International Journal of Secondary Computing and Applications Research

Volume 1, Issue 1

EDITOR-IN-CHIEF: DR. MARIA HWANG

DECEMBER, 2024

International Journal of Secondary Computing and Applications Research, Volume 1, Issue 1

December 19, 2024

Letter from the Editor-in-Chief

Welcome to the inaugural issue of the *International Journal of Secondary Computing and Applications Research*. This volume showcases the excellent work of researchers in our field. Each article reflects the growing diversity and depth of topics that define modern computing.

We hope these papers will inspire new ideas, foster collaboration, and further research among high school students. Our goal is to raise awareness and interest in the potential of high school students to bring serious research to the field of computing. As the editor, I am delighted to present this issue as a testament to the hard work and dedication of the students who contributed to it, as well as to the future impact of high school students in computing research.

Enjoy reading, and we look forward to your feedback.

Sincerely,
Maria Hwang
Editor-in-Chief

Volume 1, Issue 1 December 19, 2024
DOI: 10.5281/zenodo.14279457 | `https://ijscar.org/pubs/volume1/issue1`

Contents

Leveraging LLMs for Automated MIDI Generation

Corey Zhang
Eastlake High School
Sammamish, WA, USA
1043437@lwsd.org

Mark Santolucito
Barnard College, Columbia University
New York, USA
msantolu@barnard.edu

Abstract

This paper explores the application of large language models (LLMs) for music generation, specifically focusing on generating MIDI files, JSON representations, and music-related code. We aim to leverage LLM capabilities to automate and enhance the music composition process. We systematically examine methodologies and tools for integrating LLMs in music generation with direct music creation and code-based techniques. The challenge of automating music composition using AI remains significant due to the complexity and creativity required in the process. Our findings demonstrate the potential of LLMs to innovate and streamline music composition, offering new tools and approaches for musicians and developers.

Keywords

large language models, music generation, code generation

ACM Reference Format:
Corey Zhang and Mark Santolucito. 2024. Leveraging LLMs for Automated MIDI Generation. In *Proceedings of International Journal of Secondary Computing and Applications Research (IJSCAR VOL. 1, ISSUE 1)*. ACM, New York, NY, USA, 8 pages. https://doi.org/10.5281/zenodo.14279480

1 Introduction

Music composition traditionally requires significant expertise and creativity, making it a time-consuming and challenging endeavor. However, the advent of artificial intelligence (AI) and natural language processing (NLP) has introduced novel possibilities for automating aspects of music composition. Despite these advancements, the application of AI in music composition is still in its nascent stages, revealing a substantial gap in effectively harnessing these technologies for creative fields like music. Key areas for improvement include enhancing AI's ability to generate complex and creative compositions, developing tools for human-AI collaboration, creating user-friendly interfaces, increasing algorithmic efficiency and speed, and ensuring the quality and originality of AI-generated music. Addressing these gaps can democratize music creation and expand the horizons of musical innovation.

LLMs developed by OpenAI, such as GPT-3.5, have demonstrated impressive capabilities in generating structured data formats, including text and code [17]. LLMs offer the potential to pioneer new interfaces for musical expression by enabling the integration of powerful AI systems into networked devices via API calls. Accessing the capabilities of LLMs through simple network connections can facilitate novel interactions with music, such as real-time music generation and innovative engagement in live performance contexts. However, a significant challenge in leveraging LLMs for music generation lies in the insufficient understanding of how to effectively utilize these models to produce music that is both coherent and engaging.

This research explores the potential of these models to generate music in MIDI and JSON formats, introducing an innovative approach to music creation. The need for this research is driven by the opportunity to improve the efficiency and creativity of music composition through AI, addressing the current limitations, and exploring new applications of LLMs in the music domain.

We identify the primary contributions of this work as follows:

- Development and evaluation of a framework for integrating large language models in music generation, utilizing the OpenAI API for both direct and code-based music generation in MIDI and JSON formats.
- Identification of benchmark evaluation prompts and the assessment of generated outputs based on three key metrics: correctness, compliance, and complexity, ensuring the creation of valid, standard-compliant, and sophisticated musical compositions.
- Conducting a comparative analysis between LLM-generated music and traditional code-based music generation, providing insights into the effectiveness of these models and identifying potential areas for enhancement in AI-assisted music composition.

By explicitly developing a comparative analysis and introducing new benchmark metrics, this work extends beyond previous studies that primarily focused on either subjective evaluations or singular methods, thereby providing a more comprehensive contribution to the field of AI-driven music generation. This research seeks to advance the field, providing a foundation for future developments and applications in both computer science and the arts.

2 State of the Art

2.1 Background

The Musical Instrument Digital Interface (MIDI) is a standardized protocol used for communication between electronic musical instruments and computers. MIDI files contain musical information such as note pitch, duration, velocity, and timing, allowing for playback across any compatible device. Unlike traditional audio files, which store sound waves, MIDI files store instructions for sound production, offering significant flexibility in musical applications [12].

JavaScript Object Notation (JSON) is a lightweight, text-based data interchange format renowned for its human readability and widespread use in web applications. Within the realm of music

IJSCAR VOL. 1, ISSUE 1

technology, JSON represents intricate musical structures, encompassing notes, chords, rhythms, and dynamics. This structured format enhances software applications' ability to process musical information, enabling detailed explorations and manipulations of musical data [1].

Early AI-based music composition methods primarily involved rule-based systems and algorithmic composition, which often led to repetitive and predictable outputs due to their rigid structures. Recent advancements have shifted towards machine learning techniques, offering more flexibility and creativity in music generation. JSON offers significant advantages in this context, as it can be used to create complex musical structures by precisely defining musical elements such as tempo changes, dynamic markings, and articulation patterns, which can then be easily manipulated and interpreted by software. This detailed representation allows for more nuanced and sophisticated compositions compared to earlier methods. The ability to convert JSON data into other formats, such as MIDI, further enhances its utility in music technology, allowing for seamless integration into various musical applications. Integrating MIDI and JSON within musical applications leverages MIDI's encoding of musical attributes and JSON's capacity to represent complex structures. This combination enhances music software's capabilities for composing, editing, and analyzing musical compositions. MIDI's real-time performance instructions and JSON's structured representation enable innovative approaches to music creation and analysis, advancing digital music technologies [2].

LLMs, such as OpenAI's GPT-3.5, leverage extensive textual data to exhibit proficiency in generating coherent and contextually relevant text across diverse domains. These models transcend traditional text generation tasks, demonstrating competence in code synthesis, translation, and summarization. Moreover, LLMs' capability to comprehend and manipulate structured data positions them as viable tools for innovative applications in the arts, particularly in domains such as music composition. The inherent versatility of LLMs stems from their training on vast corpora, enabling them to interpret and generate complex outputs from natural language prompts. This ability holds promise for transformative applications in creative fields, including music composition, where LLMs can facilitate the synthesis of elaborate musical compositions based on textual descriptions. In the realm of computational creativity, LLMs represent a paradigm shift, offering computational methodologies that bridge the gap between textual prompts and creative outputs in domains traditionally associated with human creativity. By harnessing the latent potentials embedded in their training data, these models enable novel approaches to artistic expression and innovation. The integration of LLMs into creative processes not only expands the scope of computational tools available to musicians but also fosters interdisciplinary collaborations between computer science and artistic endeavors [3].

2.2 Related Work

The application of AI in music composition has evolved significantly, encompassing diverse methodologies and technologies. Early efforts centered on employing predefined rules and algorithms to generate musical sequences. Recent advancements have leveraged machine learning techniques, particularly deep learning models, to analyze and synthesize music. Deep learning, specifically recurrent neural networks (RNNs) and convolutional neural networks (CNNs), has emerged as a powerful tool for music generation. These models learn intricate patterns from extensive datasets of musical compositions. Projects like Google's Magenta and Sony's Flow Machines are prominent applications of deep learning in producing music ranging from simplistic melodies to complex compositions [5, 7]. Transformer architectures, such as OpenAI's MuseNet and Jukedeck, represent a notable advancement in music composition using AI. These models excel in capturing long-range dependencies within musical sequences, enabling the generation of diverse and cohesive musical pieces across various styles and genres [9, 20].

While there has been extensive research on generating music using AI, the specific use of JSON and MIDI formats in conjunction with LLMs is relatively new. Previous works have explored the generation of MIDI files directly from neural networks, but the integration of JSON as an intermediate representation for music generation adds a new dimension of flexibility and control [8, 10]. Research by Huang et al. (2018) on Music Transformer [11] and by Donahue et al. (2019) on LakhNES [4] have focused on generating MIDI files directly from AI models. These works highlight the feasibility of using deep learning models to create musically coherent MIDI sequences. JSON's use in music representation remains underexplored but will be further utilized for structured and detailed musical composition.

The application of LLMs in creative domains, such as text generation and code synthesis, has paved the way for their use in music composition. LLMs can generate creative content, including poetry, stories, and even programming code, leveraging their understanding of contextual prompts and structural nuances [14]. OpenAI's Codex model, a descendant of GPT-3, has demonstrated the capability to generate code snippets based on natural language descriptions. This ability is directly applicable to generating music-related code, such as scripts for creating MIDI files or JSON representations of musical pieces [18]. Projects like OpenAI's MuseNet have showcased that LLMs can generate music by leveraging their understanding of musical structure and styles. MuseNet can create compositions in various genres, demonstrating the potential of LLMs to contribute to music generation tasks [16].

3 Methodology

3.1 Generation Methods

We explored three generation methods for producing MIDI scores in JSON format. Specifically, we look into Direct Generation, Code Generation, Rich Code Generation. Each of these approaches demonstrate different design choices in structuring an LLM-powered generative system.

Direct Generation involves producing MIDI scores in a JSON format that can be directly utilized without further transformation or interpretation. This method offers the advantage of providing immediate output without intermediary steps, thereby reducing the potential for errors that could arise during format translation. However, it comes with limitations, such as restricted post-generation adjustment capabilities, which may hinder fine-tuning and adaptation to specific musical requirements. Ensuring the quality of output may also necessitate more rigorous evaluation processes to validate correctness and musical integrity.

Code Generation refers to the production of executable code, such as JavaScript, which generates MIDI scores. This approach provides enhanced control over customization and manipulation of musical elements post-generation. Modifications to the generated code can be made to optimize performance or tailor compositions to different stylistic preferences. Nevertheless, it introduces the risk of errors in code execution, particularly if scripts are not meticulously crafted or if unexpected issues arise during runtime. Proficiency in coding is essential to effectively modify scripts and ensure they meet the desired musical criteria.

Rich Code Generation involves the creation of sophisticated scripts that incorporate complex data structures and detailed instructions for generating intricate musical pieces. This method excels in providing a higher level of musical nuance and detail, facilitated by annotations that enhance understanding and modification of the generated code. Despite this, the increased complexity in both generation and execution phases demands substantial computational resources. Execution may require advanced hardware capabilities to handle the processing demands of complex algorithms and extensive musical data structures effectively.

3.2 Evaluation Metrics

In this section, we introduce three custom evaluation metrics - Correct, Conceptual Correctness, Compiled, and Complexity - developed specifically for assessing the generated MIDI pieces in this study. These metrics are not standard in the research space but were created to better evaluate the unique aspects of LLM-generated music. Below, we explain each metric and provide justification for its creation.

The Correct metric measures how well the generated MIDI pieces adhere to predefined criteria, assigning a binary value (0 or 1) depending on whether the output satisfies these criteria. This metric provides a straightforward way to compare the quality of outputs across a dataset, ensuring consistency in evaluation. However, subjectivity in defining what constitutes “correctness” may introduce variability in ratings, which could affect the reliability of assessments. Additionally, the Correct metric does not consider whether the generated music successfully compiles; it focuses solely on qualitative aspects of the compositions. It is best suited for scenarios with clear, well-defined criteria and minimal subjective bias, where multiple evaluators can assess adherence to compositional rules.

Additionally, we introduce an additional metric, Conceptual Correctness, which addresses the limitations of traditional correct metrics when applied to Rich Code Generation. While traditional correctness metrics evaluate melodic and harmonic coherence, rhythmic consistency, and tonal balance, these often fall short when assessing code generated through methods like randomization or iterative constructs (e.g., Math.random() and for loops). Conceptual Correctness assesses whether the generated code aligns with the compositional instructions—specifically randomization and iterative structures—rather than traditional musical qualities. This metric evaluates the fidelity of the code to the intended procedural structure.Traditional correctness metrics for evaluating Rich Code Generation, such as melodic and harmonic coherence, rhythmic consistency, and tonal balance, often fall short and yield very low results when assessing code generated through methods involving randomization and iterative constructs like `Math.random()` and `for` loops. The nature of these data structures can lead to the creation of incoherent and extremely dissonant music, which may not align with traditional musical conventions. For example, when using `Math.random()` to generate pitches and durations, there is no guarantee that the resulting musical phrases will adhere to recognized melodic or harmonic patterns. Similarly, employing `for` loops to iterate over musical elements may produce repetitive or non-musical sequences that lack expressive or structural integrity.

To address these limitations, we propose a modified metric: Conceptual Correctness. This metric assesses whether the generated code adheres to the compositional instructions involving randomization and iterative structures, rather than focusing solely on traditional musical qualities. Conceptual Correctness ensures that the generated music aligns with the intended compositional approach, evaluating the presence of recognizable musical forms, specified harmonic progressions, rhythmic patterns, and overall coherence and expressiveness.

Consider a prompt instructing the generation of a piece using random pitches within a specified scale and repeated rhythmic patterns via `for` loops. Traditional metrics might rate the output poorly due to lack of harmonic progression or melodic development. However, Conceptual Correctness would evaluate if the generated code correctly uses randomization to select pitches from the specified scale and employs `for` loops to create the repetitive patterns as described. For instance, if the prompt specifies generating a melody using random pitches from a C major scale and repeating a rhythmic pattern four times, the traditional metrics might penalize the output for not having a clear melodic contour or harmonic resolution. On the other hand, Conceptual Correctness would give a positive score if the code produces notes within the C major scale and correctly implements the repetitive rhythmic structure.

Using the same binary assessment as correct, Conceptual Correctness determines if the generated code qualifies as rich code per the prompt’s intentions. A score of 1 indicates the code meets the criteria for Rich Code Generation, while a score of 0 indicates it does not. This metric prioritizes procedural fidelity and structural guidelines over traditional musical aesthetics, providing a nuanced evaluation of Rich Code Generation methods.
Example 1: Passing Conceptual Correctness Prompt: Generate MIDI scores in JSON notation that can be played back by Tone.js

Code:

```
function generateMIDIScore() {
  const tracks = [];
  for (let i = 0; i < 4; i++) {
    const notes = [];
    for (let j = 0; j < 4; j++) {
      notes.push({
        pitch: Math.floor(Math.random() * 1000)
        + 100,
        duration: Math.floor(Math.random() * 4)
        + 1,
        startTime: Math.floor(Math.random() * 4)
```

```
      });
    }
    tracks.push({ notes });
  }
  return { tracks };
}

generateMIDIScore();
```

This example passes the Conceptual Correctness criteria because it adheres to the prompt's instructions by using `for` loops and `Math.random()` to generate the MIDI score. The generated code reflects the intended use of iterative constructs and randomization to produce musical elements, demonstrating rich Code Generation as specified.

Example 2: Failing Conceptual Correctness Prompt: Generate a MIDI JSON Bach chorale following counterpoint rules with four voices following a chord progression and counterpoint harmony rules

Code:

```
function generateMIDIScore() {
  const midiScore = {
    tracks: [
      {
        notes: [
          { pitch: Math.floor(Math.random() *
          (1000 - 100)) + 100, duration:
          Math.floor(Math.random() * 4) + 1,
          startTime: 0 },
          { pitch: Math.floor(Math.random() *
          (1000 - 100)) + 100, duration:
          Math.floor(Math.random() * 4) + 1,
          startTime: 1 },
          // ... more notes for voice 1
        ]
      },
      ...
}

generateMIDIScore();
```

This example fails the Conceptual Correctness criteria because the segment of the generated code shown below does not fully adhere to the prompt's instructions. Although it generates a MIDI score with four voices, it does not follow counterpoint rules, chord progressions, or harmony rules as specified. Additionally, it lacks the use of iterative constructs like `for` loops to generate the musical elements, and the comment within the code indicates incomplete implementation. Thus, it does not meet the criteria for rich Code Generation as intended by the prompt.

