DECOMPOSER: LEARNING TO DECOMPILE SYMBOLIC MUSIC TO PROGRAMS

Anonymous authors
Paper under double-blind review

ABSTRACT

Musical performance involves executing a set of high-level musical instructions, yet recovering those instructions from the performance is a challenging inverse problem. We present DECOMPOSER, a post-training framework for *symbolic music decompilation*: the task of recovering executable, editable music programs from symbolic music. We instantiate the task as MIDI-to-Strudel decompilation, where the model takes symbolic MIDI as input and produces a program in Strudel, a music programming language, that reconstructs the input when executed. The task poses two challenges: Strudel is a low-resource language with little naturally paired MIDI–code data, and optimizing faithful reconstruction of MIDI alone can collapse to unreadable note-by-note transliteration. We address these challenges in two stages. First, we construct STRUDEL-SYNTH, a synthetic corpus of paired Strudel programs and rendered MIDI, and use it for supervised fine-tuning. Second, we refine the model with reinforcement learning on unpaired MIDI, optimizing rewards for both MIDI reconstruction faithfulness and code readability. Our evaluation across synthetic and real-world MIDI benchmarks shows that DECOMPOSER achieves substantially higher MIDI reconstruction faithfulness than closed-source LLMs while producing more readable and diverse code than the heuristic converter.

1 INTRODUCTION

Decompilation, the task of generating an editable, re-executable program from a rendered artifact, has emerged as a central problem across diverse modalities: binaries to source code (Tan et al., 2024; Zou et al., 2025), raster images to SVG programs (Rodriguez et al., 2025; 2024), UI screenshots to frontend code (Jain et al., 2019; Wu et al., 2025), and 3D shapes to CAD scripts (Kolodiazhnyi et al., 2026; Chen et al., 2025). The motivation is shared across these domains: rendered artifacts are often the only representation available in practice, yet downstream use such as analysis, modification, integration with existing tooling benefits from a programmatic form that human users can read, modify, and extend (Ritchie et al., 2023; Rodriguez et al., 2024; Zou et al., 2025). A direct consequence is that functional equivalence alone is insufficient: a recovered program that reproduces the artifact but is unreadable defeats the purpose of decompilation (Buse & Weimer, 2010; Scalabrino et al., 2018). Effective decompilation thus requires jointly optimizing *faithfulness* (the program reproduces the input artifact) and *readability* (the program is easy to read and edit).

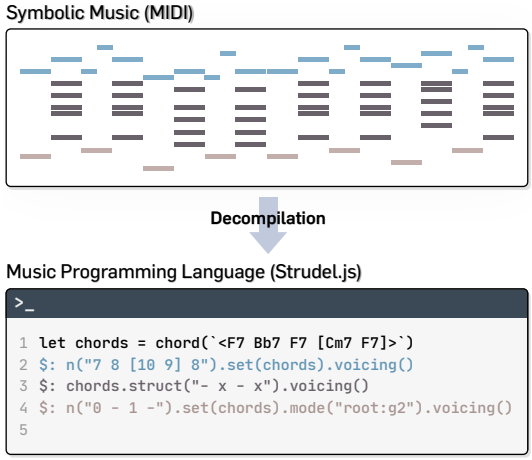


Figure 1: **Symbolic Music Decompilation.** In this task, the goal is to generate a *faithful* and *readable* music program from symbolic music input. We instantiate this task as MIDI-to-Strudel decompilation; Strudel is a domain-specific music programming language for algorithmic composition and live coding. The recovered program should reproduce the input when executed while exposing readable and editable musical structure.

Project page: decomposer-page.netlify.app

Reviewers: please read the Reviewer Guidelines (iclr.cc/Conferences/2027/ReviewerGuidelines) and the AI Policy for Reviewers (iclr.cc/Conferences/2027/AIPolicyForReviewers). If you used AI to expand, edit, or polish your review, please provide the input text to the LLM. Better still, consider skipping the LLM and submitting your original text: we, and the authors, are much more interested in your unedited thoughts than in what an LLM has to say. AI-assisted or not, you are putting your name and reputation behind your review: LLM-generated falsehoods, hallucinations or misrepresentations are subject to disciplinary action, which may include desk-rejecting all papers you have authored.

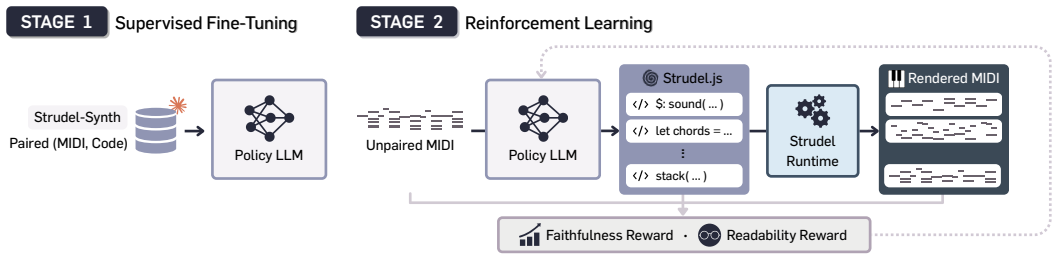


Figure 2: **DECOMPOSER Overview.** We present a framework for post-training LLMs to decompile symbolic music (MIDI) into a music programming language (Strudel). DECOMPOSER consists of two stages: (i) supervised fine-tuning equips the model with the ability to write valid Strudel code; (ii) reinforcement learning optimizes the decompilation objective directly, rewarding sampled programs for both *faithfulness* (rendered MIDI matches the input) and *readability* (concise, editable code). The paired examples used for SFT are provided by STRUDEL-SYNTH, our synthetic corpus of executable Strudel programs and rendered MIDI. Since the reward depends only on the input MIDI and executed program, RL needs no reference code and trains on any unpaired MIDI at scale.

We extend this paradigm to the music modality and introduce *symbolic music decompilation*: generating code in a music programming language (Wang et al., 2015; Puckette et al., 1996; Roos & McLean, 2023; Aaron; McLean & Wiggins, 2010; McCartney, 1996) corresponding to a symbolic music input. We instantiate the paradigm here through a MIDI-to-Strudel (Roos & McLean, 2023) task, where Strudel is a domain-specific language (DSL) for algorithmic composition (Dean, 2018) and live coding (Collins et al., 2003; Wang & Cook, 2004) with JavaScript syntax. Unlike MIDI, which represents music as a low-level sequence of note events, Strudel represents music through high-level pattern expressions and programmatic operators that can encode repetition, layering, and temporal structure (Figure 1; see Appendix A.1 for more examples). Recovering Strudel code from MIDI therefore does more than reproduce the input notes: it exposes latent musical structure that MIDI flattens into individual events, produces code that is readable and directly editable, and places music generation and editing in a code modality well suited to modern language model tooling.

MIDI-to-Strudel decompilation is challenging for both task-level and data-level reasons. At the task level, faithfulness alone does not define a useful decompilation: a heuristic converter can achieve near-perfect rendered-MIDI faithfulness (Jong, 2025), yet it does so by emitting verbose note-level code that ignores the abstractions that make Strudel useful to humans. In contrast, frontier LLMs (OpenAI, 2026; Anthropic, 2026; Google DeepMind, 2026) often generate readable Strudel programs, but their rendered MIDI is not faithful to the input. At the data level, this mapping is hard to learn because naturally paired (MIDI, Strudel) examples are not available. Moreover, Strudel itself is low-resource: our initial crawl found only 688 non-trivial human-authored programs, and open-weight LLMs rarely produce executable Strudel without adaptation, making direct execution-based optimization difficult.

Our approach. We present DECOMPOSER, a framework for post-training LLMs to decompile MIDI into Strudel code (Figure 2). The framework consists of a two-stage training pipeline. First, supervised fine-tuning (SFT) teaches a model to produce valid and idiomatic Strudel code, providing a warm start for on-policy learning. Second, reinforcement learning (RL) optimizes the decompilation objective directly. For each MIDI input, the model samples candidate programs, executes them with the Strudel runtime, and receives a composite reward that measures both rendered-MIDI faithfulness and code-level readability. Because this reward depends only on the input MIDI and the generated program, the RL stage can train on any unpaired MIDI at scale without requiring reference Strudel code. To supply the paired examples needed for the SFT warm start, we additionally construct and release STRUDEL-SYNTH, a synthetic (MIDI, Strudel) corpus obtained by generating Strudel programs with a frontier LLM and rendering each program to MIDI.

Our experiments show that DECOMPOSER substantially improves MIDI-to-Strudel decompilation, producing programs that better reconstruct the input MIDI while preserving readable code structure. On both STRUDEL-SYNTH and real-world LMD (Raffel, 2016), DECOMPOSER produces markedly more faithful renderings than frontier LLMs (Anthropic, 2026; Google DeepMind, 2026; OpenAI, 2026), which often generate plausible-looking but poorly grounded Strudel code. At the same time, unlike a heuristic converter (Jong, 2025) that achieves high faithfulness by emitting verbose note-level transcriptions, DECOMPOSER preserves readable program structure comparable to frontier LLMs.

We highlight the main contributions of this paper below:

- We introduce *symbolic music decompilation*, the task of recovering executable, editable music-programming-language code from symbolic music input.
- We propose DECOMPOSER, a two-stage post-training framework that combines supervised fine-tuning with execution-based reinforcement learning for symbolic music decompilation.
- We release STRUDEL-SYNTH, a synthetic corpus of 21,174 (Strudel, MIDI) pairs constructed by frontier LLM generation of Strudel code, followed by Strudel-to-MIDI code execution.
- We find that an 8B open-weight model trained with our pipeline substantially outperforms closed-weight frontier LLMs on decompilation faithfulness, while retaining comparable readability.

2 DECOMPOSER

We approach symbolic music decompilation as program synthesis with execution feedback, following recent decompilation work that optimizes programs against a verifiable renderer (Kolodiazhnyi et al., 2026; Rodriguez et al., 2025). Given an input MIDI segment x, the goal is to model $p(y \mid x)$ where y is a corresponding Strudel program. Specifically, for the Strudel runtime $\mathcal{C}$ and a program $\hat{y} \sim p(y \mid x)$, our goal is to improve faithfulness of the MIDI output from Strudel execution ($\mathcal{C}(\hat{y}) \approx x$), while ensuring $\hat{y}$ is human-readable. We parameterize a conditional language model policy

$$\pi_\theta(y \mid x) = \texttt{LLM}_\theta(\texttt{MidiToText}(x)), \tag{1}$$

where `MidiToText` is a helper function that converts the MIDI input into a text representation compatible with a general LLM (see Appendix B). Because many Strudel programs can render to the same or similar MIDI output (see Appendix A.1 for examples), decompilation quality is defined by both rendered-MIDI faithfulness and code-level readability.

The remainder of this section first introduces our two-stage training pipeline combining SFT and RL (Section 2.1) and details the composite reward used in the RL stage (Section 2.2). Finally, we describe STRUDEL-SYNTH, the synthetic paired (MIDI, Strudel) corpus we construct (Section 2.3).

2.1 TWO-STAGE TRAINING

Directly applying RL to this task from a general-purpose open-weight LLM is difficult: unlike common LLM post-training domains such as math or conventional code generation, MIDI-to-Strudel decompilation requires grounding a dense symbolic music input while generating a low-resource DSL that appears rarely in pretraining data. As a result, early rollouts from the unadapted model seldom produce valid Strudel programs, making reward signals sparse and unstable. We therefore use a two-stage pipeline: supervised fine-tuning first gives the model a prior over executable Strudel code, and reinforcement learning then optimizes the decompilation objective.

Stage 1: Supervised fine-tuning (SFT). We first adapt the base LLM π_θ to the MIDI-to-Strudel decompilation task using the paired dataset $\mathcal{D}_\text{S}$. Each example $(x^*, y^*) \in \mathcal{D}_\text{S}$ consists of a Strudel program y^* and its executed MIDI representation $x^* = \mathcal{C}(y^*)$, where $\mathcal{C}$ denotes the Strudel runtime.[1] Given the MIDI input x^*, the model is trained to generate the corresponding program y^* with the standard next-token prediction objective:

$$\mathcal{L}_\text{SFT}(\theta) = -\,\mathbb{E}_{y^* \sim \mathcal{D}_\text{S}} \left[\sum_{t=1}^{|y^*|} \log \pi_\theta(y_t^* \mid y_{<t}^*,\, x^*) \right]. \tag{2}$$

Without this stage, RL collapses onto a degenerate solution where the model learns to emit short, literal note lists that compile but capture almost none of the input music, and reward plateaus early at a low value. SFT instead provides a warm start that places the policy in a region of program space where rollouts express nontrivial musical structure, so that the gradient signal from the faithfulness and readability terms is meaningful (Section 3.3).

[1] We use y^* to indicate that, during SFT, the target program corresponding to $x^* = \mathcal{C}(y^*)$ is known, in contrast to a generic MIDI input x at test time or during RL where the target program is unknown.

Stage 2: Reinforcement learning. After SFT, we further train the model to optimize the decompilation objective: a generated program should faithfully reproduce the input MIDI when executed while remaining readable. We use the SFT model π_θ as the initial policy and optimize it by maximizing the expected reward

$$\mathcal{J}(\theta) = \mathbb{E}_{x\sim\mathcal{D}_{\mathrm{M}},\, \hat{y}\sim\pi_\theta(\cdot|x)}\left[R(\hat{y}, x)\right], \tag{3}$$

where $\mathcal{D}_{\mathrm{M}}$ is a dataset of unpaired MIDI and R is a composite reward combining decompilation faithfulness and code readability (see Section 2.2). Because R depends only on the input MIDI and the executed program, this stage requires no reference Strudel code and can train on any unpaired MIDI at scale.

To optimize Equation 3, we adopt Group reward-Decoupled normalization Policy Optimization (GDPO; Liu et al. 2026), a GRPO-style policy optimization method (Guo et al., 2025) for multi-reward objectives. For each input MIDI x, the policy samples a group of $G > 0$ candidate programs, executes them with the Strudel runtime, and evaluates each program with K reward components $r_1, \ldots, r_K$. In our setting, $K = 2$, corresponding to faithfulness and readability. A standard GRPO-style update first scalarizes these components as $r^{(g)} = \sum_k w_k r_k^{(g)}$ where w_k are reward weights, and then normalizes the scalar rewards across the group; GDPO instead constructs a group-relative advantage for each reward component and combines the normalized advantages only afterward.

Concretely, let $r_k^{(g)}$ denote the k-th reward component of the g-th sampled program, where $k \in \{1, \ldots, K\}$ and $g \in \{1, \ldots, G\}$. For each component, GDPO computes

$$A_k^{(g)} = \frac{r_k^{(g)} - \mu_k}{\sigma_k + \epsilon}, \quad \mu_k = \frac{1}{G}\sum_{g'=1}^{G} r_k^{(g')}, \quad \sigma_k = \mathrm{std}\Big(\{r_k^{(g')}\}_{g'=1}^{G}\Big). \tag{4}$$

The final advantage is $A^{(g)} = \sum_{k=1}^{K} w_k A_k^{(g)}$, which is used in the standard clipped group-relative policy objective to update π_θ. Full optimization details are provided in Appendix C.

2.2 Reward

We design a reward function $R(\hat{y}, x)$ that evaluates a generated Strudel program $\hat{y}$ for an input MIDI segment x along two axes: *faithfulness*, whether the rendered MIDI $\hat{x} = \mathcal{C}(\hat{y})$ matches the input x, and *readability*, whether the generated program $\hat{y}$ is easy to understand and edit.

Faithfulness. The faithfulness reward $R_{\text{faith}}(\hat{x}, x) \in [0, 1]$ measures how accurately the rendered MIDI $\hat{x} = \mathcal{C}(\hat{y})$ reproduces the input MIDI x. We adopt the instrument-aware onset metric from the multi-track automatic music transcription literature (Gardner et al., 2022; Lu et al., 2023; Chang et al., 2024; Mcleod & Steedman, 2018): R_{faith} is the multi-instrument onset F1 between $\hat{x}$ and x, where a predicted note is counted as correct only if it (i) matches the pitch of a reference note, (ii) has an onset within ± 50 ms of that reference onset, and (iii) is assigned the same General MIDI program number as the reference note. The optimal pairing between reference and predicted notes is obtained by bipartite matching, and R_{faith} is the resulting F1, computed with the standard `mir_eval` implementation (Raffel et al., 2014).

Readability. The readability reward $R_{\text{read}}(\hat{y}) \in [0, 1]$ measures whether the generated program $\hat{y}$ is easy to read and edit. This term is essential because optimizing R_{faith} alone admits a degenerate solution: the model can learn a literal, note-by-note transliteration that renders correctly but captures none of the repeated patterns or musical structure that make Strudel concise and editable.

Following the rubric-as-rewards paradigm (Viswanathan et al., 2025; Gunjal et al., 2026), we therefore score readability with a fixed set of $J = 12$ rubric items, each a binary yes/no check evaluated by an LLM judge. The items cover (i) general code readability such as formatting and structure, adapted from established code-readability metrics (Buse & Weimer, 2010; Scalabrino et al., 2018); and (ii) Strudel-specific criteria targeting concision and musical abstraction, such as using pattern operators and chord symbols rather than literal pitches (full rubric in Appendix D). To turn these checks into a scalar reward, we let $c_j : \hat{y} \mapsto \{0, 1\}$ indicate whether a sampled program $\hat{y}$ satisfies rubric item j

and aggregate with uniform weights:

$$R_{\text{read}}(\hat{y}) = \frac{1}{J}\sum_{j=1}^{J} c_j(\hat{y}). \tag{5}$$

Final reward. Both rewards presume an executable program: a non-compiling output yields no rendered MIDI and exposes no executable structure for the readability judge to credit. We therefore gate each reward on successful compilation, assigning a penalty $-\rho$ ($\rho > 0$) when the Strudel runtime $\mathcal{C}$ fails to render $\hat{y}$:

$$\hat{R}_m = \begin{cases} R_m, & \text{if } \mathcal{C}(\hat{y}) \text{ succeeds,} \\ -\rho, & \text{otherwise,} \end{cases} \qquad m \in \{\text{faith}, \text{read}\}. \tag{6}$$

The gated rewards ($\hat{R}_{\text{faith}}$, $\hat{R}_{\text{read}}$) form the two components (r_1, r_2) that GDPO normalizes separately before combining (Section 2.1).

2.3 Dataset: Strudel-Synth

Our SFT warm start requires a paired (MIDI, code) corpus, which the public web does not supply at scale: crawling public Strudel programs yields only 688 non-trivial human-authored examples. Following prior work on synthetic data generation (Doh et al., 2025; 2026; Mukherjee et al., 2023), we construct Strudel-Synth by distilling from a frontier LLM (Claude-Opus-4.6) to write Strudel programs and executing each through the Strudel runtime $\mathcal{C}$ to obtain its rendered MIDI.

Data generation pipeline. Our pipeline produces (MIDI, Strudel code) pairs in three stages (Figure 3). We begin by generating Strudel programs from an LLM, conditioned on a musical seed s_{music} and a code style seed s_{code} under a fixed task prompt. In the second stage, we clean the generated programs, as LLM-written programs frequently include dead voices: layers that compile but emit no MIDI events, typically from hallucinated instrument names or malformed operator chains. We first apply deterministic regex rewrites that correct these recurring mistakes, then use the Acorn JavaScript parser to split each program into its independent voices, compile each voice in isolation, and prune the silent ones together with the variable declarations they alone reference. A program is retained only if at least one voice survives. Lastly, we render each cleaned program to MIDI with the runtime $\mathcal{C}$. Full details of the pipeline are given in Appendix E.1.

