AI Composition Assistant using a KB-based LLM

Nathan Lim, Jeremiah Liao, Colin Ngo, and Jerry Tang
Cal Poly, San Luis Obispo, United States

Final Project: CSC 481 – Knowledge-Based Systems, Cal Poly, Spring 2025 (Instructor: Dr. Rodrigo Canaan)

***Abstract*—In this report, we present a hybrid composition assistant that automatically appends stylistically faithful continuations to solo-piano compositions in the form of MIDI files. Our system combines a compact knowledge base consisting of compositional and stylistic rules with a four-agent LLM workflow implemented in LangGraph. Compared to other deep learning based approaches, our system proves itself in its ability to remain stylistically sound with the input while still generally showing creative ability to compose more than a mere repeat of what came before. Our results demonstrate that a hybrid LLM-KB system works well for LLM-generated music compositions.**

I. Introduction

During the process of music composition, composers, like writers, may face "writer's block", unsure of how to continue a piece. During this process, having a tool, such as a composition assistant, would be helpful for composers to overcome blocks, offering suggestions on how to continue a piece. This tool must 1) adhere to music theory rules, generating suggestions that are legal within the realm of music, and 2) maintain style continuity of the original piece. A musical composition from a composer such as Bach differs heavily from that of Mozart stylistically. If a composition assistant extends a piece from Bach, it must remain in the same style as that of Bach. However, current approaches to music composition assistants, relying on a deep learning based approach, are inaccurate both in adhering to music theory rules and composition style.

Our design of a composition assistant takes a hybrid approach by feeding a knowledge base of style continuity rules to a large language model (LLM) to generate style-faithful continuations to a piece while adhering to music theory rules. This combines the generative power of LLMs with a set of constraints, optimizing the musical output it generates. To evaluate the effectiveness of our composition assistant, we took samples of classical pieces and extended those samples with our composition assistant and other models on the web. We then compared the output by ear, determining whether it adhered to music theory rules and composition style. While all the systems adhered to music theory rules, our composition assistant outperformed existing ones in style continuity to the point where the transition from written to AI generated music was indistinguishable.

We first explore the problem with current approaches to AI generated composition and then propose a design—specifications, implementation, tools used—for a more robust composition assistant. Finally, we evaluate the design, comparing and evaluating our generated outputs with existing approaches, identifying current gaps in our implementation, and suggestions for future work.

II. Related Work & References

A. Pure DL Generation – Markov, Music Transformer, Mubert

Early algorithmic composition models such as Markov Chains used simple probabilistic transitions to generate note sequences. While computationally efficient, these models lacked the ability to model long-term structure or develop coherent phrasing over time [1]. The introduction of the Music Transformer marked a significant advancement by employing relative self-attention mechanisms to capture long-range dependencies and expressive dynamics in symbolic music generation [2]. Commercial systems like Mubert use proprietary deep learning pipelines trained on massive datasets to stream continuous, AI-generated music in real time [3]. However, such models are generally black-box systems, offering limited user control over structure, harmony, or theoretical elements.

B. Commercial Tools

To test our MIDI sequences, we evaluated multiple commercial AI-assisted MIDI extension tools, including ExtendMusic.ai, Soundverse, and SkyTNT MIDI Composer. These tools allow users to upload a seed MIDI file, which is then extended by proprietary generative models trained on genre-specific or mood-specific datasets. Similarly, platforms like Staccato AI [4], AivaAI [5], and Beethoven AI [6] offer comparable services, using deep learning to generate stylistically consistent continuations. However, most of these tools lack transparency in how decisions are made and provide little to no capability for dynamic theory integration or symbolic control, limiting their utility for users seeking structured compositional guidance.

C. Rule-based Systems

Prior research has integrated knowledge-based (KB) rules—such as harmony constraints, voice leading, and key signatures—with generative models to ensure more musically valid outputs. These systems encode theory explicitly, allowing the composer to enforce stylistic or structural constraints throughout generation. While such rule-based frameworks improve interpretability, they often become overly rigid or computationally intensive when scaled without adaptive mechanisms.

D. Our Project

Our work targets the underexplored intersection of LLMs, symbolic rule systems, and modular architecture in music generation. We sought to explore the classroom insight of combining rule-based systems with DL systems (in our case, LLMs) to produce higher quality AI-based results compared to either method alone. Current research in the field of music generation seems to have very little results for LLM based or hybrid based composition, so our work is practically entirely from scratch. Instead of building off of any of the aforementioned existing research, we decided to build our own system from the ground up and instead use the existing methods as a baseline for comparison to our own.

