The Architecture and Implementation of the Flexible and Interoperable Data Transfer (FIT) Protocol for High-Fidelity Sports Telemetry

1. Introduction to the FIT Protocol Ecosystem

The Flexible and Interoperable Data Transfer (FIT) protocol stands as the preeminent standard for data serialization in the sports science and fitness tracking industries. Developed by the ANT+ Alliance (a subsidiary of Garmin), the protocol was engineered to solve a specific set of engineering challenges inherent to embedded fitness devices: the need to store high-frequency time-series data from multiple sensors (heart rate, GPS, power, hemodynamics) on devices with severely constrained battery life, processing power, and storage capacity. Unlike its predecessors, such as the Training Center XML (TCX) or the GPS Exchange Format (GPX), which rely on verbose, text-based structures, FIT utilizes a compact, binary architecture. This shift from text to binary represents a fundamental trade-off: human readability is sacrificed for a dramatic reduction in file size and I/O overhead, a critical necessity when writing 1Hz (or higher) data streams to slow flash storage during endurance events that may last anywhere from minutes to days.[1]

The implications of this architectural choice extend beyond mere file size. The FIT protocol is dynamic and extensible. It functions less like a static file format and more like a serialized stream of database rows where the schema can be redefined on the fly. This "self-defining" characteristic allows the protocol to evolve without breaking backward compatibility; a parser written in 2015 can still read the core structure of a file generated in 2025, simply ignoring the data fields it does not recognize. This forward compatibility is achieved through a mechanism of "Definition Messages" that describe the layout of subsequent "Data Messages," allowing devices to introduce new metrics—such as "Stryd" running power or "Core" body temperature—without requiring a fundamental restructuring of the file format.[1]

For developers working in health-tech, data science, or connected fitness, mastering the parsing of FIT files is non-negotiable. The data contained within is not merely a track of GPS points; it is a rich tapestry of sensor fusion, containing event markers, lap summaries, device calibration data, and increasingly, proprietary "Developer Data Fields" generated by third-party Connect IQ applications. This report serves as a comprehensive technical reference for programmatically extracting this data, managing the complexities of the binary structure, and converting proprietary streams into interoperable JSON formats using Python and Node.js environments.

2. The Binary Architecture of a FIT File

A valid FIT file is a binary stream organized into three distinct, contiguous segments. The parsing logic must respect the strict ordering of these segments to maintain the integrity of the decoding state machine. Unlike random-access formats, a FIT file is typically read sequentially, as the interpretation of any given byte depends on the "Definition Messages" that preceded it.

2.1 The File Header

The entry point for any parser is the File Header. This segment provides essential metadata required to interpret the subsequent binary stream and validates that the file is indeed a FIT file. While early versions of the protocol utilized a 12-byte header, modern Garmin devices almost exclusively generate a 14-byte header.
The header structure is defined as follows:

Offset (Bytes)	**Field Name**	**Type**	**Size**	**Description**
0	Header Size	uint8	1 Byte	Indicates the length of the header (usually 12 or 14).
1	Protocol Version	uint8	1 Byte	Specifies the FIT protocol version (e.g., 2.0).
2-3	Profile Version	uint16	2 Bytes	The version of the Global FIT Profile (Profile.xlsx) used to generate the file.
4-7	Data Size	uint32	4 Bytes	The total size of the Data Records section, excluding header and CRC.
8-11	Data Type	char	4 Bytes	The ASCII string ".FIT". Acts as a magic number.
12-13	Header CRC	uint16	2 Bytes	(Optional) Checksum for the header itself.

Parsing Implications:
A robust parser must first read Byte 0 to determine Header Size. If this value is 14, the parser reads the subsequent 13 bytes; if 12, it reads 11. The Data Size (Bytes 4-7) is particularly

critical for stream processing. It tells the parser exactly how many bytes of Data Records to consume before expecting the File CRC. If a parser relies solely on End-Of-File (EOF) detection, it risks reading the CRC as a malformed message or failing to detect truncated files. The .FIT signature at offset 8 is the primary validation check; if these four bytes do not match the ASCII values for .FIT (0x2E 0x46 0x49 0x54), the file should be rejected immediately.3

2.2 The Data Records Segment

Following the header is the Data Records segment, which constitutes the bulk of the file. This segment is an interleaved stream of two fundamental message types: **Definition Messages** and **Data Messages**. This architecture separates the *schema* (what the data is) from the *payload* (the data itself).

