Real-Time Roman Numeral Analysis from Live MIDI Performance Using a Distributed Multimodal Architecture

Oğuzhan Tuğral

Abstract

Automatic Roman numeral analysis is an important area in music information retrieval. While recent deep learning approaches have advanced the field, real-time harmonic analysis for MIDI input and live performance remains underexplored. Existing systems primarily focus on offline analysis of symbolic or audio data. Real-time settings introduce computational and latency constraints, and current tools for live MIDI processing do not provide comprehensive Roman numeral annotations. This work presents a three-tier distributed architecture integrating Python-based backend processing with tempo-adaptive temporal buffers, WebSocket bidirectional communication protocols, and JavaScript-based frontend analysis implementing pitch class arithmetic, interval vector computation, and comprehensive Roman numeral lookup algorithms. The system successfully processes MIDI input streams in real time, identifying chord qualities, inversions, diatonic and chromatic functions, and secondary dominants with latency characteristics suitable for live performance. A browser-based interface provides immediate synchronized visual feedback. The integrated pipeline enables harmonic analysis for both pre-recorded MIDI files and live keyboard performance, making real-time Roman numeral analysis accessible for pedagogical, compositional, and music cognition research applications.

November 16, 2025

Contents

1 Introduction

Automatic Roman numeral analysis (RNA) has progressed through statistical, rule-based, and deep learning approaches; however, real-time harmonic interpretation—especially from live MIDI input—remains largely unexplored. More specifically, existing systems achieve moderate accuracy on offline symbolic or audio data, yet they struggle with the latency, consistency, and contextual demands of interactive performance. Given these challenges, this work presents an integrated real-time framework combining MIDI acquisition, tempo-adaptive backend processing, bidirectional WebSocket communication, and frontend harmonic analysis. Within this unified structure, the system performs interval arithmetic, chord-quality detection, inversion and chromatic evaluation, and secondary-dominant classification, thereby providing synchronized Roman numeral feedback suitable for live performance and pedagogical use. After this present outline of the motivation and general scope of the system, the following subsection contextualizes this study within the broader research landscape.

1.1 Background

Automatic Roman numeral analysis (RNA) occupies a central place in music information retrieval (MIR), aiming to annotate harmonic function across symbolic and audio data. While early work leaned on statistical heuristics and rule-based systems, recent developments have increasingly turned toward deep learning architectures. In this regard, methods based on convolutional–recurrent neural networks (CRNNs) and graph neural networks (GNNs) have shown notable improvements by capturing both local harmonic cues and broader structural context (Fricke et al., 2024; Karystinaios & Widmer, 2023; Micchi et al., 2020). In parallel, the adoption of richer modeling strategies—such as full pitch spelling and multitask prediction of root, inversion, quality, and local key—has further expanded model expressiveness and interpretability (Karystinaios & Widmer, 2023; Micchi et al., 2020).

Progress has also been shaped by the availability of curated datasets. Collections like TAV-

ERN and various aggregated meta-corpora have enabled wider evaluation and training, though the field continues to grapple with limited stylistic diversity and inconsistencies in annotation practice (Devaney et al., 2015; Micchi et al., 2020; Tymoczko et al., 2019). Moreover, ongoing standardization efforts, including Harmalysis and RomanText, aim to alleviate these issues by promoting interoperable formats and more uniform annotation methodologies (López & Fujinaga, 2020; Tymoczko et al., 2019).

In terms of open-source RNA tools and current performance limitations, a wide range of toolkits and research frameworks now support automatic Roman numeral analysis, spanning CRNN-based models, GNN architectures, and context-free grammar parsers (Fricke et al., 2024; Karystinaios & Widmer, 2023; López & Fujinaga, 2020; Micchi et al., 2020; Tymoczko et al., 2019). AugmentedNet, for example, has been extended to operate directly on audio and MIDI data, achieving performance comparable to approaches that rely solely on symbolic input (Fricke et al., 2024). Nevertheless, even the most advanced systems typically attain only about 43% accuracy on challenging benchmarks, highlighting a significant gap between automated methods and expert human analysis (Micchi et al., 2020).

Beyond expert–annotated data, real-time automatic RNA for MIDI files and live performances remains particularly underexplored. Although tools such as Notochord and Scramble support real-time MIDI processing, generation, and corrective feedback, their focus lies in performance modeling, improvisation, or error detection rather than delivering robust harmonic interpretations (Marinov, 2020; Privato et al., 2022; Shepardson et al., 2022). Thus, despite their technical strengths, these systems do not yet provide expert-quality Roman numeral annotations in real time. Having established the background and current limitations, the next subsection outlines the specific problems that motivate the present research.

1.2 Problems

In terms of RNA accuracy and data constraints, several persistent limitations contribute to this gap. First, harmonic analysis is intrinsically challenging: establishing ground-truth annotations often involves ambiguity and subjective judgment, leading to inconsistencies across datasets that undermine model reliability (Devaney et al., 2015; Micchi et al., 2020). Compounding this issue is the limited availability of sufficiently large and stylistically diverse annotated corpora, which restricts the generalizability of supervised models across varied repertoires and performance conditions (Devaney et al., 2015; Jamshidi et al., 2024; Micchi et al., 2020).

Real-time settings introduce additional constraints. Harmonic predictions must be produced within a few milliseconds to remain musically coherent, placing considerable computational and latency demands on existing architectures (Marinov, 2020; Privato et al., 2022; Shepardson et al., 2022). At the same time, current systems for transcription, harmonic analysis, and real-time interaction tend to operate independently, leaving a noticeable absence of integrated, end-to-end frameworks capable of delivering seamless harmonic feedback (Benetos et al., 2019; Jamshidi et al., 2024; Marinov, 2020).

Although recent research—supported by deeper representations, expanded datasets, and improved learning strategies—has advanced automatic RNA, these developments have not yet translated into expert-level, real-time performance whether for MIDI or real-time input (Fricke et al., 2024; Jamshidi et al., 2024; Karystinaios & Widmer, 2023; Marinov, 2020; Micchi et al., 2020). Consequently, this remaining challenge offers a valuable direction for my work, which aims to address this need by introducing a practical, real-time framework for Roman numeral analysis capable of supporting both MIDI file playback and live performance. By emphasizing efficient representations, low-latency processing, and harmonic modeling, the system moves toward achieving expert-level real-time RNA in practical musical settings. With these problems articulated, the following section turns to the design of the proposed system and its implementation.