III. Specification

Our overall system works as follows: given an initial MIDI excerpt, the composition assistant system is tasked to generate an extended MIDI excerpt consisting of a specified number of additional measures. Unlike audio files, MIDI (Musical Instrument Digital Interface) files contain symbolic musical data that can be easily parsed, analyzed, and modified programmatically. Each note event includes parameters such as pitch (represented as MIDI note numbers 0-127), velocity, onset time, and duration. The AI composition process focuses mostly on stylistic continuity such that the generated music seamlessly aligns harmonically, rhythmically, and melodically with the original piece. The system takes the following inputs and parameters:

1) Solo-piano MIDI file: Initial musical excerpt provided by the user. This MIDI file can be either one channel, with both clefs represented by a single channel, or dual channel with treble clef in one channel and bass clef in the other.

2) Output file location: File location for the output MIDI file. This output MIDI file will always be single channel, no matter if the input was single or dual channel.

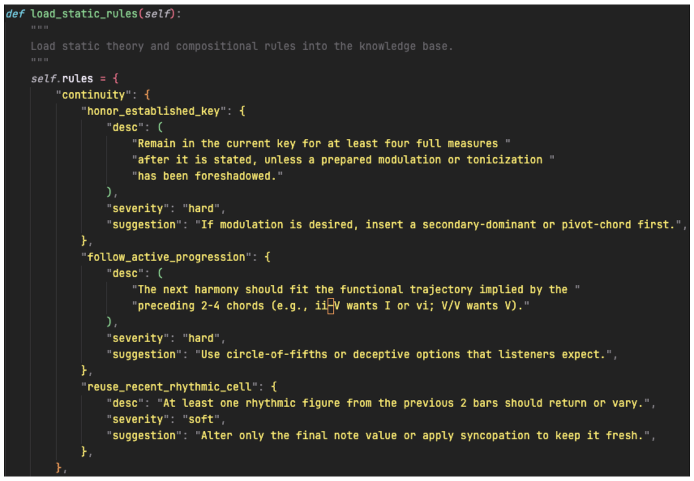
3) Optional number of additional measures: The number of measures that the system will extend the composition to. For example, if this is set to 8, the system will add 8 additional measures to the composition. This is optional because the default is set to 8.

4) Optional natural-language prompt: A brief textual description or style suggestion. If not specified, the natural-language prompt defaults to ”Extend the excerpt in a similar style.”

5) Optional recursion limit: The maximum number of recursive calls in the agentic system. Used as a safeguard against the system making too many OpenAI API calls. Default is set to 50.

6) Optional maximum review iterations: The maximum number of iterations allowed by the reviewer agent to verify generated measures against the knowledge base rules to prevent extensive runtime or LLM token costs. The reviewer agent is much more expensive than the composer agent because it needs to see the entire composition—the default for this parameter is 3.

Upon execution, the system provides a runtime log to the console of agent communication, including printing the dynamically generated rules and composer planner instructions. Upon completion, the output MIDI file is located in the directory specified by the command line argument passed into the script.



```
def load_static_rules(self):
    """
    Load static theory and compositional rules into the knowledge base.
    """
    self.rules = {
        "continuity": {
            "honor_established_key": {
                "desc": (
                    "Remain in the current key for at least four full measures "
                    "after it is stated, unless a prepared modulation or tonicization "
                    "has been foreshadowed."
                ),
                "severity": "hard",
                "suggestion": "If modulation is desired, insert a secondary-dominant or pivot-chord first.",
            },
            "follow_active_progression": {
                "desc": (
                    "The next harmony should fit the functional trajectory implied by the "
                    "preceding 2-4 chords (e.g., ii□V wants I or vi; V/V wants V)."
                ),
                "severity": "hard",
                "suggestion": "Use circle-of-fifths or deceptive options that listeners expect.",
            },
            "reuse_recent_rhythmic_cell": {
                "desc": "At least one rhythmic figure from the previous 2 bars should return or vary.",
                "severity": "soft",
                "suggestion": "Alter only the final note value or apply syncopation to keep it fresh.",
            },
        },
```

Fig. 1. Snippet of Knowledge Base Rules showing static compositional constraints organized by category

