Large Language Model approaches to Educational Video Generation using Manim

Ravidu Suien Rammuni Silva[1], Ahmad Lotfi[1], Isibor Kennedy Ihianle[1], Golnaz Shahtahmassebi[2], and Jordan J. Bird[1]

[1] Department of Computer Science, Nottingham Trent University, Nottingham, UK
Corresponding author: `ravidu.rammunisilva2024@my.ntu.ac.uk`
[2] Department of Physics and Mathematics, Nottingham Trent University, Nottingham, UK

Abstract. The automated generation of educational videos based on Large Language Models (LLMs) holds promising potential to enhance pedagogy. This study investigates the use of compact LLMs with fewer than 10 billion parameters, utilising Parameter-Efficient Fine-Tuning (PEFT) techniques to generate *Manim* Python code that produces images and videos for educational visualisations from natural language narratives. This approach is highly resource-efficient compared to diffusion-based video generation, which is often infeasible on consumer-level hardware. Due to the lack of data availability, this study curated and reviewed the first benchmarking dataset for Manim code generation, ManimBench. ManimBench features 417 paired descriptions and corresponding Manim code samples categorised as basic, intermediate, and advanced, sourced directly from the official ManimCE documentation. The curated dataset was then used to fine-tune seven open-source compact LLMs using LoRA and QLoRA. Subsequently, their performances were evaluated comparatively against their base models using both functional and semantic metrics. The results indicate that the LoRA Fine-Tuned (FT) models consistently outperform the base models, achieving a 62% Manim render success rate. By leveraging ManimBench, further comparisons were made between the FT models and larger 14B parameter LLMs, finding that the PEFT-tuned FT coder models perform comparably well in accurate Manim code generation. However, it was also discovered that LLMs frequently make hallucination-related API errors in the generated code.

Keywords: Generative AI, Large Language Models, Code Generation, Manim Video Generation, LoRA, Educational Video Generation

1 Introduction

With the rise of Large Language Models (LLMs), Artificial Intelligence (AI) has paved the way for new opportunities in enhancing traditional educational methods [1–3]. One particularly useful application of AI in education is the automated generation of educational videos [4, 5]. Projects such as Manim[3], an

[3] Manim Community Edition: https://www.manim.community/

open-source Python library for creating mathematical and scientific visualisation videos, are widely employed to produce accessible, engaging, and easily understandable content [6]. Manim is particularly favoured for this purpose due to its extensive range of built-in STEM-related animation elements that can be tailored to create innovative animations to visualise STEM concepts, alongside its easy-to-code scriptable approach [6]. However, despite the simplicity of the Manim library, writing Manim code remains a challenging and non-trivial task, since it requires programming expertise and domain knowledge. In addressing this issue, the capabilities of LLMs in understanding natural language and the intricate relationships between concepts, coupled with their ability to transform natural language descriptions into executable code, may possess the potential to generate Manim code, ultimately allowing LLMs to produce educational videos.

Commercially available general LLMs such as GPT-4 often struggle to produce executable code for niche libraries featuring domain-specific APIs such as Manim [5, 7]. To the authors' knowledge, no study has specifically benchmarked either open or closed-source LLMs for generating Manim code. Additionally, no reliable dataset has been identified for this purpose. Furthermore, only one preprint was identified that explores the ability of LLMs to explain STEM theorems, TheoremExplainAgent [5], which utilises multiple commercially available closed-source LLMs in an orchestrated approach to create explanatory videos for STEM-related theorems using Manim.