2 Automated Annotation of Roman Numeral Harmony

As its first and foremost function, traditional harmonic analysis requires recognizing pitch combinations, identifying chord qualities, interpreting their functions within an established tonal framework, and articulating these relationships through Roman numeral notation—a standardized symbolic language fundamental to Western music theory and practice. The proposed system in the present work implements a real-time Roman numeral analysis framework designed to operate seamlessly in both MIDI playback and live performance contexts, which continuously ingests live and streamed MIDI data, extracts and organizes recent pitch information, identifies harmonic structures, interprets them within a defined tonal framework, and delivers dynamic Roman numeral analysis through an interactive visual interface.

To clarify how these components interact, Fig. 1 summarizes the complete real-time analysis pipeline. As a preconditional level, Layer 0 – 2.3 Input Layer and External Systems receives live MIDI performance and other control data via OSC/UDP from Digital Audio Workstations (DAWs), hardware, or controllers, transforming raw performance input into timestamped MIDI events. Layer 1 – 2.4 Backend Layer (Python) handles these events, converts notes to pitch classes, maintains tempo-adaptive temporal buffers, manages OSC/UDP networking, and coordinates WebSocket messaging to and from the browser. Then, Layer 2 – 2.5 Transport Layer (WebSocket) provides the bidirectional messaging channel: it carries configuration changes from the frontend to the backend and broadcasts state updates—including pitch buffers, tempo, meter, and voice information—back to the browser. Finally, Layer 3 – 2.6 Frontend Analysis Layer receives these broadcasts, performs interval arithmetic and chord-template matching, computes Roman numeral labels, and renders the harmonic analysis UI in real time. Each node and sub-bullet corresponds to the numbered subsections 2.3–2.6., which will be discussed in detail in the following sections of the present work.

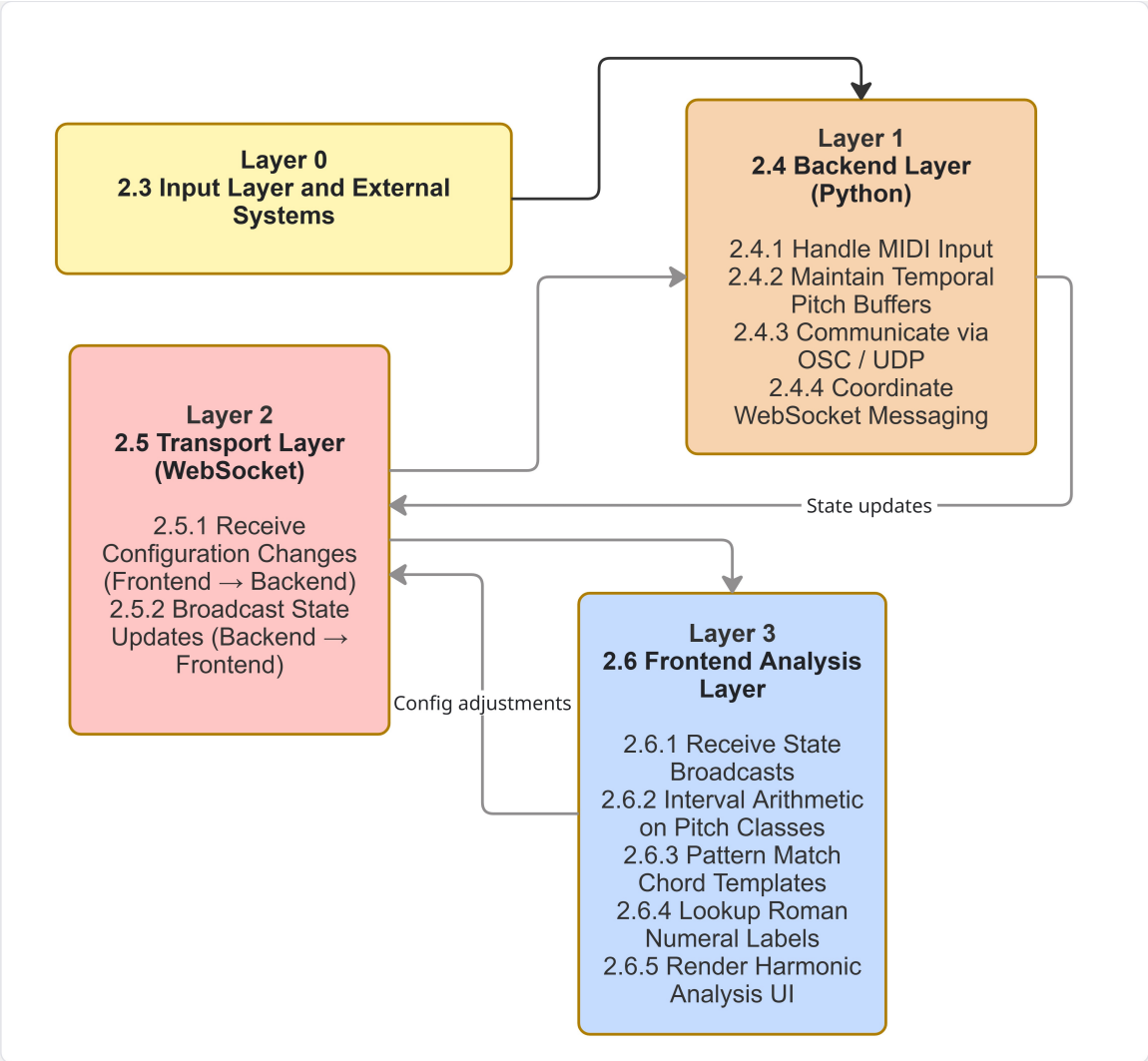

Figure 1: General Work Flow

2.1 Input Layer and External Systems

As a precondition, the input layer serves as the primary interface between external musical performance devices and the system's processing architecture. This layer establishes communication channels, receives MIDI data streams, and prepares incoming information for subsequent harmonic analysis operations. Thus, it consists of two functions: MIDI input and OSC/UDP protocols as external systems.

2.1.1 MIDI Input

The MIDI input subsystem receives real-time musical performance data from external sources, including digital audio workstations, hardware synthesizers, and MIDI controllers.

As a conceptual bridge to the next section, it is worth noting that communication occurs through Open Sound Control (OSC) and the User Datagram Protocol (UDP). Each MIDI message contains three primary components: pitch information encoded as note numbers (0–127), velocity data representing performance dynamics, and timing metadata. The system processes these incoming messages at millisecond-level temporal resolution, ensuring that rapid performance gestures and dense musical textures are accurately captured for downstream harmonic analysis.