IV. Implementation

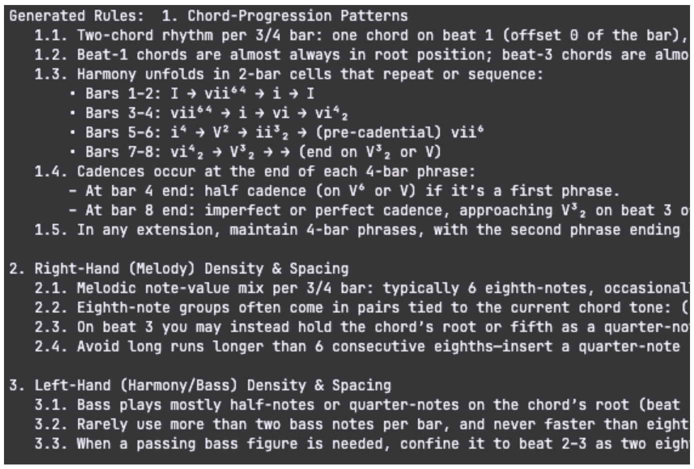
Our implementation of the composition assistant uses a hybrid artificial intelligence approach for generating music. It consists of two parts—a knowledge base containing static and dynamic music theory rules, and an agentic LLM system responsible for music composition. The system is separated into 4 distinct Python files: a main.py for operating the system, a KB.py containing the code for our knowledge base, a MidiHandler.py containing functions to parse and handle MIDI data, and a compositionAgent.py which contains the main logic for our agentic AI system.

A. Knowledge Base

The knowledge base consists of all information related to the piece, including broad music theory rules and style continuation rules (static), as well as piece specific rules such as key, tempo, chord progressions, and common motifs and patterns (dynamic). Static rules are rules we manually configured in the composition assistant for the LLM to adhere to, while dynamic rules are piece specific, generated at runtime.

Static rules, specifically, are hardcoded composition principles organized into seven categories: continuity, chord progression, voice leading, phrase structure, motivic development, melodic constraints, and rhythmic patterns. Each rule contains a description, severity level (hard or soft), and suggested corrections. As an example of a couple of the rules, the continuity rules include ”honor established key,” requiring the piece to remain in the current key for at least four measures unless a prepared modulation is foreshadowed, and ”follow active progression,” requiring that harmonic progressions follow their expected functional trajectories. These rules are implemented in the KnowledgeBase class in KB.py and serve as universal constraints that apply to all compositions regardless of the input piece provided by the user. Figure 1 shows an example of our static knowledge base rules.

Dynamic rules are generated at runtime through algorithmic analysis of the input MIDI file combined with LLM-based stylistic analysis. Our code extracts as much information algorithmically as it can, including key signature, time signature, and chord progressions. This extracted information, along with the entire piece, is then sent to a dynamic rule builder LLM that generates specific stylistic rules about chord progression patterns, note density, melodic intervals, rhythmic patterns, and other stylistic elements that should be maintained to continue the piece in the same style as the original input. These generated rules become stored as text within the knowledge base and are combined with static rules to guide the composition process. Figure 2 shows an example of dynamically generated rules for a specific piece. To integrate the knowledge base with the LLM calls, we decided to pass in all of the rules in the KB in text format with all of our prompts.



```
Generated Rules:  1. Chord-Progression Patterns
   1.1. Two-chord rhythm per 3/4 bar: one chord on beat 1 (offset 0 of the bar),
   1.2. Beat-1 chords are almost always in root position; beat-3 chords are almo
   1.3. Harmony unfolds in 2-bar cells that repeat or sequence:
      • Bars 1-2: I → vii^64 → i → I
      • Bars 3-4: vii^64 → i → vi → vi^4_2
      • Bars 5-6: i^4 → V^2 → ii^3_2 → (pre-cadential) vii^6
      • Bars 7-8: vi^4_2 → V^3_2 → → (end on V^3_2 or V)
   1.4. Cadences occur at the end of each 4-bar phrase:
      - At bar 4 end: half cadence (on V^6 or V) if it’s a first phrase.
      - At bar 8 end: imperfect or perfect cadence, approaching V^3_2 on beat 3 o
   1.5. In any extension, maintain 4-bar phrases, with the second phrase ending

2. Right-Hand (Melody) Density & Spacing
   2.1. Melodic note-value mix per 3/4 bar: typically 6 eighth-notes, occasional
   2.2. Eighth-note groups often come in pairs tied to the current chord tone: (
   2.3. On beat 3 you may instead hold the chord’s root or fifth as a quarter-no
   2.4. Avoid long runs longer than 6 consecutive eighths—insert a quarter-note

3. Left-Hand (Harmony/Bass) Density & Spacing
   3.1. Bass plays mostly half-notes or quarter-notes on the chord’s root (beat
   3.2. Rarely use more than two bass notes per bar, and never faster than eight
   3.3. When a passing bass figure is needed, confine it to beat 2-3 as two eigh
```

