Supplementary Materials

S1. Stimuli

Stimuli are original musical recordings created by a real human musician in Logic Pro X using a 2021 16" MacBook Pro (Apple M1 Pro chip), an Apollo Twin X audio interface, and Yamaha HS8 monitors. Stimuli were recorded on electric guitar (PRS McCarty Hollowbody II, Schecter Solo-6), piano (Arturia KeyLab Essential Mk3 MIDI controller with Analog Lab V software instruments), and drums (Roland TD-17 electronic kit with Superior Drummer 3 plugin). Guitar recordings were processed with Neural DSP plugins (Tim Henson Archetype, Cory Wong Archetype).

Additional excerpts were reserved for few-shot prompting (2 for syncopation, 2 for transposition, and 4 for chord ID, one per chord class) and excluded from testing.

You can access the stimuli used in this experiment on The MUSE Benchmark Github page.

The stimuli used for the Transposition Detection task can be found here and all of them have the melody number, key, and tempo in the filename (e.g., M1_EbMaj_90.wav).

The stimuli used for the Syncopation Scoring task can be found here and all of them have Sync in the name, along with the syncopation level number (e.g., NoSync_A, Sync2_B).

The stimuli used for the Chord Quality Identification task can be found here. The chords are named by number, and you can find the mapping in Table S1 below.

Table S1: Mapping of chord roots and qualities to numerical identifiers (1.wav–48.wav).

Root	Diminished	Dominant	Major	Minor
Ab	1	2	3	4
A	5	6	7	8
Bb	9	10	11	12
B	13	14	15	16
C	17	18	19	20
Db	21	22	23	24
D	25	26	27	28
Eb	29	30	31	32
E	33	34	35	36
F	37	38	39	40
Gb	41	42	43	44
G	45	46	47	48

S2. System Instructions

1a) Syncopation — Standalone

"You are an expert music transcription AI participating in a multi-turn reasoning experiment.

You will be given one short audio excerpt of a drum set per trial. Your task is to focus only on the kick and snare drums. The hi-hat plays constant 8th notes, acting as a metronome. Count the total number of kicks and snare hits that fall on off-beats.

Valid multiple-choice responses are:

A. 0 (No Syncopation)
B. 2 (Low Syncopation)
C. 4 (Medium-Low Syncopation)
D. 6 (Medium-High Syncopation)
E. 8 (High Syncopation)

End with exactly one line:
Final Answer: X
"

1b) Syncopation — Chain-of-Thought (CoT)

"You are an expert music transcription AI participating in a multi-turn reasoning experiment.

You will be given one short audio excerpt of a drum set per trial. Your task is to focus only on the kick and snare drums. The hi-hat plays constant 8th notes, acting as a metronome. Count the total number of kicks and snare hits that fall on off-beats. On-beats are the main pulses (beats 1, 2, 3, and 4) and off-beats are the "ands" in between. Ignore the on-beats and ignore the hi-hat.

Valid multiple-choice responses are:

A. 0 (No Syncopation)
B. 2 (Low Syncopation)
C. 4 (Medium-Low Syncopation)
D. 6 (Medium-High Syncopation)
E. 8 (High Syncopation)

After any reasoning, end with exactly one line:
Final Answer: X
"

1c) Syncopation — LogicLM

"You are an expert music transcription AI participating in a multi-turn reasoning experiment.

Your task is to transcribe the onsets of ONLY the kick and snare drums into the format:
`rhythm(identifier, [list_of_onsets]).`

- The 'identifier' is the filename of the audio.
- The 'list_of_onsets' is a comma-separated list of integers from 1 to 32.

- The rhythm is on a 4-bar grid, quantized to 8th notes (numbered 1 to 32). All odd numbers are on-beats, and all even numbers are off-beats.
- The hi-hat plays constant 8th notes, acting as a metronome. On-beats are the main pulses (beats 1, 2, 3, and 4 of each bar) and off-beats are the 'ands' in between.

Grid: The excerpt is 4 bars quantized to 8th notes → 32 slots numbered 1–32.

Within each bar (8 slots): 1,3,5,7 = on-beats (beats 1–4). 2,4,6,8 = off-beats ("&"s).