Table 1: MIDI Message Components

Component	**Range**	**Description**
Note Number	0–127	Pitch representation
Velocity	0–127	Performance dynamics
Timestamp	Milliseconds	Event timing

2.1.2 External Systems (OSC / UDP)

Open Sound Control (OSC) and the User Datagram Protocol (UDP) function as external systems that communicate with the architecture, providing the low-latency channels through which real-time musical data is transmitted. OSC offers structured message formatting with type-tagged arguments, enabling expressive and semantically rich parameter passing. UDP, by contrast, provides lightweight packet transmission without connection overhead or guaranteed delivery. Consequently, choosing between these protocols depends on the programming requirements of the system, particularly in contexts where millisecond-level latency is critical. External sources—including Ableton Live, Max/MSP, and hardware MIDI interfaces—transmit performance data through these channels. The system binds to specific network ports, typically port 8000 for OSC and port 9000 for UDP, and listens continuously for incoming musical event streams from connected devices. Once these external systems are configured, the three essential layers of the program become active.

2.2 Backend Layer (Python)

The backend layer, implemented in Python, serves as the system's central processing engine, receiving MIDI data from external sources, validating and transforming incoming messages, maintaining temporal buffers of pitch events, managing bidirectional communication protocols, and coordinating state distribution to frontend clients for real-time harmonic analysis. In the following subsections, I present the details of each of these steps, thereby establishing the foundation upon which the transport and analysis layers operate.

2.2.1 Handle MIDI Input

The backend receives normalized MIDI messages from external sources such as Ableton Live and Max/MSP, performing initial validation and preprocessing operations. Invalid or malformed messages are filtered to ensure system stability. The subsystem converts MIDI note numbers into pitch-class representations using modulo-12 arithmetic, where each pitch class $p \in \{0, 1, 2, \ldots, 11\}$ denotes a chromatic position independent of octave. Throughout this process, the handler monitors incoming data rates and dynamically adjusts processing parameters to maintain real-time performance across varying musical textures and note densities. These processed MIDI notes then become the input samples stored within tempo- and time-signature–adaptive buffers used for subsequent harmonic analysis. With MIDI data now validated and normalized, the system transitions naturally into the task of organizing these events within temporal structures suitable for harmonic interpretation.

2.2.2 Maintain Temporal Pitch Buffers

The backend implements circular buffer structures to store recent pitch events for harmonic analysis over configurable time windows. These buffers operate as first-in-first-out (FIFO) queues where incoming pitch events displace the oldest stored values upon reaching capacity. Critically, the system converts unordered FIFO arrivals into temporally ordered lists, ensuring chronological coherence for downstream harmonic analysis (e.g.,

bass and soprano detection).

The temporal window duration Δt directly influences analysis granularity. Shorter windows ($\Delta t \approx 200$–500 ms) provide fine-grained temporal resolution but may yield incomplete harmonic information, whereas longer windows ($\Delta t \approx 1$–3 s) capture fuller chord structures at the expense of temporal precision. At this stage of processing, it is advantageous to keep these buffers as short as possible, since a second buffering layer in the JavaScript stage later adjusts the effective window size according to user-selected parameters such as bar length and half-bar subdivisions. This design ensures that the initial Python buffer remains lightweight and responsive, while higher-level temporal organization is handled at the frontend.

Thus, the buffer capacity `smCapacity` is dynamically determined through WebSocket configuration messages from the JavaScript frontend, calculated based on current tempo (BPM) and time signature. The relationship is expressed as:

$$\texttt{smCapacity} = f(\text{tempo}, \text{time signature}, \Delta t)$$

Table 2: Temporal Buffer Configuration Parameters

Parameter	**Range**	**Description**
Δt	100–5000 ms	Window duration
`smCapacity`	50–500 events	Tempo-adaptive buffer limit
Tempo	40–240 BPM	Musical tempo

The system sorts buffered events by timestamp before forwarding them to analysis stages, transforming potentially out-of-order network arrivals into chronologically ordered pitch sequences essential for accurate harmonic analysis. Additionally, the backend identifies bass and soprano voices by determining the lowest and highest pitch values within the temporal buffer, providing essential voice-leading information for Roman numeral determination in the frontend analysis layer. Once these temporal structures are finalized, the backend is prepared to manage communication across OSC and UDP channels, forming the next stage of processing.

2.2.3 Communicate via OSC / UDP

Once the MIDI input flow is managed through tempo- and time-signature–adaptive buffers, the backend treats the prepared external sources as active OSC and UDP communication channels through which continuous MIDI data streams are received. This subsystem manages all network socket operations, including port binding, packet reception, and protocol-specific message parsing. The module operates asynchronously using event-driven programming, ensuring that network I/O operations never block critical processing tasks. Robust error-handling routines address packet loss, transmission delays, and temporary disconnections. Furthermore, timeout mechanisms detect when MIDI streams cease, triggering reconnection logic that automatically re-establishes communication with external sources without requiring manual intervention. With reliable communication established, the backend is then positioned to coordinate real-time message exchange with the frontend through WebSocket protocols.

2.2.4 Coordinate WebSocket Messaging

Before data originating from input sources is transmitted bidirectionally to and from the frontend, the backend manages the entire WebSocket connection lifecycle, including connection establishment, keepalive mechanisms, and disconnection handling. This coordinator maintains a registry of active frontend connections and implements broadcast mechanisms for efficient state distribution. Message-routing logic determines the appropriate recipients based on client configurations, ensuring correct delivery ordering for sequential state updates. At this stage, the backend operations are fully prepared to interface with the second layer of the system, enabling real-time synchronization through the transport layer.

2.3 Transport Layer (WebSocket)

WebSocket is a full-duplex communication protocol that enables persistent, bidirectional data exchange between a client and a server over a single, long-lived connection. This layer manages state broadcasts from backend to frontend and configuration updates from

frontend to backend, ensuring synchronized operation across distributed system components while maintaining low-latency message delivery essential for real-time harmonic analysis. With backend readiness in place, the transport layer assumes responsibility for transmitting system state efficiently and reliably.

2.3.1 Broadcast State Updates (Backend → Frontend)

The transport layer implements a unidirectional state broadcast mechanism that continuously transmits backend state information to connected frontend clients through WebSocket channels. These broadcasts contain current pitch buffer contents, identified bass and soprano voices, timing parameters, tempo information, time signature data, and configuration values required for frontend harmonic analysis operations.