Fig. 2. Example snippet of dynamically generated rules created by analyzing a specific input MIDI file, showing piece-specific stylistic constraints

B. Agentic LLM System

When an input MIDI file is given to the composition assistant, it is first parsed using the music21 Python library to extract piece related information, including key, time signature, notes and their corresponding offsets in the piece, and underlying chord progressions. That information, in conjunction with the static rules from the knowledge base, is fed to a dynamic rule builder LLM that analyzes the musical style and generates piece-specific composition rules. Our system uses a multi-agent architecture built using LangGraph that consists of four specialized agents that work together in a directed graph workflow:

1. Dynamic Rule Builder Agent: Initializes the system by parsing the input MIDI file, building the knowledge base with both static and dynamic rules, and setting up the system state.

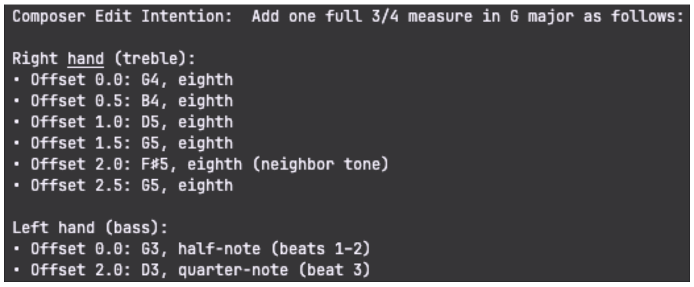
2. Composer Planner Agent: Responsible for generating new musical content by analyzing the existing piece along with the KB and proposing specific note additions that follow the established style and rules, measure by measure. These additions are plaintext instructions that are then passed to the Handler agent.

3. Handler Agent: A reactive agent equipped with MIDI manipulation tools that is able to execute specific musical instructions from the Composer and Reviewer agents via LLM function calling.

4. Reviewer Agent: Evaluates the generated composition against the knowledge base rules and identifies any violations that need correction. If corrections are required, they are passed as plaintext instructions to the Handler agent to execute.

LangGraph’s framework allows our system to be easily implemented as a state graph. The initialization phase of our graph starts with the Dynamic Rule Builder agent, which loads the input MIDI file using the MidiHandler class and leverages the music21 library to parse musical elements including notes by part, chord progressions, key signatures, and time signatures. It then constructs a comprehensive knowledge base that combines our static rules with our dynamically generated rules.

In the next phase, the Composer Planner agent operates iteratively, adding one measure at a time until the target length (in number of measures) is achieved. For each iteration, it receives the last 12 measures of the current piece in JSON format along with the complete knowledge base. The JSON format includes the length, duration, pitch, and placement of all the notes in the specific measures. The agent generates specific instructions for note placement that include exact pitches, durations, and offsets within the measure. These instructions are passed off to the Handler agent, which then is able to execute all of those individual instructions via multiple function calls to the MIDI manipulation tools. Figure 3 shows an example of the output from the Composer Planner that gets passed to the Handler agent.



```
Composer Edit Intention:  Add one full 3/4 measure in G major as follows:

Right hand (treble):
• Offset 0.0: G4, eighth
• Offset 0.5: B4, eighth
• Offset 1.0: D5, eighth
• Offset 1.5: G5, eighth
• Offset 2.0: F#5, eighth (neighbor tone)
• Offset 2.5: G5, eighth

Left hand (bass):
• Offset 0.0: G3, half-note (beats 1-2)
• Offset 2.0: D3, quarter-note (beat 3)
```

Fig. 3. Example output from Composer Planner agent showing specific note placement instructions passed to Handler agent for MIDI manipulation

Once the target length is reached, the Reviewer agent takes over the primary looping. It examines all of the generated measures against the knowledge base rules. If violations are detected, the agent provides specific correction instructions to be executed by the Handler agent. The system supports different numbers of review iterations, which is configurable via the command-line parameter, to ensure compliance with the knowledge base. Because the Reviewer agent must examine the entire piece for violations of the KB, it is the most expensive part of the system process and that is why a strict limit was placed on the maximum number of iterations.