2.2.1 The Record Header Byte

Every message in the Data Records segment begins with a 1-byte Record Header. This byte acts as a switch, telling the parser how to interpret the subsequent bytes.

- **Normal Header (Bit 7 = 0):** Indicates a standard Definition or Data message.
  - **Bit 6 (Message Type):** If 1, it is a Definition Message. If 0, it is a Data Message.
  - **Bit 5 (Developer Data):** If 1, the Definition Message includes Developer Data Fields (introduced in protocol 2.0).
  - **Bits 0-3 (Local Message Type):** A 4-bit identifier (0-15) used to associate Data Messages with their Definitions.
- **Compressed Timestamp Header (Bit 7 = 1):** A specialized header used to compress time data (discussed in Section 6).[2]

2.2.2 Definition Messages: The Schema

A Definition Message creates a template. It assigns a "Local Message Type" (e.g., 0) to a "Global Message Number" (e.g., 20, representing a record). It essentially says: "From now on, whenever you see Local Message Type 0, interpret it as a 'Record' message containing the following fields..."

The structure of a Definition Message includes:

1. **Reserved Byte:** Usually 0.
2. **Architecture:** Specifies the Endianness of the multi-byte fields in the Data Message (0 = Little Endian, 1 = Big Endian). FIT files usually use Little Endian, matching the architecture of ANT+ processors.
3. **Global Message Number (2 Bytes):** The unique ID from the Global FIT Profile (e.g., session=18, lap=19, record=20).
4. **Number of Fields (1 Byte):** How many fields are in this message.
5. **Field Definitions (3 Bytes per field):**
   - **Field Definition Number (1 Byte):** ID of the specific metric (e.g., Heart Rate = 3).
   - **Size (1 Byte):** Size of the field in bytes.
   - **Base Type (1 Byte):** The data type (e.g., uint8, sint16).

This mechanism allows the FIT file to be self-describing regarding its structure, while relying on the external "Global Profile" (Profile.xlsx) for the semantic meaning of the fields.[2]

2.2.3 Data Messages: The Payload

A Data Message contains no metadata. It is a sequence of raw bytes. The parser reads the Record Header, identifies the Local Message Type, looks up the active definition for that type, and consumes the bytes accordingly. If the definition said "Field 1 is 4 bytes, Field 2 is 1 byte," the parser reads 4 bytes, then 1 byte. This extreme compactness is why FIT files are vastly smaller than JSON; there are no repeated keys like "heart_rate": 140—just the byte 0x8C.[2]

2.3 The CRC Footer

The final two bytes of the file are the CRC-16 (Cyclic Redundancy Check). This checksum allows the parser to verify that the file has not been corrupted during transfer. The CRC is calculated over the entire Data Records segment (excluding the header). While checking the CRC is optional for data extraction, it is a best practice for production systems to prevent the ingestion of corrupted data that might lead to spurious metrics (e.g., a GPS jump to 0,0 due to bit rot).[3]

---

3. Data Types and Encoding Strategies

The FIT protocol utilizes a specific set of primitive data types. Understanding these is crucial for correctly interpreting the raw byte streams extracted from Data Messages.

3.1 Base Types and Invalid Sentinels

FIT defines base types ranging from enum and sint8 to float64. A critical concept in FIT parsing is the "Invalid Value" (or Sentinel Value). Since binary fields have a fixed width, a specific value is reserved to indicate "null" or "no data."

Base Type	**ID**	**Size (Bytes)**	**Invalid Value (Hex)**	**Description**
enum	0x00	1	0xFF	Enumerated type (mapped to strings in Profile).
sint8	0x01	1	0x7F	Signed 8-bit integer.
uint8	0x02	1	0xFF	Unsigned 8-bit integer.
sint16	0x83	2	0x7FFF	Signed 16-bit integer.
uint16	0x84	2	0xFFFF	Unsigned 16-bit integer.
sint32	0x85	4	0x7FFFFFFF	Signed 32-bit integer.

uint32	0x86	4	0xFFFFFFFF	Unsigned 32-bit integer.
string	0x07	Variable	0x00 (Null Term)	Null-terminated UTF-8 string.
float32	0x88	4	0xFFFFFFFF	IEEE 754 Single Precision Float.
float64	0x89	8	0xFFFFFFFFFFFFFFFF	IEEE 754 Double Precision Float.