The Compiled metric refines the Correct metric by evaluating only those MIDI pieces that compile successfully into executable outputs, also assigning a binary value (0 or 1) based on compilation success. This metric enhances practical usability assessment by ensuring that the evaluated pieces are not only correct in structure but also executable in real-world applications. However, Compiled may overlook high-quality pieces that fail compilation due to minor, non-musical errors, which could reduce its utility in scenarios where frequent compilation issues arise. This metric is ideal for evaluating the practical execution of music alongside its aesthetic or structural quality.

The Complexity metric quantifies the intricacy of the generated music by measuring the number of individual notes in the MIDI representation. This metric offers an objective measure of musical detail and variation, correlating with the creative richness of compositions. However, high Complexity does not always equate to superior quality or musicality, as excessively complex pieces can be chaotic and less aesthetically pleasing. Therefore, Complexity is best used in conjunction with other metrics to ensure that increased intricacy enhances the overall musical experience rather than detracting from it.

Each of these metrics provides a distinct perspective on the evaluation of generated MIDI music, forming a comprehensive assessment framework. Correct serves as a foundational criterion, assessing alignment with compositional rules or intentions. Compiled focuses on practical usability by evaluating whether the output is executable. Complexity adds depth by measuring the intricacy and variation within the compositions. Together, these metrics offer a nuanced understanding of the strengths and limitations of different generation models, providing insights into both qualitative and practical aspects of generated MIDI music.

3.3 Evaluation Process

We used three specific prompts to generate MIDI scores in JSON format. Each prompt was tested across 10 trials:

Prompt 1: Generate MIDI scores in JSON notation that can be played back by Tone.js

> This baseline prompt tests the model's ability to generate MIDI scores compatible with Tone.js, evaluating its most fundamental capability for practical application in web-based music playback and interactive environments.

Prompt 2: Generate a MIDI JSON Bach chorale adhering counterpoint rules with four voices following a chord progression and counterpoint harmony rules

> This prompt assesses the model's skill in generating standard four-part structure adhering to strict counterpoint rules, gauging its proficiency in classical music composition.

Prompt 3: Generate a MIDI JSON complex piano piece with two hands with syncopated rhythms and varied notes

> This prompt challenges the model to create more sophisticated, multi-layered music, demonstrating its capacity for intricate musical composition and rhythmic complexity.

Each of the three prompts was executed 10 times using each generation method. This means there were a total of 30 trials per method, resulting in 90 trials in total. For each trial:

(1) The generation method was applied to the prompt.
(2) The output MIDI score in JSON format was evaluated based on predefined metrics (correctness, compilation success, and complexity).

(3) The results were recorded and analyzed to determine the performance and consistency of each method.

For generating the MIDI scores, we used the OpenAI API's gpt-3.5-turbo model for all three methods [17].

Our evaluation methodology investigates the generation of MIDI scores using LLMs across three methods: Direct Generation, Code Generation, and Rich Code Generation. We evaluated the generated outputs using the Correct, Compiled, and Complexity metrics, providing a comprehensive analysis of the effectiveness and intricacy of the generated music and code. All evaluation results are derived from averaging the scores obtained across 10 trials to provide a comprehensive analysis of the musical and coding outputs.

4 Results

I provide here an analysis of my evaluation through the data I collected. I provide all code for the LLM music generator, Code Generations, and MIDI files in JSON format for Direct Generation and MIDI files as JS Code Generation JSON format for code and rich Code Generation in a Github repository at https://github.com/kazado/tonejsmidijson in the JSON MIDI folder. The latest version of the project may be used here: https://kazado.github.io/tonejsmidijson/.

In this evaluation, we measure three dimensions of our generational approach to assess our system: Correct, Compiled, and Complexity, along with the Conceptual Correctness metric that we introduced to address the limitations of standard evaluation methods. Each of these metrics offers a unique perspective on how well the generation methods produce meaningful and usable musical outputs.

4.1 Correct

The Correct metric evaluates how well the generated MIDI pieces adhere to predefined musical criteria, such as harmonic coherence, rhythmic consistency, and tonal balance. This is an important metric to ensure that the output meets basic expectations for musicality and structure.

Correct	Generation Methods		
	Direct Generation	Code Generation	Rich Code Generation
Prompt 1	0.5	0.9	0.2
Prompt 2	0.7	0.6	0
Prompt 3	0.7	0.5	0

Table 1: Comparison of calculated results for the three generation methods for each of the three prompts for Correct benchmark.

Direct Generation shows moderate consistency across prompts, achieving scores ranging from 0.5 to 0.7. Code Generation performs well with a high score of 0.9 for one prompt but exhibits variability, scoring lower on others (0.6 and 0.5). Rich Code Generation consistently scores lower across all prompts (0.2, 0, and 0), indicating challenges in producing musically accurate outputs using complex data structures. These results suggest that while Code Generation excels in certain scenarios, all methods require refinement to enhance overall correctness in MIDI score generation.

4.2 Conceptual Correctness

The Conceptual Correctness metric measures whether the generated code adheres to the procedural logic and structural guidelines outlined in the prompt. This metric is particularly useful for evaluating outputs generated using iterative constructs and randomization, which may not conform to traditional musical qualities but align with the intended compositional approach.

Prompt	Complexity
Prompt 1	0.6
Prompt 2	0.8
Prompt 3	1

Table 2: Comparison of calculated Conceptual Correctness results for each of the three prompts.

The results presented in **Table 2** underscore the need for a more context-sensitive approach, such as Conceptual Correctness, to effectively evaluate Rich Code Generation techniques.

4.3 Compiled

The Compiled metric assesses whether the generated outputs can successfully compile into executable MIDI scores. This is essential for ensuring that the generated music is usable in real-world applications without errors during playback or execution.

Compiled	Generation Methods		
	Direct Generation	Code Generation	Rich Code Generation
Prompt 1	0.8	0.9	0.8
Prompt 2	0.7	0.6	0.9
Prompt 3	0.7	0.6	1

Table 3: Comparison of calculated results for the three generation methods for each of the three prompts for Compiled benchmark.

Prompt 3 achieves the highest scores across all methods, with Rich Code Generation achieving a perfect score of 1, indicating optimal compilation success in generating MIDI scores. Prompt 2 also performs well, particularly with Rich Code Generation, achieving a score of 0.9, highlighting the method's capability to consistently produce correctly compiled outputs. Prompt 1 shows strong performance overall, with Code Generation achieving the highest score of 0.9, closely followed by Direct Generation and Rich Code Generation at 0.8. These results showcase the effectiveness of Rich Code Generation in ensuring accurate compilation of MIDI scores across diverse compositional prompts.

4.4 Complexity

The Complexity metric quantifies the intricacy of the generated music by counting the number of individual notes, rhythmic patterns, and the overall variation in the composition. Higher complexity may indicate a more sophisticated output but does not necessarily correlate with better musical quality.

The results indicate varying levels of complexity generated by each method for different prompts. Rich Code Generation consistently produces the most complex musical outputs across all

Complexity	Generation Methods		
	Direct Generation	Code Generation	Rich Code Generation
Prompt 1	6.3	6.4	13.5
Prompt 2	16.7	15.7	40.4
Prompt 3	12	9.4	52.4

Table 4: Comparison of calculated results for the three generation methods for each of the three prompts for Complexity benchmark.

prompts, with scores of 13.5, 40.4, and 52.4 for Prompts 1, 2, and 3, respectively. This method leverages detailed instructions and complex data structures to introduce nuanced musical elements, resulting in compositions that exhibit higher levels of intricacy and sophistication. In contrast, Direct Generation and Code Generation exhibit lower complexity scores, indicating less intricate musical compositions. These findings show the capability of Rich Code Generation to generate musically complex outputs, demonstrating its potential for creating sophisticated musical compositions that align closely with predefined complexity metrics.

4.5 Summary of Findings

The analysis reveals significant performance variation across generation approaches. Direct Generation produced cohesive sequences but struggled with length and pitch. Code Generation, while functional, often resulted in harmonic clashes and simplistic structures. Rich Code Generation provided complexity and length but compromised musicality and coherence. The results necessitate the evaluation of multiple dimensions of generated code, including harmonic consistency, melodic structure, duration, and pitch range, to develop robust and efficient code generation systems.

4.6 Evaluations

The Direct Generation approach revealed several recurring issues in the generated sequences. Length was a significant concern, as sequences were often excessively short, typically spanning only four beats. Harmonic consistency was generally maintained within a C major chord framework, with both voices playing scales—one starting from C and the other from G—creating a cohesive harmonic structure. However, these sequences occasionally diverged from traditional chord patterns. Voice interaction showed that while the voices generally complemented each other, note durations were overly prolonged, limiting sequences to just four notes. Chords from common keys, such as C major and E minor, were used, with note durations of either one or two beats. Yet, there were problems with numerical pitches leading to odd-sounding low-pitched notes. A consistent duration and pitch range with increasing pitch frequency were observed, but sequences often deviated from typical chord patterns, using a scale structure with random pitch increments. The chord progression typically followed a I-I-ii-V pattern in C major, although some deviations, like moving from a C major chord to an A minor chord, introduced interesting tensions but were not always diatonic to the key.

The evaluation of Code Generation focused on both functionality and musicality. Functionally, the code reliably ran a function containing the MIDI score in JS format, playable in Tone.js, and returned the object for playback. Musically, a simple C major chord structure was common, with one voice playing an ascending C major scale and the other a descending F dominant 7th chord. Despite moments of harmonic dissonance, interactions between voices generally resonated, though sustained note durations restricted sequences to merely four notes. Some sequences included a 2+ octave scale with varied note durations (0.5, 1, 1.5, 2 beats), and voices often used notes from common chords with C major as the tonic. The generated sequences displayed a I-I6-V-IV7 chord progression in C major, but these progressions sometimes resulted in dissonance and clashing chords. Additionally, some compositions were notably shorter, often containing only one chord.

Rich Code Generation produced more complex and varied results but also introduced several challenges. The generated melodies were often extremely dissonant, with no variation in note duration or discernible trends in pitches, a dissonance that persisted in both singular and dual melodies. The pitch range could be altered through prompt engineering, particularly the numerical ranges used. The use of for loops and Math.random() enabled more complex and longer compositions. Despite this complexity, the melodies often lacked a common theme, with two voices failing to harmonize. Some sequences started with dissonant melodies but branched into multiple voices, showing moderate success in replicating two-voice structures typical in piano music. There was minimal variation in note duration, and no clear trends or structures in the pitches. In some instances, the notes' durations were strictly syncopated, adding rhythmic interest but maintaining harmonic dissonance. The generated sequences did not accurately represent the structure of classical music or Bach chorales, as they lacked the expected harmonic and melodic patterns.

4.7 Summary of Findings

The analysis reveals significant performance variation across generation approaches. Direct Generation produced cohesive sequences but struggled with length and pitch. Code Generation, while functional, often resulted in harmonic clashes and simplistic structures. Rich Code Generation provided complexity and length but compromised musicality and coherence. The results necessitate the evaluation of multiple dimensions of generated code, including harmonic consistency, melodic structure, duration, and pitch range, to develop robust and efficient code generation systems.

5 Discussion

In this section, we explore the potential applications of our findings to the development of LLM-assisted tools for music generation and composition, and outline future research directions in this domain.

5.1 Enhancing Music Composition Tools

The insights offered from our evaluation of LLM-generated music sequences can significantly enhance existing music composition tools. By understanding the strengths and limitations of different generation approaches, developers can create more robust and user-friendly interfaces that empower composers to produce music more efficiently and creatively. For instance, integrating direct generation methods can provide users with quick and intuitive means to generate musical ideas, while incorporating code generation techniques can offer more granular control over the composition process [22].

5.2 Human-Centered Evaluation

A major limitation of this study is the lack of comprehensive human-centered evaluation, particularly involving feedback from musicians and composers. Such evaluation is critical for validating the practical utility and creative potential of LLM-generated music. Future research should focus on conducting user studies with professional musicians and composers to gather insights on how these tools can fit into the creative process. Additionally, user feedback can help fine-tune LLMs to meet the specific needs and preferences of human composers, ensuring the generated music aligns with artistic goals and enhances collaborative workflows.

5.3 Exploring New Musical Styles and Genres

The flexibility and adaptability of LLMs open exciting possibilities for exploring new musical styles and genres. Training models on diverse datasets that span different musical traditions and cultures can foster cross-cultural exchange and collaboration, potentially leading to the emergence of innovative hybrid genres and experimental compositions. Moreover, incorporating user feedback and preferences into the training process can enable LLMs to adapt and evolve over time, reflecting changing trends and tastes in the music industry [6].

5.4 Ethical Considerations

As AI-generated music becomes more widespread, several ethical concerns must be addressed. First, there are questions surrounding intellectual property and originality in AI-generated compositions. Determining the ownership of music created by LLMs and the potential overlap with existing compositions remains a complex issue. Furthermore, the use of LLMs raises concerns about the displacement of human composers in various sectors of the music industry. It is essential to ensure that AI augments human creativity rather than replacing it. Additionally, the biases present in the training data used for these models could influence the music they generate, potentially leading to cultural insensitivity or reinforcing stereotypes. Future work must focus on ensuring that the training data is diverse and representative of various musical traditions to avoid perpetuating such biases. Addressing these ethical concerns is crucial to fostering a responsible and inclusive approach to AI in music generation.

5.5 AI Music Cognition

AI in music within therapeutic contexts offer personalized interventions by analyzing patients' responses to music, enhancing treatment for conditions like anxiety and depression. Real-time adaptation based on biometric data optimizes therapy effectiveness, while expanding accessibility through mobile platforms benefits remote and immobile populations. AI complements traditional methods by providing data-driven insights and improving therapist-patient interactions [21].

5.6 Future Directions

Several promising avenues for future research in LLM-assisted music generation can be identified. First, utilizing JSON as an intermediary step in music generation can enhance control over the composition process, incorporating domain-specific knowledge and optimizing JSON-to-MIDI conversion to generate music that transcends conventional stylistic boundaries. Integrating additional modalities such as lyrics, images, and videos can enrich the music generation process, creating more immersive and interactive user experiences. Developing collaborative composition tools that allow multiple users to interact with LLM-generated content in real-time can facilitate remote collaboration among musicians and composers. Addressing ethical considerations such as bias, fairness, and cultural representation is crucial to ensure that LLM-generated music is inclusive and respectful of diverse communities and perspectives. Lastly, engaging end-users, including musicians, composers, educators, and music enthusiasts, in the design and evaluation of LLM-assisted tools is essential for creating solutions that meet their needs and preferences [13, 15, 19].

By pursuing these research directions and fostering interdisciplinary collaborations between experts in machine learning, musicology, cognitive science, and human-computer interaction, we can unlock the full potential of LLMs to revolutionize music creation and appreciation in the digital age.

References

[1] [n. d.]. JSON: JavaScript Object Notation. Available at https://www.json.org/.

[2] Adriano Baratè and other authors. 2016. Advances and perspectives in web technologies for music representation. *DigitCult-Scientific Journal on Digital Cultures* 1, 2 (2016), 1–18.

[3] Tom B. Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, Sandhini Agarwal, Ariel Herbert-Voss, Gretchen Krueger, Tom Henighan, Rewon Child, Aditya Ramesh, Daniel M. Ziegler, Jeffrey Wu, Clemens Winter, Christopher Hesse, Mark Chen, Eric Sigler, Matt Litwin, Scott Gray, Jack Chess, Jack Clark, Christopher Berner, Sam McCandlish, Alec Radford, Ilya Sutskever, and Dario Amodei. 2020. Language Models are Few-Shot Learners. *Advances in Neural Information Processing Systems* 33 (2020), 18714–18732.

[4] Florian Colombo, Natasha Chen, Chris Donahue, Zachary Li, and Bryan Pardo. 2019. LakhNES: Improving multi-instrumental music generation with cross-domain pre-training. *Transactions of the International Society for Music Information Retrieval (TISMIR)* 2, 1 (2019), 97–108.