Table 3: Transport Layer Message Types

Direction	**Content**	**Purpose**
Frontend → Backend	Δt, tempo, time signature, analysis mode	Runtime parameter configuration
Backend → Frontend	Pitch buffers, bass/soprano, tempo, time signature	State synchronization for analysis

These broadcasts ensure that all connected frontend clients receive consistent and timely system updates. As a complement to these outgoing streams, the transport layer must also handle user-driven configuration changes arriving from the frontend.

2.3.2 Receive Configuration Changes (Frontend → Backend)

The transport layer also implements a bidirectional channel for configuration messages flowing from frontend to backend, enabling user interfaces to modify backend parameters during runtime. Configuration messages include adjustments to temporal window duration Δt, tempo (BPM), time signature, and analysis mode settings. These messages travel through WebSocket connections, distinguished by message type identifiers for proper routing.

The backend validates incoming configuration requests to ensure parameters fall within acceptable ranges and prevent incompatible setting combinations that could compromise

system stability. Validation checks confirm that tempo remains within 40–240 BPM and Δt stays within 100–5000 ms bounds. The subsystem provides acknowledgment messages back to requesting frontends, confirming successful configuration changes or reporting validation errors. These validation and acknowledgment steps complete the transport layer's responsibility for reliable bidirectional coordination, ensuring that the system can safely progress to the subsequent processing stage.

2.4 Frontend Analysis Layer

Once input data arrives at the backend and the initial preprocessing steps described above are completed, the refined musical features are transmitted to the frontend for higher-level interpretation. The Frontend Analysis Layer processes real-time musical data received from the backend and transforms it into symbolic harmonic interpretations displayed in the user interface. Operating asynchronously over WebSocket communication, the frontend continuously updates its internal state structures containing pitch-class sets, bass and soprano identifications, tempo, meter, and key information. Interval arithmetic converts absolute pitches into normalized interval patterns, which are then matched against predefined chord templates to determine chord quality and root position. Using this information, the Roman numeral lookup subsystem assigns functional labels based on scale degree, chord quality, inversion, and secondary-dominant relationships. Finally, the rendering engine presents Roman numeral labels and related musical attributes in a responsive browser interface optimized for clarity and low-latency performance during live musical interaction.

2.4.1 Receive State Broadcasts

The frontend layer receives continuous state broadcasts from the backend through WebSocket connections, establishing the foundation for real-time harmonic analysis. These broadcasts contain temporally ordered pitch sequences, identified bass and soprano voices, current tempo (BPM), time signature, temporal window duration Δt, and system configuration parameters. The frontend implements event listeners that parse incoming JSON

messages and update internal state representations.

Upon receiving each broadcast, the frontend validates message integrity by checking sequence numbers to detect potential packet loss or out-of-order delivery. Missing sequences trigger re-synchronization requests to the backend. The received pitch data populates local data structures that subsequent analysis stages consume. While bass and soprano identifications provide voice-leading context essential for Roman numeral determination, tempo and time-signature information enable metric-aware analysis, allowing the system to align harmonic interpretations with musical meter and phrase boundaries.

From this point onward, the system activates its novel analysis pipeline, transforming the incoming musical data into its final Roman numeral interpretation.

2.4.2 Interval Arithmetic on Pitch Classes

Building directly on the received state broadcasts, the frontend performs interval arithmetic on received pitch classes to extract harmonic relationships essential for chord identification. This subsystem, implemented primarily in JavaScript files, converts absolute pitch values into relative interval structures that characterize chord quality independent of transposition.

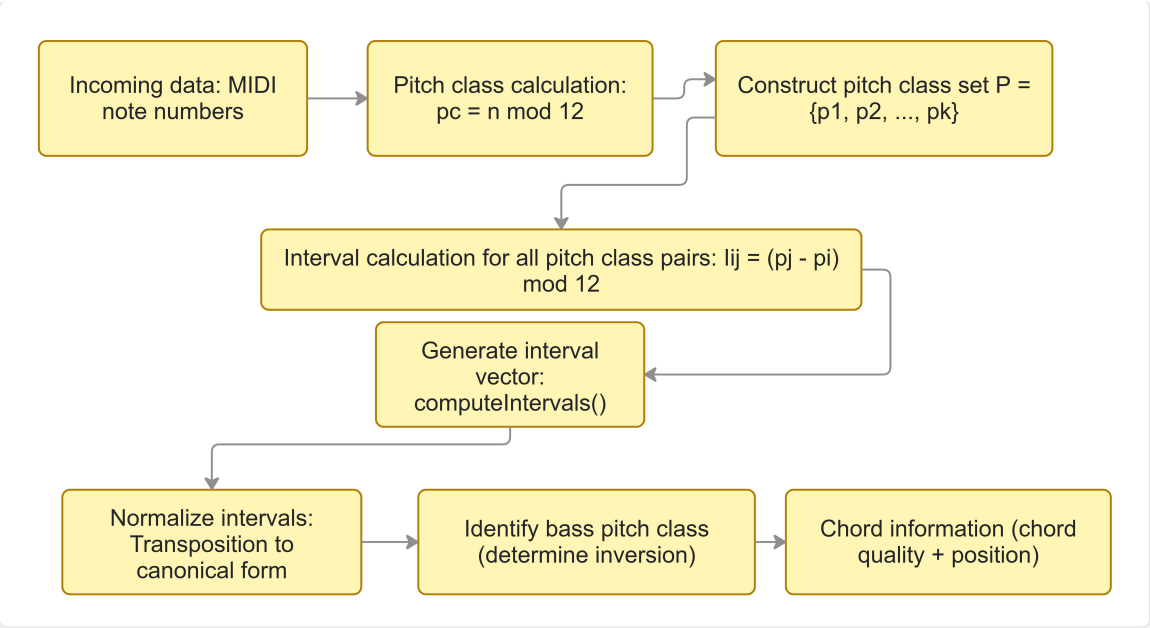

Figure 2: Interval Arithmetic on Pitch Classes: Processing Pipeline

The first node in Fig. 2 represents the data received from the backend, whose details have been described above. To summarize the process in the backend, it begins by reducing MIDI note numbers to pitch classes using modulo-12 arithmetic:

$$p_c = n \bmod 12$$

where n represents the MIDI note number (0–127) and p_c yields the pitch class (0–11), with 0 = C, 1 = C♯/D♭, continuing through 11 = B, independent of octave register.

From this point, the system constructs a pitch class set $P = \{p_1, p_2, \ldots, p_k\}$ containing all unique pitch classes present within the current temporal buffer, where k represents the cardinality of the set and each element $p_i \in \{0, 1, 2, \ldots, 11\}$ corresponds to a distinct chromatic pitch class. The temporal buffer's parameters—window duration Δt and capacity `smCapacity`—are dynamically configured through WebSocket messages from the frontend based on tempo (BPM) and time signature, determining which pitch events contribute to the set P at any given analytical moment.

