schema.py -f schema.yaml`
- ☐ Test vectors pass: `python tools/score_schema.py schema.yaml`
- ☐ JS codec generates without errors
- ☐ Cross-validation matches original codec output
- ☐ All conditional branches have test coverage

9 Implementation Status

Feature support matrix across reference implementations.

9.1 Quick Summary

Implementation	Decode	Encode	Binary Schema	Performance
Python	Full	Full	Full	45K msg/s
Go	Full	Partial	Full	2.1M msg/s
C	Full	-	Full	32M msg/s
JavaScript	Full	Partial	-	180K msg/s

9.2 Core Types Support

Feature	Python	Go	C	JS
`u8, u16, u32, u64`	✓	✓	✓	✓
`s8, s16, s32, s64`	✓	✓	✓	✓
`u24, s24`	✓	✓	✓	✓
`f16 (half-precision)`	✓	✓	✓	✓
`f32, f64`	✓	✓	✓	✓
`bool`	✓	✓	✓	✓
`ascii`	✓	✓	✓	✓
`hex`	✓	✓	✓	✓
`bytes`	✓	✓	✓	✓
`skip`	✓	✓	✓	✓
`enum`	✓	✓	✓	✓

9.3 Arithmetic Modifiers

Feature	Python	Go	C	JS
`add`	✓	✓	✓	✓
`mult`	✓	✓	✓	✓
`div`	✓	✓	✓	✓
YAML key ordering	✓	✓	✓	✓
`lookup`	✓	✓	✓	✓

9.4 Transform Pipeline

Feature	Python	Go	C	JS
`sqrt`	✓	✓	✓	✓
`abs`	✓	✓	✓	✓
`pow`	✓	✓	✓	✓
`log / log10`	✓	✓	✓	✓
`clamp`	✓	✓	✓	✓

9.5 Computed Fields

Feature	Python	Go	C	JS
`type: number`	✓	✓	✓	✓
`ref: $field`	✓	✓	✓	✓
`polynomial`	✓	✓	✓	✓
`compute: {op, a, b}`	✓	✓	-	✓
`guard` conditions	✓	✓	-	✓

9.6 Conditional Parsing

Feature	Python	Go	C	JS
`switch`	✓	✓	✓	✓
`switch` range	✓	✓	-	✓
`flagged`	✓	✓	✓	✓
`tlv`	✓	✓	✓	✓

9.7 Structures

Feature	Python	Go	C	JS
`type: object`	✓	✓	✓	✓
`type: repeat`	✓	✓	✓	✓
`definitions` / `use`	✓	✓	-	✓
`ports` (fPort routing)	✓	✓	✓	✓

9.8 Binary Schema Format

Feature	Python	Go	C	JS
Parse v1	✓	✓	✓	-
Parse v2	✓	✓	✓	-
Encode v1	✓	✓	-	-
Encode v2	✓	✓	-	-

9.9 Performance Benchmarks

Tested with typical 8-field schema:

Implementation	Throughput	Latency (p99)
C Interpreted	32M msg/s	31 ns
C Precompiled	200M msg/s	5 ns
Go Binary	2.1M msg/s	476 ns
Go YAML	350K msg/s	2.8 μs
JS Generated	180K msg/s	5.5 μs
Python	45K msg/s	22 μs

9.10 Implementation Notes

9.10.1 Python

Reference implementation - most complete and tested. Full decode and encode support.

9.10.2 Go

Production quality for high-throughput servers. Full decode support. Optimized for performance (2.1M msg/s with binary schema).

9.10.3 C

Embedded-optimized - no dynamic allocation required. Full decode support. 32M msg/s throughput.

9.10.4 JavaScript

Generated codecs for TTN/ChirpStack. Full decode support. Eval-free generated code.

10 Language Design Rationale

This section explains the design decisions behind the Payload Schema language.

10.1 Design Goals

10.1.1 Declarative Over Imperative

Schema definitions describe *what* to decode, not *how*.

Rationale:

- **Security**: No code execution eliminates injection attacks
- **Portability**: Same schema works in Python, Go, C, JavaScript
- **Tooling**: Static analysis, validation, code generation possible
- **Simplicity**: Device manufacturers don't need to write code

10.1.2 Wire-Format Agnostic Output

Schema defines binary→structured data mapping; output format is interpreter choice.