[5] Alexandre DuBreuil. 2020. *Hands-On Music Generation with Magenta: Explore the Role of Deep Learning in Music Generation and Assisted Music Composition.* Packt Publishing.

[6] Elena V. Epure, Guillaume Salha, Manuel Moussallam, and Romain Hennequin. 2020. Modeling the Music Genre Perception across Language-Bound Cultures. In *Proceedings of the 2020 Conference on Empirical Methods in Natural Language Processing (EMNLP)*, Bonnie Webber, Trevor Cohn, Yulan He, and Yang Liu (Eds.). Association for Computational Linguistics, Online, 4765–4779. https://doi.org/10.18653/v1/2020.emnlp-main.386

[7] Franco Ghedini, François Pachet, and Pierre Roy. 2016. Creating Music and Texts with Flow Machines. In *Multidisciplinary Contributions to the Science of Creative Thinking*, Giovanni Corazza and Sergio Agnoli (Eds.). Springer, Singapore. https://doi.org/10.1007/978-981-287-618-8_18

[8] Gaëtan Hadjeres, François Pachet, and Frank Nielsen. 2017. DeepBach: a steerable model for Bach chorales generation. In *Proceedings of the 34th International Conference on Machine Learning*. 1362–1371.

[9] Dorien Herremans and Cheng-Hao Chuan. 2020. The emergence of deep learning: new opportunities for music and audio technologies. *Neural Computing and Applications* 32 (2020), 913–914. https://doi.org/10.1007/s00521-019-04166-0

[10] D. Herremans, C.-H. Chuan, and E. Chew. 2017. A functional taxonomy of music generation systems. *Comput. Surveys* 50, 5 (2017), 1–30.

[11] Cheng-Zhi Anna Huang, Ashish Vaswani, Jakob Uszkoreit, Noam Shazeer, Curtis Hawthorne, Andrew M. Dai, Matthew D. Hoffman, and Douglas Eck. 2018. Music Transformer: Generating Music with Long-Term Structure. In *Proceedings of the International Conference on Machine Learning (ICML)*. 2410–2419.

[12] David Miles Huber. 2020. *Modern Recording Techniques* (9th ed.). Routledge. 225–259 pages.

[13] Eija Kaasinen et al. 2012. A User-Centric View of Intelligent Environments: User Expectations, User Experience and User Role in Building Intelligent Environments. *Computers* 2, 1 (2012), 1–33.

[14] Li-Chun Lu et al. 2024. LLM Discussion: Enhancing the Creativity of Large Language Models via Discussion Framework and Role-Play. *arXiv preprint arXiv:2405.06373* (2024).

[15] Michele Newman, Lidia Morris, and Jin Ha Lee. 2023. Human-AI Music Creation: Understanding the Perceptions and Experiences of Music Creators for Ethical and Productive Collaboration. In *Proceedings of the International Society for Music Information Retrieval Conference (ISMIR)*.

[16] OpenAI. 2019. OpenAI MuseNet. https://openai.com/index/musenet/.

[17] OpenAI. 2021. GPT-3.5 Turbo. https://beta.openai.com/docs/guides/gpt.

[18] OpenAI. 2021. OpenAI Codex. https://openai.com/index/openai-codex/.

[19] Sergio Oramas and et al. 2018. Multimodal Deep Learning for Music Genre Classification. *Transactions of the International Society for Music Information Retrieval* 1, 1 (2018), 4–21.

[20] Timothy Sexton. 2023. MuseNet. *Music Reference Services Quarterly* 26, 3–4 (2023), 151–153. https://doi.org/10.1080/10588167.2023.2247289

[21] Jingjing Sun, Jingyi Yang, Guyue Zhou, Yucheng Jin, and Jiangtao Gong. 2024. Understanding Human-AI Collaboration in Music Therapy Through Co-Design with Therapists. In *Proceedings of the CHI Conference on Human Factors in Computing Systems*.

[22] Yifei Zhang. 2024. Utilizing Computational Music Analysis And AI For Enhanced Music Composition: Exploring Pre- And Post-Analysis. *Educational Administration: Theory and Practice* 30, 5 (2024), 269–282. https://doi.org/10.53555/kuey.v30i5.2841

A Trials

Prompt	Trial	Correct		
		Direct Generation	Code Generation	Rich Code Generation
Prompt 1	Trial 1	1	1	0
	Trial 2	0	1	0
	Trial 3	0	0	0
	Trial 4	1	1	0
	Trial 5	1	1	0
	Trial 6	0	1	1
	Trial 7	1	1	1
	Trial 8	0	1	0
	Trial 9	0	1	0
	Trial 10	1	1	0
Prompt 2	Trial 1	1	1	0
	Trial 2	1	0	0
	Trial 3	0	0	0
	Trial 4	1	1	0
	Trial 5	0	0	0
	Trial 6	1	1	0
	Trial 7	1	1	0
	Trial 8	1	1	0
	Trial 9	0	0	0
	Trial 10	1	1	0
Prompt 3	Trial 1	0	1	0
	Trial 2	1	1	0
	Trial 3	0	0	0
	Trial 4	1	1	0
	Trial 5	1	1	0
	Trial 6	1	0	0
	Trial 7	1	0	0
	Trial 8	1	0	0
	Trial 9	0	0	0
	Trial 10	1	1	0

Table 5: Table of trials for the three prompts and ten trials each for Correct benchmark.

Prompt	Trials
Prompt 1	0
	1
	0
	1
	1
	1
	0
	1
	0
	1
Prompt 2	1
	1
	1
	0
	1
	1
	0
	1
	1
	1
Prompt 3	1
	1
	1
	1
	1
	1
	1
	1
	1
	1

Table 6: Table of trials for the three prompts and ten trials each for Conceptual Correctness benchmark.

Prompt	Trial	Direct Generation	Code Generation	Rich Code Generation
Prompt 1	Trial 1	1	1	1
	Trial 2	0	1	1
	Trial 3	1	0	0
	Trial 4	1	1	1
	Trial 5	1	1	1
	Trial 6	1	1	1
	Trial 7	1	1	1
	Trial 8	0	1	0
	Trial 9	1	1	1
	Trial 10	1	1	1
Prompt 2	Trial 1	1	1	1
	Trial 2	1	0	1
	Trial 3	0	0	1
	Trial 4	1	1	0
	Trial 5	0	0	1
	Trial 6	1	1	1
	Trial 7	1	1	1
	Trial 8	1	1	1
	Trial 9	0	0	1
	Trial 10	1	1	1
Prompt 3	Trial 1	0	1	1
	Trial 2	1	1	1
	Trial 3	0	0	1
	Trial 4	1	1	1
	Trial 5	1	1	1
	Trial 6	1	1	1
	Trial 7	1	0	1
	Trial 8	1	0	1
	Trial 9	0	0	1
	Trial 10	1	1	1

Table 7: Table of trials for the three prompts and ten trials each for Compiled benchmark.

Prompt	Trial	Direct Generation	Code Generation	Rich Code Generation
Prompt 1	Trial 1	6	4	6
	Trial 2	5	6	3
	Trial 3	6	8	12
	Trial 4	8	8	6
	Trial 5	8	6	22
	Trial 6	6	6	16
	Trial 7	4	8	5
	Trial 8	6	7	32
	Trial 9	8	6	24
	Trial 10	6	5	9
Prompt 2	Trial 1	16	16	64
	Trial 2	16	25	64
	Trial 3	16	16	16
	Trial 4	16	16	12
	Trial 5	23	8	64
	Trial 6	16	16	32
	Trial 7	16	16	8
	Trial 8	16	16	16
	Trial 9	16	12	64
	Trial 10	16	16	64
Prompt 3	Trial 1	24	16	100
	Trial 2	10	8	32
	Trial 3	24	8	32
	Trial 4	8	14	16
	Trial 5	12	8	32
	Trial 6	9	10	32
	Trial 7	8	8	16
	Trial 8	8	10	200
	Trial 9	8	4	32
	Trial 10	9	6	32

Table 8: Table of trials for the three prompts and ten trials each for Complexity benchmark.

Received 16 August 2024; accepted 24 September 2024

Deep Learning Based Volumetric Segmentation of Heart Ventricles for Assessment of Cardiac Disease Using MRI

Anika Pallapothu
anikap@novaltech.org
Novaltech
Hayward, CA, USA

Abstract

Diagnosis of cardiovascular diseases through cardiac MRI imaging plays a crucial role. Manual evaluation is time consuming and prone to errors. With the help of deep learning, a lot of traction has been developed for cardiac imaging diagnosis. In this study, we present a fully automated pipeline for the segmentation of left ventricle, right ventricle, myocardium, and classification of cardiovascular diseases into five classes using the cardiac MRI scans from the ACDC dataset. We adopted Segnet architecture for segmentation and made a comparative analysis using 2D and 3D approach. Best results were obtained using 2D approach with dice scores of 0.877(RV), 0.877(MYO), 0.937(LV) on the test set. We later on use the segmentation outputs to extract quantitative features to develop a robust classifier that gave us an overall accuracy of 85% on the test set and 0.81,0.89 scores of precisions and recall. Our proposed approach is computationally efficient and can be used for making critical decisions during diagnosis.

Keywords

Cardiovascular, MRI, Heart Ventricles, Segmentation, Artificial Intelligence, Deep Learning

ACM Reference Format:
Anika Pallapothu. 2024. Deep Learning Based Volumetric Segmentation of Heart Ventricles for Assessment of Cardiac Disease Using MRI. In *Proceedings of International Journal of Secondary Computing and Applications Research (IJSCAR VOL. 1, ISSUE 1).* ACM, New York, NY, USA, 4 pages. https://doi.org/10.5281/zenodo.14279490

1 Introduction

Cardiovascular diseases are one of the major factors that contribute towards the death globally as per WHO [12]. Cardiovascular magnetic resonance (CMR) is often used for diagnosis and management of cardiovascular diseases as it helps in giving a detailed and quantitative analysis of the parameters associated with heart's anatomy. Clinical changes can be quantitatively analyzed with the help of CMR imaging using which doctors can monitor and strategize further diagnosis. Manual delineation of quantitative features from these images is often time-consuming, error-prone and introduces inter-observer variability which can affect the diagnosis of patients and becomes infeasible in real life scenario where the footfall of patients is high. With the advent of deep learning, a lot of progress has been made in the automated analysis of medical imaging for various tasks which in turn can help the doctors in treatment planning. Automated delineation of cardiac features from CMRI plays a crucial role in analyzing the normal and abnormal parameters at large scale. Convolution neural networks have shown to be of great use especially for medical image segmentation-based tasks that primarily uses encoder-decoder based architectures to localize the features of interest. Current methods tend to use convolution-based approach to segment heart anatomy features and then extract quantitative features to come up with an automated diagnosis. In this study, we will be using cardiac MRI data provided as part of automated cardiac diagnosis challenge (ACDC) 2017 to segment for left ventricle (LV), right ventricle (RV) and myocardium and also to detect the presence of five types of classes namely normal, dilated cardiomyopathy, hypertrophic cardiomyopathy, prior myocardial infarction and abnormal right ventricle using deep learning and machine learning models. Our aim will be to come up with a robust deep learning framework using Segnet architecture for the segmentation task and then build on the segmentation maps obtained to build a predictive model using machine learning algorithms for the prediction of five types of classes as mentioned. We will finally validate our results against the ground truths provided from the challenge and report our results using evaluation metrics.

IJSCAR VOL. 1, ISSUE 1

2 Literature Review

A lot of advancement has been made in the field of medical image segmentation and classification tasks recently, especially after the development of Unet architectures [10]. In 2018 [6] combined Unet and M-net [14] architecture to come up with an automated cardiac segmentation model. Data augmentation in form of rotation was used after which the final dice score improved significantly for right ventricle. [2] developed a fully automated framework for segmentation of heart anatomical structures like left ventricles, right ventricles and myocardium using multiple 2D and 3D convolutional tasks using the cardiac MRI images provided as part of ACDC 2017 challenge. Out of various combination of networks, it was found that 2D networks outperformed other 3D networks due to presence of large slice thickness in terms of dice evaluation metric. [5]developed a fully automated processing pipeline for segmentation and classification using cardiac cine MRI data. An ensemble model of 2D and 3D Unet architecture was used with dice loss used as optimizer to come up with robust segmentation outputs. Geometrical features were extracted from the segmentation outputs and further prediction models were developed using an ensemble of classifiers in which an overall accuracy of 92% was achieved. [11]proposed a multi-task cardiac segmentation and diagnosis training from CMR images that had a better convergence rate. Densenets [4] and Unet models are used for classification and segmentation training in which the use of handcrafted features was completely avoided, which is generally used in clinical diagnosis. Classification error was reduced from 32% to 22% by incorporating the segmentation training block. [7] used Fourier analysis and circular Hough transform to get the region of interest in cardiac MRI images and deployed a FCN based architecture based on Densenets [4] along with long skip and short-cut connections in the up-sampling path to avoid feature map explosion. Multiscale processing of input was done at initial layers of the network in parallel paths and later ensembled as in inception networks. Weighted cross-entropy and dice loss was used as optimizers. The proposed architecture achieved an overall accuracy of 100% for cardiac disease diagnosis on the ACDC 2017 challenge.

3 Methodology

3.1 Dataset Description

The data in this study has been taken from ACDC 2017 challenge that consists of cine-MRI scans acquired at University Hospital of Dijon (France). The training set consists of almost 100 patient scans with slice thickness varying from 5mm to 10mm, with each scan having a corresponding end-systolic (ES) and end-diastolic(ED) phase volumes. Ground truths masks are provided for myocardium, left ventricle and right ventricle for ES and ED phases of each patient. The data consists of equal distribution of normal, dilated cardiomyopathy, hypertrophic cardiomyopathy, prior myocardial infarction and abnormal right ventricle cases which have been pre-annotated.

In this study, we will be using 70 cases for training, 10 cases for validation and 20 cases for testing on which the final evaluation metrics in terms of dice and accuracy will be reported. Five-fold cross validation strategy will be adopted for model hyperparameter tuning and finally the model results across all the models will be ensembled for final reporting.

3.2 Segmentation

3.2.1 Data Preprocessing. In the segmentation part of the ACDC challenge, we normalize the whole image to zero mean and unit variance. For 2D segmentation, we do not resample the image in the axial plane, but, select slices from the whole volume and resize it to a fixed size. Each scan comprised of at least 18 slices from both ED and ES phases, so a total of 700 slices was used as part of our training and a total of 190 slices was used as part of testing. Random augmentations like cropping, rotating, and flipping were performed to avoid overfitting and also to overcome lack of training data. For 3D segmentation, we resample the whole volume to 1.25x1.25x10mm. We will randomly crop the images for a patch size of 128*128*128 to be input to the model. From each scan comprising of both ED and ES phases, two random crops were selected based on a positive to negative ratio of 0.4 where probability of less than 0.4 will focus on patches containing regions where organ of interest will be found and probability greater than 0.4 will focus on patches containing regions organ of interest will not be found to make sure the model sees multiple instances of each scan.