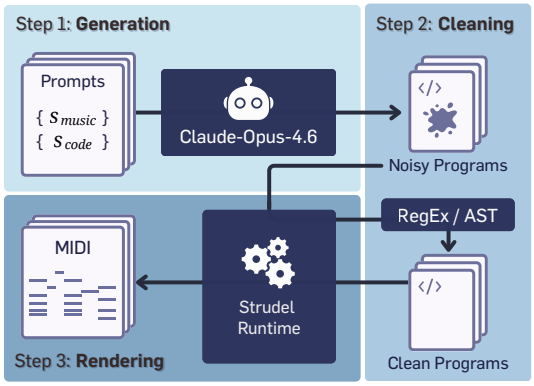


Figure 3: **Strudel-Synth generation pipeline.** Each (MIDI, Strudel code) pair is produced in three stages: (i) prompting an LLM with a musical and a code style seed, (ii) cleaning the generated program, and (iii) rendering it to MIDI through the Strudel runtime.

Conditioning inputs and tools. To ensure diversity in Strudel-Synth, we independently vary two conditioning seeds during generation: a musical seed s_{music} and a code-style seed s_{code}, which respectively specify the musical content of the piece and how it should be written. Specifically, s_{music} samples musical attributes—genre, mood, key, tempo, meter, chords, and instrumentation—from existing music metadata (McDonald, 2013; Melechovsky et al., 2024; DB, 2026), and we vary how many of these attributes are specified during generation. The code-style seed s_{code} selects among different program idioms that realize the same musical target, ranging from mini-notation (`$`) based patterns to multi-section programs using `arrange` functions[2]. We render each generated program to MIDI with a headless Strudel runtime $\mathcal{C}$ and use the same deterministic renderer to score policy rollouts during RL (Section 2). Further details on the conditioning inputs and runtime are provided in Appendix E.2 and A.2, respectively.

[2] https://strudel.cc/learn/factories

Statistics. Table 1 presents the statistics of STRUDEL-SYNTH. It comprises 21,174 (MIDI, Strudel) pairs generated from Claude-Opus-4.6 with diverse conditioning seeds, roughly 30× the 688 non-trivial programs found from public Strudel crawl. The generated programs cover 122 of the 128 General MIDI instruments and render to short musical fragments, averaging 24 seconds in duration (from 2s to 90s).

Table 1: **Statistics of STRUDEL-SYNTH dataset.**

Metric	Train Split	Test Split
# of (MIDI, Code) Pairs	20,152	1,022
# of Instruments	122	98
Avg. # Insts / Song	3.73±1.22	3.68±1.26
Avg. Lines of Code	52.20±20.88	52.46±21.70
Avg. Duration (s)	24.17±19.03	24.10±18.70

3 EXPERIMENTS

We evaluate the performance of DECOMPOSER and the effect of its proposed components on both synthetic and real-world MIDI data. Our experiments address the following questions:

- Can DECOMPOSER recover faithful, readable Strudel programs from MIDI? (Section 3.2)
- How do DECOMPOSER's components affect decompilation quality? (Section 3.3)
- Can DECOMPOSER generalize across diverse MIDI datasets? (Section 3.4)

3.1 SETUP

Metrics. We evaluate DECOMPOSER along three axes: faithfulness, readability, and diversity. For faithfulness, we report compile rate (Comp.) together with transcription metrics adapted from automatic music transcription (Gardner et al., 2022; Maman & Bermano, 2022; Chang et al., 2024): Onset F1, Frame F1, and Multi-Instrument Onset F1 (Inst F1). Readability is measured by the proposed rubric score (Rubric) and average rank from a list-wise LLM reranker (Rank; Tang et al., 2024). Diversity is measured by the average pairwise CodeBLEU self-similarity (SelfCB; Ren et al., 2020) among compiled outputs from the same method. Further details are provided in Appendix F.1.

Datasets. Our main experiments use two corpora: STRUDEL-SYNTH (synthetic MIDI–Strudel pairs) and LMD (real-world MIDI; Raffel, 2016). For LMD, we use short fragments (<30 s; ∼9K train / ∼1K test). To test generalization beyond these primary datasets, we further evaluate on four held-out datasets spanning different MIDI distributions: longer LMD (30–60 s), GigaMIDI (Lee et al., 2025), NES-MDB (Donahue et al., 2018), and Nottingham (Foxley, 2003). Except for longer LMD, all held-out corpora use short fragments (<30 s); see Appendix F.2 for details.

Baselines. Since MIDI-to-Strudel decompilation has no direct prior baseline, we compare against two classes of baselines. First, we prompt frontier LLMs to generate Strudel code from the MIDI input: Claude-Opus-4.6 (Anthropic, 2026), Gemini-3.5-Flash (Google DeepMind, 2026), and GPT-5.5 (OpenAI, 2026). Second, we include a heuristic MIDI-to-Strudel converter (Jong, 2025), which directly transliterates MIDI events into Strudel code. Baseline prompts and decoding settings are described in Appendix F.3.

Implementation details. All experiments use open-weight models from the Qwen3 family (Qwen Team, 2025b), specifically 4B (`Qwen3-4B`) and 8B (`Qwen3-8B`) models. We split the STRUDEL-SYNTH training set into two disjoint halves of approximately 10K samples each. The first half is used for SFT with paired (MIDI, Strudel) examples; the second half is used for RL, where only the MIDI input is provided. For SFT, we fine-tune each model for one epoch with LoRA (Hu et al., 2022). For RL, we initialize from the SFT checkpoint and train on a mixture of STRUDEL-SYNTH MIDI and short-fragment (<30s) LMD MIDI. Unless otherwise specified, we set the reward weights to $(w_1, w_2) = (1.0, 0.5)$ for faithfulness and readability. Additional hyperparameters and compute details are provided in Appendix F.4.

3.2 MAIN RESULTS

Table 2 summarizes the main quantitative results. Overall, DECOMPOSER is more faithful than frontier LLMs while producing more readable code than the heuristic converter.

Table 2: **Main results across synthetic and real-world MIDI.** We report results on held-out splits of STRUDEL-SYNTH and LMD, matching the distributions used for training. DECOMPOSER improves the faithfulness–readability tradeoff for symbolic music decompilation: it outperforms frontier LLMs in execution faithfulness while producing substantially more readable and diverse code than the heuristic converter. ↓ and ↑ indicate whether lower or higher values are better, respectively.

		Faithfulness				Readability		Diversity
Dataset	**Method**	↑Comp.	↑Onset	↑Frame	↑Inst	↑Rubric	↓Rank	↓SelfCB
STRUDEL -SYNTH	Heuristic	1.00	0.99	0.83	0.99	0.05	7.89	0.72
	Claude-Opus-4.6	0.73	0.50	0.59	0.42	0.42	4.41	0.46
	Gemini-3.5-Flash	0.57	0.50	0.59	0.42	0.45	3.75	0.46
	GPT-5.5	0.80	0.57	0.63	0.50	0.39	2.76	0.47
	Qwen3-4B	0.06	0.03	0.08	0.01	0.37	–	–
	+ SFT	0.58	0.41	0.46	0.39	0.58	4.76	0.42
	+ SFT+RL (ours)	0.99	0.71	0.67	0.70	0.73	3.65	0.56
	Qwen3-8B	0.19	0.05	0.10	0.02	0.33	–	–
	+ SFT	0.62	0.46	0.46	0.41	0.58	4.96	0.44
	+ SFT+RL (ours)	0.99	0.73	0.69	0.72	0.74	3.83	0.55
LMD	Heuristic	1.00	0.95	0.67	0.89	0.09	7.62	0.77
	Claude-Opus-4.6	0.75	0.28	0.33	0.21	0.36	4.47	0.60
	Gemini-3.5-Flash	0.63	0.22	0.30	0.17	0.34	4.37	0.51
	GPT-5.5	0.82	0.27	0.31	0.23	0.29	4.27	0.56
	Qwen3-4B	0.17	0.04	0.12	0.03	0.35	–	–
	+ SFT	0.45	0.09	0.13	0.07	0.55	4.01	0.38
	+ SFT+RL (ours)	0.99	0.54	0.49	0.52	0.70	3.13	0.53
	Qwen3-8B	0.18	0.04	0.11	0.02	0.28	–	–
	+ SFT	0.44	0.10	0.14	0.09	0.54	4.01	0.37
	+ SFT+RL (ours)	0.99	0.60	0.53	0.58	0.61	4.12	0.47

DECOMPOSER is more faithful than frontier LLMs. Frontier LLMs (Claude-Opus-4.6, Gemini-3.5-Flash, GPT-5.5) can generate executable Strudel code, yet their renders often diverge from the input MIDI. This gap widens on LMD: compared with the best frontier LLM, DECOMPOSER (8B) improves Onset F1 by $+0.16$ on STRUDEL-SYNTH and by $+0.32$ on LMD, suggesting that real-world MIDI requires stronger input grounding than general-purpose LLMs provide.

DECOMPOSER retains readability comparable to frontier LLMs. The heuristic converter is nearly perfectly faithful by construction, but its note-by-note transliteration is ranked least readable on both datasets. DECOMPOSER instead ranks within the range of frontier LLMs, and ahead of all of them on LMD. A human evaluation (Appendix I) further suggests that DECOMPOSER is comparable to GPT-5.5 in readability and is preferred over the heuristic converter in 79.5% of comparisons. Rubric, which also serves as our RL reward, shows the same gap to the heuristic converter.

DECOMPOSER avoids diversity collapse. Compared with SFT alone, applying RL increases SelfCB: for the 8B model, SelfCB rises by $+0.11$ on STRUDEL-SYNTH and $+0.10$ on LMD. This is consistent with prior work showing that reward optimization can reduce output diversity (Wang et al., 2026; West & Potts, 2025; Zhang et al., 2025). Importantly, its SelfCB remains well below that of the heuristic converter (-0.17 on STRUDEL-SYNTH and -0.30 on LMD) and comparable to frontier LLMs, showing that DECOMPOSER does not collapse to a fixed transliteration template.

Execution-based RL improves real-world decompilation. SFT alone transfers poorly to real-world MIDI: for the 8B model, SFT improves Onset F1 by $+0.41$ on STRUDEL-SYNTH but only by $+0.06$ on LMD. Execution-based RL shows the complementary effect, giving a larger gain on LMD than on STRUDEL-SYNTH ($+0.50$ vs. $+0.27$ Onset F1). This improvement requires no reference Strudel programs during RL, showing that DECOMPOSER can bootstrap from synthetic paired data and then adapt to real-world MIDI through label-free execution feedback.

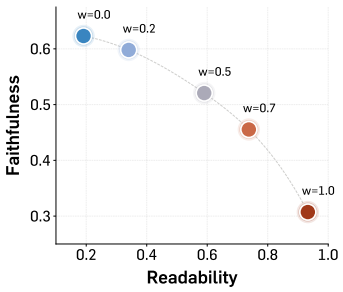


Figure 4: **Effect of readability weight (w_2).** We vary w_2 for Qwen3-8B with fixed $w_1 = 1.0$. Larger w_2 improves readability at the cost of faithfulness.

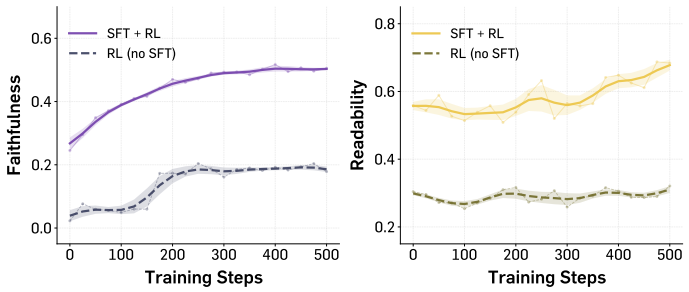


Figure 5: **Effect of SFT initialization.** We show faithfulness (left) and readability (right) rewards during RL training on Qwen3-8B, with and without SFT initialization. Without SFT, RL plateaus early at a degenerate low-reward solution; SFT initialization provides a warm start that enables both rewards to improve steadily.

3.3 Ablation Studies

Effect of readability weight. We vary the readability weight w_2 while keeping the faithfulness weight fixed at $w_1 = 1.0$. Figure 4 reveals a clear faithfulness–readability tradeoff: with $w_2 = 0$, the model produces programs that better match the input MIDI but are often verbose and close to note-by-note transcriptions. Adding a readability reward shifts the model toward more abstract and editable code, yet sacrificing faithfulness. We set $w_2 = 0.5$ for our main experiments, as it improves readability substantially without the large faithfulness drop observed at higher weights.

Effect of SFT initialization. We compare RL initialized from the SFT checkpoint against RL initialized directly from the unadapted `Qwen3-8B` model. As shown in Figure 5, RL without SFT plateaus early: the policy collapses to short, literal note lists that compile but capture little of the input, and readability does not improve. SFT places the policy into a region where sampled programs compile and express nontrivial musical structure, making execution-based RL feasible.

3.4 Generalization Across Datasets

We test whether Decomposer generalizes beyond the training distributions using the 8B model on four datasets: longer LMD (30–60 s), GigaMIDI, NES-MDB, and Nottingham, covering shifts in duration, corpus source, genre, and instrumentation (full details in Appendix H).

Table 3 shows that Decomposer is more faithful than GPT-5.5 on every distribution with comparable readability. Faithfulness transfers well to Nottingham and GigaMIDI, suggesting that it can recover compact programmatic structure beyond the synthetic Strudel distribution. Both models score lower on longer LMD excerpts and NES-MDB, as inputs grow longer or diverge from training data. Even so, Decomposer always compiles, whereas GPT-5.5 fails on 17% of longer LMD inputs (see Appendix H.1 for more details), and readability holds, with the best Rank on longer LMD. Note that closing this gap requires no new paired data: since RL needs only unpaired MIDI, such held-out corpora can be added to post-training directly.

Table 3: **Generalization across datasets.** We report Best@5 Onset F1 and Rank, selected by Inst F1. Decomposer is more faithful than GPT-5.5 on all datasets while keeping comparable readability.

	GPT-5.5		Decomposer	
Dataset	↑Onset	↓Rank	↑Onset	↓Rank
Str-Synth	0.70	1.61	**0.86**	**1.39**
LMD (<30s)	0.44	**1.44**	**0.70**	1.56
LMD (30–60s)	0.21	1.81	**0.35**	**1.19**
GigaMIDI	0.57	1.61	**0.61**	**1.39**
NES-MDB	0.37	**1.49**	**0.43**	1.51
Nottingham	0.51	1.50	**0.76**	1.50

4 Related Work

We discuss the most relevant literature here and provide more discussion in Appendix G.

Decompilation. Decompilation casts program synthesis as inferring source programs from non-code artifacts, including source code from binaries (Tan et al., 2024; Zou et al., 2025; She et al., 2024) or user interfaces (Wu et al., 2025; Beltramelli, 2018; Jain et al., 2019; Xiao et al., 2024), vector graphics programs from raster images (Rodriguez et al., 2024; 2025; Belouadi et al., 2024; Ellis et al., 2018), and CAD programs from 3D shapes (Rukhovich et al., 2025; Kolodiazhnyi et al., 2026; Chen et al., 2025; Wang et al., 2025). Earlier methods rely on SFT over paired (artifact, code) data (Tan et al., 2024; Rodriguez et al., 2024; Rukhovich et al., 2025), while recent work adds RL with execution-based rewards (Rodriguez et al., 2025; Kolodiazhnyi et al., 2026). For example, Rodriguez et al. (2025) decompile a raster image into an SVG program by rewarding a rendered SVG's visual similarity to the input, and Kolodiazhnyi et al. (2026) decompile a 3D shape into a CAD program by rewarding the executed solid's geometric agreement with the input. We extend this paradigm to music with the MIDI-to-Strudel decompilation task, using execution-based RL that rewards faithful and readable renders.

Music analysis-by-synthesis. An observed musical signal can be explained by fitting a generative process whose output reconstructs it. Prior work instantiates this idea with parametrized audio-generation processes, ranging from classic instrument tone analysis (Risset & Mathews, 1969) and spectral modeling synthesis (Serra & Smith, 1990) to modern differentiable synthesis (Engel et al., 2020a;b), physical or instrument models (Diaz et al., 2023; Wang et al., 2014; Renault et al., 2022), and audio-effect or mixing-chain models (Steinmetz et al., 2021; 2022; 2024; Vanka et al., 2024; Comunità et al., 2025). While closely related, these methods cast music analysis as parameter recovery for a predefined acoustic generator; DECOMPOSER instead operates over symbolic music, recovering Strudel programs whose structure is inferred rather than specified a priori.

5 Discussion and Conclusion

Limitations and future work. While DECOMPOSER demonstrates that symbolic music can be decompiled into faithful and readable programs, several directions remain for future work. First, our readability reward uses a fixed checklist, which may be susceptible to reward hacking and favor abstractions ill-suited to the input, such as harmonic constructs in primarily percussive music. Future work could explore more fine-grained, input-conditional rubrics (Gandhi et al., 2026). Our experiments also focus primarily on short MIDI segments; scaling to long-form music could benefit from training on longer sequences and rewards that capture sections, recurrence, and development. Diversity could likewise be encouraged explicitly through reward design (Li et al., 2025) or optimization (He et al., 2025). Beyond these methodological improvements, the music decompilation task itself could expand from recovering a single final program to inferring a *creative trace*: a sequence of program edits and executions through which musical structure emerges (Manesh et al., 2024; Hammad et al., 2026). At the application level, future work could explore live coding assistance, code-based music editing and understanding, and extensions to audio-to-code decompilation, which we discuss in more detail in Appendix J.

In this paper, we have introduced *symbolic music decompilation*, the task of recovering executable and editable music programs from symbolic music. This task requires a model to infer the higher-level musical structure that MIDI leaves implicit, such as repetition, layering, and temporal organization. We instantiated this problem as MIDI-to-Strudel decompilation and presented DECOMPOSER, a two-stage post-training framework that combines supervised learning on synthetic (Strudel, MIDI) pairs with reinforcement learning on unpaired MIDI. Across synthetic and real-world benchmarks, DECOMPOSER improves the faithfulness–readability tradeoff over both heuristic conversion and frontier LLMs, producing programs that better preserve the input music while remaining substantially more readable than note-by-note transcriptions. We hope this work helps facilitate future research on music decompilation systems that recover not only performed notes, but also the editable programmatic structure that makes them interpretable, reusable, and extensible.

AI USE STATEMENT

In this work, we used generative AI tools to generate the synthetic data used in our proposed data generation pipeline (Section 2.3). The use of LLM-generated synthetic data is an explicit component of our method and is described in detail in the corresponding section of the paper. Generative AI tools were also used to assist with parts of the code implementation. In these cases, the method design, reward designs, and implementation plans were specified by the authors, and generative AI was used to translate these specifications into code. AI tools were additionally used for code review, including suggestions for improving code readability and efficiency, and to edit portions of the manuscript for clarity and readability.

We did not use generative AI tools to develop theoretical models or conceptual frameworks, propose or refine hypotheses, design the research methodology or experiments, clean or reformat datasets, or interpret experimental results. Formulating mathematical claims, providing ingredients for mathematical proofs, writing proofs, conducting qualitative or thematic data analysis, and translation were not applicable to this work.

All AI-assisted work was reviewed by the authors. In particular, AI-assisted code included in the final implementation was manually inspected, tested, and verified for correctness by two authors. AI-generated synthetic data was processed and evaluated according to the methodology described in the paper, and all AI-edited manuscript text was reviewed and revised by the authors. We take responsibility for the final content of this work, including text, claims or artifacts produced with the aid of generative AI.

REPRODUCIBILITY STATEMENT

We provide implementation, dataset construction, training configuration, and evaluation procedures in Section 3.1 and Appendices E–F. To support reproducibility, we make our model checkpoint and dataset available at `decomposer-page.netlify.app`.

REFERENCES

Sam Aaron. Sonic Pi - The Live Coding Music Synth for Everyone. https://sonic-pi.net/. Retrieved 21 May 2026.