Rationale:

- Same schema produces JSON, SenML, LwM2M, or native structs
- Semantic hints enable format translation
- Decouples parsing from presentation

10.1.3 Explicit Over Magic

All transformations visible in schema, applied in YAML key order.

Rationale:

- Predictable: `div: 10` then `add: -40` always means `(raw / 10) - 40`
- Debuggable: Each step traceable
- No hidden conversions

10.1.4 Progressive Complexity

Simple cases are simple; complexity available when needed.

Complexity	Feature
Basic	`type: u16`, `div: 10`
Intermediate	`switch`, `flagged`, `lookup`
Advanced	`polynomial`, `guard`, `tlv`

10.2 Type System Design

10.2.1 Why Both Long and Short Type Names?

```yaml
type: u16          # Short form - common case
type: UInt         # Long form - explicit
  length: 2
```

Short forms (`u8`, `s16`, `f32`) match C conventions. Long forms allow explicit length for unusual sizes.

10.2.2 Why Bitfield Syntax Variants?

```
type: u8[0:3]     # Range syntax - bits 0-3
type: u8:4        # Width syntax - 4 bits from current position
```

Range syntax is natural for hardware engineers. Width syntax is natural for sequential extraction.

10.2.3 Why Separate udec/sdec Types?

Some sensors encode decimals as BCD-like nibbles. Temperature 23.5°C encoded as `0x235`. Dedicated types handle this common pattern.

10.3 Conditional Parsing Design

10.3.1 Why Three Conditional Constructs?

Construct	Use Case
`switch`	Dispatch on message type field
`flagged`	Bitmask presence indicators
`tlv`	Self-describing variable payloads

Each maps to common protocol patterns.

10.4 Feature Exclusions

10.4.1 Why No Formula Field?

`formula: "sqrt(x * 0.01) + offset"` was considered but rejected:

- Requires expression parser (complex, security risk)
- Platform-dependent floating point
- Covered by `polynomial` + `transform` + `compute`

10.4.2 Why No Encryption/Compression?

Schema describes payload structure only. Encryption is transport concern (handled by LoRaWAN). Compression varies by implementation.

10.5 Compatibility

10.5.1 TTN Codec Compatibility

Most TTN JavaScript codecs can be converted:

TTN Pattern	Schema Equivalent
`bytes[i]`	Sequential field reads
`<< / >>`	Bitfields or multi-byte types
Lookup object	`lookup:` array
Math expressions	`add`/`mult`/`div` + `transform`

10.5.2 Extension Points

Schema can include vendor extensions:

```
x-vendor:
  custom_property: value
```

Interpreters ignore unknown `x-` prefixed keys.

11 Formula Migration

Tracking the migration from imperative formulas (JavaScript `eval()`) to declarative schema constructs.

11.1 Why Migrate?

11.1.1 Security

Traditional TTN codecs use JavaScript with patterns like:

```
// DANGEROUS: Code execution
decoded.temp = (bytes[0] << 8 | bytes[1]) / 100 - 40;
```

Risks:

- Arbitrary code execution
- Injection attacks via malformed payloads
- Platform-dependent behavior

11.1.2 Declarative Alternative

```
- name: temp
  type: s16
  div: 100
  add: -40
```

Benefits:

- No code execution
- Validated at parse time
- Portable across languages

11.2 Migration Patterns

11.2.1 Linear Scaling

Pattern	Schema
`raw / 100`	`div: 100`
`raw * 0.01`	`mult: 0.01`

11.2.2 Offset + Scale

Pattern	Schema
`(raw - 4000) / 100`	`add: -4000` then `div: 100`
`raw / 100 - 40`	`div: 100` then `add: -40`

11.2.3 Polynomial Calibration

Before:

```
var x = raw / 1000;
decoded.temp = 0.00001 * x*x*x - 0.001 * x*x + 0.5 * x - 10;
```

After:

```yaml
- name: raw
  type: u16
  div: 1000

- name: temp
  type: number
  ref: $raw
  polynomial: [0.00001, -0.001, 0.5, -10]
```