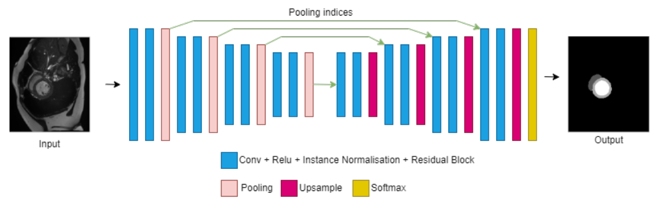

Figure 1: Standard Segnet architecture

3.2.2 Network Architecture. We approach the segmentation problem by using 2D and 3D based approaches. We will make use of Segnet as proposed in [1], that works quite similar to traditional Unet [10] which is a standard go-to algorithm that has been used mainly for biomedical image segmentation problems. Unet's capability lies in its architecture that has an encoder and a decoder path that captures the context and then successfully manages to localize it and also the skip connections that helps to efficiently integrate low level features with high level semantic information. This helps in getting better pixel level classification. Segnet works quite similar to Unet but with small changes in the decoder path that tends to use the pooling indices that have been obtained from the encoder path to eliminate the need for learning to upsample. This will help us in reducing parameter count and make the model memory efficient than Unet. Also, as our main focus is to get the overall classification accurate, we primarily use the segments as an additional input later on to the classification model We will also make slight changes in the encoding path as proposed in [9] in 2019. Here, the encoder will use ResNet blocks, where each bock consists of two convolutions, normalization and ReLu followed with a skip connection. Instance Normalization [13] will be used instead of traditional batch normalization. We will make use of four down sampling blocks and four up sampling blocks and initial filters being set to 16 followed by a final softmax layer to get final predictions (see Figure 1). We use the same architecture for 3D and 2D approaches with the difference being only in the convolution dimensions. We used a weighted multiclass dice loss [8] which tries to optimize the overlap of prediction and ground truth and also overcomes the class imbalance problem.

$$L_{dsc} = -\frac{2}{|K|} \sum_{k \in K} \frac{\sum_{i \in I} u_i^k v_i^k}{\sum_{i \in I} u_i^k + \sum_{i \in I} v_i^k} \tag{1}$$

Both the approaches were trained for 300 epochs with five-fold cross validation using Adam optimizer. Dice score efficient was used as the evaluation metric which tends to measure the overlap of two volumes. Final results were obtained after ensembling the outputs from each fold and resampling to original voxel resolution.

$$\text{Dice Score Coeffecient} = \frac{2TP}{2TP + FP + FN} \tag{2}$$

3.3 Cardiac disease classification

3.3.1 Feature Engineering. In the cardiac disease classification part of ACDC challenge, our goal was to accurately classify the cardiac MRI images into five classes namely, normal, dilated cardiomyopathy, hypertrophic cardiomyopathy, prior myocardial infarction, and abnormal right ventricle. To achieve this purpose, we extract

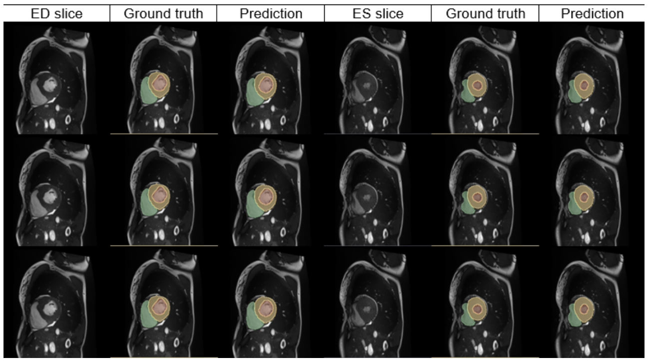

Figure 2: Segmentation results on test set using 2D Segnet model at both ED and ES phases

cardiac features from the ground truth segmentations as proposed in [7]. Primary features like myocardial wall thickness, volume of right ventricle, left ventricle, myocardium at end diastole, and systole phases were extracted using the segmentation masks. Derive features like ejection fraction, volumetric ratios, and variation profile of myocardial wall thickness were obtained from the primary features.

3.3.2 *Classification.* In addition to the primary and derived features extracted, we used the cardiac MRI images at end diastole and systole phases provided along with the segmentation outputs as input to the classification model. Classification model used was an ensemble of 3d resnet50 [3]and layer multilayer perceptron (MLP) with two hidden layers, each containing 100 units. The output from each branch was finally passed through a linear layer to get the final output. A five-fold cross-validation approach was used along with Adam optimizer. Each fold was trained for 20 epochs. Modal value across all the folds was taken to obtain the final ensemble prediction.

4 Results

We evaluated our segmentation model with respect to the ground truth segmentation provided as part of the ACDC challenge on ED and ES phases using dice score evaluation metric on the separately reserved test set. We ensembled the final outputs obtained from five folds to get the final output. 2D model clearly outperformed the 3d model in terms of final dice scores. This can be attributed to the presence of poor resolution of the z-axis. 2d model achieved an overall dice score of 0.877, 0.877, 0.937 for right ventricle, myocardium and left ventricle respectively whereas 3d model got a dice score of 0.73, 0.725 and 0.852. Figure 2 shows the results of 2d segmentation model on both ED and ES phases. Detailed results for both 2d and 3d model in dice scores are tabulated in Table 1 and 2.

We used the 2d segmentation outputs to extract the features as described in earlier section to perform classification on the test set. Along with the extracted features, input images of both the phases and their respective segmentation outputs from the 2d model were given as input to the classification model. We ensembled the results from all the folds and took the model value across the five folds to get the final prediction.We achieved an overall accuracy of 85% on the test set and 0.81,0.89 and 0.9 scores of precision, recall and auc-score respectively. Confusion matrices are provided in figure 3. We can see from figure 3, a clear difficulty in differentiating DCM from MINF and RV patients, also, a normal patient being misclassified as RV on the test set. Misclassifying DCM for MINF can have clinical implications as both diseases have different diagnosis and also misclassifying DCM for RV will have clinical implications too, as DCM would require more focus on left ventricle functioning than right ventricle as seen in RV.

Table 1: Dice scores of 2d segmentation model

	Phase	Dice		
		RV	MYO	LV
DCM	ED	0.904	0.86	0.969
	ES	0.808	0.866	0.96
HCM	ED	0.926	0.908	0.957
	ES	0.776	0.922	0.866
MINF	ED	0.916	0.868	0.944
	ES	0.788	0.9	0.933
RV	ED	0.926	0.84	0.958
	ES	0.876	0.863	0.896
NOR	ED	0.95	0.87	0.96
	ES	0.912	0.892	0.92

Table 2: Dice scores of 3d segmentation model

	Phase	Dice		
		RV	MYO	LV
DCM	ED	0.743	0.667	0.875
	ES	0.571	0.633	0.837
HCM	ED	0.81	0.8	0.894
	ES	0.65	0.687	0.662
MINF	ED	0.837	0.742	0.915
	ES	0.72	0.786	0.901
RV	ED	0.788	0.738	0.917
	ES	0.705	0.758	0.855
NOR	ED	0.91	0.787	0.932
	ES	0.762	0.81	0.855

5 Discussion

In this study, we presented a fully automated pipeline to classify cardiovascular diseases into five classes on cardiac MRI images using deep learning. We initially started out with a segmentation model that would accurately segment left ventricle, right ventricle, and myocardium using end diastole and end systole phases of cardiac MRI images. We tested our approach with 2D and 3D models of Segnet architecture with 2D model achieving the best dice score of 0.877, 0.877, 0.937 for right ventricle, left ventricle, and myocardium respectively on the test set. We used the segmentation outputs to extract quantitative features which were used as an additional input to the ensembled classification model to predict the cardiovascular diseases. We achieved an overall accuracy of 85% on the test set with a recall score of 0.86. As our main focus is to improve recall of each disease against normal, a normal case being misclassified

Figure 3: Confusion matrix

as RV would mean giving extra precaution to patient than missing the disease out. There were some cases of misclassification, which can lead to significant errors in patient management and diagnosis.

Further improvements to this study can be made by using more patient records apart from the one provided by the ACDC challenge along with different cases of cardiovascular diseases. In addition to images, patient's clinical history can be considered in the future, to get better results. Also, more robust segmentation architecture can be used with different data processing techniques to yield better results in terms of segmentation. As there were scenarios of misclassification, accurate classification needs to be achieved for ensuring appropriate patient disease management.

6 Conclusion

The present research can be elaborated by incorporating more features and patient records to design a more robust model. The patient's data can be recorded over time and create a sequential model to predict CAD with better accuracy. Along with prediction, further research can lead to the identification of the underlying cause of the disease so that better preventative measures can be taken.

Acknowledgments

I would like to express my deepest gratitude to Dr. Michael Singer for his invaluable guidance, insightful feedback, and continuous encouragement throughout the research process. His expertise and support were instrumental in the completion of this work.

References

[1] Vijay Badrinarayanan, Alex Kendall, and Roberto Cipolla. 2017. SegNet: A Deep Convolutional Encoder-Decoder Architecture for Image Segmentation. *IEEE Transactions on Pattern Analysis and Machine Intelligence* 39, 12 (2017), 2481–2495.

[2] Christian F Baumgartner, Lisa M Koch, Marc Pollefeys, and Ender Konukoglu. 2018. An Exploration of 2D and 3D Deep Learning Techniques for Cardiac MR Image Segmentation. In *Statistical Atlases and Computational Models of the Heart. ACDC and MMWHS Challenges: 8th International Workshop, STACOM 2017, Held in Conjunction with MICCAI 2017, Quebec City, Canada, September 10-14, 2017, Revised Selected Papers*. Springer, 111–119.

[3] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. 2016. Deep Residual Learning for Image Recognition. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*. 770–778.

[4] Gao Huang, Zhuang Liu, Laurens Van Der Maaten, and Kilian Q Weinberger. 2017. Densely Connected Convolutional Networks. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*. 4700–4708.

[5] Fabian Isensee, Paul F Jaeger, Peter M Full, Ivo Wolf, Sandy Engelhardt, and Klaus H Maier-Hein. 2018. Automatic Cardiac Disease Assessment on Cine-MRI via Time-Series Segmentation and Domain Specific Features. In *Statistical Atlases and Computational Models of the Heart. ACDC and MMWHS Challenges: 8th International Workshop, STACOM 2017, Held in Conjunction with MICCAI 2017, Quebec City, Canada, September 10-14, 2017, Revised Selected Papers*. Springer, 120–129.

[6] Yeonggul Jang, Yoonmi Hong, Seongmin Ha, Sekeun Kim, and Hyuk-Jae Chang. 2018. Automatic Segmentation of LV and RV in Cardiac MRI. In *Statistical Atlases and Computational Models of the Heart. ACDC and MMWHS Challenges: 8th International Workshop, STACOM 2017, Held in Conjunction with MICCAI 2017, Quebec City, Canada, September 10-14, 2017, Revised Selected Papers*. Springer, 161–169.

[7] Mahendra Khened, Varghese Alex Kollerathu, and Ganapathy Krishnamurthi. 2019. Fully Convolutional Multi-Scale Residual DenseNets for Cardiac Segmentation and Automated Cardiac Diagnosis Using Ensemble of Classifiers. *Medical Image Analysis* 51 (2019), 21–45.

[8] Fausto Milletari, Nassir Navab, and Seyed-Ahmad Ahmadi. 2016. V-Net: Fully Convolutional Neural Networks for Volumetric Medical Image Segmentation. In *2016 Fourth International Conference on 3D Vision (3DV)*. IEEE, 565–571.

[9] Andriy Myronenko. 2019. 3D MRI Brain Tumor Segmentation Using Autoencoder Regularization. In *BrainLes: Glioma, Multiple Sclerosis, Stroke and Traumatic Brain Injuries: 4th International Workshop, BrainLes 2018, Held in Conjunction with MICCAI 2018, Granada, Spain, September 16, 2018, Revised Selected Papers, Part II*. Springer, 311–320.

[10] Olaf Ronneberger, Philipp Fischer, and Thomas Brox. 2015. U-Net: Convolutional networks for biomedical image segmentation. In *Medical Image Computing and Computer-Assisted Intervention–MICCAI 2015: 18th International Conference, Munich, Germany, October 5-9, 2015, Proceedings, Part III*. Springer, 234–241.

[11] Gerard Snaauw, Dong Gong, Gabriel Maicas, Anton Van Den Hengel, Wiro J Niessen, Johan Verjans, and Gustavo Carneiro. 2019. End-to-End Diagnosis and Segmentation Learning from Cardiac Magnetic Resonance Imaging. In *2019 IEEE 16th International Symposium on Biomedical Imaging (ISBI 2019)*. IEEE, 802–805.

[12] World Health Organization. 2024. Cardiovascular Diseases. https://www.who.int/health-topics/cardiovascular-diseases. Accessed: 2024-08-30.

[13] Yuxin Wu and Kaiming He. 2018. Group Normalization. In *Proceedings of the European Conference on Computer Vision (ECCV)*. 3–19.

[14] Xiao-Yun Zhou and Guang-Zhong Yang. 2019. Normalization in Training U-Net for 2-D Biomedical Semantic Segmentation. *IEEE Robotics and Automation Letters* 4, 2 (2019), 1792–1799.

Received 01 September 2024; accepted 30 October 2024

Using a Feedback Loop for LLM-based Infrastructure as Code Generation

Mayur Amarnath Palavalli
Irvington High School
Fremont, California, USA
mayur.palavalli@gmail.com

Mark Santolucito
Barnard College, Columbia University
New York, New York, USA
msantolu@barnard.edu

Abstract

Code generation with Large Language Models (LLMs) has helped to increase software developer productivity in coding tasks, but has yet to have significant impact on the tasks of software developers that surround this code. In particular, the challenge of infrastructure management remains an open question. We investigate the ability of an LLM agent to construct infrastructure using the Infrastructure as Code (IaC) paradigm. We particularly investigate the use of a feedback loop that returns errors and warnings on the generated IaC to allow the LLM agent to improve the code. We find that, for each iteration of the loop, its effectiveness decreases exponentially until it plateaus at a certain point and becomes ineffective.

Keywords

Infrastructure as Code, Large Language Models

ACM Reference Format:
Mayur Amarnath Palavalli and Mark Santolucito. 2024. Using a Feedback Loop for LLM-based Infrastructure as Code Generation. In *Proceedings of International Journal of Secondary Computing and Applications Research (IJSCAR VOL. 1, ISSUE 1)*. ACM, New York, NY, USA, 4 pages. https://doi.org/10.5281/zenodo.14279492

1 Introduction

Infrastructure as Code (IaC) has fundamentally transformed the way cloud infrastructure is managed. With IaC, developers can provision and maintain their infrastructure through code. This ensures automation and consistency in deploying infrastructure, allowing for more effective scaling of operations. IaC also allows teams to use version control on their infrastructure, making it easier to collaborate and track changes [7]. However, a major challenge with IaC is in the difficulty of writing correct code [1, 3, 10, 11, 16].

At the same time, Large Language Models (LLMs), as applied to code generation, are enabling developers to be more effective. Code generation benchmarks, such as HumanEval [2] and SWEBench [9], have shown that LLMs are capable of assisting developers with challenging programming tasks. It is then natural to seek to extend the application of LLMs for code generation to IaC.

The combination of IaC and LLMs could allow for wider adoption of IaC and more effective infrastructure management. However, many questions remain open about the ability of LLMs to reason about the complexities of IaC. In particular, there are many implicit rules about the semantics of cloud resources that are difficult to reconcile when creating infrastructure through IaC.

In our work, we provide an analysis of the ability of LLMs to generate AWS CloudFormation code. In particular, we investigate the ability of an LLM to respond to errors and warnings generated by `cfn-lint` [5], a tool for analyzing CloudFormation code. In summary, the main contributions of this work are:

(1) The design of a feedback loop system for the generation of CloudFormation code.
(2) A set of CloudFormation code generation benchmarks built from industry-standard IaC problems.
(3) An evaluation of our feedback loop system on this benchmark, showing that LLMs struggle to fully reconcile all errors in IaC generation.

2 Related Work

The application of LLMs in generating IaC has amassed significant attention in recent years. Using LLMs for IaC comes with certain challenges that make it inefficient in large systems. Srivatsa et al. evaluate the LLM performance on functional correctness by comparing with human-written code and deciding whether or not it is an exact match. They found that the GPT-3.5-turbo model had a success rate of between 50 and 60 percent, while the Codeparrot model never exceeded 10 percent accuracy [18]. They also discuss ethical and safety concerns of developing LLMs for more accurate IaC generation.

A natural extension is the work of Ugare et al. who introduce SynCode [20], a framework that uses grammar rules to enhance LLM generation in formal coding languages. SynCode is able to reduce 96.07 percent of syntax errors in Python and Go. It particularly shines in generating JSON, where it is able to eliminate all syntax errors. This is achieved by utilizing context-free grammar rules based on discrete finite automation.