Across bars: slot = 8×(bar-1) + local_slot.

Beat positions across 4 bars:
• Beat 1 → 1, 9, 17, 25
• Beat 2 → 3, 11, 19, 27
• Beat 3 → 5, 13, 21, 29
• Beat 4 → 7, 15, 23, 31

Off-beats ("&"s):
• &1 → 2, 10, 18, 26
• &2 → 4, 12, 20, 28
• &3 → 6, 14, 22, 30
• &4 → 8, 16, 24, 32

Output format:
`rhythm(identifier.wav, [n1, n2, ..., nK])` where each `n` is an integer in 1–32.

Example of format where the kicks are on beats 1 and 3 in each bar, and the snare hits are on beats 2 and 4 in each bar (all played on the on-beats):
`rhythm(example.wav, [1, 3, 5, 7, 9, 11, 13, 15, 17, 19, 21, 23, 25, 27, 29, 31])`

Output your answer of symbolic code as a single line of plain text without code fences or explanations. After your transcription, an external tool will score it, and you will answer a question based on that score."

2a) Transposition Detection — Standalone

"You are an expert melody transcription AI participating in a multi-turn reasoning experiment.

You will be given two short monophonic audio melodies per trial.
Your job is to decide whether they represent the SAME melody up to TRANSPOSITION (i.e., identical shape/intervals but possibly in different keys).

Valid responses are exactly one of:
"Yes, these are the same melody."
"No, these are not the same melody."

Respond with exactly one of the two phrases and nothing else."

2b) Transposition Detection — Chain-of-Thought (CoT)

"You are an expert melody transcription AI participating in a multi-turn reasoning experiment.

You will be given two short monophonic audio melodies per trial. Your job is to decide whether they represent the SAME melody up to TRANSPOSITION (i.e., identical shape/intervals but possibly in different keys).

Definitions and constraints:
- Transposition equivalence: the two melodies have the same number of notes and the same sequence of pitch INTERVALS between successive notes (including 0 for repeated notes).
- Ignore absolute key/register, starting pitch, and tempo. Small timing variations are acceptable. If the rhythmic patterns are drastically different (e.g., note insertions/deletions or re-ordered phrases), they are most likely NOT the same melody.
- Treat repeated notes as separate events and include 0 in the interval sequence when a note repeats.
- If there are leading/trailing silences, ignore them.

Valid responses (exactly one of these strings):
"Yes, these are the same melody."
"No, these are not the same melody."

After any reasoning, end with exactly one line:
Final Answer: Yes, these are the same melody.
OR
Final Answer: No, these are not the same melody."

2c) Transposition Detection — LogicLM

"You are an expert melody transcription AI participating in a multi-turn reasoning experiment.

You will be given two short monophonic audio melodies per trial. Your first task is to transcribe EACH melody into the symbolic format below, using MIDI integers for pitches. If the rhythmic sequences seem drastically different, they are most likely not the same melody.

Output format (schema):
`melody(identifier, [p1, p2, ..., pK])`
- Use the exact identifiers I provide for each trial (one per audio).
- p1..pK are integers representing MIDI pitches (e.g., C4 = 60).
- Transcribe the pitch sequence only.
- Output exactly two lines of plain text: one 'melody(...)' per line, in the same

order as the audios (Audio 1 line first, then Audio 2 line).
- Do not include code fences or any extra commentary.

Example (schema only; not tied to any audio):
```
melody(Audio1, [60, 62, 64])
melody(Audio2, [65, 67, 69])
```

After your transcription, a deterministic tool will analyze the two lines to decide if the melodies are transpositions (same contour, different key). You will then answer a Yes/No question based on that decision."

3a) Chord Quality Matching — Standalone

"You are an expert chord-transcription AI participating in a multi-turn reasoning experiment.

You will be given one short audio clip per trial. Each clip first plays a chord (block), then the individual notes (arpeggiation).
All chords are in ROOT POSITION.

Your task is to identify the chord QUALITY.