C. Design Decisions

There were a few key design decisions that were made to optimize the system's performance and quality. We chose to use OpenAI's o4-mini reasoning LLM as our primary language model because of its reasoning capabilities. We found that its ability to reason led to much stronger compliance with the KB while maintaining cost efficiency for the iterative nature of our composition process—especially as compared to some of OpenAI's more expensive models like o3. We also set the temperature parameter of the LLM to be equal to 1. The temperature parameter of LLMs defines how stochastic its output is. The higher it is, the more likely it is to give less likely, more creative output. Creativity is a key part of the composition process, so we wanted to keep the temperature well above zero.

In addition, the introduction of a dedicated Handler agent creates a crucial separation of concerns that significantly improved our system's composition quality. By isolating individual MIDI manipulation operations away from musical decision-making, our system enables the Composer and Reviewer agents to focus purely on high-level musical reasoning without needing to understand the technical details of the MIDI functions or tool calling. This choice also facilitates better prompt engineering, as each agent becomes more specialized for its specific role. The Composer Planner handles creative music generation, the Reviewer handles specific rule-based evaluation, and the Handler focuses on technical execution. Both the Composer Planner and Reviewer agents are also prompt engineered to compose measure-by-measure. For the Reviewer agent, this means specifying a replacement measure for any measures that violate the KB. We found that this measure-by-measure approach helped ensure a greater level of musical continuity because it maintained constant awareness of recent musical context, rather than relying purely on note-by-note or unspecified additions.

Our initial proposal for our implementation called for 100-200 comprehensive music theory-based and dynamically generated rules, but our final system reduced this number to approximately 40 carefully curated rules based on cost-benefit analysis and context limitations. Each additional rule we add increases the token count in every LLM call, significantly impacting API costs given that our iterative, multi-agent architecture makes dozens of LLM calls per composition extension. We also found that excessive rules diluted the context window, which reduced the space available for actual musical content and piece-specific analysis. When we used lots of rules heavily weighted toward theoretical constraints (no parallel 5ths, etc.), the system produced compositions that were theoretically sound but stylistically disconnected from the input piece.

In addition to reducing the number of rules in our system, we also shifted our focus from music theory-focused rules to style continuity-based rules. We discovered that LLMs already possess substantial "built-in" music theory knowledge, and so focusing our rules on continuity led the AI focus on that part of composition and the music theory abidance would follow on its own. This insight led us to emphasize rules that prioritize maintaining the specific character, rhythmic patterns, melodic contours, and harmonic progressions of the original piece rather than enforcing abstract theory principles. This approach, we found, produced extensions that sounded much more like natural continuations of the original piece rather than theoretically correct but stylistically disconnected additions.

V. Analysis and Results

To evaluate the effectiveness of our composition assistant, we sampled three Classical excerpts—Minuet in G Major BWV Anh. 116 by J.S Bach, Sonata No. 14 in C Minor by Ludwig van Beethoven, and Fantaisie-Impromptu by Frédéric Chopin. Each excerpt ranged between 8 - 10 measures, and was, on average, 10 seconds in length. We decided to select these 3 Classical pieces as they offered a range of styles (from 3 different periods of classical music), keys, time signature, and complexity. We then fed these three excerpts into our AI composition assistant as well as a few other music generation models—StaccatoAI [4], AivaAI [5], BeethovenAI [6]—to extend each piece by 8 seconds. For brevity, we will not include all our system's generated and externally generated outputs in this report—instead, we have MIDI and MP3 files of all of our evaluation tests in our GitHub repository for this project.

With music being subjective, it is inherently difficult to evaluate how "good" a piece is. While there is no defined guide for what makes a composition better, there are rules and common patterns in composition that the piece must adhere to. Rather than evaluate how "well" the piece is as a whole, our basis of evaluation focused primarily on how well models did on maintaining the style of composition, and whether it adhered to basic music theory rules and piece patterns such as key, tempo, and chord progressions. As a listening-based art, music is difficult to analyze visually by merely looking at the music score. Additionally, due to the method in which the composition was extended, the formatting of pieces when rendering them in a music editing software made the excerpt unreadable. Our system produces one channel MIDI output—MuseScore will tend to interpret this as putting all of the notes onto one staff in most cases. Even though our output might appear to be formatted oddly on paper (the sheet music), our mode of evaluation was primarily by ear, listening for general melodic and harmonic patterns or deviations in key and standard chord progressions.

Our ultimate metric of success is that the listener of the composition would not be able to confidently identify where human composition ends and AI continuation begins.