Insight: Parsers must check for these invalid values *before* applying any scaling or offsetting. For example, if a uint8 heart rate field reads 0xFF, it implies the sensor was disconnected, not that the heart rate was 255 bpm. This value should be converted to None (Python) or null (JSON/Node.js).[2]

3.2 Scaling and Offsetting

To maximize storage efficiency, FIT avoids floating-point numbers for most metrics. Instead, it uses scaled integers. The Global Profile defines a scale and offset for each field.
Formula:

$$\text{Physical Value} = \frac{\text{Raw Value}}{\text{Scale}} - \text{Offset}$$

For example, **Altitude** (Field ID 2) is stored as a uint16 with scale = 5 and offset = 500.

- If the raw value in the file is 10435:
- Physical Altitude = $10435 / 5 - 500 = 2087 - 500 = 1587$ meters.

This allows altitude to range from -500m to +12,607m with 0.2m precision using only 2 bytes, whereas a float32 would require 4 bytes. Parsers must implement a lookup table for these scale/offset factors to return meaningful data.[4]

---

4. Programmatic Extraction Strategy: Python and Node.js

The choice of programming language and library dictates the parsing strategy. The Python ecosystem is favored for data science and batch processing, while Node.js is dominant in web-based upload pipelines.

4.1 Python: The fitdecode Approach

While fitparse has historically been the default library, fitdecode is the modern, high-performance alternative. fitparse loads the entire DOM into memory, which can be prohibitive for large history files. fitdecode utilizes a streaming (SAX-style) approach, yielding messages frame-by-frame, which is significantly more memory-efficient and faster for large datasets.

4.1.1 Installation and Setup

fitdecode is available via PyPI and requires Python 3.6+. It is thread-safe, making it suitable for parallel processing of activity archives.

Bash

```
pip install fitdecode
```

4.1.2 Extraction Logic

The following Python script demonstrates a production-grade pattern for extracting record messages using fitdecode. It handles the FIT_FRAME_DATA check, iterates through fields, and applies the necessary scale/offset conversions automatically (a feature built into fitdecode's field accessors).

Python

```
import fitdecode
from typing import List, Dict, Any, Optional

def extract_fit_records(file_path: str) -> List]:
    """
    Parses a FIT file and returns a list of dictionaries representing
    the 'record' messages (time-series data).
    """
    records =

    # Use 'with' to ensure the file stream is closed automatically.
    # check_crc=True ensures we don't process corrupted files.
    with fitdecode.FitReader(file_path, check_crc=True) as fit_file:
        for frame in fit_file:
            # We are only interested in Data Messages, not Definitions or Headers
            if frame.frame_type == fitdecode.FIT_FRAME_DATA:

                # Filter specifically for 'record' messages (Global ID 20)
                if frame.name == 'record':
                    record_data = {}

                    # Iterate over all fields in the message
            for field in frame.fields:
                # fitdecode automatically applies scale/offset.
                # field.name returns the human-readable name (e.g., 'heart_rate')
                # field.value returns the scaled physical value.

                if field.value is not None:
                    # Handle standard fields vs developer fields
                    if field.is_developer_field:
                        key = f"dev_{field.name}"
                    else:
                        key = field.name

                    record_data[key] = field.value

                    # Optional: Extract units for metadata
                    # units = field.units

            records.append(record_data)

    return records

# Example Usage
# data = extract_fit_records('activity.fit')
```

8

4.2 Node.js: The fit-file-parser Approach

For Node.js, fit-file-parser is the robust choice. It wraps the parsing logic in a callback or promise-based structure and outputs a hierarchical JavaScript Object. Unlike the streaming Python approach, this library typically parses the full buffer at once.