Valid options:
A. Major
B. Minor
C. Dominant
D. Diminished

Final Answer: X
"

3b) Chord Quality Matching — Chain-of-Thought (CoT)

"You are an expert chord-transcription assistant in a multi-turn reasoning experiment.

You will be given one short audio clip per trial containing a single chord (first block, then arpeggiated notes). All chords are in ROOT POSITION; the lowest pitch is the ROOT (treat as 0 semitones). Your task: identify the chord QUALITY by inferring pitch-class intervals above the root and ignoring octave doublings.

Valid options:
A. Major → {0,4,7}
B. Minor → {0,3,7}
C. Dominant → {0,4,7,10}
D. Diminished → {0,3,6}

Think through the identification. Once you've finished reasoning, the final line of your output should be exactly:

Final Answer: X
"

3c) Chord Quality Matching — LogicLM

"You are an expert chord-transcription assistant in a multi-turn reasoning experiment.

You will be given one short audio clip per trial containing a single chord. First the chord sounds as a block, then the notes are arpeggiated.

Your task is to transcribe the chord tones into a strict symbolic format. Use MIDI integers (0–127). Include octave doublings if you hear them. Do not add commentary.

Output format (schema):
`chord(identifier, [p1, p2, ..., pK])`

Rules:
- Use the exact identifier I provide for the trial.
- Record only the pitches you hear as MIDI integers.
- It is acceptable if the list is not sorted; a deterministic solver will normalize.
- Output EXACTLY ONE LINE of plain text with NO code fences or extra text.

Example (schema only; not tied to any audio):
`chord(Audio_X, [56, 60, 64, 67, 72, 76])`

After your line is produced, a deterministic tool will classify the chord quality (Major / Minor / Dominant / Diminished) from your symbolic line. You will then answer a multiple-choice question with: Final Answer: X"

S3. Task Schemas and Deterministic Solvers

Each task defines a single-line schema the model must emit verbatim. A hand-written, deterministic solver (`solver.py`) parses that line, makes the decision, and returns the minimal information needed for a constrained final answer.

Syncopation Scoring

Input: 4-bar drum loop with constant 8th-note hi-hat; we only score kick+snare.
Grid: 32 slots (8 per bar). Odd slots are on-beats; even slots are off-beats.

Schema (one line):

```
rhythm(<id>, [n1, n2, ..., nK])
```

Where each `n` is an integer in [1..32] (kick or snare onset).
Solver: counts off-beat onsets and maps to five categories: 0,2,4,6,8 off-beats → A–E respectively. Final answer is a single MC letter A–E.

Transposition Detection

Input: two short monophonic excerpts (guitar or piano) that are either the same melody in different keys or different melodies.

Schema (two lines, order-preserving):

```
melody(<id1>, [p1, p2, ..., pK])
melody(<id2>, [p1, p2, ..., pK])
```

Where `p*` are MIDI integers (0–127).
Solver: checks equal length and equality of adjacent-interval sequences (transposition invariance). Returns ARE / ARE NOT (transpositions). Final answer is forced to one of:

"Yes, these are the same melody."
"No, these are not the same melody."

Chord Quality Identifier

Input: a single triad or seventh chord (piano), presented as a block then arpeggiated.

Schema (one line):

```
chord(<id>, [p1, p2, ..., pK])
```

MIDI integers (0–127); octave doublings allowed.

Solver: normalizes to pitch classes, factors out the putative root, and matches the interval set to:

- Major (0, 4, 7) → A
- Minor (0, 3, 7) → B
- Dominant 7 (0, 4, 7, 10) → C
- Diminished (0, 3, 6) → D

Final answer is a single MC letter A–D.

Self-refinement (SR)

For LogicLM, we validate the line(s) with strict regex/AST checks and label errors as parse, structural, or domain. If invalid, we run up to 2 SR rounds in a separate deterministic chat (temperature=0, top_p=1, top_k=1, 256 tokens) with a fix-only prompt that:

- Echoes the prior output,
- States the specific error type/message,
- Re-states the required line(s) and constraints,
- Forbids commentary and code fences.

If the solver returns undecidable/None (e.g., empty list), we allow one extra SR pass with a synthesized parse error. This SR design follows the LogicLM self-refinement idea of using solver feedback to repair the symbolic form.