Through our evaluation, we found that all three music generation models strongly adhered to music theory rules. However, not all models performed equally well when it came to style continuity. In comparison to the other 2 models, our

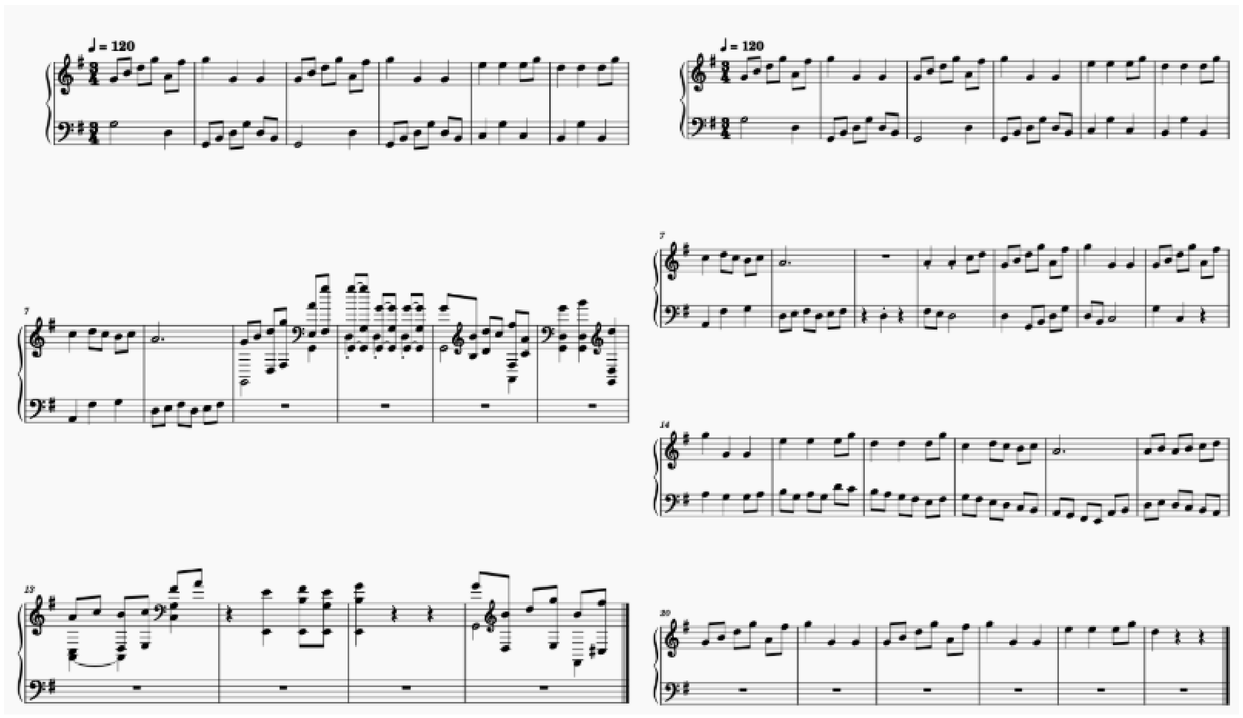


Fig. 4. Comparison of extension of Bach's Minuet in G Major BWV Anh. 116 between our composition assistant (left) and StaccatoAI [4] (right). Extension starts in measure 9. Our system maintains better counterpoint and voice independence characteristic of Bach's style.

AI composition assistant performed the best in maintaining the style of the piece while having the creative freedom to compose unique music rather than repeating previous measures. With Chopin Fantaisie-Impromptu, for example, our system composes something that is faithful to the original style of the piece while not being a complete repeat of what came before. StaccatoAI [4] seemed to produce something that was rhythmically similar to the original, but nonsensical in terms of actual notes and it did not seem like a fluid continuation.

One difference we noticed, especially when comparing extending Bach's Minuet in G Major, was the degree of cohesiveness between the left hand and right hand parts. One major characteristic of compositions by Bach and other similar composers during that time period was the usage of counterpoint—the relationship of two or more simultaneous musical lines that are harmonically dependent on each other, yet independent in rhythm and melodic contour. StaccatoAI [4], perhaps due to the method of music generation (extending individual parts), did not perform well with maintaining counterpoint, lacking cohesiveness between voices. Our model, on the other hand, maintained this fairly well, with both voices sounding fairly cohesive while maintaining individual melodies.

