How MIDI Data Mirrors a Cloud Data Pipeline

1 Summary

When I first learned about the structure of MIDI data, I was surprised by how much it resembles a cloud ETL (Extract, Transform, Load) data pipeline. Both systems process structured streams of messages that start with configuration, flow through ordered steps, and end with a transformed output.

A **MIDI file** is not audio — it is a sequence of *events* that describe what to play, when, and how. It begins with a **header chunk** defining global settings such as file format, number of tracks, and timing division. This is analogous to a pipeline's configuration step, where schemas and parameters are set. After the header come one or more **track chunks**, each carrying its own ordered stream of events (e.g., one track for piano, another for drums). In a cloud ETL process, this resembles multiple data streams being processed in parallel and synchronized later.

Each MIDI event has two components: a **delta time** (the time since the previous event) and a **message** describing the action. Messages begin with a *status byte* (which encodes type and channel) and are followed by one or two *data bytes* for parameters such as pitch or velocity. This binary structure acts like a compact log of musical actions.

Parsing libraries such as `mido` abstract these low-level details. Instead of manually decoding bytes, developers work with high-level objects like `Message(type='note_on', note=60, velocity=100, time=480)`. This abstraction parallels how cloud tools treat data files as collections of parsed records rather than raw bytes.

Thinking of a MIDI file as a data object helped me connect it to ETL logic:

- **Extract**: Read structured binary data from disk (the MIDI file, like an S3 object).
- **Transform**: Convert event bytes into meaningful fields (note, velocity, time).
- **Load**: Use or export the interpreted data — for instance, generating sound or training an ML model.

Just as data pipelines emphasize precise ordering and timing of operations, MIDI ensures musical precision through delta times and tempo control. In both domains, consistent structure and timing are what make complex systems — or songs — run smoothly.

2 Deep Dive: Binary Format and Timing Concepts

As I explored the MIDI standard more deeply, I found that beneath its simple concept lies an elegant binary structure and precise timing model. Understanding these elements made me appreciate how MIDI balances compactness with flexibility.

2.1 Delta Time and Absolute Time

Each MIDI event is preceded by a **delta time**, which represents the number of ticks since the previous event. A delta of zero means the event happens simultaneously with the last one, while larger values introduce delays. These delta values are stored as variable-length quantities, allowing efficient encoding of small and large intervals.

To obtain an event's **absolute time**, one simply accumulates all previous delta times:

$$t_{\text{absolute}} = \sum_{i=1}^{n} \text{delta}_i$$

This approach reminded me of timestamp reconstruction in streaming data pipelines, where events are logged by offsets rather than absolute times.

2.2 Ticks per Beat (Resolution)

The timing precision of a MIDI file is defined by its **ticks per beat** (also known as pulses per quarter note, or PPQ). This value appears in the header chunk and defines how many ticks make up one quarter-note. For example, if the resolution is 480, then 480 ticks correspond to one beat. Higher values give finer control of timing, much like increasing sampling resolution in digital data.

2.3 Tempo Meta-Message and Timing Conversion

The actual tempo of playback is defined by a special meta-event: `FF 51 03 tt tt tt`, where the last three bytes specify the number of microseconds per quarter note. The default tempo (if none is set) is 500,000 µs per quarter, or 120 BPM. To convert a given number of ticks into seconds, I used the following formula:

$$\text{seconds} = \frac{\text{ticks} \times \text{tempo}}{\text{ticks\_per\_beat} \times 1{,}000{,}000}$$

For instance, at 480 ticks/beat and a tempo of 500,000 µs/beat, 480 ticks corresponds to half a second — one quarter note at 120 BPM.

2.4 Velocity and Normalization

The **velocity** parameter ranges from 0 to 127 and reflects how strongly a note was played. In data-processing terms, it's a 7-bit integer that many systems normalize to [0, 1] by dividing by 127. For example, a velocity of 102 corresponds to roughly 0.8 on a normalized scale.

2.5 Binary Structure of a MIDI File

A Standard MIDI File is divided into **chunks**: a single header chunk and one or more track chunks. The header (`MThd`) specifies file type, number of tracks, and time division, while each track (`MTrk`) contains a series of delta-time + event pairs. Each event starts with a status byte (identifying message type and channel) followed by one or more data bytes (parameters). Data bytes always have the most significant bit = 0, distinguishing them from status bytes ($\geq 0 \times 80$).

3 MIDI Concept Reference Table

Table 1: *
MIDI Concept Reference Table

Concept	Definition	Example
Delta time	Time interval (in ticks) between consecutive events. Stored as a variable-length quantity.	480 ticks = one quarter note at 480 PPQ. A delta of 0 means "play immediately."
Velocity	Intensity (0–127) of a Note On event, representing how hard a note is played. Often normalized to [0, 1].	102 → ≈ 0.80 intensity. A velocity of 0 is treated as Note Off.
Note On vs Note Off	Commands that start and stop notes. Note On = 0x9n, Note Off = 0x8n (or Note On with velocity 0).	`90 3C 64` = Note On (C4, vel 100). Later `80 3C 40` = Note Off.
Ticks per beat (PPQ)	Resolution of timing — number of ticks in one quarter note. Stored in header.	480 PPQ → 480 ticks = 1 beat. Allows 1/480-beat precision.
Tempo meta-message	Meta event `FF 51 03 tt tt tt` specifying µs per quarter note. Controls speed of playback.	`FF 51 03 07 A1 20` → 500,000 µs/qn = 120 BPM.
Status + data bytes	Low-level structure: 1 status byte (event type + channel) followed by 1–2 data bytes. Data bytes are 7-bit.	`91 40 7F` → Note On, channel 2, note 64 (E4), velocity 127.

4 Conclusion

Looking back on this exploration, I realized that studying MIDI was more than just learning a music-tech format — it reshaped how I think about data itself. At first, I approached MIDI as a musician trying to understand why my files behaved a certain way. But as I read through the specification and parsed messages byte by byte, I began seeing the same principles that power the data systems I study.

MIDI turned out to be a language of structure and timing. Each event, every delta, and even the way bytes are packed reflects deliberate design choices for efficiency and clarity. That's the same logic that drives cloud data pipelines: small, well-defined units flowing through a system in order, transforming along the way to produce something greater. When I compared a song's sequence of notes to an ETL process, it suddenly made sense — music is data, and data can be music if you process it right.

This connection between creativity and computation keeps me fascinated. It reminds me that technical precision doesn't kill expression; it enables it. Understanding how a note's velocity or timing can change a melody taught me as much about engineering as about art. Now, when I design a dataset, model, or backend system, I see rhythm and structure — the same patterns I first discovered inside a simple MIDI file.