To characterize the internal structure of this set, the system calculates intervals between all pitch class pairs within set P using the formula:

$$I_{ij} = (p_j - p_i) \bmod 12$$

where p_i and p_j represent any two pitch classes in the set, and I_{ij} yields the ascending interval from p_i to p_j expressed in semitones (0–11). This exhaustive pairwise comparison generates a complete intervallic profile characterizing the harmonic relationships within the current pitch collection.

On the basis of these pairwise relationships, the `computeIntervals()` function in the program generates an interval vector that quantifies the frequency of each interval class present in the pitch class set. This vector counts occurrences of interval classes 0 through 6 (unison, minor second, major second, minor third, major third, perfect fourth, and tritone), providing a compact numerical representation of the set's intervallic content that

characterizes chord quality independent of voicing or octave placement. For example, a C major triad with pitch class set $P = \{0, 4, 7\}$ (C, E, G) produces an interval vector indicating one major third (4 semitones), one minor third (3 semitones), and one perfect fifth (7 semitones).

Subsequently, the `normalizeIntervals()` function transposes an interval set into a canonical form, enabling comparison against chord templates regardless of the chord's root position or transposition level. This comparison matches to an item of a finite set which represents all combinational possibilities in the Western music system, as explained in Appendix I. This normalization process rotates the pitch-class set to a standard reference point—typically aligning the lowest pitch class to 0 or establishing a consistent ordering criterion. By standardizing interval structures, the system can match harmonically equivalent chords that differ only in their absolute pitch level, facilitating efficient template-based chord recognition in subsequent processing stages. For example, an E major triad $P = \{4, 8, 11\}$ (E, G♯, B) normalizes to $P' = \{0, 4, 7\}$ by transposing down 4 semitones, yielding the same canonical form as a C major triad and thus enabling recognition of the major-triad structure independent of root pitch.

In parallel, the system identifies the bass pitch class by determining the lowest pitch value within the temporal buffer, providing essential information for inversion analysis. The bass pitch class, denoted as p_{bass}, combined with the normalized chord structure, enables the system to distinguish between root position and inverted chords. For instance, a C major triad with $p_{\text{bass}} = 0$ (C) indicates root position, while $p_{\text{bass}} = 4$ (E) indicates first inversion (C/E), and $p_{\text{bass}} = 7$ (G) indicates second inversion (C/G), critical distinctions for accurate Roman numeral representation and voice-leading analysis.

Taken together, the output of the interval arithmetic subsystem provides comprehensive chord information encompassing chord quality (major, minor, diminished, augmented, seventh chords, etc.) derived from the interval vector analysis, and chord position (root position, first inversion, second inversion) determined by the bass pitch class identification. This structured chord information serves as the primary input for the subsequent Roman

numeral label lookup stage, where the system assigns functional harmonic labels based on the identified chord quality, inversion, and tonal context established by key signature and harmonic progression analysis.

2.4.3 Pattern Match Chord Templates

Building on the interval vector representation, the frontend performs pattern matching by comparing the normalized interval vector against a predefined library of chord templates, which is the selected number of items from all possible 2048 templates explained in the appendix. This template library contains canonical interval vectors for standard chord types including major triads, minor triads, diminished triads, augmented triads, dominant seventh chords, major seventh chords, minor seventh chords, half-diminished seventh chords, and fully diminished seventh chords, among other extended harmonies.

The matching algorithm, implemented in the `matchChordTemplate()` function, computes similarity scores between the observed interval vector and each template using distance metrics or exact matching criteria. The system employs a threshold-based approach where the template with the highest similarity score above a minimum threshold is selected as the chord identification. When multiple templates yield comparable scores, the algorithm prioritizes simpler chord structures following principles of parsimony in music-theoretic analysis.

Importantly, the template matching process accounts for enharmonic equivalence and voice doubling variations common in keyboard performance. For instance, a C major triad remains identifiable whether voiced as $\{0, 4, 7\}$ or with octave doublings as $\{0, 0, 4, 7, 7\}$, since the interval vector computation focuses on pitch class content rather than absolute note counts. Successfully matched templates return chord quality labels (e.g., “major”, “minor7”, “dim”) that combine with bass pitch class information to form complete chord symbols ready for Roman numeral assignment based on the established tonal center. With chord templates matched and chord qualities identified, the system is now prepared to embed these chords within a tonal framework through Roman numeral labeling.

2.4.4 Lookup Roman Numeral Labels

While the processes described thus far enable the system to identify chord qualities, the primary challenge remains determining how these qualities are situated within a tonal context in order to derive an appropriate Roman numeral analysis—the central goal of the present work. As shown in Figure 3, the Lookup Roman Numeral Labels subsection processes the identified chord quality and bass pitch class (determined in previous sections) by mapping them to functional harmonic designations through root identification, tonic comparison, scale-degree calculation, diatonic evaluation, inversion analysis, and secondary-function detection. These stages collectively yield the system's complete Roman numeral annotations.