While our model maintains style in the music it generates—often better than the competing AI systems (as seen with Chopin Fantaisie-Impromptu)—it tends to not perform as well when extending the input for too long. As seen in Figure 4, the style of the piece drops significantly in measure 14, changing keys momentarily and pausing suddenly in the music. One potential explanation for this could be due to the shortened input given to the composition assistant. Limited context windows with LLMs are a key limitation of the system, and we also trim the number of previous measures we provide to the LLM to keep token costs low. LLMs are effective at analyzing patterns within data, hence its effectiveness at extending music while maintaining style. When the context window becomes too diluted, our composition assistant cannot further infer patterns or chord progressions to continue in. On the other extreme, as seen in Figure 6 with Sonata No. 14, it maintains the style too closely, essentially repeating the same notes with no creativity. Overall, our model tended to meet our metric of success—being that the listener is unable to confidently identify where human composition ends and AI continuation begins—in most of our test cases.

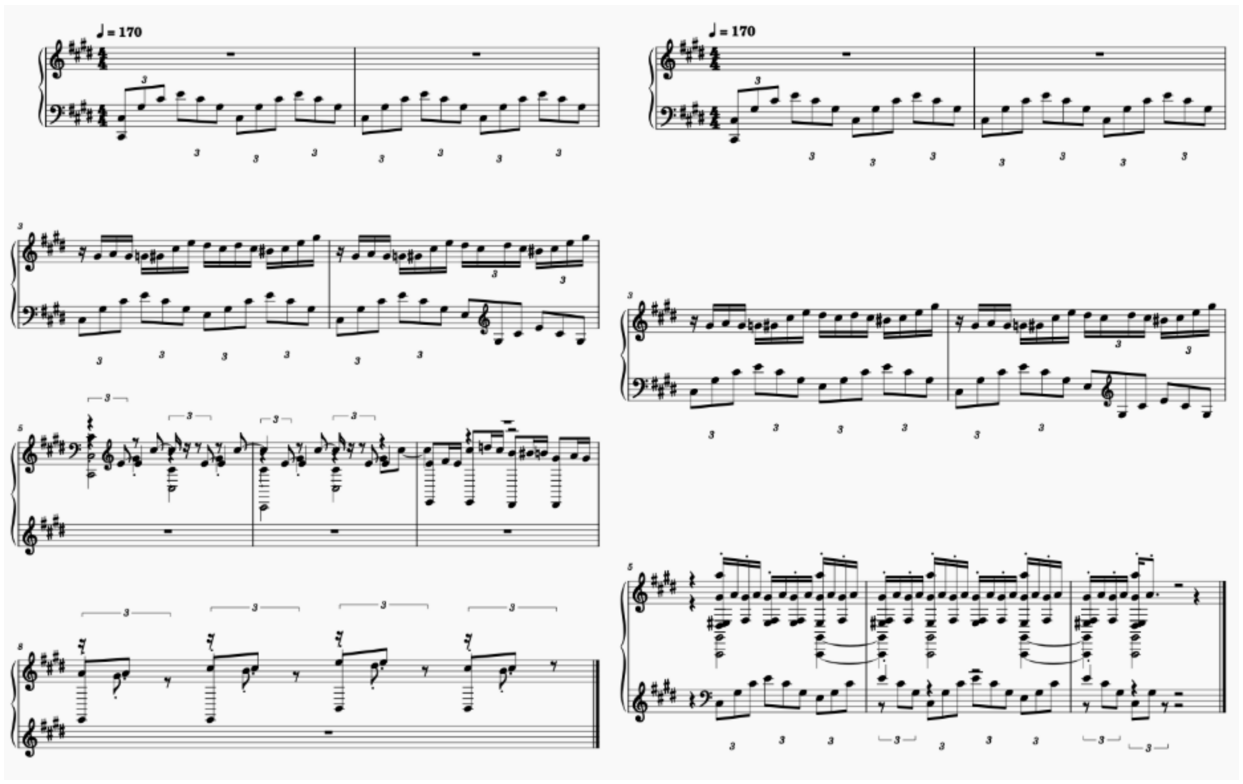


Fig. 5. Comparison of extension of Chopin's Fantaisie-Impromptu between our composition assistant (left) and StaccatoAI [4] (right). Extension starts in measure 5. Our system demonstrates better stylistic continuity while StaccatoAI [4] produces rhythmically similar but melodically nonsensical output.