11.2.4 Cross-Field Computation

Before:

```
decoded.power = decoded.voltage * decoded.current;
```

After:

```yaml
- name: power
  type: number
  compute:
    op: mul
    a: $voltage
    b: $current
```

11.2.5 Safe Division

Before:

```
if (denominator > 0) {
  decoded.ratio = numerator / denominator;
} else {
  decoded.ratio = 0;
}
```

After:

```yaml
- name: ratio
  type: number
  compute:
    op: div
    a: $numerator
    b: $denominator
  guard:
    when:
      - field: $denominator
        gt: 0
    else: 0
```

11.3 Migration Statistics

11.3.1 Coverage

Category	Migrated	Partial	Not Possible
Temperature/Humidity	95%	5%	0%
GPS Trackers	80%	15%	5%
Industrial	70%	20%	10%

Category	Migrated	Partial	Not Possible
Custom/Proprietary	60%	25%	15%

"Partial" = Core decode works; some computed fields require post-processing.

"Not Possible" = Requires string parsing, complex state machines.

11.4 Constructs Coverage

Formula Pattern	Schema Construct	Coverage
`+, -`	`add:`	100%
`*, /`	`mult:, div:`	100%
Bitwise	Bitfield types	100%
`Math.sqrt()`	`transform: sqrt`	100%
`Math.pow()`	`transform: pow`	100%
`Math.log()`	`transform: log`	100%
Polynomial	`polynomial:`	100%
Cross-field	`compute:`	90%
String parsing	-	0%
Regex	-	0%

11.5 Remaining Gaps

11.5.1 String Operations

Some codecs parse ASCII/string data. Workaround: `ascii` type extracts string, post-processing for parsing.

11.5.2 State Machines

Some protocols require multi-message state (e.g., GPS with delta encoding). Workaround: Application-level state management.

11.6 Conclusion

~85% of existing TTN codecs can be fully migrated to declarative schemas. The remaining 15% either require minor post-processing (10%) or device firmware changes (5%).

Glossary

Binary Schema Compact binary representation of a payload schema for OTA transfer and embedded systems.
Codec Software component that encodes and/or decodes payloads between binary and structured formats.
Computed Field A field whose value is derived from other fields rather than read from the payload.
Declarative Schema A schema that describes what data looks like rather than how to process it.
fPort LoRaWAN frame port, used to route payloads to different decoders.
Guard Condition A condition that must be true for a computed field to be evaluated.
IPSO IP for Smart Objects - OMA standard for IoT data modeling.
LwM2M Lightweight M2M - OMA protocol for IoT device management.
Modifier Arithmetic operation (add, mult, div) applied to decoded values.
Polynomial Mathematical formula ($ax^3 + bx^2 + cx + d$) used for sensor calibration.
Schema Interpreter Runtime engine that decodes payloads using schema definitions.
SenML Sensor Measurement Lists - IETF RFC 8428 standard for sensor data.
Semantic Annotation Metadata (IPSO, SenML) that maps fields to standard IoT ontologies.
Test Vector Input/output pair used to verify correct codec behavior.
TLV Type-Length-Value - encoding format for variable-length data.
TS013 LoRa Alliance technical specification for Payload Codec API.
TTN The Things Network - LoRaWAN network infrastructure provider.

References

1. **LoRaWAN L2 1.0.4 Specification** LoRa Alliance, 2020. https://lora-alliance.org/resource_hub/lorawan-104-specification-package/
2. **TS013 Payload Codec API** LoRa Alliance Technical Specification, 2021. https://resources.lora-alliance.org/technical-specifications/ts013-1-0-0-payload-codec-api
3. **RFC 8428 - Sensor Measurement Lists (SenML)** IETF, August 2018. https://datatracker.ietf.org/doc/html/rfc8428
4. **IPSO Smart Objects** OMA SpecWorks. https://technical.openmobilealliance.org/OMNA/LwM2M/LwM2MRegistry.html
5. **The Things Network Device Repository** The Things Industries. https://github.com/TheThingsNetwork/lorawan-devices
6. **YAML Specification 1.2** yaml.org, 2009. https://yaml.org/spec/1.2/spec.html
7. **IEEE 754 Floating Point Standard** IEEE Computer Society, 2019. https://ieeexplore.ieee.org/document/8766229