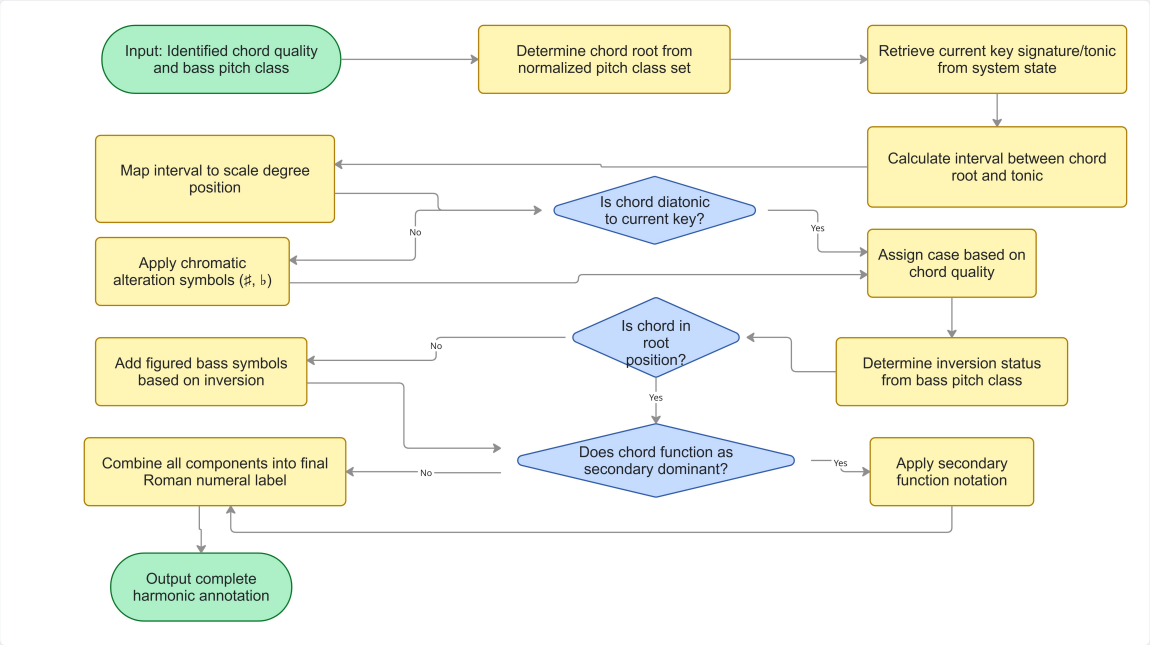

Figure 3: Lookup Roman Numeral Labels: Processing Pipeline

After the previous stages produce refined musical data, the system begins by determining the chord root from the normalized pitch-class set obtained through interval-arithmetic analysis. The chord root represents the fundamental pitch on which the chord is built, serving as the reference for scale-degree identification and Roman numeral classification. For chords in root position—where the bass pitch class corresponds directly to the chord root—the identification is straightforward: $p_{\text{root}} = p_{\text{bass}}$. In contrast, inverted chords require additional evaluation, as the bass pitch class no longer coincides with the root

and must therefore be inferred from the internal interval structure of the chord.

For triadic structures, the root identification algorithm analyzes interval relationships within the pitch class set $P = \{p_1, p_2, p_3\}$. The system searches for the pitch class that generates a third-stacked structure when intervals are measured upward. Specifically, it identifies the pitch class p_{root} such that:

$$p_{\text{root}} + I_1 \equiv p_2 \pmod{12}$$

$$p_{\text{root}} + I_2 \equiv p_3 \pmod{12}$$

where $I_1 \in \{3, 4\}$ (minor or major third) and $I_2 \in \{6, 7, 8\}$ (diminished fifth, perfect fifth, or augmented fifth), representing the characteristic interval structures of triadic harmony.

For seventh chords with four distinct pitch classes $P = \{p_1, p_2, p_3, p_4\}$, the algorithm extends this principle by identifying the pitch class that generates a complete tertian stack:

$$p_{\text{root}} + I_1 \equiv p_2 \pmod{12}$$

$$p_{\text{root}} + I_2 \equiv p_3 \pmod{12}$$

$$p_{\text{root}} + I_3 \equiv p_4 \pmod{12}$$

where $I_1, I_2 \in \{3, 4\}$ and $I_3 \in \{9, 10, 11\}$ (diminished seventh, minor seventh, or major seventh).

The `findChordRoot()` function in `script.js` implements these algorithms, testing each pitch class in the set as a potential root candidate and evaluating whether the remaining pitches form valid tertian intervals above it. When multiple candidates satisfy the tertian criteria, the system applies heuristic rules prioritizing simpler interpretations and favoring roots that align with common-practice harmonic progressions within the established tonal context.

Once the root has been determined, the process advances to the next stage. The system then obtains the active key signature and tonic pitch class from backend state updates

delivered over the WebSocket transport layer. This data, maintained within the frontend's state management framework, establishes the tonal context required for computing scale degrees and assigning Roman numerals. The tonic pitch class $p_{\text{tonic}} \in \{0, 1, 2, \ldots, 11\}$ functions as the reference for all harmonic function analysis, while the key signature (major or minor) specifies the underlying diatonic set against which any chromatic deviations are detected.

After the user selects the key and key quality, the next stage computes the interval between the identified chord root and the tonic pitch class to determine the chord's scale-degree position within the key. This interval is calculated using modulo-12 arithmetic:

$$I_{\text{scale}} = (p_{\text{root}} - p_{\text{tonic}}) \bmod 12$$

where I_{scale} represents the number of semitones above the tonic. This value directly maps to scale degrees: 0 = I, 2 = ii, 4 = iii, 5 = IV, 7 = V, 9 = vi, 11 = vii°, establishing the foundational Roman numeral designation before quality and inversion symbols are applied.

At the next stage, which can be followed by arrows in Fig. 3, the program maps the calculated interval I_{scale} to its corresponding scale degree position using a lookup table that associates semitone distances with Roman numeral designations. The mapping follows standard diatonic scale theory: 0 semitones maps to I (tonic), 2 to ii (supertonic), 4 to iii (mediant), 5 to IV (subdominant), 7 to V (dominant), 9 to vi (submediant), and 11 to vii° (leading tone). This translation converts the numerical interval calculation into a music-theoretically meaningful scale degree representation that forms the basis of the Roman numeral label.

At this step, a decision point emerges. The program evaluates whether the identified chord belongs to the diatonic collection of the current key by comparing its pitch class content against the expected scale degrees. A chord is considered diatonic if all its constituent pitch classes can be found within the major or minor scale defined by the current key signature, without requiring chromatic alterations. If the chord is diatonic, the system

proceeds directly to assign Roman numeral case based on chord quality: major chords receive uppercase numerals (I, IV, V), minor chords receive lowercase numerals (ii, iii, vi), and diminished chords receive lowercase numerals with degree symbols (vii°). Augmented chords, though rare in diatonic contexts, receive uppercase numerals with plus signs (III+).

If the chord is non-diatonic, containing one or more chromatic alterations, the system applies accidental symbols ($\sharp, \flat, \natural$) to the Roman numeral to indicate the altered scale degree. For example, in C major, an A♭ major chord would be notated as ♭VI, indicating the lowered sixth scale degree. The system compares each pitch class in the chord against the diatonic collection, determines which scale degrees have been raised or lowered, and prefixes the appropriate accidental to the Roman numeral.

Then, a second decision point determines whether the chord is in root position or inversion. If it is not in root position, the program determines inversion status by comparing the bass pitch class p_{bass} with the identified chord root p_{root}. If $p_{\text{bass}} = p_{\text{root}}$, the chord is in root position and receives no inversion symbol. If the bass matches the chord's third, the chord is in first inversion (notated with ^{6} or $^{6}_{3}$). If the bass matches the fifth, the chord is in second inversion (notated with $^{6}_{4}$). For seventh chords, third inversion occurs when the seventh appears in the bass (notated with $^{4}_{2}$ or ^{2}), providing complete figured bass notation for the harmonic analysis.