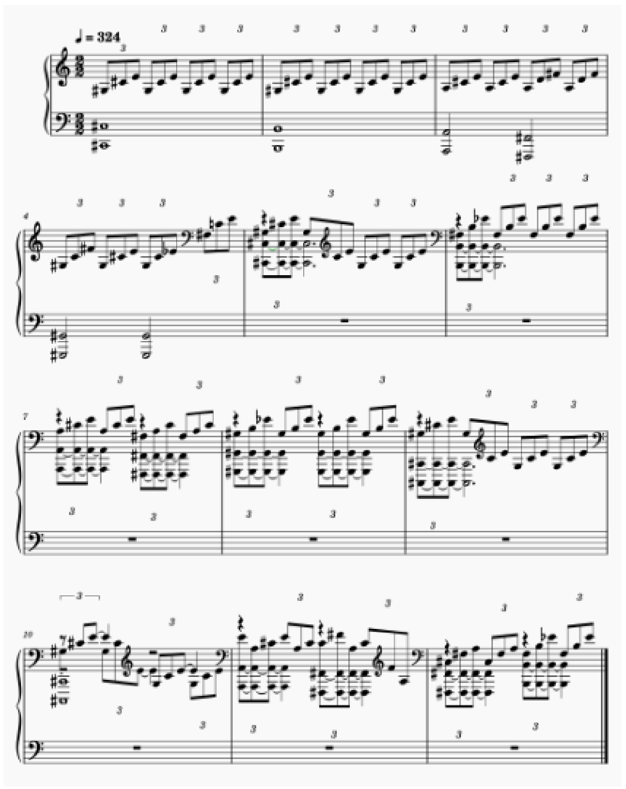


Fig. 6. Beethoven's Sonata No. 14 in C Minor extended by our AI composition assistant. Extension starts in measure 5. The system maintains Beethoven's style, but suffers from repeating figures too often and sounding repetitive.

VI. Summary and Future Work

The trajectory of our project remained largely unchanged throughout the quarter. Our end goal of developing a system that could compose an extended version of an input MIDI file was never changed; however, our most notable shifts during the project were the focus of our KB and the design of our agentic AI system. We had anticipated implementing 100-200 comprehensive rules centered on detailed music theory principles, but our experiments showed that fewer, more focused rules created better results. Additionally, including a dedicated Handler agent was one of the key developments of our agentic system that made it into an effective composer.

Our findings indicate that LLMs are able to compose music effectively and compete with existing, non-LLM-based tools especially with regard to stylistic continuity. When combined with a KB focused on prioritizing the stylistic continuity of the existing piece, the generated musical quality is actually comparable to that of the original human composer. We found that prioritizing stylistic continuity led to much better performance than focusing on pure theory-based constraints for our KB, and our multi-agent design was particularly effective to allow each of the individual agents to specialize in specific tasks of the composition process.

Given more time, there are plenty of ways that the system could be improved. The biggest of which continues to be reducing runtime latency. While generation times averaging between 5 and 10 minutes for 8 measures of complex extension is more or less acceptable, with more input to analyze and more measures to generate, the time taken to generate music will increase significantly; this limits the system's practicality and severely limits composition extension capabilities when it comes to larger compositions. The current model's context window also restricts effective stylistic coherence to about 12 measures, which seems to still pose challenges for generating longer musical extensions. We think that a future approach, which uses hierarchical "section-level" planning after the measure-level planner, could increase long-range compositional coherence. This would necessitate a much more complex agentic system. Our current implementation is also limited to solo piano. It would be interesting to see how it could expand to different instrumentation, including chamber music (string quartets, etc.), wind ensembles, or full symphonies. This, however, would likely first require runtime improvements, since adding instrumentation is linearly proportional to system runtime.

Ultimately, our system demonstrates the class insight that hybrid systems which combine explicit rules via a knowledge base with deep-learning approaches like LLMs can generate higher-quality, more stylistically faithful music extensions than purely deep-learning or pure rule-based systems.

References

[1] M. Allan and C. K. I. Williams, "Harmonising chorales by probabilistic inference," in *Advances in Neural Information Processing Systems*, vol. 17, 2005.

[2] C.-Z. A. Huang, A. Vaswani, J. Uszkoreit, I. Simon, C. Hawthorne, N. Shazeer, A. M. Dai, M. D. Hoffman, M. Dinculescu, and D. Eck, "Music transformer: Generating music with long-term structure," *arXiv preprint arXiv:1809.04281*, 2018.

[3] "Mubert," https://mubert.com, accessed: 2025-01-26.

[4] "Staccato ai," https://staccato.ai/, accessed: 2025-01-26.

[5] "Aiva ai," https://www.aiva.ai/, accessed: 2025-01-26.

[6] "Beethoven ai," https://www.beatoven.ai/, accessed: 2025-01-26.