Pranjal Aggarwal and Sean Welleck. L1: Controlling how long a reasoning model thinks with reinforcement learning. In *Second Conference on Language Modeling*, 2025. URL https://openreview.net/forum?id=4jdIxXBNve.

Anthropic. Claude opus 4.6 system card. Technical report, Anthropic, February 2026. URL https://www.anthropic.com/claude-opus-4-6-system-card.

Jordi Armengol-Estapé, Jackson Woodruff, Chris Cummins, and Michael F. P. O'Boyle. Slade: A portable small language model decompiler for optimized assembly. In *Proceedings of the 2024 IEEE/ACM International Symposium on Code Generation and Optimization*, CGO '24, pp. 67–80. IEEE Press, 2024. ISBN 9798350395099. doi: 10.1109/CGO57630.2024.10444788. URL https://doi.org/10.1109/CGO57630.2024.10444788.

Daman Arora and Andrea Zanette. Training language models to reason efficiently. In D. Belgrave, C. Zhang, H. Lin, R. Pascanu, P. Koniusz, M. Ghassemi, and N. Chen (eds.), *Advances in Neural Information Processing Systems*, volume 38, Main Conference, pp. 60770–60808. Curran Associates, Inc., 2025. doi: 10.52202/085713-2032. URL https://proceedings.neurips.cc/paper_files/paper/2025/file/579b5b84311e122584dedab1d3b7613f-Paper-Conference.pdf.

Rahul K. Arora, Jason Wei, Rebecca Soskin Hicks, Preston Bowman, Joaquin Quiñonero-Candela, Foivos Tsimpourlas, Michael Sharman, Meghan Shah, Andrea Vallone, Alex Beutel, Johannes Heidecke, and Karan Singhal. Healthbench: Evaluating large language models towards improved human health, 2025. URL https://arxiv.org/abs/2505.08775.

Jonas Belouadi, Simone P Ponzetto, and Steffen Eger. Detikzify: Synthesizing graphics programs for scientific figures and sketches with tikz. *Advances in Neural Information Processing Systems*, 37: 85074–85108, 2024.

Tony Beltramelli. pix2code: Generating code from a graphical user interface screenshot. In *Proceedings of the ACM SIGCHI Symposium on Engineering Interactive Computing Systems*, EICS '18, New York, NY, USA, 2018. Association for Computing Machinery. ISBN 9781450358972. doi: 10.1145/3220134.3220135. URL https://doi.org/10.1145/3220134.3220135.

Ole Martin Bjørndalen, Raphaël Doursenaud, et al. Mido: Midi objects for python. URL https://mido.readthedocs.io/en/stable/. Accessed: 2026-06-20.

Sebastian Böck, Filip Korzeniowski, Jan Schlüter, Florian Krebs, and Gerhard Widmer. madmom: a new Python Audio and Music Signal Processing Library. In *Proceedings of the 24th ACM International Conference on Multimedia*, pp. 1174–1178, Amsterdam, The Netherlands, 10 2016. doi: 10.1145/2964284.2973795.

Peter Boot, Anja Volk, and W. Bas de Haas. Evaluating the role of repeated patterns in folk song classification and compression. *Journal of New Music Research*, 45(3):223–238, 2016. doi: 10.1080/09298215.2016.1208666. URL https://doi.org/10.1080/09298215.2016.1208666.

Raymond P.L. Buse and Westley R. Weimer. Learning a metric for code readability. *IEEE Transactions on Software Engineering*, 36(4):546–558, 2010. doi: 10.1109/TSE.2009.70.

Sungkyun Chang, Emmanouil Benetos, Holger Kirchhoff, and Simon Dixon. Yourmt3+: Multi-instrument music transcription with enhanced transformer architectures and cross-dataset stem augmentation, 2024. URL https://arxiv.org/abs/2407.04822.

Cheng Chen, Jiacheng Wei, Tianrun Chen, Chi Zhang, Xiaofeng Yang, Shangzhan Zhang, Bingchen Yang, Chuan-Sheng Foo, Guosheng Lin, Qixing Huang, and Fayao Liu. Cadcrafter: Generating computer-aided design models from unconstrained images, 2025. URL https://arxiv.org/abs/2504.04753.

Nick Collins, Alex McLean, Julian Rohrhuber, and Adrian Ward. Live coding in laptop performance. *Organised sound*, 8(3):321–330, 2003.

Marco Comunità, Christian J. Steinmetz, and Joshua D. Reiss. Differentiable black-box and gray-box modeling of nonlinear audio effects, 2025. URL https://arxiv.org/abs/2502.14405.

WJ Conover. *Practical nonparametric statistics*. John Wiley & Sons, Inc, New York, NY, USA, 1999.

Josef Dai, Xuehai Pan, Ruiyang Sun, Jiaming Ji, Xinbo Xu, Mickel Liu, Yizhou Wang, and Yaodong Yang. Safe RLHF: Safe reinforcement learning from human feedback. In *The Twelfth International Conference on Learning Representations*, 2024. URL https://openreview.net/forum?id=TyFrPOKYXw.

TheoryTab DB. Theorytab db: Tabs that show the theory behind songs. https://www.hooktheory.com/theorytab, 2026.

Roger T Dean. *The Oxford handbook of algorithmic music*. Oxford University Press, 2018.

Rodrigo Diaz, Ben Hayes, Charalampos Saitis, György Fazekas, and Mark Sandler. Rigid-body sound synthesis with differentiable modal resonators. In *ICASSP 2023 - 2023 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*, pp. 1–5, 2023. doi: 10.1109/ICASSP49357.2023.10095139.

Jacob Dineen, Aswin Rrv, Qin Liu, Zhikun Xu, Xiao Ye, Ming Shen, Zhaonan Li, Shijie Lu, Chitta Baral, Muhao Chen, and Ben Zhou. QA-LIGN: Aligning LLMs through constitutionally decomposed QA. In Christos Christodoulopoulos, Tanmoy Chakraborty, Carolyn Rose, and Violet Peng (eds.), *Findings of the Association for Computational Linguistics: EMNLP 2025*, pp. 20619–20642, Suzhou, China, November 2025. Association for Computational Linguistics. ISBN 979-8-89176-335-7. doi: 10.18653/v1/2025.findings-emnlp.1123. URL https://aclanthology.org/2025.findings-emnlp.1123/.

Seungheon Doh, Keunwoo Choi, and Juhan Nam. Talkplay: Multimodal music recommendation with large language models, 2025. URL https://arxiv.org/abs/2502.13713.

SeungHeon Doh, Junghyun Koo, Marco A. Martínez-Ramírez, Woosung Choi, Wei-Hsiang Liao, Qiyu Wu, Juhan Nam, and Yuki Mitsufuji. LLM2fx-tools: Tool calling for music post-production. In *The Fourteenth International Conference on Learning Representations*, 2026. URL https://openreview.net/forum?id=OyIJvyyB3R.

Chris Donahue, Huanru Henry Mao, and Julian McAuley. The nes music database: A multi-instrumental dataset with expressive performance attributes. In *ISMIR*, 2018.

Chris Donahue, Antoine Caillon, Adam Roberts, Ethan Manilow, Philippe Esling, Andrea Agostinelli, Mauro Verzetti, Ian Simon, Olivier Pietquin, Neil Zeghidour, and Jesse Engel. Singsong: Generating musical accompaniments from singing, 2023. URL https://arxiv.org/abs/2301.12662.

Kevin Ellis, Daniel Ritchie, Armando Solar-Lezama, and Joshua B. Tenenbaum. Learning to infer graphics programs from hand-drawn images, 2018. URL https://arxiv.org/abs/1707.09627.

Jesse Engel, Lamtharn (Hanoi) Hantrakul, Chenjie Gu, and Adam Roberts. Ddsp: Differentiable digital signal processing. In *International Conference on Learning Representations*, 2020a. URL https://openreview.net/forum?id=B1x1ma4tDr.

Jesse Engel, Rigel Swavely, Lamtharn Hanoi Hantrakul, Adam Roberts, and Curtis Hawthorne. Self-supervised pitch detection by inverse audio synthesis. In *ICML 2020 Workshop on Self-supervision in Audio and Speech*, 2020b. URL https://openreview.net/forum?id=RlVTYWhsky7.

Michael Eskin. General midi instrument program numbers. https://michaeleskin.com/abctools/general_midi_extended_v7.pdf. Accessed: 2026-06-20.

FluidSynth Developers. FluidSynth: A soundfont synthesizer. https://www.fluidsynth.org/. Accessed: 2026-06-20.

Eric Foxley. Nottingham music database, 2003. URL http://abc.sourceforge.net/NMD/.

Apurva Gandhi, Vishwas Suryanarayanan, Firoz Shaik, Raja Hasnain Anwar, Shubhang Desai, Thong Q. Nguyen, Muhammad Taqi Raza, Vishal Chowdhary, and Graham Neubig. Ppt-eval: A benchmark for computer-use agents on powerpoint tasks. In *International Conference on Machine Learning (ICML)*, Seoul, South Korea, July 2026. URL https://openreview.net/forum?id=GmeK95WQQ4.

Josh Gardner, Simon Durand, Daniel Stoller, and Rachel Bittner. Llark: A multimodal instruction-following language model for music. *Proc. of the International Conference on Machine Learning (ICML)*, 2024.

Joshua P Gardner, Ian Simon, Ethan Manilow, Curtis Hawthorne, and Jesse Engel. MT3: Multi-task multitrack music transcription. In *International Conference on Learning Representations*, 2022. URL https://openreview.net/forum?id=iMSjopcOn0p.

Sreyan Ghosh, Arushi Goel, Lasha Koroshinadze, Sang gil Lee, Zhifeng Kong, Joao Felipe Santos, Ramani Duraiswami, Dinesh Manocha, Wei Ping, Mohammad Shoeybi, and Bryan Catanzaro. Music flamingo: Scaling music understanding in audio language models, 2025. URL https://arxiv.org/abs/2511.10289.

Google DeepMind. Gemini 3.5 flash, May 2026. URL https://deepmind.google/technologies/gemini/.

Boyu Gou, Zanming Huang, Yuting Ning, Yu Gu, Michael Lin, Weijian Qi, Andrei Kopanev, Botao Yu, Bernal Jimenez Gutierrez, Yiheng Shu, Chan Hee Song, Jiaman Wu, Shijie Chen, Hanane Nour Moussa, TIANSHU ZHANG, Jian Xie, Yifei Li, Tianci Xue, Zeyi Liao, Kai Zhang, Boyuan Zheng, Zhaowei Cai, Viktor Rozgic, Morteza Ziyadi, Huan Sun, and Yu Su. Mind2web 2: Evaluating agentic search with agent-as-a-judge. In *The Thirty-ninth Annual Conference on Neural Information Processing Systems Datasets and Benchmarks Track*, 2025. URL https://openreview.net/forum?id=AUaW6DS9si.

Anisha Gunjal, Anthony Wang, Elaine Lau, Vaskar Nath, Yunzhong He, Bing Liu, and Sean M. Hendryx. Rubrics as rewards: Reinforcement learning beyond verifiable domains. In *The Fourteenth International Conference on Learning Representations*, 2026. URL https://openreview.net/forum?id=c1bTcrDmt4.

Daya Guo, Dejian Yang, Haowei Zhang, Junxiao Song, Ruoyu Zhang, Runxin Xu, Qihao Zhu, Shirong Ma, Peiyi Wang, Xiao Bi, et al. Deepseek-r1: Incentivizing reasoning capability in llms via reinforcement learning. *arXiv preprint arXiv:2501.12948*, 2025.

Noor Hammad, David Chuan-En Lin, Amy Smith, Max Kreminski, Erik Harpstead, and Jessica Hammer. Tracing creativity: A design space for creative activity traces in hci. In *Proceedings of the 2026 CHI Conference on Human Factors in Computing Systems*, CHI '26, New York, NY, USA, 2026. Association for Computing Machinery. ISBN 9798400722783. doi: 10.1145/3772318.3791263. URL https://doi.org/10.1145/3772318.3791263.

Helia Hashemi, Jason Eisner, Corby Rosset, Benjamin Van Durme, and Chris Kedzie. LLM-rubric: A multidimensional, calibrated approach to automated evaluation of natural language texts. In Lun-Wei Ku, Andre Martins, and Vivek Srikumar (eds.), *Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pp. 13806–13834, Bangkok, Thailand, August 2024. Association for Computational Linguistics. doi: 10.18653/v1/2024.acl-long.745. URL https://aclanthology.org/2024.acl-long.745/.

Marijn Haverbeke et al. Acorn: a small, fast, javascript-based javascript parser. https://github.com/acornjs/acorn. Accessed: 2026-06-20.

Andre He, Daniel Fried, and Sean Welleck. Rewarding the unlikely: Lifting grpo beyond distribution sharpening, 2025. URL https://arxiv.org/abs/2506.02355.

Edward J Hu, Yelong Shen, Phillip Wallis, Zeyuan Allen-Zhu, Yuanzhi Li, Shean Wang, Lu Wang, and Weizhu Chen. LoRA: Low-rank adaptation of large language models. In *International Conference on Learning Representations*, 2022. URL https://openreview.net/forum?id=nZeVKeeFYf9.

Chengyu Huang, Zhengxin Zhang, and Claire Cardie. Hapo: Training language models to reason concisely via history-aware policy optimization, 2025a. URL https://arxiv.org/abs/2505.11225.

Yichen Huang, Zachary Novack, Koichi Saito, Jiatong Shi, Shinji Watanabe, Yuki Mitsufuji, John Thickstun, and Chris Donahue. Aligning text-to-music evaluation with human preferences, 2025b. URL https://arxiv.org/abs/2503.16669.

Vanita Jain, Piyush Agrawal, Subham Banga, Rishabh Kapoor, and Shashwat Gulyani. Sketch2code: Transformation of sketches to UI in real-time using deep neural network. *CoRR*, abs/1910.08930, 2019. URL http://arxiv.org/abs/1910.08930.

Joel Jang, Seungone Kim, Bill Yuchen Lin, Yizhong Wang, Jack Hessel, Luke Zettlemoyer, Hannaneh Hajishirzi, Yejin Choi, and Prithviraj Ammanabrolu. Personalized soups: Personalized large language model alignment via post-hoc parameter merging. *arXiv preprint arXiv:2310.11564*, 2023.

Emanuel de Jong. Midi-to-strudel: Convert midi files to strudel code. https://github.com/Emanuel-de-Jong/MIDI-To-Strudel, 2025. Accessed: 2026-05-25.

Yewon Kim, Sung-Ju Lee, and Chris Donahue. Amuse: Human-ai collaborative songwriting with multimodal inspirations. In *Proceedings of the 2025 CHI Conference on Human Factors in Computing Systems*, CHI '25, New York, NY, USA, 2025. Association for Computing Machinery. ISBN 9798400713941. doi: 10.1145/3706598.3713818. URL https://doi.org/10.1145/3706598.3713818.

Donald E. Knuth, James H. Morris, Jr., and Vaughan R. Pratt. Fast pattern matching in strings. *SIAM Journal on Computing*, 6(2):323–350, 1977. doi: 10.1137/0206024. URL https://doi.org/10.1137/0206024.

Maksim Kolodiazhnyi, Denis Tarasov, Dmitrii Zhemchuzhnikov, Alexander Nikulin, Ilya Zisman, Anna Vorontsova, Anton Konushin, Vladislav Kurenkov, and Danila Rukhovich. cadrille: Multi-modal cad reconstruction with reinforcement learning, 2026. URL https://arxiv.org/abs/2505.22914.

Keon Ju Maverick Lee, Jeff Ens, Sara Adkins, Pedro Sarmento, Mathieu Barthet, and Philippe Pasquier. The gigamidi dataset with features for expressive music performance detection. *Transactions of the International Society for Music Information Retrieval (TISMIR)*, 8(1):1–19, 2025.

Gang Li, Yan Chen, Ming Lin, and Tianbao Yang. DRPO: Efficient reasoning via decoupled reward policy optimization. In *The Fourteenth International Conference on Learning Representations*, 2026. URL https://openreview.net/forum?id=GP5RHZnEsw.

Tianjian Li, Yiming Zhang, Ping Yu, Swarnadeep Saha, Daniel Khashabi, Jason Weston, Jack Lanchantin, and Tianlu Wang. Jointly reinforcing diversity and quality in language model generations. *arXiv preprint arXiv:2509.02534*, 2025. URL https://arxiv.org/abs/2509.02534.

Shih-Yang Liu, Xin Dong, Ximing Lu, Shizhe Diao, Peter Belcak, Mingjie Liu, Min-Hung Chen, Hongxu Yin, Yu-Chiang Frank Wang, Kwang-Ting Cheng, Yejin Choi, Jan Kautz, and Pavlo Molchanov. Gdpo: Group reward-decoupled normalization policy optimization for multi-reward rl optimization, 2026. URL https://arxiv.org/abs/2601.05242.

Ryan Louie, Andy Coenen, Cheng Zhi Huang, Michael Terry, and Carrie J. Cai. Novice-ai music co-creation via ai-steering tools for deep generative models. In *Proceedings of the 2020 CHI Conference on Human Factors in Computing Systems*, CHI '20, pp. 1–13, New York, NY, USA, 2020. Association for Computing Machinery. ISBN 9781450367080. doi: 10.1145/3313831.3376739. URL https://doi.org/10.1145/3313831.3376739.

Wei-Tsung Lu, Ju-Chiang Wang, and Yun-Ning Hung. Multitrack music transcription with a time-frequency perceiver, 2023. URL https://arxiv.org/abs/2306.10785.

Ben Maman and Amit H Bermano. Unaligned supervision for automatic music transcription in the wild. In Kamalika Chaudhuri, Stefanie Jegelka, Le Song, Csaba Szepesvari, Gang Niu, and Sivan Sabato (eds.), *Proceedings of the 39th International Conference on Machine Learning*, volume 162 of *Proceedings of Machine Learning Research*, pp. 14918–14934. PMLR, 17–23 Jul 2022. URL https://proceedings.mlr.press/v162/maman22a.html.

Daniel Manesh, Douglas Bowman Jr., and Sang Won Lee. Sharp: Exploring version control systems in live coding music. In *Proceedings of the 16th Conference on Creativity & Cognition*, C&C '24, pp. 426–437, New York, NY, USA, 2024. Association for Computing Machinery. ISBN 9798400704857. doi: 10.1145/3635636.3656195. URL https://doi.org/10.1145/3635636.3656195.

James McCartney. Supercollider, a new real time synthesis language. In *Proceedings of the 1996 International Computer Music Conference*, pp. 257–258. The International Computer Music Association, 1996.

Glenn McDonald. Every noise at once (1d). https://www.everynoise.com/everynoise1d.html, 2013. Accessed: 2026-05-31.

Alex McLean and Geraint Wiggins. Tidal–pattern language for the live coding of music. In *Proceedings of the 7th sound and music computing conference*, pp. 331–334, 2010.

Andrew Mcleod and Mark Steedman. Evaluating automatic polyphonic music transcription. In Emilia Gómez, Xiao Hu, and Eric Humphrey (eds.), *Proceedings of the 19th International Society for Music Information Retrieval Conference, ISMIR 2018*, pp. 42–49, November 2018. ISBN 9782954035123. URL http://ismir2018.ircam.fr/. 19th International Society for Music Information Retrieval Conference, ISMIR 2019 ; Conference date: 23-09-2018 Through 27-09-2018.

Zhiyu Mei, Wei Fu, Kaiwei Li, Guangju Wang, Huanchen Zhang, and Yi Wu. Real: Efficient rlhf training of large language models with parameter reallocation. In *Proceedings of the Eighth Conference on Machine Learning and Systems, MLSys 2025, Santa Clara, CA, USA, May 12-15, 2025*. mlsys.org, 2025.