If the chord is in root position, the system next determines whether it functions as a secondary dominant. Rather than relying solely on the chord's actual resolution, the analysis first checks whether the chord contains dominant-function pitch content—typically a major triad (optionally with a minor seventh) that includes at least one pitch class lying outside the home-key collection. When such chromatic alterations are present, the system evaluates which possible tonal center the four pitch classes most strongly imply, selecting the candidate key whose dominant-function template best matches the chord's interval structure. The relationship between this implied temporary key and the global home key is then used to assign the appropriate secondary dominant label, expressed in slash

notation (e.g., V/V or V^7/IV). This evaluation is carried out within a temporal window by the `analyzeSecondaryFunction()` routine, which assesses the harmonic context to refine the final annotation.

When secondary dominant function is confirmed, the system applies secondary function notation using slash syntax that indicates the target chord of the temporary tonicization. The notation format is [function]/[target], where the function represents the dominant or leading-tone relationship (V, V^7, vii°, vii°^{7}) and the target represents the diatonic chord being tonicized (ii, iii, IV, V, vi). For example, V/V indicates the dominant of the dominant, while vii°^{7}/vi represents a fully diminished seventh chord functioning as the leading-tone chord of the submediant.

If the chord does not function as a secondary dominant, or after secondary function notation has been applied, the system combines all analytical components into the final Roman numeral label. This combination assembles the scale degree Roman numeral, chord quality indicators (M7 for major seventh, ø7 for half-diminished seventh), chromatic alteration symbols (♯, ♭), figured bass inversion symbols (^{6}, $^6_4$, $^6_5$, $^4_3$, $^4_2$), and secondary function notation into a complete, standardized representation.

The final output produces complete harmonic annotations such as "I", "ii^6", "V^6_5/IV", "♭VI", or "vii°^{7}/V", encoding comprehensive functional harmonic information in compact symbolic form. These annotations are then rendered in the user interface display, synchronized with the real-time musical performance through the continuous WebSocket state broadcast mechanism from the backend processing layer, providing immediate visual feedback of harmonic analysis results to performers and analysts.

2.4.5 Render Harmonic Analysis UI

To present these analytical results to the user, the frontend rendering subsystem displays complete harmonic analysis results through a browser-based user interface implemented in HTML, CSS, and JavaScript. This subsystem, primarily managed in `index.html` and styled through associated CSS files, presents Roman numeral labels, chord symbols, pitch

class information, and temporal context in a synchronized, real-time visualization that updates continuously as the musical performance progresses.

The user interface organizes information hierarchically across multiple display regions. The primary analysis region shows the current Roman numeral annotation in large, readable typography, ensuring immediate visibility during live performance. Secondary regions display supporting information including the identified chord quality, bass and soprano pitch classes, current tempo (BPM), time signature, and temporal window duration Δt. Additional interface elements provide access to configuration controls allowing users to adjust analysis parameters and system settings through frontend-to-backend WebSocket messages.

The rendering engine, implemented in JavaScript, updates the display asynchronously upon receiving state broadcasts from the backend. The `updateUI()` function parses incoming JSON messages and maps data fields to corresponding DOM elements using JavaScript DOM manipulation methods. Animation and transition effects provide visual continuity between harmonic changes, with configurable fade durations and color coding that distinguishes diatonic harmonies from chromatic alterations.

Table 4: User Interface Display Components

Component	**Information Displayed**
Primary Analysis Region	Current Roman numeral label
Chord Quality Display	Major, minor, diminished, augmented, seventh
Voice Information	Bass pitch class, soprano pitch class
Temporal Context	Tempo (BPM), time signature, Δt
Configuration Panel	Parameter adjustment controls
History Timeline	Previous harmonic annotations

The interface implements responsive design principles ensuring compatibility across various display sizes and devices. The rendering system maintains consistent frame rates through efficient DOM update strategies that minimize reflow and repaint operations, ensuring smooth visual performance even during rapid harmonic changes or dense polyphonic textures that generate frequent analysis updates.

3 Conclusion

Throughout this work, I presented a comprehensive real-time system for automatic Roman numeral harmonic analysis operating on live MIDI input. Through the integration of Python-based backend processing, WebSocket communication protocols, and JavaScript-based frontend analysis, I developed a distributed architecture capable of delivering expert-level harmonic analysis with latency characteristics suitable for live musical performance.

The system's three-tier architecture successfully addresses the fundamental challenges of real-time harmonic analysis. The backend layer efficiently manages MIDI input streams, maintains tempo-adaptive temporal buffers, and identifies bass and soprano voices essential for accurate Roman numeral determination. The transport layer provides reliable, low-latency bidirectional communication enabling dynamic parameter adjustment during performance. The frontend layer implements sophisticated algorithms for interval arithmetic, chord template matching, and Roman numeral lookup, producing complete harmonic annotations including diatonic and chromatic functions, inversions, and secondary dominants.

Key contributions of this work include the temporal buffering mechanism with dynamic capacity adjustment based on musical tempo and time signature, the integration of pitch class set theory with pattern matching algorithms for robust chord identification, and the comprehensive Roman numeral lookup pipeline that handles complex harmonic phenomena including modal mixture and tonicization. The system's browser-based user interface provides immediate visual feedback synchronized with ongoing musical performance, making harmonic analysis accessible in real-time pedagogical and performance contexts.

Future research directions include extending the system to support more complex harmonic vocabularies beyond common-practice tonality, incorporating machine learning approaches for improved chord recognition in ambiguous contexts, and developing automated key detection algorithms that adapt to modulation and tonal ambiguity. Additionally, in-

tegration with multimodal data sources such as EEG signals could enable investigation of neurological correlates of harmonic perception during real-time analysis, contributing to music cognition research methodologies.

Appendix

A Combinatorial Possibilities in 12-Tone Diatonic and Chromatic Music

One of the crucial components described in this work is the interval-arithmetic subsystem. At the end of this process, the system produces a numerical vector that functions as a structural fingerprint of the chord. This vector is then matched against a finite dictionary of predefined interval patterns, enabling the system to determine the appropriate Roman numeral label. In Western music theory, this finite dictionary corresponds to the twelve pitch classes and the full set of their possible combinations. When encoded as ordered pitch-class sets anchored at center 0, these combinations yield 2048 distinct configurations—the complete lexical space available for harmonic structures in Western classical music. This section explains the mathematical foundations that support this analytical approach.