This study investigates the capabilities of compact LLMs in translating natural language descriptions of educational animations into executable and meaningful Manim code using a self-curated dataset. It further explores the effectiveness of employing Parameter-Efficient Fine-Tuning (PEFT) techniques, specifically Low-rank Adaptation (LoRA) [8], for fine-tuning compact LLMs for Manim-based code generation. The key contributions of this study are threefold:

1. **ManimBench**: This study presents a dataset comprising 417 examples of human-reviewed paired explanations and Manim code snippets extracted from the official ManimCE documentation, covering basic, intermediate, and advanced visualisation tasks. The dataset can be freely accessed via Huggingface Hub[4].
2. **Compact Manim fine-tuned LLMs**: This study presents seven compact LLMs with fewer than 10B parameters, fine-tuned and evaluated on the curated dataset using LoRA. An overview of the Manim video generation is illustrated in Figure 1. Figure 2 provides a few prompt-output examples generated using the top FT model.
3. **Comprehensive Evaluation**: Using ManimBench, this study comprehensively evaluates contemporary LLMs that are appropriate in complexity for consumer-level hardware and their LoRA fine-tuning efficacy based on four statistical metrics, Pythonic syntactical correctness, and, most importantly, the success of Manim rendering.

[4] ManimBench Dataset: https://huggingface.co/datasets/SuienR/ManimBench-v1

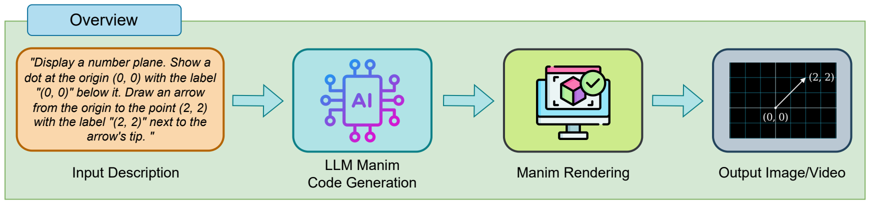

Fig. 1: Overview of the LLM-backed Automated Manim Video Creation Process.

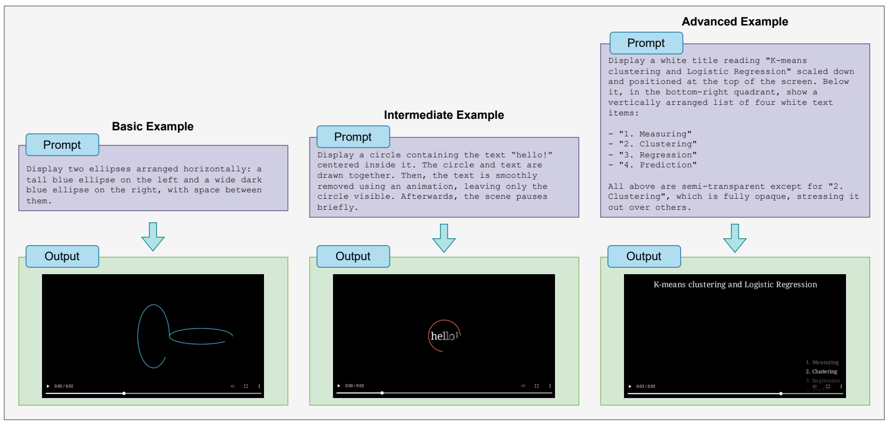

Fig. 2: Example generations constructed from the SeedCoder 8B FT model.

The remainder of this paper is organised as follows: Section 2 provides a concise overview of related work carried out in the field. Section 3 details the theoretical, technical, and implementation aspects of the contributions outlined above, followed by Section 4, which presents the findings from the experiments conducted as part of this study, along with a comprehensive discussion of the results. Finally, potential future avenues for research are discussed, concluding the paper in Section 5.

2 Related Work

Recent advances in LLM capabilities for Natural Language Understanding (NLU) have made them a valuable and effective tool for translating natural language descriptions into code [9–11]. This capability of LLMs has also enabled the generation of code for libraries with domain-specific APIs, such as Manim, which cater to specific use cases, such as mathematical video generation. This section presents related work on employing LLMs to generate Manim code and parameter-efficient methods to enhance their capabilities.