Jan Melechovsky, Abhinaba Roy, and Dorien Herremans. Midicaps: A large-scale midi dataset with text captions. *arXiv:2406.02255*, 2024.

Subhabrata Mukherjee, Arindam Mitra, Ganesh Jawahar, Sahaj Agarwal, Hamid Palangi, and Ahmed Awadallah. Orca: Progressive learning from complex explanation traces of gpt-4. *arXiv preprint arXiv:2306.02707*, 2023.

OpenAI. Gpt-5.5 system card. Technical report, OpenAI, April 2026. URL https://openai.com/index/gpt-5-5-system-card/.

Kishore Papineni, Salim Roukos, Todd Ward, and Wei-Jing Zhu. Bleu: a method for automatic evaluation of machine translation. In *Proceedings of the 40th Annual Meeting on Association for Computational Linguistics*, ACL '02, pp. 311–318, USA, 2002. Association for Computational Linguistics. doi: 10.3115/1073083.1073135. URL https://doi.org/10.3115/1073083.1073135.

Aditya Pathak, Rachit Gandhi, Vaibhav Uttam, Arnav Ramamoorthy, Pratyush Ghosh, Aaryan Raj Jindal, Shreyash Verma, Aditya Mittal, Aashna Ased, Chirag Khatri, Yashwanth Nakka, Devansh, Jagat Sesh Challa, and Dhruv Kumar. Rubric is all you need: Improving llm-based code evaluation with question-specific rubrics. In *Proceedings of the 2025 ACM Conference on International Computing Education Research V.1*, ICER '25, pp. 181–195, New York, NY, USA, 2025. Association for Computing Machinery. ISBN 9798400713408. doi: 10.1145/3702652.3744220. URL https://doi.org/10.1145/3702652.3744220.

F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and E. Duchesnay. Scikit-learn: Machine learning in Python. *Journal of Machine Learning Research*, 12:2825–2830, 2011.

Christopher Petrie. Programming music with sonic pi promotes positive attitudes for beginners. *Computers & Education*, 179:104409, 2022. ISSN 0360-1315. doi: https://doi.org/10.1016/j.compedu.2021.104409. URL https://www.sciencedirect.com/science/article/pii/S0360131521002864.

Miller Puckette et al. Pure data: another integrated computer music environment. *Proceedings of the second intercollege computer music concerts*, pp. 37–41, 1996.

Qwen Team. Qwen2.5-omni technical report. *arXiv preprint arXiv:2503.20215*, 2025a.

Qwen Team. Qwen3 technical report, 2025b. URL https://arxiv.org/abs/2505.09388.

Qwen Team. Qwen3.6-35B-A3B: Agentic coding power, now open to all, April 2026. URL https://qwen.ai/blog?id=qwen3.6-35b-a3b.

Colin Raffel. *Learning-based methods for comparing sequences, with applications to audio-to-midi alignment and matching*. Columbia University, 2016.

Colin Raffel, Brian McFee, Eric J Humphrey, Justin Salamon, Oriol Nieto, Dawen Liang, Daniel PW Ellis, and C Colin Raffel. Mir_eval: A transparent implementation of common mir metrics. In *ISMIR*, volume 10, pp. 2014, 2014.

Shuo Ren, Daya Guo, Shuai Lu, Long Zhou, Shujie Liu, Duyu Tang, Neel Sundaresan, Ming Zhou, Ambrosio Blanco, and Shuai Ma. Codebleu: a method for automatic evaluation of code synthesis. *arXiv preprint arXiv:2009.10297*, 2020.

Lenny Renault, Rémi Mignot, and Axel Roebel. Differentiable Piano Model for MIDI-to-Audio Performance Synthesis. In *25th International Conference on Digital Audio Effects (DAFx20in22)*, Vienna, Austria, September 2022. doi: 10.5281/zenodo.7092602. URL https://hal.science/hal-03779081.

Mitchel Resnick, Brad Myers, Kumiyo Nakakoji, Ben Shneiderman, Randy Pausch, and Mike Eisenberg. Design principles for tools to support creative thinking. *Report of Workshop on Creativity Support Tools*, 20, 01 2005.

Jean-Claude Risset and Max V Mathews. Analysis of musical-instrument tones. *Physics today*, 22 (2):23–30, 1969.

Daniel Ritchie, Paul Guerrero, R Kenny Jones, Niloy J Mitra, Adriana Schulz, Karl DD Willis, and Jiajun Wu. Neurosymbolic models for computer graphics. In *Computer graphics forum*, volume 42, pp. 545–568. Wiley Online Library, 2023.

Juan A. Rodriguez, Abhay Puri, Shubham Agarwal, Issam H. Laradji, Pau Rodriguez, Sai Rajeswar, David Vazquez, Christopher Pal, and Marco Pedersoli. Starvector: Generating scalable vector graphics code from images and text, 2024. URL https://arxiv.org/abs/2312.11556.

Juan A. Rodriguez, Haotian Zhang, Abhay Puri, Rishav Pramanik, Aarash Feizi, Pascal Wichmann, Arnab Kumar Mondal, Mohammad Reza Samsami, Rabiul Awal, Perouz Taslakian, Spandana Gella, Sai Rajeswar, David Vazquez, Christopher Pal, and Marco Pedersoli. Rendering-aware reinforcement learning for vector graphics generation. In *The Thirty-ninth Annual Conference on Neural Information Processing Systems*, 2025. URL https://openreview.net/forum?id=2Twzlf6qFv.

Felix Roos and Alex McLean. Strudel: Live coding patterns on the web, April 2023. URL https://doi.org/10.5281/zenodo.7842142.

Simon Rouard, Michael Krause, Axel Roebel, Carl-Johann Simon-Gabriel, and Alexandre Défossez. Muscriptor: An open model for multi-instrument music transcription, 2026. URL https://arxiv.org/abs/2607.08168.

Jie Ruan, Inderjeet Jayakumar Nair, Shuyang Cao, Amy Liu, Sheza Munir, Micah Pollens-Dempsey, Yune-Ting Tiffany Chiang, Lucy R. Kates, Nicholas David, Sihan Chen, Ruxin Yang, Yuqian Yang, Jihyun Jasmine Gump, Tessa Bialek, Vivek S Sankaran, Margo Schlanger, and Lu Wang. Expertlongbench: Benchmarking language models on expert-level long-form generation tasks with structured checklists. In *The Fourteenth International Conference on Learning Representations*, 2026. URL https://openreview.net/forum?id=nJvgBolRcR.

Danila Rukhovich, Elona Dupont, Dimitrios Mallis, Kseniya Cherenkova, Anis Kacem, and Djamila Aouada. Cad-recode: Reverse engineering cad code from point clouds. In *Proceedings of the IEEE/CVF International Conference on Computer Vision*, pp. 9801–9811, 2025.

Simone Scalabrino, Mario Linares-Vásquez, Rocco Oliveto, and Denys Poshyvanyk. A comprehensive model for code readability. *Journal of Software: Evolution and Process*, 30(6):e1958, 2018. doi: https://doi.org/10.1002/smr.1958. URL https://onlinelibrary.wiley.com/doi/abs/10.1002/smr.1958. e1958 smr.1958.

John Schulman, Filip Wolski, Prafulla Dhariwal, Alec Radford, and Oleg Klimov. Proximal policy optimization algorithms, 2017. URL https://arxiv.org/abs/1707.06347.

Xavier Serra and Julius Orion Smith. Spectral modeling synthesis. *Computer Music Journal*, 14: 12–24, 1990. URL https://api.semanticscholar.org/CorpusID:59309314.

Xinyu She, Yanjie Zhao, and Haoyu Wang. Wadec: Decompiling webassembly using large language model, 2024. URL https://arxiv.org/abs/2406.11346.

Saebyeol Shin, Chao Wan, Zhenzhen Liu, Justin Lovelace, Daniel C Lin, Kilian Q Weinberger, and John Thickstun. Music transcription with (almost) no supervision. *arXiv preprint arXiv:2605.24193*, 2026.

C. Steinmetz, Nicholas J. Bryan, and Joshua D. Reiss. Style transfer of audio effects with differentiable signal processing. *ArXiv*, abs/2207.08759, 2022.

C. Steinmetz, Shubhr Singh, Marco Comunità, Ilias Ibnyahya, Shanxin Yuan, Emmanouil Benetos, and Joshua D. Reiss. St-ito: Controlling audio effects for style transfer with inference-time optimization. In *International Society for Music Information Retrieval Conference*, 2024. URL https://api.semanticscholar.org/CorpusID:273653798.

Christian J. Steinmetz, Jordi Pons, Santiago Pascual, and Joan Serrà. Automatic multitrack mixing with a differentiable mixing console of neural audio effects. In *ICASSP 2021 - 2021 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*, pp. 71–75, 2021. doi: 10.1109/ICASSP39728.2021.9414364.

Jinyan Su and Claire Cardie. Thinking fast and right: Balancing accuracy and reasoning length with adaptive rewards, 2025. URL https://arxiv.org/abs/2505.18298.

Hanzhuo Tan, Qi Luo, Jing Li, and Yuqun Zhang. LLM4Decompile: Decompiling binary code with large language models. In Yaser Al-Onaizan, Mohit Bansal, and Yun-Nung Chen (eds.), *Proceedings of the 2024 Conference on Empirical Methods in Natural Language Processing*, pp. 3473–3487, Miami, Florida, USA, November 2024. Association for Computational Linguistics. doi: 10.18653/v1/2024.emnlp-main.203. URL https://aclanthology.org/2024.emnlp-main.203/.

Raphael Tang, Crystina Zhang, Xueguang Ma, Jimmy Lin, and Ferhan Türe. Found in the middle: Permutation self-consistency improves listwise ranking in large language models. In *Proceedings of the 2024 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies (Volume 1: Long Papers)*, pp. 2327–2340, 2024.

Soumya Vanka, Christian Steinmetz, Jean-Baptiste Rolland, Joshua Reiss, and György Fazekas. Diff-mst: Differentiable mixing style transfer. In *Proc. of the 25th Int. Society for Music Information Retrieval Conf. (ISMIR)*, San Francisco, USA, 2024. Int. Society for Music Information Retrieval (ISMIR).

Vijay Viswanathan, Yanchao Sun, Xiang Kong, Meng Cao, Graham Neubig, and Tongshuang Wu. Checklists are better than reward models for aligning language models. In *The Thirty-ninth Annual Conference on Neural Information Processing Systems*, 2025. URL https://openreview.net/forum?id=RPRqKhjrr6.

Cheng-i Wang, Tamara Smyth, and Zachary C Lipton. Estimation of saxophone control parameters by convex optimization. In *CIM14, Conference on Interdisciplinary Musicology: proceedings. Conference on Interdisciplinary Musicology (9th: 2014: Berlin, Germany)*, volume 2014, pp. 280, 2014.

Ge Wang and Perry R Cook. " on-the-fly programming: Using code as an expressive musical instrument". In *NIME*, volume 4, pp. 138–143, 2004.

Ge Wang, Perry R. Cook, and Spencer Salazar. Chuck: A strongly timed computer music language. *Computer Music Journal*, 39(4):10–29, 12 2015. ISSN 0148-9267. doi: 10.1162/COMJ_a_00324. URL https://doi.org/10.1162/COMJ_a_00324.

Siyu Wang, Cailian Chen, Xinyi Le, Qimin Xu, Lei Xu, Yanzhou Zhang, and Jie Yang. Cad-gpt: Synthesising cad construction sequence with spatial reasoning-enhanced multimodal llms. *Proceedings of the AAAI Conference on Artificial Intelligence*, 39(8):7880–7888, April 2025. ISSN 2159-5399. doi: 10.1609/aaai.v39i8.32849. URL http://dx.doi.org/10.1609/aaai.v39i8.32849.

Yichen Wang, Chenghao Yang, Tenghao Huang, Muhao Chen, Jonathan May, and Mina Lee. Optimizing diversity and quality through base-aligned model collaboration. In *Proceedings of the 43rd International Conference on Machine Learning*, Proceedings of Machine Learning Research. PMLR, 2026.

Peter West and Christopher Potts. Base models beat aligned models at randomness and creativity. In *Second Conference on Language Modeling*, 2025. URL https://openreview.net/forum?id=vqN8uom4A1.

Wai Kin Wong, Huaijin Wang, Zongjie Li, Zhibo Liu, Shuai Wang, Qiyi Tang, Sen Nie, and Shi Wu. Refining decompiled c code with large language models, 2023. URL https://arxiv.org/abs/2310.06530.

Fan Wu, Cuiyun Gao, Shuqing Li, Xin-Cheng Wen, and Qing Liao. Mllm-based ui2code automation guided by ui layout information. *Proc. ACM Softw. Eng.*, 2(ISSTA), June 2025. doi: 10.1145/3728925. URL https://doi.org/10.1145/3728925.

Violet Xiang, Chase Blagden, Rafael Rafailov, Nathan Lile, Sang Truong, Chelsea Finn, and Nick Haber. Just enough thinking: Efficient reasoning with adaptive length penalties reinforcement learning, 2025. URL https://arxiv.org/abs/2506.05256.

Shuhong Xiao, Yunnong Chen, Jiazhi Li, Liuqing Chen, Lingyun Sun, and Tingting Zhou. Prototype2code: End-to-end front-end code generation from ui design prototypes, 2024. URL https://arxiv.org/abs/2405.04975.

Adrien Ycart, Lele Liu, Emmanouil Benetos, and Marcus T. Pearce. Investigating the perceptual validity of evaluation metrics for automatic piano music transcription. *Transactions of the International Society for Music Information Retrieval*, Jun 2020. doi: 10.5334/tismir.57.

Yiming Zhang, Harshita Diddee, Susan Holm, Hanchen Liu, Xinyue Liu, Vinay Samuel, Barry Wang, and Daphne Ippolito. Noveltybench: Evaluating language models for humanlike diversity, 2025. URL https://arxiv.org/abs/2504.05228.

Shuyan Zhou, Uri Alon, Sumit Agarwal, and Graham Neubig. Codebertscore: Evaluating code generation with pretrained models of code. In *Conference on Empirical Methods in Natural Language Processing (EMNLP)*, Singapore, December 2023. URL https://arxiv.org/abs/2302.05527.

Yaoming Zhu, Sidi Lu, Lei Zheng, Jiaxian Guo, Weinan Zhang, Jun Wang, and Yong Yu. Texygen: A benchmarking platform for text generation models. In *The 41st international ACM SIGIR conference on research & development in information retrieval*, pp. 1097–1100, 2018.

Muqi Zou, Hongyu Cai, Hongwei Wu, Zion Leonahenahe Basque, Arslan Khan, Berkay Celik, Dave, Tian, Antonio Bianchi, Ruoyu, Wang, and Dongyan Xu. D-lift: Improving llm-based decompiler backend via code quality-driven fine-tuning, 2025. URL https://arxiv.org/abs/2506.10125.

A STRUDEL BACKGROUND AND RUNTIME

A.1 PROGRAM REPRESENTATIONS

A central property of the MIDI-to-Strudel task is that many distinct Strudel programs can render to the same (or nearly the same) musical content (MIDI), while exposing different degrees of musical structure. Figure 6 shows four Strudel programs that produce the same short loop of drums, bass, and electric guitar, but differ in how they represent the underlying musical ideas:

- **Event-level transliteration.** A program can stay close to the MIDI event representation by converting note events into fixed-resolution mini-notation patterns (Figure 6a). At each grid step, the code emits the corresponding drum hit, pitch, or rest. This representation can preserve precise timing and pitches, but it produces long event lists that obscure repeated musical structure.
- **Pattern-level abstraction.** The same material can be written more compactly by using pattern operations and symbolic musical units (Figure 6b). Repeated events can be compressed into shorter patterns, while explicit pitch collections can be replaced with chord symbols (`chord()`) shaped by rhythmic operators such as `struct()`. This exposes rhythm and harmony hidden in a literal event list.
- **Shared structure and layering.** A program can further factor out musical information that is shared across voices (Figure 6c). For example, a chord progression can be defined once and reused by different layers, so that each layer is expressed relative to the current chord rather than as a separate list of absolute pitches. Simultaneous voices can also be grouped with `stack()`, making the layered structure of the music explicit and reusable in the program.
- **Arrangement-level organization.** At a higher level, a program can separate reusable voice definitions from the temporal organization of the piece (Figure 6d). Individual parts can be named once and then combined over time using `arrange()`, allowing users to edit each layer independently and rearrange sections without rewriting the note content.

A.2 RUNTIME IMPLEMENTATION

The runtime $\mathcal{C}$ is a headless execution server built around the open-source Strudel implementation.[3] It provides two interfaces used throughout our pipeline. First, it executes generated Strudel programs and returns symbolic note events, which are serialized to MIDI for dataset construction, reward computation, and evaluation. Second, it parses programs into abstract syntax trees (ASTs), which we use for data cleaning and code-level analyses.

In implementation, a Node.js process loads the Strudel JavaScript engine and handles program execution and AST parsing, while a FastAPI wrapper exposes these functions to Python workers and performs MIDI serialization. Programs that fail to compile or exceed the timeout (30s) are treated as invalid and receive the compilation penalty in Equation 6 during RL.

MIDI construction. Unlike MIDI, a Strudel program does not explicitly specify a finite duration. It instead represents a time-queryable musical pattern (Roos & McLean, 2023): given a requested time span, the runtime returns the events active within that span, including their onset, pitch, and sound name. To obtain a finite MIDI rendering, $\mathcal{C}$ chooses a segment length from the program's musical content rather than from an arbitrary query window: it first queries the pattern over a sufficiently long fixed window (90 s in our experiments) and trims the resulting event sequence to one complete loop. Specifically, we use the Knuth–Morris–Pratt algorithm (Knuth et al., 1977) to identify the fundamental period of the queried event sequence, and retain only the events within the first period. The trimmed events are then serialized to a MIDI file by the FastAPI wrapper using the `mido` package (Bjørndalen et al.). Each Strudel sound is mapped to a General MIDI program for melodic instruments or to a channel-9 drum note for percussion samples using fixed lookup tables (Eskin). Note names and numeric pitches are resolved to MIDI note numbers, and Strudel `gain` and `velocity` attributes are soft-clipped to the MIDI velocity range.

[3] https://codeberg.org/uzu/strudel

```
$: s(`<[bd ~ ~ ~ ~ bd bd ~]
       [bd ~ ~ ~ ~ bd bd ~]>`)
 .bank("RolandTR808")

$: s("<[~ sd ~ sd] [~ sd ~ sd]>")
 .bank("RolandTR808")

$: note("<[c2 c2 a1 a1] [d2 d2 g2 g2]>")
 .sound("gm_synth_bass_2")

$: note(`<[- [c3,e4,g4,b4]
           - [g3,c#4,f4,g4,a4]]
         [- [f3,c4,d4,f4,c5]
           - [g3,d4,f4,g4,b4]]>`)
 .sound("gm_electric_guitar_jazz")
```

(a) **Event-level transliteration.** Musical contents are quantized into mini-notation strings that explicitly list drum hits, pitches, and rests.

```
// kick
$: s("bd ~ [~ bd] bd").bank("RolandTR808")

// snare
$: s("~ sd ~ sd").bank("RolandTR808")

// bassline
$: note("<c2*2 a1*2 d2*2 g2*2>*2")
 .s("gm_synth_bass_2")

// guitar strum
$: chord("<C^7 A7b13 Dm7 G7>*2")
 .struct("[~ x]*2").voicing()
 .s("gm_electric_guitar_jazz")
```

(b) **Pattern-level abstraction.** The bassline compacts events with pattern operators, and the guitar layer represents harmony with `chord()` and rhythm with `struct()`.

```
let chords = chord("<C^7 A7b13 Dm7 G7>*2")

stack(
 // kick
 s("bd ~ [~ bd] bd").bank("RolandTR808"),

 // snare
 s("~ sd ~ sd").bank("RolandTR808"),

 // bassline
 n("0*4").set(chords).mode("root:g2").voicing()
 .s("gm_synth_bass_2"),

 // guitar strum
 chords.struct("[~ x]*2").voicing()
 .s("gm_electric_guitar_jazz")
)
```

(c) **Shared structure and layering.** A shared `chords` variable anchors the bassline and the guitar harmony, while `stack()` groups the layers as simultaneous voices.

```
let chords = chord("<C^7 A7b13 Dm7 G7>*2")

let kick = s("bd ~ [~ bd] bd")
 .bank("RolandTR808")

let snare = s("~ sd ~ sd")
 .bank("RolandTR808")

let bass = n("0*4").set(chords).mode("root:g2")
 .voicing().s("gm_synth_bass_2")

let guitar = chords.struct("[~ x]*2")
 .voicing().s("gm_electric_guitar_jazz")

$: arrange(
 // [2, stack(kick, snare)],
 [16, stack(kick, snare, bass, guitar)]
)
```

(d) **Arrangement-level organization.** Voices are factored into named variables, and their combinations are sequenced into sections with `arrange()`.