4.2.1 Installation

Bash

```
npm install fit-file-parser
```

4.2.2 Extraction Logic

The library allows configuration options to force unit conversions (e.g., speed to km/h) during the parse. However, for raw data analysis, it is often better to keep defaults and convert later.

JavaScript

```
const fs = require('fs');
const FitParser = require('fit-file-parser').default;

const parseFitFile = (filePath) => {
  const content = fs.readFileSync(filePath);

  // Configure parser options
  const fitParser = new FitParser({
    force: true,           // Continue parsing even if errors occur
    speedUnit: 'm/s',      // Keep standard units
    lengthUnit: 'm',
    temperatureUnit: 'celsius',
    elapsedRecordField: true, // Calculate elapsed time
    mode: 'cascade'        // Organize data hierarchically
  });

  fitParser.parse(content, (error, data) => {
    if (error) {
      console.error("Parsing Error:", error);
    } else {
      // Data is structured as: data.activity.sessions.laps.records
      // We need to flatten this for time-series usage.

      const records =;

      // Navigate the hierarchy safely
      if (data.activity && data.activity.sessions) {
        data.activity.sessions.forEach(session => {
          if (session.laps) {
            session.laps.forEach(lap => {
              if (lap.records) {
                lap.records.forEach(record => {
                  // Extract core fields
                  const point = {
                    timestamp: record.timestamp,
                    lat: record.position_lat,
                    lon: record.position_long,
                    hr: record.heart_rate,
                    power: record.power
                  };
```

```
                                    // Handle Developer Fields if present
                                    if (record.developerFields) {
                                        Object.keys(record.developerFields).forEach(key => {
                                            point[`dev_${key}`] = record.developerFields[key];
                                        });
                                    }

                                    records.push(point);
                                });
                            }
                        });
                    }
                });
            }

            console.log(`Parsed ${records.length} records.`);
        }
    });
};

parseFitFile('./activity.fit');
```

[11]

5. Core Metric Extraction: Temporal and Spatial Logic

Extracting the numbers is only half the battle; interpreting them requires specific domain knowledge of the FIT protocol's coordinate systems and epochs.

5.1 Timestamps: The Garmin Epoch

The timestamp field (ID 253) in a FIT file is a uint32. However, it is **not** a Unix Timestamp. The FIT protocol uses the **Garmin Epoch**, defined as 00:00:00 UTC on December 31, 1989. The Unix Epoch (standard in most computing) is January 1, 1970.
The difference between the two epochs is **631,065,600 seconds**.
Implementation Insight:
When converting FIT timestamps to standard datetime objects or JSON ISO strings, you must add this offset.

- **Python:** datetime.fromtimestamp(fit_time + 631065600)
- **JavaScript:** new Date((fit_time + 631065600) * 1000) (JS uses milliseconds).

Most libraries like fitdecode handle this conversion internally if you request the value as a Python datetime object, but raw value extraction will return the Garmin integer.[13]

5.2 Compressed Timestamp Headers

To save bandwidth, FIT utilizes "Compressed Timestamp Headers." In a dense stream of 1Hz data, repeating the full 4-byte timestamp is redundant. A Compressed Header (Bit 7 = 1) uses a 5-bit time offset (0-31 seconds) relative to a previously established reference timestamp.
Parsing Logic:

1. Parser stores last_timestamp.
2. Header indicates compressed time with offset X.
3. current_timestamp = (last_timestamp & 0xFFFFFFE0) + X.
4. If X < (last_timestamp & 0x1F), it implies a rollover (we moved to the next 32-second block), so add 32 seconds to the base.

This complex logic is why writing a raw parser from scratch is discouraged compared to using fitdecode or fit-file-parser, which implement this state machine correctly.[2]

5.3 Geospatial Data: Semicircles to Degrees

Latitude and Longitude (Fields 0 and 1) are stored as 32-bit signed integers (sint32) representing "Semicircles." This unit maps the full range of the 32-bit integer to the 360 degrees of the globe.

- Range of sint32: $-2^{31}$ to $2^{31}-1$.
- Range of Longitude: $-180^\circ$ to $180^\circ$.