2.1 Domain-specific Code Generation using LLMs

Recent general-purpose LLMs, such as GPT-4 and OpenAI o3, have performed notably well on general coding tasks [9–11]. However, their performance with domain-specific libraries has been suboptimal, largely due to a lack of API knowledge and their limited capacity to incorporate API knowledge into their responses [7,12,13]. Gu et al. [12] introduced DomCoder, a code generation approach, which enhances the quality of generated code for less popular domain-specific libraries. DomCoder encourages the LLM to integrate API knowledge while developing its responses, employing techniques such as chain-of-thought prompting. However, it focuses mainly on web and game development domains. Zheng et al. [7] introduced DomainCodeBench, a code generation benchmark that includes code in multiple programming languages from various domains, such as deep learning and game development. The authors demonstrate that higher scores on general benchmarks do not ensure equivalent performance across domain-specific code generation tasks.

2.2 Visualisation Code Generation using LLMs

Although Manim has not been explicitly focused on in existing studies, several other visualisation libraries, primarily those which generate static images, have been considered for LLM-based generation. Galimzyanov et al. [14] introduced PandasPlotBench, which concentrates on static visualisation tasks across three libraries: Matplotlib, Plotly, and Seaborn. The authors highlight that leading closed-source LLMs perform notably well on widely used libraries such as Matplotlib and Seaborn, but not consistently on the relatively less popular Plotly. Similarly, Tian et al. [15] presented a dataset that can be utilised to fine-tune and evaluate LLMs on visualisation chart generation. Furthermore, the authors emphasise that hallucination issues can frequently arise when generic LLMs are employed for chart generation. Vázquez [16] analysed the performance of ChatGPT with GPT-4 in generating charts in Matplotlib, Plotly, and Altair, noting that GPT-4 was able to produce code for an appropriate graph at least 80% of the time. The author further observes that the open-source models such as LlamaCode were "significantly inferior" to GPT-4. Lee et al. [17] focus on utilising the capabilities of LLM in the field of education, investigating the generation of SVG-based diagrams to visualise mathematical problems. These studies show the potential in LLMs to generate code that can render various animations and visualisations.

However, no study has been found that attempts to benchmark the performance of Manim code generation, apart from TheoremExplainAgent [5], which primarily focuses on explaining STEM theorems by generating Manim videos. Ku et al. [5], in TheoremExplainAgent, present an orchestrated approach using multiple closed-source, general LLMs—one for planning the video and another for code generation. The authors further introduce TheoremExplainBench, which comprises 240 STEM theorems and evaluate their approach based on this dataset. However, the study does not explicitly measure the performance of the LLM in Manim code generation, even with the closed-source LLMs.

2.3 Parameter-Efficient Fine-Tuning (PEFT) for Code Generation

PEFT techniques such as LoRA [8] have been utilised to adapt LLMs for code generation in recent studies. Weyssow et al. [18] investigated how PEFT can enhance code generation in LLMs and found that PEFT techniques such as LoRA are superior to alternatives such as In-Context Learning (ICL) and Retrieval-Augmented Generation (RAG) [19]. Similarly, Hajipour et al. [20] demonstrate that fine-tuning with LoRA can effectively improve the out-of-distribution generalisation of coding LLMs, while Esmaeili et al. [21] show how PEFT can benefit Transfer Learning for coding languages, explicitly focusing on R. However, in contrast, Ayupov and Chirkova [22] found that although the performance advantage of LoRA fine-tuning may achieve results comparable to full fine-tuning in code understanding tasks, it is inferior in code generation tasks. However, the study's findings are based on older models such as CodeT5 and PLBART.

3 Methodology

This section outlines the details of the curated dataset, the experimental setup used in conducting the research, and the results obtained.

3.1 Dataset Curation

During the initial research conducted for this study, no reliable dataset was identified that could evaluate LLMs on their ability to produce executable and accurate Manim code, i.e. code that executes without compilation errors and generates a video representative of the prompt description. This represents a clear gap in the field, prompting the creation of a dataset comprised of 417 code snippets extracted from the official ManimCE documentation[5].