Figure 6: **Different Strudel representations of the same musical loop.** The same musical content can be expressed by many distinct Strudel programs. Among many possible programs for the same musical content, we show four illustrative examples that render to the same short loop of drums, bass, and electric guitar. They differ in their levels of abstraction and coding idioms, ranging from literal event-level representation to pattern-, layer-, and arrangement-level abstractions.

AST parsing. For code-level analyses, $\mathcal{C}$ exposes a parsing interface that returns the AST of a Strudel program. Since Strudel is implemented as a JavaScript-based DSL, we use the Acorn JavaScript parser (Haverbeke et al.) to parse generated programs into an AST before execution. We use the resulting ASTs in two parts. During dataset construction (Section 2.3; Appendix E), the parser identifies top-level musical voices and traces the variable declarations on which each voice depends. We use this information to clean LLM-generated programs by retaining the code needed for each extracted voice while removing invalid fragments. During evaluation, we use ASTs to compute the SelfCB diversity metric (Section 3.1; Appendix F.1). Specifically, each program is parsed into a labeled tree of its syntax-node types, which CodeBLEU uses for subtree and data-flow matching between program pairs, alongside token-level n-gram matching.

B MIDI REPRESENTATION

This section details the `MidiToText` helper of Equation 1, which converts a MIDI segment into the textual representation used as input to the language model. An example is shown below:

```
[BPM=120 Meter=4]
[acoustic_grand_piano] C4@0.00 E4@0.25 G4@0.50 C5@1.00
[acoustic_bass] C2@0.00 G2@0.50 C2@1.00
[drum] bass_drum@0.00 closed_hi_hat@0.25 closed_hi_hat@0.50 snare@0.75
```

Event serialization. We serialize MIDI as symbolic, instrument-wise note events, making the input closer to musical notation and to the layered structure of Strudel programs. Specifically, we first parse the input MIDI into a flat list of note-on events, where each event is represented by its onset, instrument, and pitch. Events are then grouped by instrument and serialized as a sequence of `pitch@onset` tokens sorted by onset. For melodic instruments, pitches are rendered as note names (e.g., `C4`, `F#3`); for drums, pitches are rendered as drum names (e.g., `closed_hi_hat`). This representation choice yields tokens closer to common textual music notation and idiomatic Strudel code than raw MIDI indices, while also keeping each musical part contiguous and exposing the segment's layered structure without requiring additional notation.

Cycle-based onsets. Strudel represents time in its native unit called *cycles* rather than seconds: here, tempo specifies how many cycles occur per second or minute, and events are placed at positions within these cycles.[4] For our MIDI representation, we adopt the convention that one cycle corresponds to one measure of the input segment. Under this convention, onset positions become normalized coordinates within the measure: for example, in $4/4$, notes on the four beats occur at `0.00`, `0.25`, `0.50`, and `0.75`, instead of at absolute timestamps in seconds.

To convert MIDI timestamps to cycle positions, we use the tempo and meter of the segment, where BPM denotes beats per minute and meter denotes the number of beats per measure. Under our one-cycle-per-measure convention, one cycle spans $T_{\text{cycle}} = (60/\text{BPM}) \times \text{meter}$ seconds. We then convert each MIDI onset time t_{sec} to its cycle coordinate onset t_{cycle} by

$$t_{\text{onset}} = \frac{t_{\text{sec}}}{T_{\text{cycle}}} = \frac{t_{\text{sec}}}{(60/\text{BPM}) \times \text{meter}}. \tag{7}$$

In the text representation, we prepend a tempo–meter header (e.g., `[BPM=120 Meter=4]`) and express each onset using this cycle coordinate. When BPM and meter metadata are available in the MIDI file, we use them directly. Otherwise, we render the MIDI to audio with FluidSynth (FluidSynth Developers) and use `madmom` (Böck et al., 2016) to estimate beat and downbeat locations, from which we derive BPM and meter.

Decompilation prompt. The serialized MIDI representation is inserted into a chat prompt for MIDI-to-Strudel decompilation. The system instruction and user message are shown below.

DECOMPOSER System Instruction

You are an expert live coder in Strudel, a JavaScript-based platform that ports the TidalCycles pattern language to the web. Your task is to decompile MIDI events into idiomatic Strudel code.

STRUDEL QUICK REFERENCE

Basic syntax

- Chain functions with dot notation, e.g., `s("bd").gain(0.8).fast(2)`.
- Use double quotes for mini-notation strings.
- Use single quotes for ordinary JavaScript strings.
- Use backticks for multi-line mini-notation strings.

[4] https://strudel.cc/understand/cycles

Core functions

Tempo	`setcpm(bpm/bpc)`, e.g., `setcpm(120/4)`
Sound	`s()`, `bank()`, `n()`, `begin()`, `end()`, `loop()`
Pitch	`note()`, `scale()`, `chord()`, `voicing()`, `rootNotes()`
Timing	`fast()`, `slow()`, `off()`, `stack()`, `rev()`
Dynamics	`gain()`, `velocity()`, `attack()`, `delay()`, `sustain()`, `release()`, `postgain()`

Core mini-notation

`"bd*4"`	Repeat four times per cycle
`"<bd sd>"`	Alternate across cycles
`"[bd sd]"`	Sequence within one cycle
`"bd, hh"`	Layer simultaneously
`"~"`	Silence or rest
`"x(5,16)"`	Euclidean rhythm with five hits over sixteen steps

INPUT FORMAT

Header

The first line gives the tempo and meter of the source MIDI: `[BPM=120 Meter=4]`.
Use this information to set the tempo at the top of the program: `setcpm(120/4)`.
If the header is missing, use `setcpm(120/4)`.

Events

Each following line contains one instrument part: `[instrument_name] pitch@cycle_onset pitch@cycle_onset ...` Here, `instrument_name` is a General MIDI program name or `drum`; `pitch` is either a note name, such as `C4` or `F#3`, or a drum name, such as `bass_drum` or `closed_hi_hat`; and `cycle_onset` is the onset position in cycle units. For example, in $4/4$, `0.00`, `0.25`, `0.50`, and `0.75` correspond to the four beats of one measure.

Example

```
[BPM=120 Meter=4]
[acoustic_grand_piano] C4@0.00 E4@0.25 G4@0.50 C5@1.00
[drum] bass_drum@0.00 closed_hi_hat@0.25 closed_hi_hat@0.50 snare@0.75
```

TRANSLATION GUIDELINES

1. Structure

- Group instruments into separate musical layers, usually inside `stack()`.
- Identify repeated bars or motifs and express them with concise mini-notation.
- MIDI note durations are not explicitly encoded; infer reasonable durations from onset gaps and musical context.

2. Timing

- Map cycle onsets directly to rhythmic subdivisions.
- Quantize to the smallest common subdivision that preserves the rhythm.
- Use `~` for rests, `*N` for repeated hits, `[...]` for subdivisions, and `<...>` for alternation.

3. Pitch

- For melodic instruments, use `note()` with note names or `n().scale()` when a scale is clear.
- For drums, use `s()` with sound banks.
- For same-instrument notes that occur at the same onset, prefer `chord()` with `voicing()` when the harmony is recognizable; otherwise use explicit grouped notes.
- Do not merge simultaneous notes from different instruments into a single chord layer.

4. Musical structure

- Prefer pattern abstraction over event-by-event transliteration.
- Use `stack()` for simultaneous parts and `arrange()` for section-level organization.
- Use `<...>` for alternating cycles and `fast()` for intra-cycle density.
- Separate bass, chords, melody, and drums when they play distinct musical roles.

STYLE GUIDELINES

- Ensure readability: voices, structure, and harmonic intent should be visible at a glance.

- Add short comments that label each major voice, such as `// walking bass` or `// backbeat drum groove`.
- Extract reused chord progressions, rhythms, motifs, or sounds into named `const` variables.
- Break long method chains across lines, with one method per line.
- Prefer `chord()` and `voicing()` over long explicit note lists when the harmony is clear.
- Prefer scale degrees with `scale()` when the line belongs to a clear key.
- Avoid raw MIDI numbers such as `n(60)`.
- Use Strudel repetition operators such as `hh*8` or `<[bar1] [bar2]>` instead of spelling out repeated events.
- Separate voice blocks with blank lines.

CRITICAL CONSTRAINTS

1. **Output format:** return only raw Strudel code. Do not use Markdown code fences, explanations, or conversational text.
2. **Sound equivalence:** the rendered program should be audibly equivalent to the input MIDI in rhythm, pitch, density, and instrumentation.
3. **Idiomatic code:** prefer clean, readable Strudel patterns over verbose transliterations.

DECOMPOSER User Message Template

Analyze the following note events and write corresponding Strudel code.

```
{midi_representation}
```

During training, we vary the natural-language request in the user message while keeping the underlying MIDI representation fixed. The request is sampled from a set of paraphrased task instructions:

1. Analyze the following MIDI events and write corresponding Strudel code.
2. Can you help me turn this MIDI into Strudel live coding syntax?
3. Convert the following MIDI events into Strudel code.
4. Encode the following MIDI events into Strudel notation.
5. Express this MIDI sequence as a Strudel program.
6. Generate Strudel code from the MIDI below.
7. Generate the Strudel code that reproduces the following MIDI events.
8. Given the following MIDI events, write the equivalent Strudel pattern.
9. Interpret this MIDI and express it as Strudel live coding syntax.
10. Reproduce the following MIDI pattern in Strudel.
11. Rewrite the following MIDI events using Strudel live coding syntax.
12. Take a look at the MIDI below and write Strudel code that reproduces it.
13. Transform this MIDI representation into Strudel.
14. Translate this MIDI sequence into Strudel code.
15. Write Strudel code that is functionally equivalent to the following MIDI.

C DESCRIPTIONS FOR OPTIMIZATION METHODS

We provide a description of the two policy-optimization methods referenced in the main paper (Section 2.1). We first explain Group Relative Policy Optimization (GRPO) in Section C.1 and Group reward-Decoupled normalization Policy Optimization (GDPO) in Section C.2. For detailed explanations and ablations, please refer to the original papers (Guo et al., 2025; Liu et al., 2026).

We consider a general verifiable generation setting in which a policy π_θ maps an input x to an output $\hat{y}$, and an external evaluator assigns rewards after generation. The evaluator may be non-differentiable, since the policy-gradient objectives below only require scalar reward values and do not differentiate through the evaluation procedure. This setting covers our training setup, where rewards are computed from the executed output and auxiliary evaluation metrics.

C.1 GROUP RELATIVE POLICY OPTIMIZATION (GRPO)

GRPO (Guo et al., 2025) is a policy-gradient method that estimates advantages relative to a *group* of completions sampled for the same input prompt, removing the separately trained value network of PPO (Schulman et al., 2017). For each input x, the old policy samples G candidates $\{\hat{y}^{(g)}\}_{g=1}^{G} \sim \pi_{\theta_{\text{old}}}(\cdot \mid x)$. Each candidate is scored with a scalar reward $r^{(g)} = r(x, \hat{y}^{(g)})$, and the rewards are normalized within the group:

$$A^{(g)} = \frac{r^{(g)} - \mu}{\sigma + \varepsilon}, \quad \mu = \frac{1}{G}\sum_{g=1}^{G} r^{(g)}, \quad \sigma = \text{std}\Big(\{r^{(g)}\}_{g=1}^{G}\Big). \tag{8}$$

The policy is then updated by maximizing a token-level clipped surrogate with a KL penalty to a reference policy π_{ref}:

$$\mathcal{J}_{\text{GRPO}}(\theta) = \mathbb{E}\left[\frac{1}{G}\sum_{g=1}^{G}\frac{1}{|\hat{y}^{(g)}|}\sum_{t=1}^{|\hat{y}^{(g)}|}\min\Big(\rho_t^{(g)}A^{(g)},\ \text{clip}\Big(\rho_t^{(g)},\,1-\epsilon,\,1+\epsilon\Big)\,A^{(g)}\Big)\right] - \beta\,\mathbb{D}_{\text{KL}}\big[\pi_\theta \,\big\|\, \pi_{\text{ref}}\big], \tag{9}$$

where t indexes positions in the generated token sequence and

$$\rho_t^{(g)} = \frac{\pi_\theta(\hat{y}_t^{(g)} \mid \hat{y}_{<t}^{(g)}, x)}{\pi_{\theta_{\text{old}}}(\hat{y}_t^{(g)} \mid \hat{y}_{<t}^{(g)}, x)} \tag{10}$$

denotes the per-token importance ratio between the current and old policies. Here, ϵ is the clipping threshold, β is the KL coefficient, and $\mathbb{D}_{\text{KL}}\big[\pi_\theta \,\big\|\, \pi_{\text{ref}}\big]$ denotes the averaged token-level KL regularization term.

C.2 GROUP REWARD-DECOUPLED NORMALIZATION POLICY OPTIMIZATION (GDPO)

GRPO assumes a scalar reward for each sampled output; when the training signal has multiple reward components, a straightforward way to apply GRPO is to first scalarize them and then apply the group-wise normalization in Equation 8. Concretely, for each sampled output $\hat{y}^{(g)}$, one may form

$$r^{(g)} = \sum_{k=1}^{K} w_k r_k^{(g)}, \tag{11}$$

where $r_k^{(g)} = r_k(x, \hat{y}^{(g)})$ is the k-th reward component and w_k is its corresponding weight. Because normalization is applied only after scalarization, high-variance components can dominate both the scalar reward and the resulting advantage, even when the intended objective assigns non-negligible weight to multiple components.

GDPO (Liu et al., 2026) instead normalizes each reward component within the group before aggregation. For each component k, GDPO computes a component-wise group-relative advantage

$$A_k^{(g)} = \frac{r_k^{(g)} - \mu_k}{\sigma_k + \varepsilon}, \quad \mu_k = \frac{1}{G}\sum_{g=1}^{G} r_k^{(g)}, \quad \sigma_k = \text{std}\big(\{r_k^{(g)}\}\big), \tag{12}$$

aggregates the normalized advantages with the weights w_k:

$$A_{\text{sum}}^{(g)} = \sum_{k=1}^{K} w_k A_k^{(g)}, \tag{13}$$

and finally applies a batch-wise normalization across all rollouts in the batch $\mathcal{B}$ to keep the advantage magnitude from growing with the number of objectives:

$$\hat{A}_{\text{sum}}^{(g)} = \frac{A_{\text{sum}}^{(g)} - \text{mean}\big\{A_{\text{sum}}^{(g')} \,\big|\, g' \in \mathcal{B}\big\}}{\text{std}\big\{A_{\text{sum}}^{(g')} \,\big|\, g' \in \mathcal{B}\big\} + \varepsilon}. \tag{14}$$

The policy is updated with the clipped surrogate of Equation 9, with $A^{(g)}$ replaced by $\hat{A}_{\text{sum}}^{(g)}$.

D READABILITY REWARD DESIGN

The readability reward R_{read} (Section 2.2) is computed by an LLM judge using a fixed binary checklist of $J = 12$ items. We use rule-based binary criteria to obtain a stable reward signal in a domain where many programs can be valid decompilations, following the rubric-as-rewards paradigm (Gunjal et al., 2026; Viswanathan et al., 2025; Gandhi et al., 2026). The checklist combines general code readability criteria, adapted from prior readability metrics (Buse & Weimer, 2010; Scalabrino et al., 2018), with Strudel-specific criteria for concision and musical abstraction. Specifically, items 1–5 capture general readability, such as variable reuse, comments, and formatting, while items 6–12 reward the use of Strudel's pattern operators and musical abstractions over note-by-note transliteration. The score is the fraction of satisfied items: $R_{\text{read}} = \frac{1}{12}\sum_j c_j \in [0, 1]$.

We use `Qwen3.6-35B-A3B` (Qwen Team, 2026) as the judge. We follow the Qwen recommended temperature setting of 0.7, set the maximum output length to 4000 tokens, and disable thinking (`enable_thinking=false`). The full judge system instruction, including the rubric, is shown below.

Readability Judge System Instruction

You are an expert Strudel code reviewer. Strudel is a JavaScript-based live coding language for music. Given a Strudel program, your task is to evaluate its readability using the 12 binary checklist items below. Apply each criterion literally and independently. Return only valid JSON in the following format:

```
{"items":[{"id":"1","reasoning":"...","score":0},...,{"id":"12",...}]}
```

Include all 12 items in order. Each score must be exactly 0 or 1. Keep each reasoning to one short sentence.

Item 1: Variable usage. Score 1 iff the code declares at least one top-level `const X = ...` or `let X = ...` and reuses `X` elsewhere.

Item 2: Comment usage. A meaningful comment is a `//` or `/* */` comment with at least two words (e.g., , `// syncopated drum groove`). Decoration-only comments do not count. Score 1 iff meaningful_comments/ $\max$(code_lines, 1) ≥ 0.10.

Item 3: Repeated method chains. Score 0 iff the code repeats the same contiguous sequence of ≥ 4 chained dot-method calls at least twice. The repeated sequence must match exactly in both method names and arguments. Otherwise score 1. Do not count expression heads such as `note(...)`, `stack(...)`, or `chord(...)`.

Item 4: Line length. Over non-empty lines, score 1 iff the average line length is at least 10 characters and the maximum line length is at most 90 characters.

Item 5: Blank line usage. Score 1 iff blank lines make up at least 10% of all code lines, i.e., blank_lines/ $\max$(code_lines, 1) ≥ 0.10.

Item 6: Mini-notation compression. Inspect each mini-notation string passed to `note`, `n`, `sound`, or `s`. Score 0 iff it contains any avoidable repetition: the same token/group appears four or more times in a row, a phrase repeats in a row, or a `<...>` alternation repeats the same element. Ignore repetitions already expressed with `*N` or `!N`. Otherwise score 1.

Item 7: Time transformation. Score 1 iff the code uses at least one timing-change methods: `.fast`, `.slow`, `.ply`, `.iter`, `.early`, `.late`.

Item 8: Rhythm structure. Score 1 iff the code uses at least one of rhythm-structure method: `.struct`, `.mask`, `.euclid`, `.euclidLegato`, `.euclidRot`, `.swing`, `.swingBy`.

Item 9: Pattern layering. Score 1 iff the code uses at least one of pattern-layering method: `.jux`, `.layer`, `.superimpose`.

Item 10: Scale usage. Score 1 iff the code uses chained `.scale(...)` to map scale degrees to pitches (e.g., `n("0 2 4").scale("c:major")`).

Item 11: Chord voicing. Score 1 iff the code uses `chord(...)` with a voicing-related method in the same expression chain: `.voicing(...)` or `.rootNotes(...)`.