Conversion Formula:

$$\text{Degrees} = \text{Semicircles} \times \left(\frac{180}{2^{31}} \right)$$

Precision Analysis:
A standard float32 (single precision) has about 7 decimal digits of precision. At the equator, this equates to roughly 1.1 meters of error. Semicircles, using 31 bits for the value, provide a resolution of approximately $1.11 \text{ cm}$. This allows FIT files to maintain survey-grade precision without the storage overhead of float64 (double precision).7
Python Helper Function:

Python

```python
def semicircles_to_degrees(semicircles: Optional[int]) -> Optional[float]:
    if semicircles is None:
        return None
    # 2^31 = 2147483648
    return semicircles * (180.0 / 2147483648)
```

6. Advanced Handling: Developer Data Fields

The "Developer Data Field" mechanism is the FIT protocol's method for extensibility. It allows third-party hardware (like the **Stryd** foot pod) or software (Connect IQ apps) to inject custom data streams into the file structure that are not part of the official Garmin Profile.

6.1 The Meta-Schema Architecture

Developer fields are not pre-defined. The FIT file contains "Definition Messages" for them *inside* the file stream. This is a "Schema-on-Read" architecture. Two specific global messages facilitate this:

1. **Field Description Message (Global ID 206):** This message provides the metadata for the custom field. It maps a developer_data_index and a field_definition_number to a Field Name (e.g., "Stryd Power"), Units (e.g., "Watts"), and Base Type.
2. **Developer Data ID Message (Global ID 207):** This maps the developer_data_index to a specific Application ID (UUID) or Manufacturer ID (e.g., Stryd's Manufacturer ID).

Parsing Flow:
When the parser encounters a record message with a field ID that is not standard (e.g., Field ID 0 for a record is Lat, but Field ID 12 might be custom):

1. It checks the Definition Message for that Local Message Type to find the developer_data_index associated with Field 12.
2. It looks up the field_description message that matches that index and field number.
3. It retrieves the field name ("Air Power") and units ("Watts") to label the data correctly.[3]

6.2 Case Study: Stryd Power Extraction

Stryd devices are notorious for their complex data integration. They use two methods to write power, which the developer must handle:

1. **Native Num Override:** The field_description message may contain a native_field_num field. For Stryd, this is often set to **7** (the ID for the standard power field). This tells the parser: "Treat this developer field exactly like the native Power field." High-quality parsers will automatically map this value to the standard power key.[17]
2. **Pure Developer Field:** If native_field_num is not used, the data appears strictly as a developer field. The parser must iterate through the developer_fields collection of the message object.

Python Extraction Code (Handling both cases):

Python

```python
def get_power_data(frame):
    # Case 1: Check for native power (ID 7)
    # This covers standard power meters and Stryd using native_field_num
    if frame.has_field('power'):
        return frame.get_value('power')

    # Case 2: Iterate developer fields for fallback
    # fitdecode exposes these via the.developer_fields attribute or generic iteration
    for field in frame.fields:
        if field.is_developer_field:
            # Check names. Stryd often uses "Power" or "Stryd Power"
            if field.name.lower() in ['power', 'stryd power', 'rwr_power']:
                return field.value
    return None
```

18

6.3 Connect IQ Metrics

Connect IQ apps function identically to Stryd. An app calculating "Hydration Loss" will generate a field_description naming the field "hydration_loss" with units "ml". Libraries like fit-file-parser in Node.js typically group these into a nested object: record.developerFields['hydration_loss']. In Python's fitdecode, they are accessible via the is_developer_field property on the FieldData object.[20]

---

7. Interoperability: Converting FIT to JSON Time-Series

For web visualization (D3.js, Chart.js) or NoSQL storage (MongoDB), converting binary FIT data to JSON is the final step.

7.1 JSON Schema Design: Flat vs. Hierarchical

While the FIT file is hierarchical (File -> Activity -> Session -> Lap -> Record), a **Flat Schema** is generally preferred for time-series analysis. A flat array of objects, where each object is a single timestamped record, allows for easier indexing and filtering in dataframes (Pandas) or JavaScript mapping libraries.
Recommended JSON Structure:

JSON

7.2 Handling Data Anomalies in JSON

JSON has strict limitations compared to binary formats or Python dictionaries:

1. **No NaN/Infinity:** Calculated fields (e.g., grade or speed) might result in NaN or Infinity.