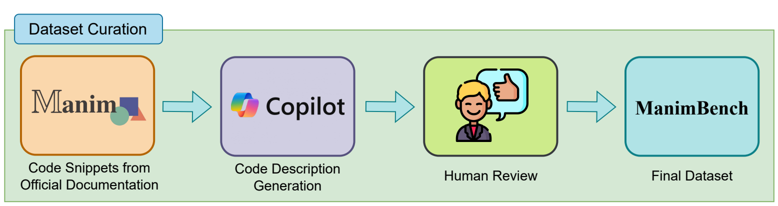

Fig. 3: Overview of the ManimBench Dataset creation.

However, the documentation only contained code snippets with their rendered output, lacking a textual description of the code. To address this, Microsoft Copilot, equipped with the GPT-4o model, was utilised. The generated textual

[5] ManimCE Docs: https://docs.manim.community/

descriptions are then reviewed by the authors, with changes made where necessary. During the review process, it was found that approximately 10% of the generated descriptions required minor edits. The overall process is illustrated in Figure 3. The prompt template used in the dataset curation process is presented along with an example in Figure 4. The curated dataset is available online from Huggingface[6].

Fig. 4: Representative Prompt-response example used in the dataset curation.

Furthermore, description-code pairs were manually categorised, considering the complexity of the code generation task. The snippets were categorised into three categories and divided among train-test splits, ensuring consistency as shown in Table 1. In the categorisation process, three factors were taken into account: 1) the length of the expected code to be generated, 2) the length of the related description, and 3) the complexity of translating between the text-based description and Manim code, considering the ready-to-use components available in the Manim library. Point 3) is particularly important regarding code complexity, as some complex visualisations were readily available out of the box

[6] ManimBench Dataset: https://huggingface.co/datasets/SuienR/ManimBench-v1

with Manim, while others can be challenging to animate within Manim. Figure 5 illustrates the description lengths and code lengths, respectively.

Table 1: Split counts of the ManimBench dataset.

	Split	
Task Type	**Train**	**Test**
Basic	199	60
Intermediate	104	35
Advanced	14	5
Total	**317**	**100**

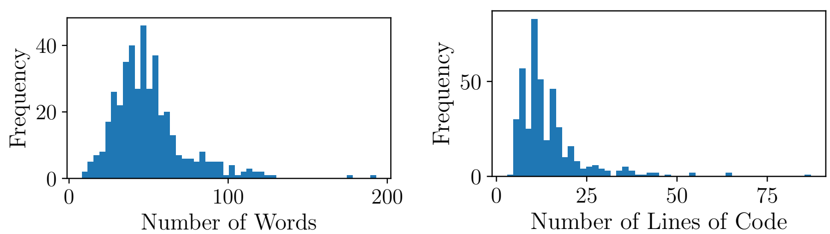

Fig. 5: Histograms of the number of words in Code Description and the number of lines in the expected code.

3.2 Experimental Setup

The models were selected from the Unsloth library[7] for fine-tuning purposes, due to their resource-efficient fine-tuning capabilities. Unsloth's built-in 4-bit quantisation was employed for all the fine-tuned models, with the exception of the Deepseek 1.3B coder model. Kaggle Notebooks equipped with two Nvidia T4 GPUs, each possessing 15 GB of VRAM, were utilised in the fine-tuning process. During the evaluation, an Ubuntu server featuring four Nvidia A2000 GPUs, each equipped with 12 GB of VRAM, was used. ManimCE v0.19.0 was used for rendering evaluation purposes. The code written in this research is available open-source via GitHub[8].