Item 12: Pitch shift. Score 1 iff the code uses at least one pitch-shift operation: `.transpose(...)`, `.add(note(...))`, `.sub(note(...))`, or `.add/.sub` with an integer semitone count. Small-float `.add(<1)` for detuning and non-pitch strings do not count.

E STRUDEL-SYNTH

This section provides additional details on the construction of STRUDEL-SYNTH, the synthetic paired corpus used to train DECOMPOSER (Section 2.3). We describe the overall generation pipeline in Section E.1 and the conditioning inputs used to control musical content and coding style of the generated Strudel programs in Section E.2.

E.1 GENERATION PIPELINE DETAILS

We detail each stage of the data generation pipeline used to construct STRUDEL-SYNTH.

Stage 1: Conditioned generation. We prompt Claude-Opus-4.6 (Anthropic, 2026) with a system prompt instructing it to write a complete, idiomatic Strudel program, conditioned on a musical seed s_{music} and a code-style seed s_{code} supplied in the user message (detailed in Appendix E.2). To ensure deterministic MIDI rendering, we forbid random modifiers and constructs unsupported by the headless engine, including visuals, sliders, and external samples. We sample with temperature 1.0 and up to 4,096 new tokens, with extended thinking disabled. The system prompt is shown below.

STRUDEL-SYNTH Generation System Instruction

You are an expert live coder specializing in Strudel, a JavaScript-based platform that ports the TidalCycles pattern language to the web. Your task is to translate natural language music descriptions into executable, idiomatic Strudel code.

STRUDEL QUICK REFERENCE

Basic syntax

- Chain functions with dot notation.
- Use double quotes (`"`) for single-line mini-notation patterns.
- Use backticks (`` ` ``) for multi-line mini-notation strings.
- Use single quotes (`'`) for regular strings that are not parsed as mini-notation.

Core mini-notation

`"bd*4"`	Repeat four times per cycle
`"<bd sd>"`	Alternate across cycles
`"[bd sd]"`	Sequence within one cycle
`"bd, hh"`	Layer simultaneously
`"~"`	Silence or rest
`"bd(3,8)"`	Euclidean rhythm shorthand
`"c@3 e"`	elongate a note with `@`

Core functions

Tempo	`setcpm(bpm/bpc)`, e.g., `setcpm(120/4)`
Sound	`s()`, `bank()`, `n()`, `begin()`, `end()`, `loop()`
Pitch	`note()`, `scale()`, `chord()`, `voicing()`, `rootNotes()`
Timing	`fast()`, `slow()`, `off()`, `stack()`, `arrange()`, `rev()`, `jux()`
Rhythm	`struct()`, `mask()`, `ply()`, `euclid()`, `swingBy()`
Dynamics	`gain()`, `velocity()`, `attack()`, `sustain()`, `release()`, `adsr()`, `postgain()`
Modulation	`fm()`, `fmh()`, `vib()`
Filters	`lpf()`, `hpf()`, `bpf()`, `bpq()`
Effects	`room()`, `delay()`, `chorus()`, `compressor()`

General advice

- Use `$:` labeled statements or `stack()` to layer simultaneous parts.
- Use `arrange()` for song-level section structure.
- Use `voicing()` for chord progressions and `rootNotes(n)` for bass lines from chord symbols.
- Use `struct()` for rhythmic gating; use `off()` or `superimpose()` for harmonic echoes.
- Use Euclidean rhythms `(n,k)` and `swingBy()` for natural-feeling grooves.
- Use synths such as `sawtooth`, `square`, `sine`, and `triangle`, or GM soundfonts such as `gm_acoustic_bass` and `gm_epiano_1`.

INPUT FORMAT: MUSIC DESCRIPTION

The input is a natural-language music description specifying any combination of genre, mood, instrumentation, tempo, key, time signature, or structural ideas. Descriptions range from terse prompts such as "chill lo-fi beat" to detailed requests. Treat missing details as creative latitude.

GENERATION GUIDELINES

1. Interpret the description

- Infer genre conventions: tempo, instruments, rhythmic feel, and harmonic language.
- Plan the main sound layers from the description, such as melody, drums, bass, and pads.

2. Build harmony

- Express chord progressions symbolically, e.g., `chord("<Am7 Fmaj7 G C>").voicing()`.
- Control register with anchor (e.g., `.anchor("c4")`) and mode (e.g., `.mode("below")`, `.mode("duck")`, or `.mode("above")`).
- Derive basslines using chord symbols, e.g., `chord(chords).rootNotes(2).s("gm_acoustic_bass")`.
- Use interval stacks such as `note("<[c3,e3,g3] [a2,c3,e3]>")` for explicit voicings.

3. Build rhythm and structure

- Use `stack()` or `$:` for simultaneous parts, and `arrange([n, section], ...)` for song sections.
- Use Euclidean rhythms for organic feels, e.g., `s("bd(3,8)")` or `s("hh(5,8)")`.
- Add swing with `.swingBy(1/8, 4)` or `.swing(4)`.
- Add variation with pattern effects, e.g., `.add("<0 1 2 1>")`, or `.off(1/8, add(7))`.

4. Apply style and effects

- Match effects to genre, such as reverb for ambient or jazz, compression for electronic music, and delay for dub.
- Keep patterns concise by using mini-notation repetition and `<...>` alternation instead of spelling out every bar.
- Use `layer()` or `superimpose()` to add harmonic richness without writing extra voices manually.

FORBIDDEN CONSTRUCTS

The following constructs must not be used, even if they appear in a few-shot example.

- **Visuals:** `.color()`, `._scope()`, `._pianoroll()`, `._waveform()`, `Hydra`, `initHydra`, `src()`, `.out()`
- **Randomness:** `rand`, `irand`, `perlin`, `Math.random()`, `choose`, `wchoose`, `chooseCycles`, `randcat`, `wchooseCycles`, `wrandcat`
- **Probabilistic modifiers:** `degradeBy`, `degrade`, `undegradeBy`, `undegrade`, `sometimesBy`, `sometimes`, `someCyclesBy`, `someCycles`, `often`, `rarely`, `almostNever`, `almostAlways`, `never`, `always`, `"bd?"`, `"bd | sd"`
- **Interactivity:** `slider()`, `button()`, `xy()`, `midin()`, `cc()`, `initMidi()`, `await midin()`
- **External samples:** `loadSamples()`, `samples('url')`, and any explicit `.wav` or `.mp3` path string

CRITICAL CONSTRAINTS

1. **Output format:** return only raw Strudel code. Do not use Markdown code fences, conversational text, or explanations. The output must be directly copy-pasteable into the Strudel REPL.
2. **Fidelity:** closely follow the musical intent of the description.
3. **Idiomatic code:** prefer clean, readable Strudel patterns with mini-notation abstractions rather than verbose literal sequences.
4. **Tempo:** always use `setcpm(bpm/bpc)`, e.g., `setcpm(120/4)` or `setcpm(90/2)`.

Table 4: **Deterministic regex rewrites.** Regex-based fixes applied to LLM-generated Strudel programs during data cleaning. Each rule targets a recurring syntactic mistake observed in initial generations generated by Claude-Opus-4.6; together, they correct $\sim$37% of generated programs.

Rewrite	Description
Invalid sound aliases	
`gm_electric_piano_1` → `gm_epiano1`	Correct the invalid e-piano alias to the valid Strudel sound name.
`\bshaker\b(?!_)` → `sh`	Replace the invalid bare `shaker` alias with `sh`; keep `shaker_small` and `shaker_large` as-is.
Voicing misuse	
`\.voicings\(".*?"\)` → `.voicing()`	Collapse `.voicings(arg)` calls to `.voicing()`.
`\.note\(("[\w$]+)\.voicing\(\)\)` → `.chord($1).voicing()`	Repair `note(X.voicing())` by rewriting it as a `chord().voicing()` call.
`\b(?:note|n)\s*\((.*?)\)(?=\s*\.voicing\(\))` → `chord($1)`	Rewrite voiced `note()` / `n()` heads as `chord()`.
Erroneous note calls	
`\.note\(x => [^)]*\)` → ∅	Remove lambda-style `.note()` calls.
`\.(rootNotes|voicing|scale)\([^)]*\).note()` → remove `.note()`	Remove empty `.note()` calls after tonal operators.
`stack(...).note()` → `stack(...)`	Remove a `.note()` appended to the `stack(...)` call.
Broken chain structure	
`rootNotes` restructure	Reorder `s(sample).note(expr.rootNotes(N))` to `expr.s(sample)`, correcting broken bass-line chains.
Length and trigger guards	
`arrange()` section lengths	Cap each `arrange()` section at $\leq$ 2 cycles and the total arrangement at $\leq$ 12 cycles to prevent overly long outputs.
`.struct("0 0")` → `.struct("1 0")`	When a `.struct(...)` pattern contains no trigger character (`1`, `t`, or `x`), replace the first `0` with `1`.

Stage 2: Cleaning and dead-voice pruning. LLM-written programs frequently contain *dead voices*: music layers that compile successfully but emit no audible events. These typically arise from hallucinated instrument names or malformed operator chains. For example, if a voice in the generated program references a sound name that does not exist in the Strudel sound library, such as `$: note("c# e g a").sound("sound_404")`, the program may still compile during rendering (Stage 3), but that voice would not produce any audible events. Such dead voices are problematic for training: the code suggests an active musical layer, while the rendered MIDI contains nothing, creating a mismatch between the program and the music that the model would otherwise learn as a valid pattern. We address this in two passes: regex rewriting and voice-level pruning.

We first apply a fixed sequence of deterministic regular expression rewrites that correct recurring syntactic mistakes we observed in early generations. The rewrites used in our pipeline are summarized in Table 4; together, they correct $\sim$37% of programs at least once before any compilation is attempted[5].

We then parse each rewritten program with Acorn JavaScript parser (Haverbeke et al.) and use the resulting AST to identify top-level musical voices, following the representative Strudel structures introduced earlier in Appendix A.1: mini-notation statements (`$:`), `stack` arguments, and `arrange` tracks (Figures 6b-6d). For each voice, we record its character span in the original source and collect the top-level variable declarations (e.g., `let chords = chord("<Bbm9 Fm9>/4")`) and helper definitions (e.g., `samples('github:eddyflux/crate')`) it depends on. We render this isolated program with the runtime $\mathcal{C}$ using a 60 s window, and remove the voice if it produces zero audible events. Finally, we remove top-level declarations that are no longer referenced and retain a program only if at least one voice survives. This full parsing-and-pruning procedure is illustrated in Figure 7.

[5]These rewrites are heuristic and target the most common failure modes observed. They are not guaranteed to be sound and may occasionally modify already-valid programs; cleaned outputs are re-validated afterward.

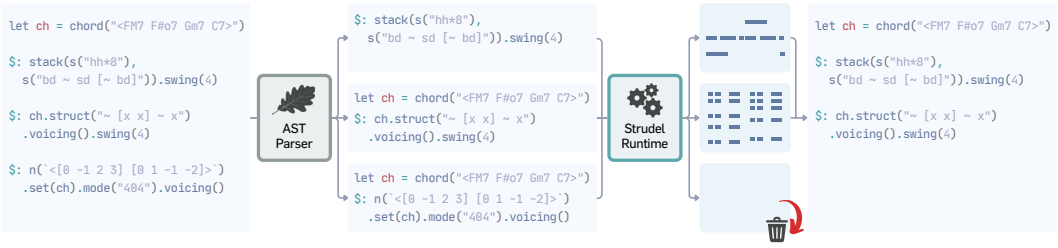


Figure 7: **Voice-level cleaning pipeline.** After regex-level cleaning of LLM-generated programs, we use an AST parser to extract each top-level musical voice along with the declarations and configuration calls it depends on. Each voice is rendered in isolation and removed if it produces no note-on events. We then remove unused declarations and keep programs with at least one surviving voice.

Stage 3: Rendering. Each cleaned program is rendered to MIDI with the runtime $\mathcal{C}$, which queries the pattern over a fixed window (90 s) and trims the result to one fundamental loop (Appendix A.2). We discard renderings shorter than 2 s, which are typically degenerate programs that produce a very short event sequence. The surviving (MIDI, Strudel) pairs constitute STRUDEL-SYNTH; summary statistics are reported in Table 1.

E.2 CONDITIONING INPUTS

We condition each generation on two inputs: a musical seed s_{music} that specifies what kind of piece to write, and a code-style seed s_{code} that specifies how the piece should be expressed as a Strudel program. These two seeds vary independently so that the same musical target can be realized in different program idioms and a given idiom can be applied to diverse music.

Musical seed s_{music}. The musical seed specifies the intended musical content of each generated piece and is injected into the user message of the generation prompt. We construct these seeds from three metadata sources: (i) EveryNoise (McDonald, 2013), a fine-grained vocabulary of genres from Spotify listening data; (ii) TheoryTab (DB, 2026), a collection of popular song analyses; and (iii) MidiCaps (Melechovsky et al., 2024), a captioned MIDI dataset built on LMD (Raffel, 2016) that combines MIR-extracted metadata with LLM-generated natural-language descriptions. To control how much musical information is given to the LLM generator, we use prompts at several levels of musical detail, from genre-only prompts to richer prompts that include multiple musical attributes. Table 5 summarizes these conditioning modes and their exposed attributes.

Table 5: **Musical conditioning seeds.** The conditioning modes range from genre-only prompts to richer multi-attribute prompts. Text color indicates the source of each exposed musical attribute: EveryNoise, TheoryTab, and MidiCaps.

# Attrs	Source	Music Attributes
1	EveryNoise	`genre`
5	EveryNoise TheoryTab	`genre`, `key`, `tempo (BPM)`, `beats per bar`, `chord progression`
7	MidiCaps	`genre`, `key`, `mood`, `tempo (BPM)`, `beats per bar`, `instrumentation`, `chord progression`

Code-style seed s_{code}. The code-style seed selects among program idioms that realize similar musical targets with different code structure: (i) labeled mini-notation patterns using `$:`; (ii) layered programs built with `stack`; and (iii) multi-section arrangements using `arrange` (see Figures 6b–6d for examples). To further ground STRUDEL-SYNTH in real Strudel practice, we additionally draw structural templates from web-crawled programs and provide them as references for a randomly selected subset of generations. Specifically, using the AST-based procedure in Appendix E.1, we parse each program into top-level musical voices, sample one or two voices, and include the resulting fragments as a reference for Strudel coding patterns. These templates are used only to guide code structure and operator use; the model is still instructed to compose new musical content. Each code-style seed is implemented with a style-specific user prompt with an optional reference-template block for template-conditioned generations; we list the prompts below.

STRUDEL-SYNTH Generation User Prompt: (1) `$:` label syntax

STRUCTURE
Your output's top-level structure must use labeled Strudel statements of the form `$: pattern`, with one statement per musical layer.

- Use 3–5 top-level `$:` statements, corresponding to musical layers such as drums, bass, chords, melody, or pads.
- Each layer should be written primarily with concise mini-notation patterns.
- Do not use a top-level `stack` or `arrange` call; keep layers as separate `$:` statements.
- Use idiomatic pattern operations such as `struct()`, `fast()`, `slow()`, `off()`, `ply()`, `swingBy()`, `chord(...).voicing()`, and `n(...).scale(...)` when useful.
- Keep the program readable by reusing variables for shared chord progressions, scales, rhythms, or sound choices.

{*% Optional block: included only when a reference program is provided.%*}

REFERENCE TEMPLATE
Use the following reference only as a guide for coding style and program organization. Write entirely new musical content from scratch, including your own choice of key, harmony, rhythm, and instrumentation.

```
{reference_code}
```

The reference may contain forbidden constructs, such as randomness, visuals, interactivity, or external samples. Do not reproduce those constructs; use deterministic and headless-compatible alternatives instead.

{*% End optional block. %*}

Create music in Strudel live coding language using the following specifications:

```
{music_attributes}
```

STRUDEL-SYNTH Generation User Prompt: (2) `stack`

STRUCTURE
Your output's top-level structure must be a single `stack(...)` call containing 3–5 inner streams.

- Each stream should loop within 4–8 cycles.
- Inside each stream, you may use any idiomatic Strudel pattern freely.
- Do not use top-level `$:` statements; wrap all streams inside a single `stack(...)` instead.
- Use clear variable names and add comments when they improve readability.

{*% Optional block: included only when a reference program is provided.%*}

REFERENCE TEMPLATE
Use the following reference only as a guide for coding style and program organization. Write entirely new musical content from scratch, including your own choice of key, harmony, rhythm, and instrumentation.

```
{reference_code}
```

The reference may contain forbidden constructs, such as randomness, visuals, interactivity, or external samples. Do not reproduce those constructs; use deterministic and headless-compatible alternatives instead.

{*% End optional block. %*}

Create music in Strudel live coding language using the following specifications:

```
{music_attributes}
```

STRUDEL-SYNTH Generation User Prompt: (3) `arrange`

STRUCTURE

Your output's top-level structure must be an `arrange([cycles, section_expr], ...)` call.

- Use 2–4 sections.
- Each section should last 2–3 cycles, with a total length of at most 12 cycles.
- Each `section_expr` could be any idiomatic Strudel expression, including `stack(...)`, mini-notation, scale-degree melodies, symbolic chord progressions, drum banks, polyrhythms, and effects.
- Use clear variable names and add comments when they improve readability.

{% Optional block: included only when a reference program is provided.%}

REFERENCE TEMPLATE

Use the following reference only as a guide for coding style and program organization. Write entirely new musical content from scratch, including your own choice of key, harmony, rhythm, and instrumentation.

`{reference_code}`

The reference may contain forbidden constructs, such as randomness, visuals, interactivity, or external samples. Do not reproduce those constructs; use deterministic and headless-compatible alternatives instead.

{% End optional block. %}

Create music in Strudel live coding language using the following specifications:

`{music_attributes}`

F Evaluation Details

F.1 Metric Definitions

We organize evaluation metrics along the three axes of faithfulness, readability, and diversity introduced in Section 3.1. The main result table (Table 2) reports the metric mean over the evaluation set under a single generation per input. For the generalization datasets (Section 3.4, Table 3) and additional evaluations (Appendix H), we instead report Best@K ($K = 5$): each method generates five candidate programs per input, and we select the candidate with the highest Inst F1. All remaining metrics are computed on the candidates selected by Inst F1.

Faithfulness. We measure how accurately the rendered MIDI $\hat{x} = \mathcal{C}(\hat{y})$ reproduces the input MIDI x, adapting metrics from the automatic music transcription literature (Gardner et al., 2022; Maman & Bermano, 2022; Chang et al., 2024; Shin et al., 2026) together with executability check from decompilation literature (Tan et al., 2024; Wong et al., 2023; Armengol-Estapé et al., 2024).

- **Comp. (compile rate)** is the fraction of generated programs that the runtime $\mathcal{C}$ successfully executes to a non-empty event list within the 30 s timeout. Programs that error out, time out, or emit zero events are counted as failures.
- **Onset F1** treats each note as a (pitch, onset) tuple and counts a predicted note as correct if it has the same pitch as a reference note whose onset lies within ± 50 ms of the prediction. We use the standard `mir_eval` implementation (Raffel et al., 2014) to perform optimal bipartite matching between reference and predicted notes, and compute per-piece precision, recall, and F1 from the resulting match counts. We report Onset F1 as the primary metric, following evidence that onset correctness is the principal determinant of perceptual transcription accuracy (Ycart et al., 2020).
- **Frame F1** measures pianoroll-level overlap of sustained notes. A pianoroll is a $[T \times 128]$ binary matrix where the entry at frame t and pitch p is 1 iff at least one note of pitch p is sounding during the 10 ms span covered by frame t (i.e., we use a frame rate of 100 frames per second); each note's velocity is binarized so that any non-zero velocity counts as active. Frame F1 is the element-wise binary F1 between the reference and predicted pianorolls, computed using `precision_recall_fscore_support` from `scikit-learn` (Pedregosa et al., 2011).