An important area of investigation related to LLM code generation is constrained decoding [13, 21]. At a high level, grammar constrained decoding (GCD) helps language models (LLMs) in producing structured results without requiring additional fine tuning. GCD utilizes grammars to guarantee that the generated sequences follow a predefined structure. The technique greatly improves the performance of LLMs in such settings without the need for expensive task-specific training efforts. Since we are generating highly structured JSON documents (CloudFormation files), GCD could be a way to further optimize the results. However, the main goal of our work is to provide an initial baseline accounting of the viability of LLM code generation for IaC. We leave such optimizations to future work.

IJSCAR VOL. 1, ISSUE 1

Another critical challenge in LLM code generation is fixing syntax errors. Tsai et al. address this with RTLFixer [19], a framework designed to fix syntax errors in Verilog code. After finding that 55 percent of errors in LLM-generated Verilog code were syntax errors, RTLFixer was designed to utilize Retrieval-Augmented Generation and ReAct (Reasoning and Action framework) techniques to improve error correction. The framework achieves a 98.5 percent success rate in fixing syntax errors after testing on 212 syntactically invalid Verilog implementations.

These studies demonstrate the evolving role of LLMs in code generation and highlight the importance of addressing syntactic correctness and complexity.

3 Background

```
{"AWSTemplateFormatVersion": "2010-09-09",
"Resources": {
  "MyEC2Instance": {
    "Type": "AWS::EC2::Instance",
    "Properties": {
      "InstanceType": "t2.micro",
      ...
```

Figure 1: An example AWS CloudFormation JSON template.

IaC is a diverse space with many existing languages and tools. The most widely adopted tools are Terraform [8] and Amazon Web Service's CloudFormation [17]. Both Terraform and CloudFormation files are specified as JSON (or YAML) documents, where the various fields define properties of the cloud resource to be deployed. These files are generally declarative - giving a specification of the desired cloud infrastructure state. This is in contrast to some other IaC languages, such as Pulumi, which allow users to write imperative programs that construct operations that should take place upon the user's infrastructure [15]. An example of a CloudFormation JSON is given in Fig. 1.

```
E1015 {'Fn::GetAZs': ''} is not of type 'string'
Error location - path/to/my_iac.json:1:4575
```

Figure 2: An example error message from `cfn-lint`.

In this work, we focus solely on the LLM generation of AWS CloudFormation [17] in JSON. We additionally use the AWS CloudFormation linter `cfn-lint` [6]. `cfn-lint` allows us to validate CloudFormation JSON templates against the resource provider schemas provided by AWS as well as other best-practice IaC rules. A schematically valid JSON template returns nothing when run through `cfn-lint`. In any other case, `cfn-lint` returns errors and/or warnings. The example error in **Fig. 2** outlines the basic structure of an error produced by `cfn-lint`. The error message code starts with a letter followed by a string of numbers which forms a error code. This is followed by a brief description of the error and the line and character number at which it occurs.

4 Methodology

To describe our system, we first describe the methodology we used for collecting a benchmark set, then we describe the feedback loop that we constructed with `cfn-lint`. We make all of our code and evaluation results available open-source: https://github.com/Mayur-Palavalli/LLM-IaC-generation.

```
Create a AWS CloudFormation template that deploys a VPC
with a pair of private subnets spread across two
Availabilty Zones. It deploys a VPC Endpoint for
CloudFormation so an instance in the private subnet can
use cfn-signal for its CreationPolicy.
```

Figure 3: An example prompt from the official AWS CloudFormation Template Schema repository.

4.1 Benchmark Set

To create a dataset of prompts, we took 33 descriptions of AWS CloudFormation problems from the official AWS CloudFormation Template Schema repository [4]. This repository converts existing Resource Specifications files into a formatted JSON (or YAML) schema document which can be integrated in an IDE. An example prompt from this repository is displayed in Fig. 3

4.2 Feeback Loop

We queried an LLM for a solution to this problem five times for each description, yielding a dataset of 165 implemented CloudFormation templates. We generated these templates in JSON format.

The tool `cfn-lint` is able to process a JSON CloudFormation template, and returns errors describing schema violations, invalid resource properties, and best practices. After each generation, we run `cfn-lint`, and send the error message(s) and JSON template back to the LLM, instructing it to modify the template to fix the error generated by `cfn-lint`. In this manner, we create the feedback loop outlined in **Fig. 4** that sends each JSON CloudFormation file in our dataset to the LLM, each time providing the new file and corresponding error message(s). This process is repeated ten times. We hypothesized that the LLM would be able to fix all errors produced by `cfn-lint` after a certain number of iterations.

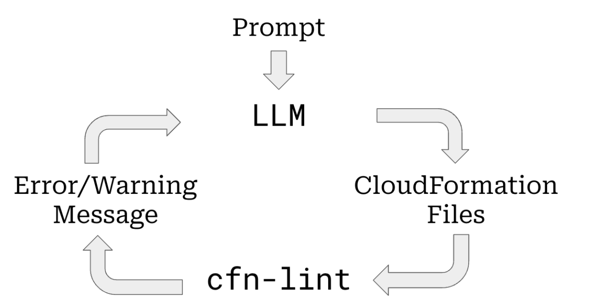

Figure 4: A diagram of the feedback loop: we provide the LLM with a prompt for an AWS CloudFormation file, which is run through `cfn-lint` to produce error/warning message(s) which are given back to the LLM.

We keep track of the total number of errors and warnings across all 165 files after each iteration of the feedback loop (for all 10 iterations) in order to identify the point at which it becomes ineffective. This process of running each file ten times through the feedback loop is repeated six times. We choose to repeat it six times because most of the error bars are narrow, indicating a small standard deviation in the number of errors after each of the ten iterations. A small standard deviation implies low variance, which demonstrates that additional repetitions are unlikely to change the overall pattern.

5 Results

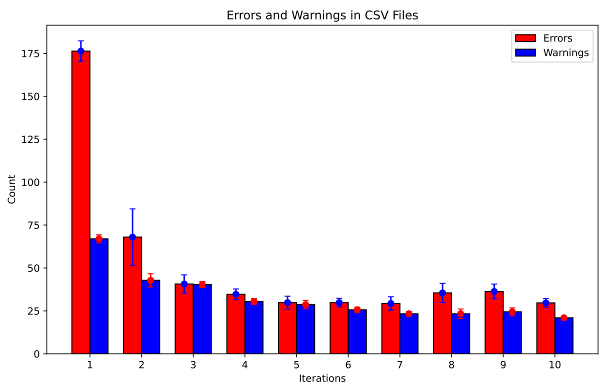

Figure 5: A histogram of errors over multiple `cfn-lint` feedback iterations showing error bars representing the distribution over six trials.

The entire process of iterating each file ten times through the feedback loop is repeated six times. **Fig. 5** summarizes the results of all six sets of iterations in a bar graph by displaying the total number of errors and warnings in all 165 JSON files after each iteration. The graph's plateau beginning at approximately the fifth iteration indicates the point at which the feedback loop is no longer effective.

The plateau is caused by the feedback loop's inconsistency in fixing errors. The LLM is unable to correctly fix certain individual errors, which occasionally result in several new errors being produced from each old error between iterations. After approximately five iterations, this anomaly is enough to yield no significant net change in the number of errors across all 165 files. The slight peak in the number of errors at the eighth and ninth iterations indicates at least one of two things:

(1) An unusually high number of files generated more errors than it had after the seventh iteration.
(2) Certain files had an unusually high increase in the number of errors generated after the seventh iteration.

We predict the cause of this to be the LLMs incapability to properly understand all the error messages from `cfn-lint`.

6 Discussion

Perfecting the use of LLMs in generating valid AWS CloudFormation could enable developers to more quickly build the infrastructure they need for their systems. Automating the generation of CloudFormation templates could drastically increases the speed and efficiency of setting up complex cloud environments.

We do not use ChatGPT's structured output mode [12]. Structured output mode allows users to provide the desired schema of JSON output from ChatGPT in addition to the prompt. This mode guarantees that the generated response matches the schema. However, for the purposes of CloudFormation generation this is not a viable option. Not only is the complete CloudFormation schema is over 200,000 lines long, it uses features of JSON schemas that are outside the scope of ChatGPT's structured output mode [12].

While our feedback loop dramatically increases the chance of generating valid IaC, we still do not have a formal guarantee that the generated code will be schematically valid. The remaining uncertainty is enough for LLM code generation of IaC to remain hard to use in large-scale development. There is a further question of semantic validity. Semantic validity captures the idea that not only is the CloudFormation deployable, but is also what the user wants (e.g. an empty file is always schematically valid, but not semantically valid). A schematically valid CloudFormation is not guaranteed to be semantically valid. We are left with the task of ensuring schematically valid CloudFormation are also semantically valid before LLMs are safe to use at large scale. This, however, is a challenging problem. Measuring semantic validity requires a way to determine how well the generated infrastructure adheres to the prompt. There is currently no simple tool like `cfn-lint` that does this.

6.1 Threats to Validity

One threat to the generalizability of our work is the question of the extent to which we would see the same pattern if using a different LLM. At the time of writing, OpenAI's gpt-4o is among the best LLMs capable of working with JSON. We believe that the use of a different LLM for our work would the rate of decrease in the number of errors, but not change the overall pattern. We hypothesize that the plateau on the graph in Fig. 5 is due to the LLM's inability to reconcile error messages with the high-level intent of the prompt. If this is true, we would need a structurally different LLM than OpenAI's gpt-4o model.

Another threat to generalizability is the extent to which the IaC provider impacts the pattern of our results. AWS CloudFormation is one of the most well-documented IaC services, and thus has large corpus of training data related to it, making it well-suited to generation with LLMs. Other well-documented IaC services, such as Terraform, would likely yield similar results due to similar compatibility with LLMs. We hypothesize that using a less documented services would likely still produce a plateau, but would yield more slowly declining error bars due to a smaller training set of relevant data.

Furthermore, our work utilizes only one method of receiving feedback. We use error messages from `cfn-lint` as feedback to demonstrate a proof of concept, but other means of testing and receiving feedback could yield different results. The plateau, for instance, might not exist with a more structured feedback strategy.

6.2 Future Work

We find that an LLM has a limited ability to respond to error messages and correct code in the context of IaC configurations. A future task would be to customize error messages to a format that an LLM like ChatGPT more clearly understands so errors can be be fixed without creating new ones. The creation of a tool like cfn-lint to check for semantic validity would take IaC generation via LLMs to the next level, allowing for widespread commercial use.

In the context of IaC, Pulumi [15] is a an open-source IaC tool that allows developers to define, deploy, and manage cloud infrastructure in familiar programming languages. All AWS services, including CloudFormation, are fully supported by Pulumi. An especially useful tool is Pulumi AI [14], an experimental feature that allows developers to generate IaC in familiar programming languages via natural-language prompts. As far as we know, the Pulumi AI tool does not incorporate any feedback during its generation from static analysis tools such as `cfn-lint`. A future task might be to use Pulumi AI to generate Pulumi code which maps to AWS CloudFormation, which is run through `cfn-lint`, and send the generated errors back to correct the Pulumi code.

7 Conclusion

Our investigation of the use of feedback loop for Infrastructure as Code generation demonstrates the potential of agentic LLM systems for real-world software development. Our results indicate that using LLMs to generate IaC provides a significant advantage, but not yet enough to be make it scale-able. The feedback loop offers a method to increase the rate of successful IaC generation from LLMs, as well as an opportunity for further research to fix the anomalies that cause the loop to become eventually ineffective. Even if OpenAI's structured output mode were to support large complex schema like that of AWS CloudFormation, there are still issues of semantic correctness that are not captured in the schema and only appear through checks with tools like `cfn-lint`.

References

[1] Claudia Cauli. 2022. Pre-deployment Description Logic-based Reasoning for Cloud Infrastructure Security. (2022). University of Gothenburg. IT Faculty. Dissertation.

[2] Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde de Oliveira Pinto, Jared Kaplan, Harri Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, Alex Ray, Raul Puri, Gretchen Krueger, Michael Petrov, Heidy Khlaaf, Girish Sastry, Pamela Mishkin, Brooke Chan, Scott Gray, Nick Ryder, Mikhail Pavlov, Alethea Power, Lukasz Kaiser, Mohammad Bavarian, Clemens Winter, Philippe Tillet, Felipe Petroski Such, Dave Cummings, Matthias Plappert, Fotios Chantzis, Elizabeth Barnes, Ariel Herbert-Voss, William Hebgen Guss, Alex Nichol, Alex Paino, Nikolas Tezak, Jie Tang, Igor Babuschkin, Suchir Balaji, Shantanu Jain, William Saunders, Christopher Hesse, Andrew N. Carr, Jan Leike, Josh Achiam, Vedant Misra, Evan Morikawa, Alec Radford, Matthew Knight, Miles Brundage, Mira Murati, Katie Mayer, Peter Welinder, Bob McGrew, Dario Amodei, Sam McCandlish, Ilya Sutskever, and Wojciech Zaremba. 2021. Evaluating Large Language Models Trained on Code. arXiv:2107.03374 [cs.LG] https://arxiv.org/abs/2107.03374

[3] Michele Chiari, Michele De Pascalis, and Matteo Pradella. 2022. Static Analysis of Infrastructure as Code: a Survey. In *2022 IEEE 19th International Conference on Software Architecture Companion (ICSA-C)*. 218–225. https://doi.org/10.1109/ICSA-C54293.2022.00049

[4] AWS CloudFormation. 2024. AWS CloudFormation Templates Collection. https://github.com/aws-cloudformation/aws-cloudformation-templates. [Accessed 02-09-2024].

[5] AWS CloudFormation. 2024. cfn-lint: CloudFormation Linter. https://github.com/aws-cloudformation/cfn-lint. [Accessed 12-09-2024].

[6] Firefly. 2024. AIAC: Artificial Intelligence Infrastructure-as-Code Generator. https://github.com/gofireflyio/aiac. [Accessed 02-09-2024].

[7] MIchele Guerriero, Martin Garriga, Damian A. Tamburri, and Fabio Palomba. 2019. Adoption, Support, and Challenges of Infrastructure-as-Code: Insights from Industry. In *2019 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. 580–589. https://doi.org/10.1109/ICSME.2019.00092

[8] HashiCorp. 2024. Terraform. https://github.com/hashicorp/terraform. [Accessed 30-09-2024].

[9] Carlos E Jimenez, John Yang, Alexander Wettig, Shunyu Yao, Kexin Pei, Ofir Press, and Karthik Narasimhan. 2023. Swe-bench: Can language models resolve real-world github issues? *arXiv preprint arXiv:2310.06770* (2023).

[10] Julien Lepiller, Ruzica Piskac, Martin Schäf, and Mark Santolucito. 2021. Analyzing Infrastructure as Code to Prevent Intra-update Sniping Vulnerabilities. In *Tools and Algorithms for the Construction and Analysis of Systems*, Jan Friso Groote and Kim Guldstrand Larsen (Eds.). Springer International Publishing, Cham, 105–123.

[11] Ruben Opdebeeck, Ahmed Zerouali, and Coen De Roover. 2023. Control and Data Flow in Security Smell Detection for Infrastructure as Code: Is It Worth the Effort?. In *2023 IEEE/ACM 20th International Conference on Mining Software Repositories (MSR)*. 534–545. https://doi.org/10.1109/MSR59073.2023.00079

[12] OpenAI. 2024. Introducing Structured Outputs in the API. https://openai.com/index/introducing-structured-outputs-in-the-api/. [Accessed 02-09-2024].

[13] Kanghee Park, Jiayu Wang, Taylor Berg-Kirkpatrick, Nadia Polikarpova, and Loris D'Antoni. 2024. Grammar-Aligned Decoding. arXiv:2405.21047 [cs.AI] https://arxiv.org/abs/2405.21047

[14] Pulumi. 2024. pulumi-ai. https://github.com/pulumi/pulumi-ai. [Accessed 12-09-2024].

[15] Pulumi. 2024. Pulumi Documentation. https://www.pulumi.com/docs/. [Accessed 12-09-2024].

[16] Feitong Qiao, Aryana Mohammadi, Jürgen Cito, and Mark Santolucito. 2024. Statically Inferring Usage Bounds for Infrastructure as Code. *arXiv preprint arXiv:2402.15632* (2024).