S3.1. solver.py

```
# solver.py
import re
from typing import List, Optional, Tuple, Dict

class SyncopationSolver:
    """
    A deterministic logic solver that calculates a syncopation score
    based on a simplified on-beat/off-beat rule for a 4-bar (1-32) 8th-note grid.
    """
    def __init__(self):
        self.on_beats = set()
        self.off_beats = set()

        for bar_offset in [0, 8, 16, 24]:
            self.on_beats.update([
                1 + bar_offset, 3 + bar_offset, 5 + bar_offset, 7 + bar_offset
            ])
            self.off_beats.update([
                2 + bar_offset, 4 + bar_offset, 6 + bar_offset, 8 + bar_offset
            ])

    def parse_llm_output(self, llm_text: str) -> Optional[List[int]]:
        """
        Parses the LLM's symbolic output to extract a list of onsets.
        Returns the list of integers if successful, or None if parsing fails.
        """
        match = re.search(r'rhythm\s*\(\s*[^,]+\s*,\s*\[([\d,\s]*)\]\s*\)',
            llm_text)
        if not match:
            return None
        numbers_str = match.group(1)
        if not numbers_str.strip(): # Check if the string is empty or just
            whitespace
            return []
        try:
            # Handle potential trailing commas by filtering out empty strings
                after split
            return [int(num.strip()) for num in numbers_str.split(',') if
                num.strip()]
        except ValueError:
            return None

    def score_onset(self, onset: int) -> int:
        if onset in self.off_beats:
            return 1
        return 0

    def calculate_total_score(self, onset_list: list[int]) -> int:
        if not onset_list:
            return 0
        total_score = sum(self.score_onset(onset) for onset in onset_list)
        return total_score

class TranspositionSolver:
    """
    A deterministic solver for melody transposition detection.
    Two melodies are considered transpositions if:
      - They have the same number of notes, and
      - Their interval sequences (adjacent pitch differences in semitones) are
          identical.
    Rhythm is ignored. Pitches must be integers (MIDI numbers).
    """

    MELODY_PATTERN = re.compile(
        r"melody\s*\(\s*([A-Za-z0-9_.\-]+)\s*,\s*\[\s*([^\]]*?)\s*\]\s*\)",
        flags=re.IGNORECASE
    )

    def _extract_pitches(self, pitches_str: str) -> Optional[List[int]]:
        """
        Extracts integer pitches from an arbitrary list content that may include
        parentheses or spaces, e.g. '[(60), (62), (64)]' or '60, 62,64'.
        """
        nums = re.findall(r"-?\d+", pitches_str)
        if not nums:
            return []
        try:
            return [int(n) for n in nums]
        except ValueError:
            return None

    def parse_llm_output(self, llm_text: str) -> Optional[List[Dict[str,
        List[int]]]]:
        """
        Parses any 'melody(ID, [ ... ])' lines found in the LLM's output, in order.
        Returns a list of dicts: [{'id': <ID>, 'pitches': [..]}, ...]
        or None if nothing parseable is found.
        """
        if not llm_text:
            return None

        text = llm_text.replace("```", "").replace("`", "").strip()

        melodies = []
        for m in self.MELODY_PATTERN.finditer(text):
            ident = m.group(1)
            plist_str = m.group(2)
            pitches = self._extract_pitches(plist_str)
            if pitches is None:
                return None
```

```
            melodies.append({"id": ident, "pitches": pitches})

        return melodies or None

    def _intervals(self, pitches: List[int]) -> List[int]:
        return [pitches[i+1] - pitches[i] for i in range(len(pitches) - 1)]

    def are_transpositions(self, p1: List[int], p2: List[int]) -> Optional[bool]:
        """
        Returns True/False if a decision is possible, or None if inputs are
            degenerate.
        Policy:
          - Require same length (>0). If lengths differ, return False.
          - If length == 1 on both, return True (single note can be transposed
              anywhere).