The JSON standard does not support these. They must be converted to null before serialization. In Python, use json.dumps(data, allow_nan=False) or explicit sanitation.
2. **Date Serialization:** JSON has no native Date type. **ISO 8601 strings** (e.g., "2023-10-27T08:00:00Z") are the gold standard for interoperability. While Unix timestamps (integers) save space, ISO strings are human-readable and automatically parsed by most databases.[22]

7.3 Geospatial Transformation: GeoJSON

For mapping applications (Leaflet, Mapbox), standard JSON is insufficient. **GeoJSON** is the required format.

Transformation Logic:

1. Extract the position_lat and position_long fields.
2. Convert Semicircles to Degrees (Section 5.3).
3. Construct a LineString feature for the track.
4. **Critical Warning:** GeoJSON uses [Longitude, Latitude] coordinate order (X, Y). FIT and most colloquial usage is (Latitude, Longitude). Reversing this is the most common error in FIT conversion.[25]

Python Implementation for GeoJSON:

Python

```
import json
import fitdecode

def fit_to_geojson(fit_path, output_path):
    coordinates =

    with fitdecode.FitReader(fit_path) as fit:
        for frame in fit:
            if frame.frame_type == fitdecode.FIT_FRAME_DATA and frame.name == 'record':
                if frame.has_field('position_lat') and frame.has_field('position_long'):
                    lat = semicircles_to_degrees(frame.get_value('position_lat'))
                    lon = semicircles_to_degrees(frame.get_value('position_long'))
                    if lat is not None and lon is not None:
                        # GeoJSON order: [Lon, Lat]
                        coordinates.append([lon, lat])

    geojson = {
        "type": "FeatureCollection",
        "features":
    }
```

```
with open(output_path, 'w') as f:
  json.dump(geojson, f)
```

25

---

8. Best Practices and Optimization

1. **Stream Processing:** Always use streaming parsers (fitdecode, fit-file-parser) over DOM parsers (fitparse) for web applications. A user uploading a 200MB Ultra-Marathon file can crash a server if the entire file is loaded into RAM.
2. **Strict Profile Updates:** The Global FIT Profile (Profile.xlsx) changes frequently. Ensure your parsing library is updated regularly or allows manual injection of new profile types. Without this, new devices (like the latest Garmin Fenix) may produce "Unknown Message" errors.
3. **Error Resilience:** CRC mismatches are common in files recovered from crashed devices. Configure your parser to WARN rather than RAISE on CRC errors if your goal is data recovery.[8]
4. **Privacy Scrubbing:** Before sharing FIT data or converting to JSON for public APIs, scrub the first and last 200 meters of GPS data (Privacy Zones) and remove serial_number fields from the File ID message to protect user privacy.

By adhering to these architectural principles and leveraging the robust libraries available in the Python and Node.js ecosystems, developers can build scalable, high-fidelity applications that fully exploit the rich data landscape of the modern fitness tracking world.

Alıntılanan çalışmalar

1. Flexible and Interoperable Data Transfer (FIT) SDK - Garmin Developers, erişim tarihi Aralık 3, 2025, https://developer.garmin.com/fit/
2. FIT Protocol | FIT SDK - Garmin Developers, erişim tarihi Aralık 3, 2025, https://developer.garmin.com/fit/protocol/
3. File Types | FIT SDK - Garmin Developers, erişim tarihi Aralık 3, 2025, https://developer.garmin.com/fit/file-types/
4. The structure of a FIT file. · dvmarinoff/Auuki Wiki - GitHub, erişim tarihi Aralık 3, 2025, https://github.com/dvmarinoff/Flux/wiki/The-structure-of-a-FIT-file.
5. Official Garmin FIT Python SDK - GitHub, erişim tarihi Aralık 3, 2025, https://github.com/garmin/fit-python-sdk
6. Java Garmin/ANT FIT file reader - Studio Blue Planet, erişim tarihi Aralık 3, 2025, https://blog.studioblueplanet.net/software/java-garminant-fit-file-reader
7. What CRS does the python sdk decode to (eg, 'position_lat': 485072248, 'position_long': -882385675)? - Discussion - Garmin Forums, erişim tarihi Aralık 3, 2025, https://forums.garmin.com/developer/fit-sdk/f/discussion/325061/what-crs-does-