3.3 Model Training and Validation

During model training, the prompt template shown in Figure 6 was utilised. The template was designed to complement the prompt used in dataset curation,

[7] Unsloth Model Library: https://huggingface.co/unsloth
[8] GitHub Repo: https://github.com/SuienS/manim-fine-tune-v1

as shown in Figure 4. The prompt template primarily comprises three sections: 1) instructions, 2) an example, and 3) task information similar to the Alpaca format [23].

````
Prompt Template

Instructions
You are an expert Manim Community Edition (ManimCE) educator and Python
developer.

Please follow these steps precisely:
1. Read the `<TEXT_SCRIPT>` block which contains a description of a Manim
animation that I want to create.
2. DO NOT think out loud or provide any explanations.
3. Generate only executable Python code for Manim, wrapped between `<CODE>` and
`</CODE>`.
4. Do not include any explanations, comments, or instructions on how to run the
code. Only include the code.

Example format:

Example
<TEXT_SCRIPT>
Display a red square centered on screen, then transform it into a circle.
</TEXT_SCRIPT>

<CODE>
```python
from manim import *
class RedSquareToCircle(Scene):
    def construct(self):
        square = Square(color=RED)
        self.play(Create(square))
        circle = Circle(color=RED)
        self.play(Transform(square, circle))
        self.wait()
```
</CODE>

Task Info
Now, generate code for the following text script:

<TEXT_SCRIPT>
{text_script}
</TEXT_SCRIPT>

{response}"""
````

Fig. 6: Inference prompt template used during fine-tuning and evaluation.

All models were trained for three epochs, except for CodeGemma 7B, which was trained for two epochs, and SeedCoder 8B, which was trained for only one epoch. The optimal epoch setting was empirically selected after conducting evaluations in Table 2. The batch size was set to 32 for all models, with the exception of DeepSeek Coder 1.7B, which was configured to 16 due to resource constraints. The learning rate was set to $2e-5$ for all models, accompanied by a warmup ratio of 0.1 and a weight decay of 0.01.

Quantised LoRA [24] was employed for model fine-tuning, primarily for two reasons: 1) the small dataset size and 2) resource constraints. By utilising LoRA, less than 1% of the model's total parameters [8] were fine-tuned. Based on the empirical findings, LoRA configurations were set to $\alpha = 16$ and $r = 8$. LoRA parameters were initialised to zero, and no bias was used for the parameters.

For a comprehensive evaluation, two functional metrics were selected – Manim Render Success Rate and Syntax Error Rate – along with six semantic metrics, as shown in Table 2. The N-Gram Match Score calculates the overlap of a continuous sequence of tokens between the reference and generated code snippets, whereas the Weighted N-Gram Match Score weighs the N-Gram score based on the frequency of the matched N-grams. The Syntax Match Score and AST Distance represent the similarity and difference between the reference and generated code according to the syntactical structure and grammar rules of the programming language. CodeBLEU combines the N-Gram match with the Syntax Match score to provide a code-aware BLEU score.

This wide range of metrics enabled us to holistically evaluate the output based on both the relevance of the generated code compared to the expected code and the successful rendering of the generated Manim code.

4 Results and Discussion

The results in Table 2 demonstrate that LoRA FT models consistently outperform their base models, except for the significantly smaller Deepseek Coder 1.3B model. The Coder models, Qwen2.5 7B Coder and SeedCoder 8B, as anticipated, exhibited considerably better performance than the others and were also comparable to the 14B general models, as shown in Figure 7. Although syntax errors were prevalent among the general models, they matched the coder models in semantic metrics, indicating their comprehension of the task. One notable observation is that the FT version of Qwen 2.5 Coder, with 3B parameters, managed to surpass all non-coder 8B models and equal CodeGemma with 7B in terms of render success, making no syntax errors. Furthermore, higher scores in metrics such as CodeBERT similarity across all models suggest that the chosen models possess some limited knowledge of Manim and produce relevant code compared to the expected code, regardless of the parameter count.