[17] Amazon Web Services. 2024. What is AWS CloudFormation? https://docs.aws.amazon.com/AWSCloudFormation/latest/UserGuide/Welcome.html. [Accessed 02-09-2024].

[18] Kalahasti Ganesh Srivatsa, Sabyasachi Mukhopadhyay, Ganesh Katrapati, and Manish Shrivastava. 2024. A Survey of using Large Language Models for Generating Infrastructure as Code. *arXiv preprint arXiv:2404.00227* (2024). https://arxiv.org/abs/2404.00227

[19] YunDa Tsai, Mingjie Liu, and Haoxing Ren. 2023. Rtlfixer: Automatically fixing rtl syntax errors with large language models. *arXiv preprint arXiv:2311.16543* (2023). https://arxiv.org/abs/2311.16543

[20] Shubham Ugare, Tarun Suresh, Hangoo Kang, Sasa Misailovic, and Gagandeep Singh. 2024. SynCode: LLM Generation with Grammar Augmentation. arXiv:2403.01632 [cs.LG] https://arxiv.org/abs/2403.01632

[21] Xi Ye, Lihu Chen, Juntao Chen, Weijia Shi, and Kai-Wei Chang. 2023. Grammar-Constrained Decoding for Structured NLP Tasks without Finetuning. https://arxiv.org/abs/2305.13971. [Accessed 25-10-2024].

Received 30 September 2024; accepted 30 October 2024

A Case Study on LLM Code Generation in Sonic Pi and Its Impact on Student Attitudes towards Computer Science

Josch Wang
joschzw99@gmail.com
Manhasset High School
Manhasset, New York, USA

Abstract

Research has demonstrated that creating music and soundscapes with tools like Sonic Pi can enhance the appeal of computer science and music composition as fields of study. However, with current advancements in artificial intelligence (AI), the nature of computer science is evolving rapidly, particularly with AI's ability to generate code from a single prompt. Similarly, AI technologies have already begun to transform music composition by enabling the generation of music. This study investigates whether live coding with large language models (LLMs) such as ChatGPT and Copilot influences perceptions of entering the fields of music composition and computer science. Understanding the impact of AI is crucial, as it increasingly shapes various domains of our lives. To explore this, we instructed a random sample of high school students in the basics of live coding, emphasizing the inclusion of AI tools. The results indicate a positive shift in students' attitudes toward programming and music creation, suggesting that AI serves as a beneficial tool rather than a detrimental one.

Keywords

LLM Code Generation, Live Coding, Sonic Pi, CS Education

ACM Reference Format:
Josch Wang. 2024. A Case Study on LLM Code Generation in Sonic Pi and Its Impact on Student Attitudes towards Computer Science. In *Proceedings of International Journal of Secondary Computing and Applications Research (IJSCAR VOL. 1, ISSUE 1).* ACM, New York, NY, USA, 5 pages. https://doi.org/10.5281/zenodo.14279494

1 Introduction

Programming has traditionally been perceived as a challenging subject to introduce to K-12 students, especially because the outcome of many programming tasks does not immediately manifest in a way that is relevant to the students. Recent research has explored innovative approaches to introduce programming concepts to novices, with live coding emerging as a promising method for engaging learners through culturally responsive computing by combining computer science and music composition [1, 4, 21]. Studies have demonstrated that tools like Sonic Pi, designed for live coding, can increase engagement and foster positive attitudes towards programming in young students [5, 7]. The ability to quickly create music with Sonic Pi, as compared to conventional programming languages, make it particularly appealing to new learners. Concurrently, artificial intelligence (AI) is increasingly influencing various fields, including computer science education. Inspired by prior work on Sonic Pi and it's impact on student attitudes [21], we aim to explore the potential of AI to enhance live coding environments.

While there is a growing body of research on AI-assisted code generation and, separately, on the pedagogical benefits of live coding, there remains a gap in understanding how AI, particularly Large Language Models (LLMs), affects learning outcomes in the context of live coding for computer science and music composition. This study aims to address the following research question: What impact do LLMs have on the learning process of computer science and music composition through live coding? Investigating this question is crucial, as the intersection of programming education and live coding has had positive outcomes for many students. With the introduction of LLMs, there is both potential for positive and negative outcomes for students learning experiences. For example, LLMs can provide help to generate code that students - getting students to music creation more quickly. However, there is a concern that LLMs may hinder students' creativity and deep thinking, since the LLM will provide the student with the answer, rather than them having to struggle to find the answer themselves.

To explore this question, we conducted a two-day case study, with each session lasting 90 minutes, to observe the effects of AI integration on students' learning experiences. In summary, the key contributions of our work are as follows:

(1) We propose and evaluate a lesson plan that incorporates AI tools for music coding tasks
(2) We assess the impact of LLMs on student creativity in the context of live coding
(3) We analyze and interpret observed changes in student performance and attitudes, providing potential explanations for these outcomes

2 Related Work

2.1 Music and Computer Science Education

Research has consistently shown that students' attitudes play a crucial role in their motivation and learning effectiveness across various educational contexts [6, 9, 22]. Programming skills are increasingly seen as a core part of K-12 education, and are being integrated into curricula [11, 12, 18]. With this, there is a need to understand how to foster positive attitudes towards programming to ensure these curricula are effective [3, 6, 16, 19].

IJSCAR VOL. 1, ISSUE 1

2.2 Tools for Live Coding

Domain-specific programming platforms that combine music and programming have shown potential in motivating fostering positive attitudes towards programming [1, 2, 4, 17, 23]. These tools enable culturally responsive curriculum - allowing students to connect programming with a domain in which they can express themselves, such as music[5, 7, 17, 20, 23].

Three major platforms that combine music and programming for beginners at the school level are:

1. EarSketch: A tool that focuses on remixing pre-existing audio loops using Python, JavaScript, or drag-and-drop code blocks [8].

2. TunePad is a platform that enables users to create music through coding, primarily using Python, and a visual interface. It provides tools for writing, sharing, and remixing musical compositions [10].

3. Sonic Pi: A integrate language and development environment that uses a customized version of Ruby along with SuperCollider for audio synthesis [1].

These platforms differ in their programming languages, design approaches, and interfaces, but all aim to introduce programming concepts through music creation. One major open question is how these platforms, or others for music based programming education, can be augmented with LLM code generation tools.

2.3 LLMs in Programming Education

The use of Large Language Models (LLMs) in programming education is an emerging area of research. As AI technologies continue to advance, tools like ChatGPT and GitHub Copilot are increasingly being integrated into educational settings [15]. Thus far, research has found that LLMs can be used to support students' learning in programming education. For example, Kazemitabaar et al. found that LLMs can be used to support students' learning in programming education [13, 14].

3 Methodology

We designed a quantitative case study to answer the research question: What effect do LLMs have on learning computer science and music composition through live coding? A case study was best suited for this question since it allows us to closely investigate the changes of multiple people at once which is helpful considering our goal.

For this case study, we designed a two day lesson plan for high school students who were new to computer science or music. The lesson plan was based upon prior work on Sonic Pi education [21], however we reduced the scope of the lesson plan to fit into a two day program rather than a three week program. The plan focused on the basic syntax of Sonic Pi in addition to common live coding practices with students free to ask any questions.

The lesson plan design is as follows:

(1) Day 1: students learned the basics of Sonic Pi including looping and modifying samples to get familiar with live coding concepts
(2) Day 2: students learned about the importance of randomness in live coding to then create a final project with the assistance of LLMs like ChatGPT

To make the lessons more engaging, we utilized demonstrations to pique the interest of the students and when explaining specific concepts, we arranged concrete examples that the students could follow along to or ask questions about any confusion they may have with the given examples.

3.1 Tooling

For our study, we utilized the programming environment Sonic Pi[1] to teach students the basics of computer music and computer science. Sonic Pi is a live coding environment that allows students to create music and computer programs in a live setting. Because we were particularly interested in the effect of LLMs on learning computer science and music composition, we utilized Github Copilot and ChatGPT to assist the students live coding. The lesson plan instructed students to install the following software:

- Sonic Pi
- VSCode
- The Github Copilot VSCode Extension
- The Sonic Pi VSCode Extension[1]

3.1.1 Data Collection. To collect the necessary data to answer our research question, we instructed participants to fill out a questionnaire to see their initial level and experience with both programming and music. Our questionnaire was designed as a shortened version of the questionnaire used in prior work [21]. It measured how much programming the participants did in their lives and the same was applied to music. The participants were then asked to briefly describe their current excitement to learn more about programming and music. The responses were then compared to another questionnaire after the study to see if students made any changes in the way that they think as a result of the study. The second questionnaire included most of the questions from the first questionnaire for the most direct comparison possible. For the questions themselves, they were mostly simple in order to decrease the amount of unnecessary variability from a more specific question. For example, "I'm excited to learn how to program".

Students were recruited through direct outreach over email lists and word of mouth. A total of 10 students participated in the study. However, only 5 students completed the both days of the lesson plan. Thus, our sample consisted of 5 people (2 male and 3 female). The research site was online over Zoom. Additionally, the case study method worked well in collecting data that could be used for quantitative and qualitative analysis.

For the quantitative data, "Strongly Disagree" was represented by the value of 1 while "Strongly Agree" represented the value of 5. We then took the average of all the values submitted by the students to get a mean that would be compared between the questionnaires.

4 Results

Three major categories were tested for change and two different questions were asked to measure each category. The three main categories to be looked into are enjoyment, future significance, and difficulty. A sum of 2 would be considered as a "bad" score in a category while a sum of 10 would be a "good" score. Each different colored bar represents a student's rating in each category.

[1] https://marketplace.visualstudio.com/items?itemName=s00500.sonic-pi-extension

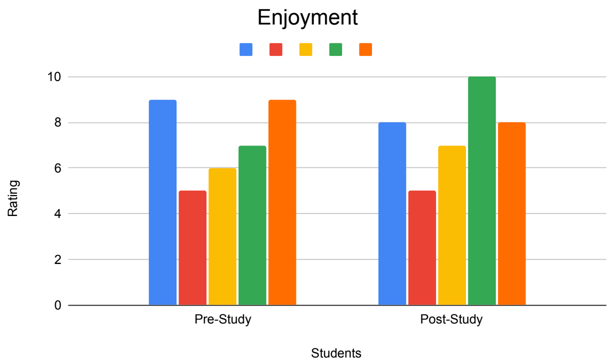

Figure 1: Total Enjoyment in Programming

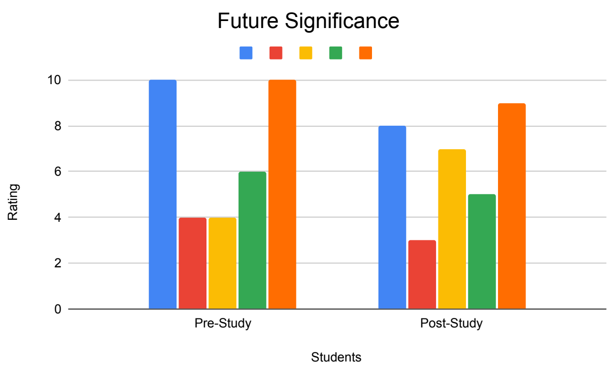

Figure 2: Future Relevance in Programming

Enjoyment was based on how fun the subject was and how excited the students were to learn about it. Future significance was based on how much the subject would be relevant in the future for the student and if they would be willing to study it in college. Finally, difficulty was based on how easy the subject felt and how well the student understood what the subject involves.

Regarding Figure 1, a few students showed a visible increase while others showed no change and a few also rated slightly lower on the second questionnaire compared to the first. According to the surveys, excitement is the main cause for the drop. This could be explained by the fact that people lose some excitement to learn something new if they already recently did so. However, any positive increases were because of an increase in fun while programming.

Unfortunately, programming would be slightly less likely to be chosen as a topic for study in college with the exception of one outlier. This could be an effect of AI and the hindrance of creativity making future studies in programming less worthwhile. However the changes are minuscule so immediate conclusions cannot be made regarding what students think of programming in the future with regards to AI.

In Figure 3, students rated how easy it was to understand the material and how much did they understand about topics involved

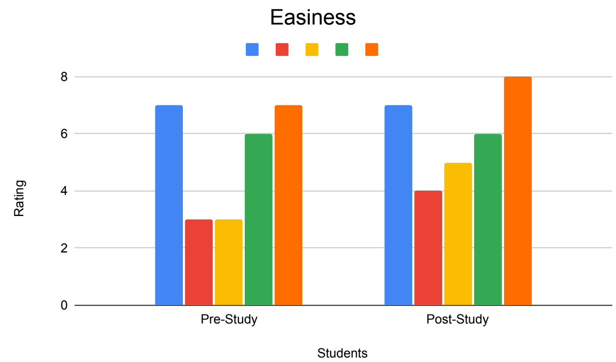

Figure 3: Perceived Difficulty in Programming

with programming. There was an overall increase in student understanding of programming with a few who saw no change. The net positive outcome can be explained by the LLMs ability to generate code which many probably believed made code easier to write and understand with explanations.

Each participant involved in both sessions gave a worded account of their computer and musical experience which acted as data for qualitative analysis. We asked the students to "briefly describe [their] feelings towards learning to code/program". One student responded saying, "I don't know much about it, so there is really nothing to say". Her account matches up with her relatively low ratings in terms of the three main categories. However, when asked the same question again, the student showed a greater interest in programming. As the same student said, "I felt that coding and program is extremely interesting and using ai makes it so much easier". Thus, it is evident that students felt more comfortable with programming as supported by the quantitative data.

5 Discussion

Our results show that LLMs can be useful tools for learning computer science and music composition through live coding. However, we also found that students were not able to use the LLMs as effectively as we hoped. The main challenge was the hurdle of tool installation and setup. We suspect that this was the main reason why some students did not complete the both days of the lesson plan. Furthermore, conducting the study online unfortunately can result in a lower rate of participation, especially for the second day. Some participants noted they did not see the reminder email we had sent for the second session.

While students were able to install Sonic Pi, VSCode, and the VSCode extensions, they had difficulties in getting the Sonic Pi VSCode extension to work. Future work should consider how to best support students in getting the necessary tools to work together on a short timeline. One potential solution could be to create an alternative way for students to interact with Sonic Pi - perhaps by using a fully-self contained Sonic Pi extension that runs a WebAudio Sonic Pi backed in VSCode itself. In addition to better preparation of the necessary tools, a more effective method of communication should be utilized. Besides employing emails, a text message or an app that sends notifications for important events can also be put to use with a higher chance of succeeding. For instance, clarifying the schedules of all participating students before the study would reduce the risk of dropouts by rescheduling.

Additionally, the students were not all able to install the Github Copilot VSCode extension. Future work could consider other editors that are AI-native and do not require additional extensions for LLM integration, such as Cursor[2].

6 Conclusions

This study sought to see what effect LLMs have on learning computer science and music. The results shows a general increase in some areas or a slight decrease or no change in others. Quantitative and qualitative data provided useful information for answering this question with the qualitative data revealing more significant results about the three main categories of enjoyment, future significance, and difficulty.

However, this data in this study is not statistically significant as there are not enough students on both days to make an accurate comparison. Despite the few visible changes, because of this lack of data, no concrete conclusion can be reached. Nevertheless, with more people and better preparation in the setup of tools, a study with more reliable data would be possible. A control group of students who won't have access to AI to complete the same lesson plan would also make for a better comparison to see effects of AI directly.

Acknowledgments

The author would like to to thank Prof. Mark Santolucito for his support in this research project.

References

[1] S. Aaron and A.F. Blackwell. 2013. From sonic Pi to overtone: Creative musical experiences with domain-specific and functional languages. (2013), 35. https://doi.org/10.1145/2505341.2505346

[2] S. Aghaee, A.F. Blackwell, D. Stillwell, and M. Kosinski. 2015. Personality and intrinsic motivational factors in end-user programming. (2015), 29–36. https://doi.org/10.1109/VLHCC.2015.7357195

[3] N. Anderson, C. Lankshear, C. Timms, and L. Courtney. 2008. 'Because it's boring, irrelevant and I don't like computers': Why high school girls avoid professionally-oriented ICT subjects. *Computers & Education* 50, 4 (2008), 1304–1318. https://doi.org/10.1016/j.compedu.2006.12.003

[4] J. Bell and T. Bell. 2018. Integrating computational thinking with a music education context. *Informatics in Education* 17, 2 (2018), 151–166. https://doi.org/10.15388/infedu.2018.09

[5] P. Burnard, Z. Lavicza, and C.A. Philbin. 2016. Strictly coding: Connecting mathematics and music through digital making. (2016), 345–350.