the-python-sdk-decode-to-eg-position_lat-485072248-position_long--882385675
8. fitdecode - PyPI, erişim tarihi Aralık 3, 2025, https://pypi.org/project/fitdecode/
9. Change Log - fitdecode, erişim tarihi Aralık 3, 2025, https://fitdecode.readthedocs.io/en/latest/history.html
10. 1.3. records — fitdecode, erişim tarihi Aralık 3, 2025, https://fitdecode.readthedocs.io/en/latest/reference/records.html
11. Top 10 Examples of fit-file-parser code in Javascript - CloudDefense.AI, erişim tarihi Aralık 3, 2025, https://www.clouddefense.ai/code/javascript/example/fit-file-parser
12. fit-file-parser - NPM, erişim tarihi Aralık 3, 2025, https://www.npmjs.com/package/fit-file-parser
13. The fitparse API Documentation - Pythonhosted.org, erişim tarihi Aralık 3, 2025, https://pythonhosted.org/fitparse/api.html
14. How to convert to a Python datetime object with JSON.loads? - Stack Overflow, erişim tarihi Aralık 3, 2025, https://stackoverflow.com/questions/8793448/how-to-convert-to-a-python-datetime-object-with-json-loads
15. dtcooper/python-fitparse: Python library to parse ANT/Garmin .FIT files - GitHub, erişim tarihi Aralık 3, 2025, https://github.com/dtcooper/python-fitparse
16. Conversion between semicircles and latitude units - GIS StackExchange, erişim tarihi Aralık 3, 2025, https://gis.stackexchange.com/questions/156887/conversion-between-semicircles-and-latitude-units
17. Hacking the FIT - Connect IQ App Development Discussion - Garmin Forums, erişim tarihi Aralık 3, 2025, https://forums.garmin.com/developer/connect-iq/f/discussion/7931/hacking-the-fit
18. Firmware 9.24 and Stryd Native Power? : r/Garmin - Reddit, erişim tarihi Aralık 3, 2025, https://www.reddit.com/r/Garmin/comments/wayr6q/firmware_924_and_stryd_native_power/
19. Decoding developer fields in FIT files : r/learnrust - Reddit, erişim tarihi Aralık 3, 2025, https://www.reddit.com/r/learnrust/comments/1cil2dd/decoding_developer_fields_in_fit_files/
20. example/decode.cpp doesn't show developer fields in sample FIT file [noob] - Discussion, erişim tarihi Aralık 3, 2025, https://forums.garmin.com/developer/fit-sdk/f/discussion/322523/example-decode-cpp-doesn-t-show-developer-fields-in-sample-fit-file-noob
21. FIT developer fields decoding example code - Suunto API Zone, erişim tarihi Aralık 3, 2025, https://apizone.suunto.com/fit-developer-fields-decoding-example-code
22. pandas.DataFrame.to_json — pandas 2.3.3 documentation - PyData |, erişim tarihi Aralık 3, 2025,

https://pandas.pydata.org/docs/reference/api/pandas.DataFrame.to_json.html
23. sending NaN in json - python - Stack Overflow, erişim tarihi Aralık 3, 2025, https://stackoverflow.com/questions/6601812/sending-nan-in-json
24. Working With JSON Data in Python - Real Python, erişim tarihi Aralık 3, 2025, https://realpython.com/python-json/
25. FIT to GeoJSON Converter Online | MyGeodata Cloud, erişim tarihi Aralık 3, 2025, https://mygeodata.cloud/converter/fit-to-geojson
26. Convert a Shapefile to GeoJSON Using Python and GeoPandas Library | by Kam - Medium, erişim tarihi Aralık 3, 2025, https://medium.com/@kambytes/convert-a-shapefile-to-geojson-using-python-and-geopandas-library-b20347c5ea92
27. Convert `Lctns.fit` to GeoJSON - arran-nz arran-nz - GitHub, erişim tarihi Aralık 3, 2025, https://github.com/arran-nz/fit-geojson