- **Inst F1 (multi-instrument onset F1)** is the same as Onset F1 but with notes partitioned by their assigned General MIDI program number, treating each program as a distinct instrument and the channel-9 percussion track as a separate drum partition. Each partition is scored independently with the ± 50 ms onset criterion, and the per-partition results are micro-averaged (i.e., each partition's precision and recall are weighted by its predicted and reference note counts respectively before the F1 is taken). Inst F1 is also the faithfulness reward R_{faith} used during RL (Section 2.2).

Readability. We report two complementary readability measures, both by LLM judges: an absolute per-program score (Rubric) and a relative cross-method score (Rank).

- **Rubric** is the 12-item factual checklist score (Section 2.2; Appendix D) in $[0, 1]$, computed with the same `Qwen3.6-35B-A3B` (Qwen Team, 2026) judge used during RL training. Each program is scored independently; we report the mean rubric across the evaluation set.
- **Rank** is the average rank from a listwise LLM reranker (Tang et al., 2024), for which we use Claude-Opus-4.7. For each input, the (compiling) outputs of all methods are shown together to an LLM judge that orders them from most to least readable; to control position and label bias we average over five random permutations of the candidate order and labels and aggregate the per-permutation orderings into a consensus rank. Lower average rank is better.

Diversity. In creative settings, useful AI suggestions are often expected to surface multiple plausible directions rather than returning near-identical outputs (Resnick et al., 2005; Louie et al., 2020; Kim et al., 2025). For symbolic music decompilation, this means that a model should not collapse to a single programming template: the same musical input can often be expressed through different Strudel idioms, such as chained patterns, shared variables, layering operators, or section-level structure. We therefore evaluate diversity as a measure of whether a method explores different coding idioms across generations.

Among different similarity measurement method (Zhou et al., 2023; Papineni et al., 2002; Ren et al., 2020) we adopt an AST-based metric for programming languages. Pure n-gram measures (Papineni et al., 2002) can be dominated by mini-notation pitches and rhythms, which obscures idiom-level diversity, whereas embedding-based code-similarity metrics (Zhou et al., 2023) are trained primarily on high-resource, mainstream languages and are thus less reliable for a low-resource DSL such as Strudel. We therefore use the AST-based metric of Ren et al. (2020) for diversity computation, detailed below.

- **SelfCB** measures intra-method self-similarity using CodeBLEU (Ren et al., 2020). For a method with programs $\{p_1, \ldots, p_N\}$ over the shared input set, CodeBLEU$(\cdot, \cdot) \in [0, 1]$ scores code similarity as a weighted combination of standard BLEU, weighted n-gram matching, syntactic AST matching, and semantic data-flow matching. We weight the four terms by $(\alpha, \beta, \gamma, \delta) = (0.1, 0.1, 0.4, 0.4)$ which Ren et al. (2020) find correlate most strongly with human judgments. We parse programs with the Acorn JavaScript AST parser (Haverbeke et al.). As CodeBLEU is asymmetric, we symmetrize each unordered pair and average over all such pairs:

$$\text{SelfCB} = \binom{N}{2}^{-1} \sum_{i<j} \tfrac{1}{2}(\text{CodeBLEU}(p_i, p_j) + \text{CodeBLEU}(p_j, p_i)) \,. \tag{15}$$

  Analogous to Self-BLEU (Zhu et al., 2018), a lower SelfCB indicates more diverse generations. To compare methods fairly, we restrict the pair set to the inputs on which all methods compile successfully (and as such exclude vanilla Qwen models which has less than 20% compile rate to ensure sample size).

F.2 Dataset

This section describes the MIDI datasets and sampling method used in our evaluation. Our experiments cover two settings: (i) the main evaluation (Section 3.2) and ablation studies (Section 3.3) use Strudel-Synth and short LMD fragments, matching the distributions used during post-training; and (ii) the generalization evaluation (Section 3.4) uses additional held-out corpora that introduce distribution shifts in duration, source, and instrumentation.

Main and ablation studies. The main experiments (Section 3.2) and ablation studies (Section 3.3) use two corpora: STRUDEL-SYNTH and LMD. STRUDEL-SYNTH is our synthetic paired corpus of executable Strudel programs and rendered MIDI outputs, as described in Section 2.3. LMD (Raffel, 2016) is a large-scale real-world MIDI corpus of multi-instrument, multi-genre music, including arrangements of modern pop songs and transcriptions of Western classical compositions. For LMD, we sample short fragments shorter than 30 s for evaluation, yielding approximately 9K training segments and 1K test segments. Together, STRUDEL-SYNTH and short LMD fragments define the in-domain distribution against which we evaluate the effects of post-training.

Generalization evaluation. To evaluate whether DECOMPOSER transfers beyond the training distributions, we additionally evaluate on four held-out settings. For each setting, we sample 200 evaluation inputs and report Best@5 results: each method generates five candidate programs per input, and we select the candidate with the highest Inst F1. All remaining metrics are computed on the candidates selected by Inst F1.

- **LMD** (30–60 s; Raffel 2016) consists of longer LMD excerpts and tests generalization to inputs longer than those used during RL post-training and main evaluation.
- **GigaMIDI** (Lee et al., 2025) is a large-scale multi-instrument, multi-genre symbolic music corpus that consolidates several major MIDI resources. It provides a source-level generalization setting over real-world MIDI files drawn from a broader collection than LMD; because GigaMIDI includes LMD among its sources, we exclude LMD-derived files when sampling evaluation inputs. We use short fragments shorter than 30 s.
- **NES-MDB** (Donahue et al., 2018) is a multi-instrument symbolic music corpus of approximately 5K chiptune pieces from 397 Nintendo Entertainment System games. Each song is represented with four NES instrument voices, corresponding to two pulse channels, one triangle channel, and one noise channel, with annotations for expressive dynamics and timbre attributes. This setting tests generalization to chiptune music with a source and instrumentation distribution that differs substantially from STRUDEL-SYNTH and LMD. We use short fragments shorter than 30 s.
- **Nottingham** (Foxley, 2003) is a collection of over 1K British and American traditional folk tunes, originally represented in ABC notation as lead sheets with monophonic melody and chord accompaniment. We use the available MIDI versions of the original ABC files. This setting tests generalization to homophonic lead-sheet music, where melodic and chordal roles may be encoded within a compact, single-instrument representation rather than separated into distinct instrumental tracks. Since the pieces are generally longer than the short-fragment setting used for post-training, we crop each piece to a 16 s segment, matching the mean duration of the short LMD fragments used during RL post-training.

F.3 BASELINES

- **Heuristic converter** (Jong, 2025) transliterates MIDI directly into Strudel mini-notation. Starting from the open-source converter, which quantizes each note onset to a fixed grid and emits the nearest Strudel sound, we extend it for our setting with: (i) *drum support*, mapping General MIDI channel-9 percussion to Strudel drum tokens (e.g., `bd`, `sd`, `hh`); (ii) *multiple-meter support*, reading time-signature changes to compute the beats-per-cycle of each segment; and (iii) broader *heuristic instrument coverage*, mapping MIDI programs to Strudel sound names through an expanded lookup table (with a piano fallback for unmatched melodic programs and a synth-drum fallback for percussion). This provides a strong reference point for low-level faithfulness but a weak baseline for readable program structure.
- **Frontier LLMs.** We compare against three closed-weight general-purpose models: Claude-Opus-4.6 (Anthropic, 2026), Gemini-3.5-Flash (Google DeepMind, 2026), and GPT-5.5 (OpenAI, 2026). These models represent strong general-purpose code-generation systems but are not specialized for Strudel or for symbolic music decompilation. We query each model through its provider's chat API using the same MIDI-to-text input representation and decompilation prompt described in Appendix B. For all models, we decode with a maximum of 4096 new tokens and disable model-specific extended reasoning modes when available.

F.4 HYPERPARAMETERS

For SFT, we fine-tune both models for one epoch with LoRA (Hu et al., 2022) using rank 64 and $\alpha = 128$. We use a learning rate of 5×10^{-4} with a cosine schedule, weight decay 0.05, gradient clipping 1.0, and batch size 128. SFT training is performed on a single node with 8×H100 GPUs and completes in approximately 1.5 hours.

For RL, we initialize from the SFT checkpoint and optimize with rollout group size 16, batch size 64, and learning rate 5×10^{-6} for 10 epochs. We use reward weights $(w_1, w_2) = (1.0, 0.5)$, compilation-failure penalty $\rho = 0.1$ (Equation 6), KL coefficient 0.05, clipping threshold 0.4, sampling temperature 1.0, and a maximum generation length of 4096 new tokens. Thinking mode is disabled during rollout generation (`enable_thinking=false`). We select checkpoints using validation reward on a held-out mixture of STRUDEL-SYNTH and LMD MIDI, stopping when the validation faithfulness–readability tradeoff no longer improves. RL training is performed on a single node with 8×H100 GPUs and takes approximately 8 days. We implement the training backend using AReaL (Mei et al., 2025).

G EXTENDED RELATED WORK

Multi-objective reinforcement learning. RL for language models often optimizes multiple, partially conflicting rewards, such as task accuracy versus reasoning length (Arora & Zanette, 2025; Aggarwal & Welleck, 2025; Huang et al., 2025a; Su & Cardie, 2025; Xiang et al., 2025; Li et al., 2026) or multiple dimensions of human preference (Dai et al., 2024; Jang et al., 2023). A common approach is to collapse these objectives into a single weighted sum and directly optimize it (Rodriguez et al., 2025; Kolodiazhnyi et al., 2026; Aggarwal & Welleck, 2025; Arora & Zanette, 2025; Huang et al., 2025a); yet it often removes distinctions across reward dimensions and leads to suboptimal reward convergence. Recent methods therefore decouple reward signals: DRPO separates the learning signals of correct and incorrect rollouts (Li et al., 2026), while GDPO normalizes each reward component independently before aggregation (Liu et al., 2026). Our work adopts GDPO to optimize execution faithfulness and program readability as separate reward components.

Rubric-based evaluation and training. Structured rubrics and checklists score LLM outputs along multiple criteria in settings without a single verifiable answer. They are widely used for evaluation (Arora et al., 2025; Hashemi et al., 2024; Ruan et al., 2026; Gandhi et al., 2026; Gou et al., 2025; Pathak et al., 2025; Dineen et al., 2025), and recent work uses them directly as RL reward signals: Rubrics as Rewards (Gunjal et al., 2026) and RLCF (Viswanathan et al., 2025) decompose a task into binary checklist items scored by an LLM judge, yielding more stable training signal than scalar preference rewards in non-verifiable domains. We use the same design for our readability reward, with a fixed checklist combining general code-quality items and Strudel-specific criteria.

H ADDITIONAL EVALUATION RESULTS

This section provides additional results and analysis that complement the experiments in the main paper (Section 3). We first provide the full generalization results behind the summary in Section 3.4 (Appendix H.1). We then evaluate DECOMPOSER across genres to analyze how decompilation faithfulness varies with musical style (Appendix H.2).

H.1 FULL GENERALIZATION RESULTS

Table 6 reports the full generalization results corresponding to the summary in Table 3 of Section 3.4, where we evaluate whether DECOMPOSER generalizes beyond the training distributions. The table includes the in-domain STRUDEL-SYNTH and short-fragment LMD settings for reference, followed by out-of-domain datasets that vary duration, source, genre, and instrumentation: longer LMD excerpts (30–60 s; Raffel 2016), GigaMIDI (Lee et al., 2025), NES-MDB (Donahue et al., 2018), and Nottingham (Foxley, 2003). For the generalization study, we compare DECOMPOSER against GPT-5.5 as the representative frontier-LLM baseline. Among the frontier models in our main evaluation, GPT-5.5 provides the strongest overall tradeoff, achieving the best Inst F1 and readability rank on

Table 6: **Full generalization results across diverse MIDI datasets.** We compare GPT-5.5 and DECOMPOSER (8B) on in-domain and out-of-domain datasets, reporting Best@5 results selected by Inst F1. DECOMPOSER achieves stronger overall faithfulness scores than GPT-5.5 across datasets, while maintaining comparable readability and diversity. ↑/↓ indicate whether higher/lower is better; **bold** marks the better result between the two methods, and underline marks a tie.

		Faithfulness				Readability		Diversity
Dataset	**Method**	↑Comp.	↑Onset	↑Frame	↑Inst	↑Rubric	↓Rank	↓SelfCB
In-domain evaluation								
STRUDEL-SYNTH	GPT-5.5	0.97	0.70	0.67	0.62	0.37	1.61	**0.45**
	DECOMPOSER	**1.00**	**0.86**	**0.74**	**0.78**	**0.69**	**1.39**	0.47
LMD (< 30 s)	GPT-5.5	0.94	0.44	0.41	0.39	0.29	**1.44**	0.54
	DECOMPOSER	**1.00**	**0.70**	**0.57**	**0.68**	**0.56**	1.56	**0.46**
Out-of-domain evaluation								
LMD (30–60 s)	GPT-5.5	0.83	0.21	0.25	0.13	0.31	1.81	0.43
	DECOMPOSER	**1.00**	**0.35**	**0.31**	**0.31**	**0.68**	**1.19**	**0.42**
GigaMIDI	GPT-5.5	0.99	0.57	**0.38**	0.57	0.25	1.61	0.55
	DECOMPOSER	**1.00**	**0.61**	0.35	**0.61**	**0.56**	**1.39**	**0.47**
NES-MDB	GPT-5.5	0.91	0.37	0.33	0.28	0.28	**1.49**	0.49
	DECOMPOSER	**1.00**	**0.43**	**0.35**	**0.40**	**0.61**	1.51	**0.48**
Nottingham	GPT-5.5	1.00	0.51	0.46	0.51	0.28	1.50	**0.46**
	DECOMPOSER	1.00	**0.76**	**0.65**	**0.76**	**0.45**	1.50	0.49

both STRUDEL-SYNTH and LMD while remaining competitive in diversity (Table 2). We report best@5 results selected by Inst F1; all other metrics are computed on the selected candidates.

Compared with GPT-5.5, DECOMPOSER achieves stronger execution faithfulness on most datasets, while maintaining comparable readability and diversity. The largest gap between GPT-5.5 and DECOMPOSER appears on Nottingham ($+0.25$ Onset F1). This may reflect dataset-specific structure rather than genre alone: Nottingham is a single-instrument, ABC-derived folk collection with comparatively regular symbolic timing, and our instrument-grouped MIDI representation (Appendix B) collapses melodic and chord accompaniment into the same track. Such inputs may make melodic and harmonic organization easier for the learned decompiler to recover. At the same time, although DECOMPOSER outperforms GPT-5.5, absolute faithfulness remains lower on LMD (30–60 s) and NES-MDB than in the other settings. These cases highlight different aspects of decompilation: longer LMD excerpts require structure to be recovered over a wider temporal span, while NES-MDB introduces specialized chiptune timbres and dense note patterns. For readability and diversity, DECOMPOSER obtains higher Rubric scores than GPT-5.5 across the held-out datasets while Rank and SelfCB remain comparable. Overall, these results support the main claim of Section 3.4: the learned decompiler transfers beyond the synthetic paired data, yet its robustness remains sensitive to the temporal and stylistic structure of the input.

H.2 GENRE-WISE FAITHFULNESS

To complement the corpus-level generalization study with a finer-grained analysis of stylistic variation, we evaluate DECOMPOSER on ModArchive,[6] a public archive of tracker-format music. A tracker module is a self-contained music file that stores symbolic note patterns together with instrument samples and sequencing information. By combining sample-level audio material with explicit symbolic structure, tracker modules provide a natural intermediate representation for the broader audio-to-code decompilation setting (Appendix J). ModArchive is also well matched to our evaluation because tracker music and Strudel are both closely tied to electronic and sample-based music-making practices, and many modules include community-provided genre tags which allow us to analyze decompilation faithfulness across fine-grained musical styles.

[6] https://modarchive.org

Table 7: **Per-genre faithfulness on ModArchive.** We report Best@5 faithfulness of DECOMPOSER (8B) across 51 ModArchive genres, grouped by ModArchive's top-level genre families. Group rows report sample-weighted averages. Compile rate is omitted because it is 1.00 for all genres except Electronic – Gabber (0.99) and Classical (0.98).

Genre	n	Onset	Frame	Inst	Genre	n	Onset	Frame	Inst
Overall	**4463**	**0.38**	**0.37**	**0.37**	–	–	–	–	–
Electronica	**1763**	**0.40**	**0.38**	**0.39**	**Pop / Rock**	**925**	**0.38**	**0.37**	**0.37**
Electronic – Jungle	93	0.47	0.38	0.46	Disco	97	0.48	0.42	0.46
Electronic – Minimal	66	0.46	0.37	0.45	Ballad	92	0.42	0.35	0.40
Trance – Acid	93	0.44	0.41	0.41	Pop – Synth	82	0.41	0.41	0.39
Trance – Dream	84	0.44	0.38	0.42	Pop – Soft	97	0.41	0.36	0.39
Electronic (general)	91	0.43	0.42	0.42	Rock (general)	92	0.37	0.37	0.36
Electronic – House	92	0.42	0.43	0.41	Pop (general)	93	0.37	0.35	0.35
Trance (general)	77	0.42	0.40	0.40	Easy Listening	95	0.36	0.36	0.34
Trance – Goa	71	0.41	0.37	0.38	Rock – Hard	91	0.36	0.38	0.35
Electronic – Industrial	86	0.40	0.40	0.39	Rock – Soft	94	0.34	0.33	0.34
Electronic – Gabber	90	0.40	0.40	0.40	Funk	92	0.31	0.36	0.29
Trance – Hard	61	0.40	0.38	0.38	**Other**	**978**	**0.37**	**0.34**	**0.35**
Electronic – Rave	92	0.40	0.39	0.38	New Age	89	0.41	0.37	0.40
Electronic – Progressive	80	0.39	0.38	0.38	Experimental	92	0.40	0.37	0.40
Electronic – Drum & Bass	85	0.39	0.38	0.38	Piano	94	0.38	0.30	0.37
Electronic – Dance	89	0.38	0.40	0.37	Comedy	94	0.37	0.37	0.36
Electronic – IDM	77	0.38	0.37	0.38	Video Game	92	0.37	0.37	0.35
Electronic – Breakbeat	88	0.38	0.38	0.37	Soundtrack	79	0.36	0.33	0.35
Electronic – Techno	90	0.37	0.37	0.35	Fantasy	86	0.36	0.35	0.35
Electronic – Ambient	86	0.37	0.38	0.36	Folk	91	0.36	0.34	0.35
Chillout	89	0.35	0.32	0.33	World	91	0.34	0.32	0.32
Electronic – Hardcore	83	0.33	0.33	0.32	Classical	88	0.33	0.30	0.32
					Orchestral	82	0.32	0.30	0.31
Seasonal	**93**	**0.39**	**0.36**	**0.38**	**Urban / Hip-Hop**	**167**	**0.36**	**0.38**	**0.33**
Christmas	93	0.39	0.36	0.38	Hip-Hop	91	0.39	0.38	0.36
					Reggae	76	0.33	0.38	0.31
Alternative	**185**	**0.39**	**0.38**	**0.37**	**Jazz / Blues**	**255**	**0.32**	**0.36**	**0.31**
Alternative	94	0.39	0.39	0.38	Jazz – Acid	67	0.34	0.39	0.33
Metal (general)	91	0.38	0.38	0.36	Jazz – Modern	93	0.34	0.35	0.32
					Jazz (general)	95	0.30	0.35	0.28
					Demo / Tracking Scene	**97**	**0.31**	**0.34**	**0.30**
					Chiptune	97	0.31	0.34	0.30

We crawl tracker modules from ModArchive across 51 genre tags, recording each module's genre, format, and rating. For each genre, we target the top 100 modules by rating. Because tracker formats (e.g., `.mod`, `.xm`, `.it`, `.s3m`) differ substantially from the General MIDI inputs used elsewhere in this paper, we convert each module to GM MIDI using a headless wrapper around the open-source MilkyTracker engine.[7] Each tracker instrument is mapped to a General MIDI program using `Qwen2.5-Omni` (Qwen Team, 2025a); samples without a clear pitched or percussive identity, such as speech or sound effects, are discarded. We then extract a 16 s segment from each valid module, matching the mean duration of the short LMD fragments used during RL post-training (Appendix F). The resulting benchmark contains 4,463 inputs across all 51 genres, with an average of 87.5 examples per genre (range: 61–97).[8] As in the generalization study, we evaluate DECOMPOSER-8B using Best@5 metrics (Section 3.4).