[6] C.S. Cheah. 2020. Factors contributing to the difficulties in teaching and learning of computer programming: A literature review. *Contemporary Educational Technology* 12, 2 (2020), Article ep272. https://doi.org/10.30935/cedtech/8247

[7] L. Cheng. 2018. Teaching live coding of electronic dance music: A case study. *Dancecult* 10, 1 (2018). https://doi.org/10.12801/1947-5403.2018.10.01.10

[8] S. Engelman, B. Magerko, T. McKlin, M. Miller, D. Edwards, and J. Freeman. 2017. Creativity in authentic STEAM education with EarSketch. (2017), 183–188. https://doi.org/10.1145/3017680.3017763

[9] M. Fishbein and I. Ajzen. 1975. Belief, attitude, intention, and behavior: An introduction to theory and research. (1975).

[10] J. Gorson, N. Patel, E. Beheshti, B. Magerko, and M. Horn. 2017. TunePad: Computational thinking through sound composition. (2017), 484–489. https://doi.org/10.1145/3078072.3084313

[11] S. Grover, R. Pea, and S. Cooper. 2016. Factors influencing computer science learning in middle school. (2016), 552–557. https://doi.org/10.1145/2839509.2844564

[12] K. Hava and Z. Koyunlu Ünlü. 2021. Investigation of the relationship between middle school students' computational thinking skills and their STEM career interest and attitudes toward inquiry. *Journal of Science Education and Technology* (2021). https://doi.org/10.1007/s10956-020-09892-y

[13] Majeed Kazemitabaar, Xinying Hou, Austin Henley, Barbara Jane Ericson, David Weintrop, and Tovi Grossman. 2023. How novices use LLM-based code generators to solve CS1 coding tasks in a self-paced learning environment. In *Proceedings of the 23rd Koli Calling International Conference on Computing Education Research*. 1–12.

[14] Majeed Kazemitabaar, Runlong Ye, Xiaoning Wang, Austin Zachary Henley, Paul Denny, Michelle Craig, and Tovi Grossman. 2024. Codeaid: Evaluating a classroom deployment of an llm-based programming assistant that balances student and educator needs. In *Proceedings of the CHI Conference on Human Factors in Computing Systems*. 1–20.

[15] Vassilka D Kirova, Cyril S Ku, Joseph R Laracy, and Thomas J Marlowe. 2024. Software engineering education must adapt and evolve for an llm environment. In *Proceedings of the 55th ACM Technical Symposium on Computer Science Education V. 1*. 666–672.

[16] S.-C. Kong, M.M. Chiu, and M. Lai. 2018. A study of primary school students' interest, collaboration attitude, and programming empowerment in computational thinking education. *Computers & Education* 127 (2018), 178–189. https://doi.org/10.1016/j.compedu.2018.08.026

[17] C. Köppe. 2020. Program a hit – using music as motivator for introducing programming concepts. (2020), 266–272. https://doi.org/10.1145/3341525.3387377

[18] S.Y. Lye and J.H.L. Koh. 2014. Review on teaching and learning of computational thinking through programming: What is next for K-12? *Computers in Human Behavior* 41 (2014), 51–61. https://doi.org/10.1016/j.chb.2014.09.012

[19] J. Margolis and A. Fisher. 2002. *Unlocking the clubhouse: Women in computing*. MIT Press, Cambridge, Massachusetts, USA.

[20] T. McKlin, B. Magerko, T. Lee, D. Wanzer, D. Edwards, and J. Freeman. 2018. Authenticity and personal creativity: How EarSketch affects student persistence. (2018), 987–992. https://doi.org/10.1145/3159450.3159523

[21] Christopher Petrie. 2022. Programming music with Sonic Pi promotes positive attitudes for beginners. *Computers & Education* 179 (2022), 104409.

[22] T. Teo. 2007. Perceived importance, enjoyment, and anxiety as correlates of computer attitudes. *Psychological Reports* 100, 1 (2007), 127–135. https://doi.org/10.2466/pr0.100.1.127-135

[23] D.L. Wanzer, T. McKlin, J. Freeman, B. Magerko, and T. Lee. 2020. Promoting intentions to persist in computing: An examination of six years of the EarSketch program. *Computer Science Education* 30, 4 (2020), 394–419. https://doi.org/10.1080/08993408.2020.1714313

[2]https://www.cursor.com/

A Materials

We list all the materials for the study below.

A.1 Lesson Plan

Our lesson plan covered two consecutive days of 1.5 hour sessions.

Day 1

Suggested Lesson Plan

Introduction: Talking generally about the nature of live coding (10 min)
Part 1: Learn the basic Sonic Pi syntax necessary for live coding (30 min)
Part 2: Learn how to modify samples to create a more elaborate rhythm (40 min)
End: Students may ask any questions about the material (10 min)

Sonic Pi Syntax Learning Goals

- `live_loop`
- `sample`
- `end`
- `sleep`
- `amp`
- `loop do`
- `times do`
- `rate`
- `pitch`
- `use_synth_defaults`

Learning Objectives:

- All students are expected to utilize pre-recorded samples to create short melodies and soundscapes.
- All students should be able to modify samples to create more distinctive sounds.

Day 2

Suggested Lesson Plan

Introduction: Short review of Day 1 material (5 min)
Part 1: Learn how to add randomness in live coding to add interesting variation (30 min)
Part 2: Students will work on an independent project on music composition with their current knowledge of Sonic Pi. They may ask questions during the process (45 min)
End: Students can voluntarily present their projects and share final thoughts (10 min)

Sonic Pi Syntax Learning Goals

- `rrand`
- `choose`
- `lists ([])`

Learning Objectives:

- All students are expected to be creative and make unique soundscapes utilizing randomization.

A.2 Survey Questions

The following questions were asked on a 7-point Likert scale.

1. I think programming is a lot of fun.
2. I think making music is a lot of fun.
3. I am excited to learn how to program.
4. I am excited to learn how to make music.
5. I think programming skills will be relevant in my future.
6. I think making music will be relevant in my future.
7. I might want to study programming in college.
8. I might want to study music composition in college.
9. I think programming is easy.
10. I think making music is easy.
11. I understand the kinds of activities involved with programming.
12. I understand the kinds of activities involved with music composition.

We additionally asked the following short answer questions. These responses were not scored in our study, but only used as a source of quotes and understanding of the participants' experience.

1. Briefly describe your feelings towards learning to code/program.
2. Briefly describe your feelings towards learning how to make music.
3. Have you had any experience in learning how to code before? If so, how long?
4. If you have experience programming, what languages did you use?
5. Have you ever received music education? If so, briefly describe what education and for how long (e.g., piano for 1 year).
6. Have you ever combined programming and music?

Received 07 October 2024; accepted 30 October 2024

Using Model Counting for Game Development: Quantifying Difficulty of 2D Platformer Levels for Diverse Playable Characters

Aditya Patil
aadtiyapatil@gmail.com
MIT Vishwashanti Gurukul
Pune, Maharashtra, India

Mark Santolucito
msantolu@barnard.edu
Barnard College, Columbia University
New York, New York, USA

Abstract

Game development is a field that benefits immensely from using Generative Artificial Intelligence. In particular, level design is a labor-intensive task that can be assisted by AI. However, one challenge in level design is ensuring the balance of the level. To further support level designers, we propose a technique using model counting for the automatic analysis of a level for multiple different playable characters, in order to quantify the difficulty of the level for each character. By using model counting we can analyze a level without the need for human play testing. We implement a prototype of our tool and show the viability of this approach for a 2D platformer.

Keywords

Game Design, Model Counting, Balance, Level Difficulty

ACM Reference Format:
Aditya Patil and Mark Santolucito. 2024. Using Model Counting for Game Development: Quantifying Difficulty of 2D Platformer Levels for Diverse Playable Characters. In *Proceedings of International Journal of Secondary Computing and Applications Research (IJSCAR VOL. 1, ISSUE 1).* ACM, New York, NY, USA, 5 pages. https://doi.org/10.5281/zenodo.14279496

1 Introduction

Level design is a critical aspect of game development, however, it is very labor-intensive. With the emergence of procedural content generation (PCG) techniques, it is possible to automate or create assistance, in the level design process. One of the main challenges however in good level design is ensuring that the levels are balanced, i.e. that they are neither too easy nor too difficult. In particular, one of the challenges in balance for level design is ensuring that the levels are balanced for different playable characters.

For example, in a 2D platformer, the levels should be balanced for different characters with different abilities. In a game like Super Mario Bros. [11], the levels are designed with Mario's abilities in mind. However, in a game like Super Mario Bros. 2 [9], the levels are designed with the abilities of the four playable characters in mind (Mario, Luigi, Toad, and Princess Peach). Each character has different abilities - for example, Luigi can jump higher than Mario but is slower overall. The levels in Super Mario Bros. 2 are designed to be balanced for all four characters, such that the difference in abilities creates varied gameplay, but the overall challenge remains mostly the same for each character.

The question we are interested in is: how can we automatically generate balanced levels for different playable characters? While there exist many techniques for generating levels, there are few techniques for ensuring that the levels are balanced in this way. Thus, the key challenge we tackle is in the quantification of level difficulty for different playable characters. In particular, we aim to develop a level difficulty quantification tool that can be run statically - that is, without requiring players to play test the generated levels. This static analysis for level difficulty is critical for enabling the construction of a fully automated level generation system.

To do this, we propose using model counting to quantify the difficulty of a level. Model counting is a technique for counting the number of solutions to a logical formula. We encode the level as a logical formula, where a satisfying assignment to the formula represents a possible play through of the level. We then use model counting to count the number of possible play through of the level. The intuition is that the more possible play through a level has, the easier it is.

We identify the main contributions of this paper as the following:

(1) We propose a novel approach to quantifying level difficulty for different playable characters using model counting.
(2) We developed a tool that can automatically quantify the difficulty of a level for different playable characters.
(3) We evaluate our tool on a game of Tic-tac-toe as a proof of concept, and show that our tool can accurately quantify the difficulty of the levels. We also identify the need for more scalable model counting techniques and more usable tools for model counting to enable further work in this direction.

2 Related Work

In modern video games *balance* refers to the relative difficulty of a game for various playable characters [1]. The aim of designing a game for good balance is to ensure that no character or strategy has a massive advantage over others. A game is said to have good balance when the success on a level is determined more strongly by skill and strategy rather than inherent character strengths [11].

However, one of the ongoing challenges in video games is maintaining a good balance among different playable characters, especially through regular updates and patches. Games with a wide range of characters often find it hard to make sure each one is equally useful and powerful, which can lead to some characters being too strong while others are rarely used. This issue can be seen in games like Overwatch [6], where the developers have to

IJSCAR VOL. 1, ISSUE 1

constantly tweak character abilities to ensure that no character is too powerful or too weak [16]. One type of game update that should maintain balance is level design (specifically, introduction of new levels as an update).

Automated level design is an area that has seen significant interest in recent years [4]. In automated level design, the main strategy for ensuring balance is to use either play-testing or search-based techniques. In the case of play-testing, the levels are generated and then played by human testers to determine if they are balanced - however this approach requires significant human effort and is too slow to scale [13]. In the case of search-based techniques, the levels are first generated by procedural algorithm, like Perlin noise. Then a search algorithm (often A*) is used to evaluate the candidate levels that have been generated by Perlin noise on measures such as playability and difficulty [4]. After evaluating all the candidate levels with A*, we can then select the one that demonstrates the best playability. Specifically, A* quantifies level difficulty by evaluating the optimal path through each level. It considers path length, where longer paths would mean that the level has a higher difficulty or short paths which would mean the level is comparatively easier. Level difficulty quantification with A* also takes other factors into account like obstacle density and path complexity. Obstacle density is defined as the concentration of obstacles within a specific area or along a path. Path complexity is the number of decision points as each point adds to the challenge of finding the most optimal path of finishing the level.

The main challenge in using search-based techniques that the search algorithm may not fully explore the game space - there may be paths through a given level that are unexplored because the search algorithm did not find them [12]. In contrast, by using model counting, we can quantify the difficulty of a level in a fully automated way, and additionally ensure that we are exploring the entire game space.

3 Background

To make this work self-contained, we give here a background on model counting - the core techniques we use for our level difficulty quantification. Model counting is a computational technique used to determine the number of solutions or configurations that satisfy a given set of constraints [9]. These constraints come in the form of logical statements, often in SAT (Boolean Satisfiability) or SMT (Satisfiability Modulo Theories).

3.1 SAT Solvers

Boolean Satisfiability (SAT) is a well-known NP-complete problem that asks if there is a satisfying assignment to a given Boolean formula. Because of its importance in computer science, many SAT solvers have been developed that can solve SAT problems efficiently, even though it it an NP-complete problem in general [3].

To provide one example of the type of problem that a SAT solver can solve, we can describe the available equipment of a NPC with the following formula:

$$(shield \land sword \land \neg bow) \lor (\neg shield \land \neg sword \land bow)$$

This states that the NPC can either equip a shield and a sword, or a bow, but cannot have, for example, a sword and a bow.

$$(shield \implies sword) \land (\neg(sword \land bow))$$

A slightly modified version of this says that if the NPC has a shield, the NPC must have a sword, and that the NPC cannot have both a sword and a bow. We can then query a SAT solver for satisfying assignments of these variables to generate configurations of equipment for the NPC that obey these rules. For example, an NPC may have only a bow, or a shield and sword, or nothing.

3.2 SMT Solvers

SMT (Satisfiability Modulo Theories) solvers extend the ability of SAT solvers by providing what are called theories. These theories can give the user the ability to write formulas that involve more complex datatypes, for example number or arrays.

To encode a game level into an SMT (Satisfiability Modulo Theories) solver, we must first translate the game's elements, rules, mechanics, and objectives into formal logical constraints and variables. These constraints allow the solver to evaluate properties such as the difficulty of the level or the feasibility of completing it. Each game element—such as platforms, obstacles, enemies, and the player's position—must be represented in a manner that the SMT solver can process and reason about effectively.

As an example again, we can look to generate valid configurations of NPC equipment. This formula describes that the sum of the power of the sword and the power of the shield must be 10:

$$swordPower + shieldPower = 10$$

Then, to find allowable game configurations, we can query an SMT solver for satisfying assignments of these variables.

3.3 Model Counting

Model counting is a technique that extends SAT and SMT solvers to count the number of satisfying assignments to a given formula[8]. To understand how model counting works, we first think through the process of solving the previous SMT problem. The SMT solver would a single possible solution to the formula, for example, $swordPower = 5$ and $shieldPower = 5$. To then do model counting, we can add this solution as a constraint to the formula - telling the SMT solver that is has to find a different solution. Our new formula would look like this:

$$(swordPower + shieldPower = 10) \land$$
$$(swordPower \neq 5 \lor shieldPower \neq 5)$$

In the next round, the SMT solver would find a different solution, for example, $swordPower = 3$ and $shieldPower = 7$. It will continue this process until it finds all the possible solutions to the logical formula.

4 Model Counting Games

We provide an running example of our strategy for level difficulty quantification with Tic-Tac-Toe to show the basic technique. As Tic-Tac-Toe is well studied from a game theory and combinatorics perspective, it serves as a good a proof of concept domain.

The first step is to encode the semantics of the game. For this, we have the following constraints:

- Grid Setup (cf. Sec 4.1: Each cell must either be empty, an "X", or an "O" but not both.
- Turn Alternation (cf. Sec 4.2: The number of "X's" and "O's" on the board must differ by at most 1 to ensure correct turn alternation.
- Winning Condition (cf. Sec. 4.3: Only one player can win, either "X" or "O", but not both simultaneously.