          - Otherwise compare interval sequences.
        """
        if p1 is None or p2 is None:
            return None
        if len(p1) == 0 and len(p2) == 0:
            return None
        if len(p1) != len(p2):
            return False
        if len(p1) == 1: # single-note melodies
            return True

        return self._intervals(p1) == self._intervals(p2)

    def decide_same_melody(self, llm_text: str) -> Optional[bool]:
        """
        Convenience: parse two melodies from LLM output and decide True/False.
        Returns None if fewer than 2 melodies parsed or if undecidable.
        """
        parsed = self.parse_llm_output(llm_text)
        if not parsed or len(parsed) < 2:
            return None
        p1 = parsed[0]["pitches"]
        p2 = parsed[1]["pitches"]
        return self.are_transpositions(p1, p2)


# ---------- Chord Quality (deterministic) ----------

class ChordQualitySolver:
    """
    Deterministic chord-quality classifier for LogicLM.
    Expects ONE schema line produced by the LLM:
        chord(identifier, [p1, p2, ..., pK])

    Behavior:
    - Parses the line and extracts MIDI integers (duplicates allowed).
    - Sorts pitches, treats the lowest as the root, and computes (p - root) % 12.
    - Deduplicates + sorts the pitch-class intervals and matches one of the
      four target fingerprints:
        (0,4,7)     -> ("Major", "A")
        (0,3,7)     -> ("Minor", "B")
        (0,4,7,10)  -> ("Dominant", "C")
        (0,3,6)     -> ("Diminished", "D")

    Returns:
        (identifier, quality_str, letter) or None if undecidable.
    """
    CHORD_PATTERN = re.compile(
        r"chord\s*\(\s*([A-Za-z0-9_.\-]+)\s*,\s*\[\s*([^\]]*?)\s*\]\s*\)",
        flags=re.IGNORECASE
    )

    QUALITY_BY_PCS: Dict[Tuple[int, ...], Tuple[str, str]] = {
        (0, 4, 7):     ("Major", "A"),
        (0, 3, 7):     ("Minor", "B"),
        (0, 4, 7, 10): ("Dominant", "C"),
        (0, 3, 6):     ("Diminished", "D"),
    }

    def _extract_pitches(self, pitches_str: str) -> Optional[List[int]]:
        """
        Robust integer pull; accepts '60,64,67', '[(60), 64, 67]', etc.
        Returns list[int] or None if malformed.
        """
        nums = re.findall(r"-?\d+", pitches_str or "")
        try:
            return [int(n) for n in nums]
        except Exception:
            return None

    def parse_llm_output(self, llm_text: str) -> Optional[Dict[str, List[int]]]:
        """
        Parse the first chord(...) line found. Returns {'id': <ID>, 'pitches':
            [...]}
        or None if not found / ill-formed.
        """
        if not llm_text:
            return None
        text = llm_text.replace("```", "").strip()
        m = self.CHORD_PATTERN.search(text)
        if not m:
            return None
        ident = m.group(1)
        pitches = self._extract_pitches(m.group(2))
        if pitches is None:
            return None
        return {"id": ident, "pitches": pitches}
```

```
def _normalize_to_pcs(self, pitches: List[int]) -> Optional[Tuple[int, ...]]:
    """
    Sort, take lowest as root, compute pitch-class intervals modulo 12,
    then deduplicate and sort.
    """
    if not pitches:
        return None
    root = min(pitches)
    pcs = tuple(sorted({(p - root) % 12 for p in pitches}))
    return pcs

def classify_quality(self, pitches: List[int]) -> Optional[Tuple[str, str]]:
    """
    Map normalized pitch-class interval set to (quality, letter).
    """
    pcs = self._normalize_to_pcs(pitches)
    if pcs is None:
        return None
    return self.QUALITY_BY_PCS.get(pcs)

def decide_quality(self, llm_text: str) -> Optional[Tuple[str, str, str]]:
    """
    End-to-end convenience used by the runner:
      - parse -> classify
    Returns (identifier, quality_str, letter) or None if undecidable.
    """
    parsed = self.parse_llm_output(llm_text)
    if not parsed:
        return None
    ident = parsed["id"]
    result = self.classify_quality(parsed["pitches"])
    if result is None:
        return None
    quality, letter = result
    return ident, quality, letter
```