Table 7 and Figure 8 report per-genre faithfulness, sorted by Onset F1. Performance varies substantially across genres, while compile rates remain almost saturated: all genres achieve a compile rate of 1.00 except Electronic–Gabber (0.99) and Classical (0.98). The genre-wise gaps therefore reflect differences in musical reconstruction rather than failures to produce executable Strudel programs.

At the level of broad genre families, Table 7 shows that Electronica achieves the highest aggregated scores, followed by Seasonal, Alternative, and Pop/Rock. The fine-grained ranking in Figure 8 shows a similar trend: dance- and electronic-oriented genres tend to score higher, with Disco ranked first

[7] https://milkytracker.org/

[8] The final number of examples per genre is below the initial crawl target of 100 because some genres contain fewer than 100 modules, or because some modules become empty after MIDI conversion. See https://modarchive.org/index.php?request=view_genres for the genre categories and their listed module counts.

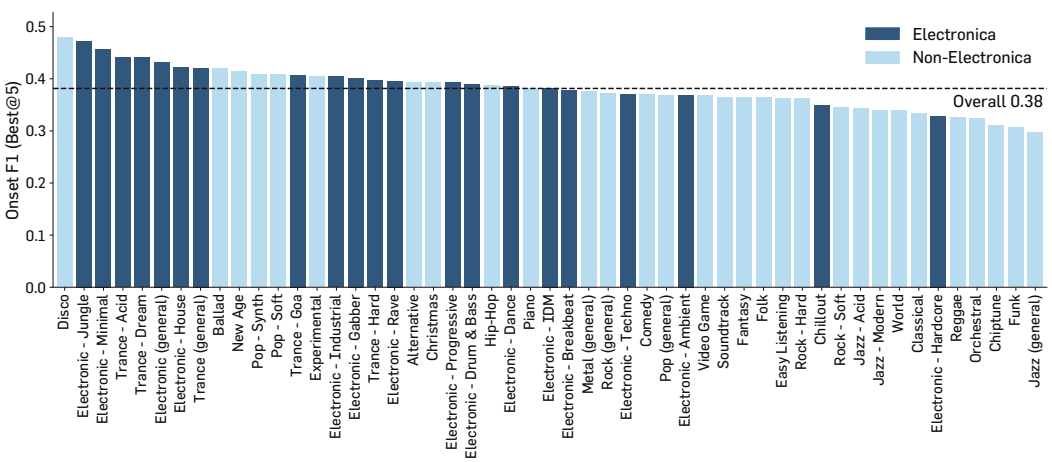


Figure 8: **Genre-wise faithfulness on ModArchive.** We show Best@5 Onset F1 of DECOMPOSER (8B) across 51 ModArchive genres, sorted in descending order. Electronica genres are highlighted separately from other genres, and the dashed line indicates the overall average. The distribution shows that performance varies substantially across musical styles, with several electronic and dance–oriented genres above average and genres with more varied instrumentation or less regular rhythmic structure more often appearing near the lower end.

and Electronica genres such as Jungle, Minimal, Acid, and House concentrated near the top. This pattern is consistent with the close fit between Strudel's pattern-based abstractions and rhythmically regular electronic and dance music. By contrast, lower-scoring genres include Jazz, Funk, Chiptune, Orchestral, and Reggae. These genres often involve expressive timing, off-beat rhythmic patterns such as syncopation, or dense note and instrumentation patterns, which can make the input harder to recover as a concise executable Strudel program. Overall, the results suggest that DECOMPOSER transfers best to genres whose structure is well captured by repeated pattern abstractions, while highlighting rhythmic variation and dense textures as key directions for improving faithful decompilation.

I HUMAN EVALUATION OF READABILITY

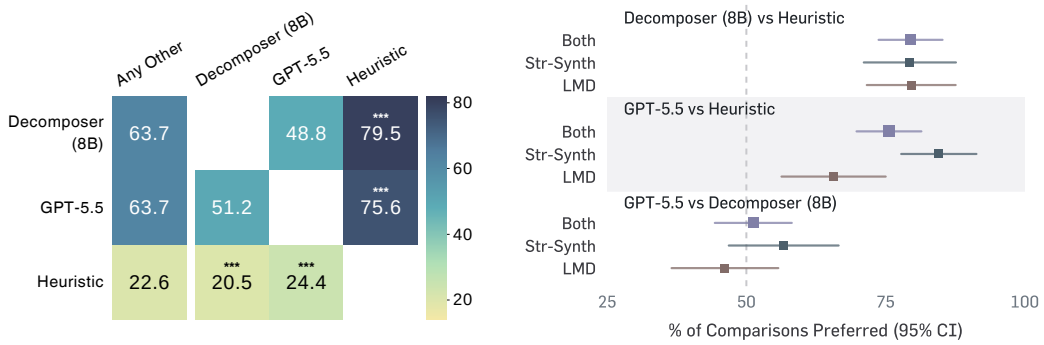
To complement the automatic evaluation of readability with the LLM-as-judge metrics (Rubric and Rank), we conduct a human evaluation to measure qualitative performance of DECOMPOSER against baselines.

I.1 STUDY DESIGN

We evaluate code readability through pairwise human preferences, following the evaluation approach adopted in previous studies (Donahue et al., 2023; Huang et al., 2025b; Kim et al., 2025). Each pair consists of outputs generated by two different systems from the same MIDI input. Participants are shown only the code, without the source MIDI or audio, to avoid confounding readability judgments with musical quality. Each pair is presented with the following reminder and question:

> Reminder: readability refers to how easily you can understand the structure and intent of the code, including how clearly it is organized and how easy it would be to inspect or modify. Focus on how easy the code is to follow, rather than how many musical operations, effects, or behaviors it contains.
>
> **Question:**
> **Which code is easier to read and understand?**

We compare DECOMPOSER (8B) with two baselines: the heuristic converter (Jong, 2025) and GPT-5.5 (OpenAI, 2026). Consistent with Section 3.4 and Appendix H.1, we select GPT-5.5 as the representative frontier model as it achieves the best Rank score among frontier LLMs (Table 2). To



(a) Overall preferences heatmap.

(b) Pairwise preferences by dataset.

Figure 9: **Results from the human evaluation of code readability. (a)** Preference heatmap showing the percentage of comparisons in which the row system was preferred over the column system, with preferences aggregated across opponents in the first column. Both DECOMPOSER and GPT-5.5 are preferred over the heuristic converter ($^{***}p < .001$), with no statistically significant preference between them ($p = .97$). **(b)** A further breakdown of pairwise preferences by dataset. Error bars indicate 95% confidence intervals, and the dashed line marks 50% (equal preference). Against the heuristic converter, preference rates for DECOMPOSER remain similar across datasets, whereas those for GPT-5.5 vary more.

construct the comparison set, we randomly sample 60 MIDI inputs from each of STRUDEL-SYNTH and LMD and construct all three pairwise comparisons for every input, yielding 360 comparison items in total. The programs are presented with system identities hidden and their left/right order counterbalanced.

We recruited 34 participants through academic communities, of whom 30 met our pre-specified eligibility criterion of being comfortable with both programming and creating/performing music. Each eligible participant rated 20 comparison items sampled uniformly at random from the 360 available items and completed two attention checks. Each attention check paired a clearly written implementation with a functionally identical version deliberately made difficult to read through a dense chain of single-letter variables and raw MIDI numbers, without comments. All 30 eligible participants passed both checks. In total, we obtained 600 readability judgments covering 289 of the 360 comparison items and all 120 MIDI inputs. For analysis, we calculate each participant's preference rate for each system pair and compare these rates against 50% (equal preference) using Wilcoxon signed-rank tests (Conover, 1999).

I.2 RESULTS

Figure 9 summarizes the pairwise preferences. In the overall results (Figure 9a), raters prefer DECOMPOSER and GPT-5.5 over the heuristic converter in the large majority of comparisons (79.5% and 75.6% respectively, both $^{***}p < .001$), indicating that both systems produce Strudel code that is substantially easier to read than a note-by-note transliteration. Between DECOMPOSER and GPT-5.5, we find no detectable preference (GPT-5.5 chosen in 51.2% of their comparisons, $p = .97$), and each wins an identical share of all its comparisons (63.7%). These results suggest that DECOMPOSER produces code that readers find competitive with GPT-5.5, consistent with the Rank results in Table 2.

Splitting by dataset (Figure 9b), DECOMPOSER and GPT-5.5 remain closely matched in their direct comparison: GPT-5.5 is chosen in 56.7% of comparisons on STRUDEL-SYNTH, while DECOMPOSER is chosen in 53.8% on LMD. Neither dataset shows a statistically significant preference for either system ($p = .92$ and $p = .50$, respectively). Against the heuristic converter, DECOMPOSER is preferred at nearly the same rate on both datasets (79.3% and 79.6%) whereas GPT-5.5 is preferred less often on LMD than on STRUDEL-SYNTH (65.7% vs. 84.5%; $p = .001$). The dataset split therefore reveals a steadier preference for DECOMPOSER over the heuristic converter, despite the two systems being closely matched in their direct comparisons. Together with the faithfulness results in Table 2, these findings suggest that DECOMPOSER can preserve the input MIDI more faithfully than frontier LLMs while maintaining comparable readability.

J Applications of Decomposer

We discuss how Decomposer can support downstream music workflows by turning music into executable, editable code. We also provide proof-of-concept examples on our project page.

Music decompilation (audio-to-code). Combined with automatic music transcription (Gardner et al., 2022; Chang et al., 2024; Shin et al., 2026; Rouard et al., 2026), Decomposer provides a path toward audio-to-code decompilation: audio is first transcribed to MIDI, and the resulting MIDI is then decompiled into a Strudel program. Since MIDI only partially captures acoustic attributes such as timbre, dynamics, and audio effects, extending music decompilation to audio would require modeling these attributes, potentially using audio production modeling methods (Steinmetz et al., 2024; Comunità et al., 2025) or audio language models with music understanding (Qwen Team, 2025a; Ghosh et al., 2025).

Music creation as code editing. Decomposer provides a programmatic editing interface for MIDI by converting sequential note events into Strudel programs that expose musical structures such as repetition, layering, harmony, and rhythm. These programs could be used in live coding settings or alongside DAW workflows, allowing users to edit higher-level musical structures such as chord progressions or scales in code rather than adjusting individual notes manually. Moreover, in creative settings where exploring multiple alternatives matters more than a single fixed output (Resnick et al., 2005; Kim et al., 2025), the faithfulness–readability tradeoff (Section 3.3) could be a controllable feature: users can dial the system toward faithful transcriptions for precise editing, or toward abstraction for creative exploration.

Music understanding through code. Recovering code from MIDI can make musical structure easier to inspect and reason about. Unlike a low-level MIDI sequence, a Strudel program with explicit musical structure could serve as an intermediate representation for LLM-based music understanding (Gardner et al., 2024; Ghosh et al., 2025), allowing models to reason over compact, interpretable programs rather than long sequences of note events. Such programs could also benefit human users: decompiled code could provide interactive material for music education (Petrie, 2022), and its explicit repetition could aid analysis tasks such as motif discovery (Boot et al., 2016).

K Qualitative Examples

Figures 10 and 11 show qualitative examples comparing decompilations of the same MIDI input across representative methods: Decomposer (8B), the vanilla Qwen3-8B model (`Qwen3-8B`; Qwen Team 2025b), Qwen3-8B after SFT, the heuristic converter (Jong, 2025), and the three frontier models used in our evaluation: Claude-Opus-4.6 (Anthropic, 2026), Gemini-3.5-Flash (Google DeepMind, 2026), and GPT-5.5 (OpenAI, 2026). To make structural differences easier to compare, we omit sound effect parameters such as `gain()` and `room()`. For additional examples with sound, please refer to the project page: `https://decomposer-page.netlify.app`.



(a) Piano roll visualization of the input MIDI

```
setcpm(85/4)

// piano chords
$: n(`<[0,4,7] [5,9,12] [3,7,10] [7,11,14]>`)
 .scale("Bb3:minor").sound("gm_epiano1")

// bass
$: note("<bb2 gb3 eb3 bb3>")
.sound("gm_acoustic_bass")

// minor melody
$: n(`<[0 ~ 4 7] [5 ~ 9 12]
[3 ~ 7 10] [7 ~ 11 ~]>`)
 .scale("Bb4:minor").sound("gm_music_box")

// kick + snare
$: sound("bd ~ sd bd ~ bd sd ~").bank("TR707")

// hats
$: sound("hh*4").bank("TR707")

// open hat
$: sound("oh").struct("0 0 1 0 0 0 1 0")
.bank("TR707")
```

(b) DECOMPOSER (8B)

```
setcpm(85/4)

// drum — syncopated groove
stack(
  s("bass_drum"), s("closed_hi_hat"),
  s("acoustic_snare"), s("open_hi_hat")
).fast(16).rev()

// acoustic_bass — walking line
stack(
  note("a#2").loop(4), note("f#3").loop(4),
  note("d#3").loop(4), note("a#3").loop(4)
).loop(4)

// electric_piano_1 — melodic line
stack(note("a#3").loop(4), note("f4").loop(4),
  note("a#4").loop(4), note("f#4").loop(4),
  note("c#5").loop(4), note("f#5").loop(4)
).loop(4)

// music_box — arpeggiated line
stack(
  note("a#4").loop(4), note("f5").loop(4),
  note("d#6").loop(4), note("a#5").loop(4),
  note("f6").loop(4), note("a#6").loop(4)
).loop(4)
```

(c) Vanilla Qwen3-8B

```
setcpm(85/4)

// drums — lo-fi boom bap
const drums_zhp = x => x.sound("RolandTR808")

$: drums_zhp("bd:5 ~ [~ bd:5] ~ ~ bd:5 ~ ~")

$: drums_zhp("[~ sd:4 ~ ~ ~ sd:4 ~ ~]")

$: n("<[3 <7 14>] [~ <10 18> ~ ~]>")
  .sound("korgm1")
  .scale("Bb4:minor")
```

(d) Qwen3-8B after SFT

Figure 10: **Qualitative examples: trained model variants.** We show the input MIDI visualization and the corresponding Strudel decompilations produced by DECOMPOSER (8B), the vanilla Qwen3-8B model, and Qwen3-8B after SFT. Sound-effect parameters such as `gain()` and `room()` are omitted so that the comparison focuses on how each method encodes the underlying MIDI structure.

```
setcpm(85/4)

$: note(`<a#2 f#3 d#3 a#3>`)
.sound("gm_acoustic_bass")

$: note(`<[a#3,f4,a#4] [f#4,c#5,f#5]
[d#4,a#4,d#5] [a#4,f5,a#5]>`)
 .sound("gm_epiano1")

$: note(`<[a#4 - f5 a#5] [f#5 - c#6 f#6]
[d#5 - a#5 d#6] [a#5 f6]>`)
 .sound("gm_music_box")

$: s(`<[bd ~ ~ bd ~ bd ~ ~] [bd ~ ~ bd ~ bd ~ ~]
[bd ~ ~ bd ~ bd ~ ~] [bd ~ ~ bd ~ bd ~ ~]>`)

$: s(`<[hh hh hh hh] [hh hh hh hh]
[hh hh hh hh] [hh hh hh hh]>`)

$: s(`<[~ oh ~ oh] [~ oh ~ oh]
[~ oh ~ oh] [~ oh ~ oh]>`)

$: s(`<[~ sd ~ sd] [~ sd ~ sd]
[~ sd ~ sd] [~ sd ~ sd]>`)
```

(a) Heuristic converter

```
setcpm(85/4)

// chord progression: Bb, F#, D#, Bb
const chords = `<[a#3,f4,a#4] [f#4,c#5,f#5]
[d#4,a#4,d#5] [a#4,f5,a#5]>`
stack(
 // kick pattern — 8 sixteenths grid
 s("[bd ~ ~ ~ ~ bd ~bd]*2").bank("RolandTR909"),
 // snare on beats 2 and 4 (each half)
 s("[~ ~ sn ~ ~ ~ ~ sn]*2").bank("RolandTR909"),
 // closed hi-hat on 1, 2, 3, 4 of each half
 s("[hh ~ hh ~ hh ~ ~ ~]*2")
 .bank("RolandTR909"),
 // open hi-hat on beats 2 and 4 alongside snare
 s("[~ ~ oh ~ ~ ~ ~ oh]*2").bank("RolandTR909"),

 // bass — quarter note roots
 note("a#2 f#3 d#3 a#3").s("sawtooth"),

 // electric piano — chords, one per beat
 note(chords).s("square"),

 // music box — arpeggiated melody
 note(`a#4 ~ f5 a#5 f#5 ~ c#6
 f#6 d#5 ~ a#5 d#6 a#5 ~ f6 ~`).s("triangle")
)
```

(b) Claude-Opus-4.6

```
setcpm(85 / 4)

// chords — electric piano 1 progression
const epChords = [
 "[a#3, f4, a#4]", // Bb/A# minor triad
 "[f#4, c#5, f#5]", // F# major
 "[d#4, a#4, d#5]", // D# minor
 "[a#4, f5, a#5]" // Bb/A# minor
]
stack(
 // drums — crisp neo-soul hip-hop beat
 s(`[bd, hh] [sd, hh, oh] [~ bd]
[hh, bd] [bd] [sd, hh, oh]`),

 // bass — deep acoustic bassline
 note("<a#2 f#3 d#3 a#3>").s("acoustic_bass"),

 // ep — warm electric piano chord stabs
 note(epChords).s("electric_piano_1"),

 // music box — sparkling, resonant
 // high-register arpeggios
 note(`<[a#4 ~ ~ f5 ~ a#5] [f#5 ~ ~ c#6 ~ f#6]
[d#5 ~ ~ a#5 ~ d#6] [a#5 ~ ~ f6 ~ ~]>`)
.s("music_box")
)
```

(c) Gemini-3.5-Flash

```
setcpm(85/4)

stack(
 // drums — steady backbeat
 // with syncopated kicks and hat accents
 stack(
 s("bd ~ ~ bd ~ bd ~ ~").fast(2),

 s("~ ~ sn ~ ~ ~ ~ sn").fast(2),

 s("hh ~ hh ~ hh ~ hh ~").fast(2),

 s("~ ~ oh ~ ~ ~ ~ oh").fast(2)
 ),

 // bass — sustained root movement
 note("a#2 f#3 d#3 a#3").s("bass"),

 // electric piano — bright triad stabs
 note(`[a#3,f4,a#4] [f#4,c#5,f#5]
[d#4,a#4,d#5] [a#4,f5,a#5]`).s("piano"),

 // music box — sparse arpeggiated top line
 note(`[a#4 ~ f5 a#5] [f#5 ~ c#6 f#6]
[d#5 ~ a#5 d#6] [a#5 ~ f6 ~]`).s("musicbox")
)
```

(d) GPT-5.5

Figure 11: **Qualitative examples: baselines.** For the same MIDI input as Figure 10, we show decompilations produced by the heuristic converter and the frontier models (Claude-Opus-4.6, Gemini-3.5-Flash, and GPT-5.5). Sound effect parameters are omitted for consistency.