To encode this constraints into an SMT solver, we need to turn these natural language constraints into first order logic. To do this, we the Python bindings for the Z3 solver[5].

4.1 Setting Up the Grid

```
!pip install z3-solver
from z3 import *

board_size = 3
Cells = [[Int(f"Cell_{i}_{j}") for j in range(board_size)
    ] for i in range(board_size)]
solver = Solver()

# Each cell can be 0 (empty), 1 (X), or 2 (O)
for i in range(board_size):
    for j in range(board_size):
        solver.add(Cells[i][j] >= 0, Cells[i][j] <= 2)
```

This code initializes our solver and a model of the board, where each cell in the board can be in one of three states.

The solver then adds constraints ensuring that each cell on the board is either empty, occupied by X or O (but not both), enforcing that no cell can be occupied by both players simultaneously.

4.2 Turn Alternation

```
num_X = Sum([If(Cells[i][j] == 1, 1, 0) for i in range(
    board_size) for j in range(board_size)])
num_O = Sum([If(Cells[i][j] == 2, 1, 0) for i in range(
    board_size) for j in range(board_size)])

# The number of X's and O's should differ by at most 1,
    and X goes first
solver.add(num_X - num_O <= 1)
solver.add(num_O - num_X <= 0)
```

Next, we add constraints to ensure that the number of moves made by each player ('X' and 'O') follows the standard turn-taking rules of Tic-Tac-Toe. First, it counts how many times 'X' and 'O' have been placed on the board. Then, it adds rules to ensure the game progresses correctly: 'X' must always go first, and the number of moves made by 'X' and 'O' can differ by at most one. This ensures that players alternate turns properly, with 'X' never having more than one extra move over 'O', and 'O' never having more moves than 'X'. These constraints prevent invalid game states from being considered. Here we are leveraging the theory of Linear Integer Arithmetic of the SMT solver.

4.3 Defining the Winning Condition

```
def is_winner(player_value):
    row_wins = [And([Cells[i][j] == player_value for j in
      range(board_size)]) for i in range(board_size)]
    col_wins = [And([Cells[i][j] == player_value for i in
      range(board_size)]) for j in range(board_size)]
    diag1_win = And([Cells[i][i] == player_value for i in
      range(board_size)])
    diag2_win = And([Cells[i][board_size - i - 1] ==
     player_value for i in range(board_size)])
    return Or(*row_wins, *col_wins, diag1_win, diag2_win)

# Both players cannot win simultaneously
solver.add(Not(And(is_winner(1), is_winner(2))))

# If X wins, X has played one more move than O
solver.add(Implies(is_winner(1), num_X == num_O + 1))

# If O wins, X and O have played the same number of moves
solver.add(Implies(is_winner(2), num_X == num_O))
```

The `is_winner` function checks if a given player (X or O) has won the game by forming a complete row, column, or diagonal. It returns True if any of these win conditions are met. We also add constraints to ensure the game ends when a player wins. In particular, if someone has won, no further moves are made after the win.

4.4 Model Counting

While tools for model counting exist [2, 14, 15], they are largely focused on SAT solving, and do not work well with Python Z3 bindings. To keep this proof of concept work simple, we want to use Python Z3 bindings, but this means we needed to implement our own model counting technique. To do this we use the below code that creates additional constraints to avoid game configurations that have already been found. It does this by comparing each cell's value in the current model (board) and generating a constraint that forces at least one cell to have a different value in the next solution, ensuring uniqueness.

```
# Exclude the current model from future iterations
current_model_constraints = []
for i in range(board_size):
    for j in range(board_size):
        val = model.evaluate(Cells[i][j])
        current_model_constraints.append(Cells[i][j] ==
    val)
solver.add(Not(And(current_model_constraints)))
```

This is the most basic approach to model counting. It does not scale well, which is why model checking is an active area of research [7], however for the purposes of our work, this is sufficient.

4.5 Assessing Level Difficulty

```
difficulty_o = 1 - (o_wins / total_moves)
difficulty_x = 1 - (x_wins / total_moves)
```

Finally, to use these model counting results to estimate level difficulty between O and X, we compare the number of winning paths for each player to the total number of outcome possibilities for a game. This gives us a sense of how often a random play strategy would win for each player.

```
Results:
Total number of valid configurations: 5478
Total number of X win options: 626
Total number of O win options: 316
Difficulty for O: 0.9423
Difficulty for X: 0.8857
```

The system generates 5478 different valid Tic-Tac-Toe configurations - matching established literature in combinatorics [10].

In summary, the difficulty for player 'X' is calculated to be 0.9423, while O is 0.8857. This suggests that 'X' has a higher chance of winning with a random strategy, thus facing an easier path to victory. This aligns with the notion of the first-move advantage present in Tic-Tac-Toe.

Through this example, we have shown we model counting can be used to assess game difficulty for different playable characters with certain constraints. By changing the constraints for different playable characters, such as player 'O' and player 'X', we can quantify how difficult a level is based on the given constraints

5 Model Counting for a 2D Platformer

We now extend our prior example from Tic-Tac-Toe to a 2D platformer game. Similar to how we encoded player position, winning conditions, and the number of possible solutions as variables in Tic-Tac-Toe, a similar approach can be used in a 2D platformer by encoding multiple variables such as character abilities, enemy behavior, level layout, and collectibles.

In this section, we explore how different constraints, such as movement rules and enemy avoidance, can be encoded and solved using an SMT solver. We also demonstrate how model counting can help estimate the complexity of different characters' paths to a goal.

5.1 Grid Setup and Character Movement

Our platformer is represented as a grid with rows and columns. Each character starts at an initial position and attempts to reach a predefined goal. Different characters have distinct movement abilities, which are modeled using logical constraints.

For instance, in the case of the Tank character, movement is restricted to horizontal directions (left and right), while the Agility character has the ability to move both horizontally and vertically, making it more versatile. The initial setup for the grid and the movement rules for each character are shown in the following code:

```
# Define the grid dimensions and parameters
rows, cols = 2, 4
moves = 5
initial_position = [0, 0]
goal_position = [0, 3]

# Characters' movement abilities
movement_rules = {
    'Tank': [(0, 0), (0, -1), (0, 1)],  # Can stay or move left/right
    'Agility': [(0, 0), (0, -1), (0, 1), (-1, 0), (1, 0)],  # Can move in all directions
}
```

We then encode the constraints for each character's movement across the grid, ensuring that they can only move according to their abilities while avoiding obstacles such as enemies.

5.2 Enemy Avoidance and Path Constraints

A key challenge in platformer games is avoiding enemies or obstacles. In our model, we define enemy positions as static cells on the grid. Characters like the Agility type can have additional constraints to avoid these enemy cells at each time step.

```
# Define enemy positions (static positions on the map)
enemy_positions = [(0, 1), (1, 3)]  # Example enemy positions

# Avoid enemies while moving
add_constraints('Agility', movement_rules['Agility'], avoid_enemies=True)
```

This allows the solver to generate valid paths for characters by avoiding cells that contain enemies, adding an extra layer of complexity to the solution space.

5.3 Model Counting and Path Enumeration

Just as in the Tic-Tac-Toe example, we use model counting to enumerate all possible valid paths for each character to reach the goal. The solver iterates through each possible configuration, ensuring that every path found is unique. We also calculate the total number of paths to give a sense of how many valid solutions exist for each character.

```
# Find paths for each character
tank_paths = find_paths('Tank', max_paths=3)
agility_paths = find_paths('Agility', max_paths=3)
```

The model counting process helps quantify how many paths are available for each character based on their movement abilities and constraints. For example, characters with more versatile movement rules, such as Agility, typically have more potential solutions, while characters like Tank, with more limited movement, may have fewer solutions.

5.4 Level Difficulty Assessment

To assess the relative difficulty of the level for each character, we can compare the number of valid paths that exist for them. Similar to the Tic-Tac-Toe example, a character with fewer valid paths will likely find the level more challenging, whereas a character with many paths will have an easier time.

For this proof-of-concept platformer, we generated several paths for the Tank and Agility characters. Below are the results:

```
Total number of valid paths for Tank: 3
Total number of valid paths for Agility: 1
```

This method of model counting provides a quantitative way to measure and compare the difficulty of different characters navigating the same level. By encoding movement abilities and constraints in an SMT solver, we can systematically explore and analyze the solution space for complex game mechanics.

To test the scalability of this method, we run an experiment where we vary the size of the board and the number of allowed moves. We ran this on a Google Colab online environment, which had an Intel Xeon CPU @ 2.20GHz and 12GB of RAM. We run on boards of increasing size, with the classic dimensions of a 2D side-scrolling platformer - a small height with a long board. The results, as shown in Fig. 1, show that while there is an exponential trend (as expected, as the model counting problem is of #SAT complexity), the values are not unreasonable. The classic 2D platformer, World 1-1 of Super Mario Bros, is 14 tiles tall and 211 tiles long. Although our implementation is too inefficient to handle this (already running

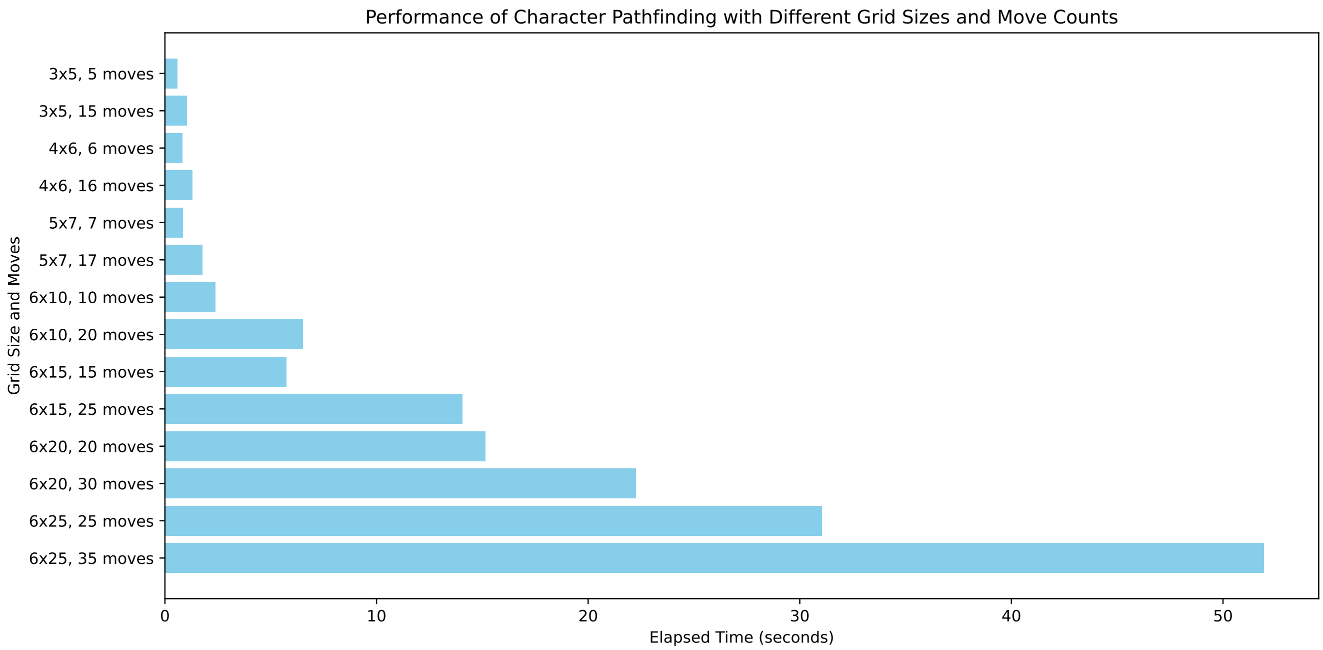

Figure 1: Scalability test of model counting for game level balancing

for 50 seconds on a board of 6 tiles by 25 tiles), with a proper model counting tool [7], we expect this would be possible.

The full implementation of this 2D platformer model counting example is available open-source here: https://github.com/adinotfound11/ModelCounting/blob/main/Tic_tac_toe_Z3.ipynb.

6 Conclusion

We have presented a novel approach to quantifying the difficulty of 2D platformer levels for diverse playable characters through the use of model counting. By encoding game mechanics and constraints into logical formulas and applying model counting techniques, we are able to give a measure of difficulty of a level without the need for extensive human playtesting. We envision this tool being used by level-designers to confirm the balance of levels, either manually designed or designed with the assistance of AI. Ideally, such a tool would be integrated directly into the workflow of the developers. Although further studies are required to validate this approach from a user-testing perspective, we offer model counting as a a first step towards more automated level generation while ensuring game balance.

References

[1] Alexander Becker and Daniel Görlich. 2019. Game Balancing – A Semantical Analysis.
[2] Supratik Chakraborty, Kuldeep S Meel, and Moshe Y Vardi. 2013. A scalable approximate model counter. In *Principles and Practice of Constraint Programming: 19th International Conference, CP 2013, Uppsala, Sweden, September 16-20, 2013. Proceedings 19*. Springer, 200–216.
[3] Koen Claessen, Niklas Een, Mary Sheeran, and Niklas Sorensson. 2008. SAT-solving in practice. *Proceedings - 9th International Workshop on Discrete Event Systems, WODES' 08*, 61 – 67. https://doi.org/10.1109/WODES.2008.4605923
[4] Eric Dallmeier. 2018. *Automated Level Generation for General Video Games*. Ph. D. Dissertation. https://doi.org/10.13140/RG.2.2.24292.71045
[5] Leonardo de Moura and Nikolaj Bjørner. 2008. Z3: an efficient SMT solver. *Tools and Algorithms for the Construction and Analysis of Systems* 4963, 337–340. https://doi.org/10.1007/978-3-540-78800-3_24
[6] Blizzard Entertainment. 2019. Overwatch 2. https://overwatch.blizzard.com/en-us/
[7] Johannes K Fichte, Markus Hecher, and Florim Hamiti. 2021. The model counting competition 2020. *Journal of Experimental Algorithmics (JEA)* 26 (2021), 1–26.
[8] Carla Gomes and Ashish Sabharwal. 2006. Model Counting: A New Strategy for Obtaining Good Bounds. *Proceedings of the National Conference on Artificial Intelligence* 1.
[9] Carla P Gomes, Ashish Sabharwal, and Bart Selman. 2021. Model counting. In *Handbook of satisfiability*. IOS press, 993–1014.
[10] Immanuel (https://math.stackexchange.com/users/93547/immanuel). [n. d.]. TicTacToe State Space Choose Calculation. Mathematics Stack Exchange. arXiv:https://math.stackexchange.com/q/486548 https://math.stackexchange.com/q/486548 URL:https://math.stackexchange.com/q/486548 (version: 2018-05-23).
[11] Florian Rupp. 2023. Learning the Generation of Balanced Game Levels. DC@KI2023: Proceedings of Doctoral Consortium at KI 2023. , 40–49 pages. https://doi.org/10.18420/ki2023-dc-05
[12] Noor Shaker, Julian Togelius, and Mark Nelson. 2016. *Procedural Content Generation in Games*. https://doi.org/10.1007/978-3-319-42716-4
[13] Noor Shaker, Georgios Yannakakis, and Julian Togelius. 2010. Towards Automatic Personalized Content Generation for Platform Games. *Proceedings of the 6th AAAI Conference on Artificial Intelligence and Interactive Digital Entertainment, AIIDE 2010*.
[14] Shubham Sharma, Subhajit Roy, Mate Soos, and Kuldeep S Meel. 2019. GANAK: A Scalable Probabilistic Exact Model Counter.. In *IJCAI*, Vol. 19. 1169–1176.
[15] Mate Soos, Stephan Gocht, and Kuldeep S Meel. 2020. Tinted, detached, and lazy CNF-XOR solving and its applications to counting and sampling. In *International Conference on Computer Aided Verification*. Springer, 463–484.
[16] Xiaofang Zhong and Jinjie Xu. 2022. Measuring the effect of game updates on player engagement: A cue from DOTA2. *Entertainment Computing* 43 (06 2022), 100506. https://doi.org/10.1016/j.entcom.2022.100506

Received 14 October 2024; accepted 12 November 2024