In the context of real-time symbolic harmonic recognition, pitch-class sets serve as the foundational building blocks for detecting and labeling chordal structures. The complete chromatic collection in Western music consists of 12 pitch classes $\{0, 1, 2, \ldots, 11\}$, representing the notes from C to B. In computational terms, any musical event within a temporal window (as defined by `smCapacity`) can be represented as a subset of these pitch classes. It is the very first step for the program to store these subsets or combinations in `smCapacity`.

A.1 Derivation of Pitch Class Set Combinations

Let $S = \{0, 1, 2, \ldots, 11\}$ denote the full chromatic set. The number of k-element subsets that can be chosen from S is defined by the binomial coefficient:

$$\binom{12}{k} = \frac{12!}{k!(12-k)!}, \quad \text{for } 1 \leq k \leq 12 \tag{1}$$

The total number of non-empty subsets is:

$$\sum_{k=1}^{12} \binom{12}{k} = 2^{12} - 1 = 4095 \tag{2}$$

To reduce analytical redundancy and enforce a consistent bass-based orientation, this system filters only those subsets where the first pitch class is 0 (C). This constraint emphasizes bass-normalized pitch-class sets and reduces the search space significantly while retaining representative harmonic structures across all cardinalities.

In this constrained model, we fix 0 as the first element of each subset and select the remaining $k-1$ elements from the remaining 11 pitch classes $\{1, 2, \ldots, 11\}$. Therefore, the number of valid k-element subsets starting with 0 is:

$$\binom{11}{k-1} = \frac{11!}{(k-1)!(11-(k-1))!}, \quad \text{for } 1 \leq k \leq 12 \tag{3}$$

The total number of such subsets across all values of k is:

$$\sum_{k=1}^{12} \binom{11}{k-1} = \sum_{j=0}^{11} \binom{11}{j} = 2^{11} = 2048 \tag{4}$$

For example:

- $k = 2$: Combinations starting with 0 include $\{0, 1\}, \{0, 2\}, \ldots, \{0, 11\}$ totaling 11 subsets.
- $k = 3$: Combinations include $\{0, 1, 2\}, \{0, 1, 3\}, \ldots$ yielding $\binom{11}{2} = 55$ subsets.

- $k = 4$: Combinations include $\{0, 1, 2, 3\}, \{0, 1, 2, 4\}, \ldots$ yielding $\binom{11}{3} = 165$ subsets.
- In general, the number of k-subsets starting with 0 is $\binom{11}{k-1}$.

In the proposed real-time system, combinatorial pitch-class sets are directly used in the chord recognition pipeline, which includes dyadic, triadic, and tetradic combinations of pitch class sets.

Thus, this section introduces the foundational combinatorial logic used in the system to recognize dyadic, triadic, and tetradic pitch-class structures in real time. By systematically constraining all harmonic evaluations to pitch-class subsets that begin with pitch class 0 (C), the system reduces the theoretical complexity of the chromatic pitch space and focuses on musically functional combinations. Each level—dyads ($k = 2$), triads ($k = 3$), and tetrads ($k = 4$)—is defined by a binomial formulation $\binom{11}{k-1}$, yielding a finite and analyzable vocabulary of 11, 55, and 165 structures, respectively. This range of combinations holds all possible chords which form the basis for interval classification and chord labeling, and paves the way for Roman numeral analysis in the main text.

The specifics of how each type is identified, labeled, and integrated into the system's harmonic grammar are explained in detail in the preceding sections. Before presenting these dyads, triads, and tetrads, I categorize these 230 all possible combinations in diatonic music; there are two categories: primary and secondary subsets. These categories allow us to observe chord distribution in the metric space of compositions, e.g., distribution of primary and secondary chords on downbeats and upbeats in measures.

References

Benetos, E., Dixon, S., Duan, Z., & Ewert, S. (2019). Automatic music transcription: An overview. *IEEE Signal Processing Magazine*, *36*(1), 20–30.

Devaney, J., Arthur, C., Condit-Schultz, N., & Nisula, K. (2015). Theme and variation encodings with roman numerals (tavern): A new data set for symbolic music analysis. *Proceedings of the 16th International Society for Music Information Retrieval Conference (ISMIR)*, 727–734.

Fricke, L., Gotham, M., Ostermann, F., & Vatolkin, I. (2024). Adaptation and optimization of augmentednet for roman numeral analysis applied to audio signals. *Artificial Intelligence in Music, Sound, Art and Design: 13th International Conference, EvoMUSART 2024*, *14633*, 146–157. https://doi.org/10.1007/978-3-031-56992-0_12

Jamshidi, F., Pike, G., Das, A., & Chapman, R. (2024). Machine learning techniques in automatic music transcription: A systematic survey [Licensed under CC BY 4.0]. *Proceedings of the 25th International Society for Music Information Retrieval Conference (ISMIR)*, 1–9.

Karystinaios, E., & Widmer, G. (2023). Roman numeral analysis with graph neural networks: Onset-wise predictions from note-wise features. *Proceedings of the 24th International Society for Music Information Retrieval Conference (ISMIR 2023)*. https://github.com/manoskary/chordgnn

López, N. N., & Fujinaga, I. (2020). Harmalysis: A language for the annotation of roman numerals in symbolic music representations. *Proceedings of the Music Encoding Conference (MEC)*, 83–85.

Marinov, G. (2020). Real-time error correction and performance aid for midi instruments [BSc Computer Science Dissertation].

Micchi, G., Gotham, M., & Giraud, M. (2020). Not all roads lead to rome: Pitch representation and model architecture for automatic harmonic analysis. *Transactions of the International Society for Music Information Retrieval*, *3*(1), 42–54. https://doi.org/10.5334/tismir.45

Privato, N., Rampado, O., & Novello, A. (2022). A creative tool for the musician combining lstm and markov chains in max/msp. *Artificial Intelligence in Music, Sound, Art and Design. 11th International Conference, EvoMUSART 2022*, *13271*, 228–242. https://doi.org/10.1007/978-3-031-05978-4_15

Shepardson, V., Armitage, J., & Magnusson, T. (2022). Notochord: A flexible probabilistic model for real-time midi performance. *Proceedings of the 3rd Conference on AI Music Creativity (AIMC)*.

Tymoczko, D., Gotham, M., Cuthbert, M. S., & Ariza, C. (2019). The romantext format: A flexible and standard method for representing roman numeral analyses. *Proceedings of the 20th International Society for Music Information Retrieval Conference (ISMIR)*, 123–129.