For an in-depth view of the results presented in Table 2, Table 3 offers additional statistics on the top-performing model, the fine-tuned SeedCoder 8B. The statistics reveal that the models entirely fail to produce an identifiable code script for certain test samples, whilst generating very similar code scripts to the expected output, achieving high CodeBLEU scores. Further exploration into the errors made by these models revealed that the majority of the Manim render failures occur due to the lack of knowledge of the ManimCE's API, confirming the findings of previous work [7, 12, 13]. It was identified that in the absence of API knowledge, the models tend to hallucinate API parameters, resulting in errors such as 'ValueError'.

5 Conclusion and Future Work

This study explored the significant potential of compact LLMs for the automated generation of educational videos using Manim. By employing the open-sourced self-curated benchmarking dataset, ManimBench, it was found that LoRA can be effectively utilised to enhance the capabilities of compact LLMs in translating natural language narratives into functional Manim Python code. The findings

Table 2: Pass@1 evaluation results summary of the LoRA fine-tuned (FT) models against base models. The mean value for each metric is presented. The 4-bit quantised version from unsloth was used for each model except for Deepseek Coder 1.7B.

	Functional Metrics		Semantic Metrics					
	Manim Render Success	Syntax Error	CodeBERT Similarity	CodeBLEU	N-gram Match	Weighted N-gram Match	Syntax Match	AST Distance (Norm)
DeepSeekCoder 1.3B Base	**0.25**	0.01	0.9195	0.3063	0.1376	0.1989	0.3585	0.6427
DeepSeekCoder 1.3B FT	0.22	0.01	0.9226	0.3077	0.1415	0.2021	0.3625	0.6357
Qwen3 8B Base	0.30	0.02	0.9132	0.3508	0.1463	0.2182	0.4221	0.5926
Qwen3 8B FT	**0.34**	0.01	0.9152	0.3465	0.1402	0.2169	0.4292	0.6057
Llama3.1 8B Base	0.31	0.01	0.9155	0.3392	0.1443	0.2167	0.4086	0.6170
Llama3.1 8B FT	**0.38**	0.00	0.9184	0.3463	0.1545	0.2209	0.4042	0.6070
CodeGemma 7B Base	0.38	0.01	0.9275	0.3469	0.1550	0.2230	0.4034	0.5931
CodeGemma 7B FT	**0.40**	0.01	0.9184	0.3450	0.1553	0.2194	0.4019	0.5987
Qwen2.5 3B Coder Base	0.32	0.00	0.3384	0.9217	0.1405	0.2110	0.4033	0.6036
Qwen2.5 3B Coder FT	**0.40**	0.00	0.3349	0.9273	0.1409	0.2168	0.3948	0.6063
Qwen2.5 7B Coder Base	0.59	0.00	0.9318	0.3520	0.1579	0.2314	0.4281	0.5546
Qwen2.5 7B Coder FT	**0.61**	0.00	0.9292	0.3483	0.1516	0.2278	0.4320	0.5650
SeedCoder 8B Base	0.61	0.00	0.9285	0.3669	0.1712	0.2291	0.4396	0.5798
SeedCoder 8B FT	**<u>0.62</u>**	0.00	0.9227	0.3676	0.1729	0.2312	0.4381	0.5816

Table 3: Further evaluation statistics for the top performing FT model: Seed-Coder 8B.

	Manim Render Success	Syntax Error	CodeBLEU	CodeBERT Similarity	N-gram Match	Weighted N-gram Match	Syntax Match	AST Distance (Norm)
Base Mean	0.61	0.00	0.3669	0.9285	0.1712	0.2291	0.4396	0.5798
Finetuned Mean	0.62	0.00	0.3676	0.9227	0.1729	0.2312	0.4381	0.5816
Base Std	0.4902	0.00	0.1351	0.1009	0.1380	0.1607	0.1807	0.1940
Finetuned Std	0.4878	0.00	0.1320	0.1096	0.1374	0.1607	0.1827	0.1955
Base Min	False	False	0.1378	0.5256	0.00	0.00	0.00	0.129
Finetuned Min	False	False	0.1455	0.5256	0.00	0.00	0.00	0.1290
Base Max	True	False	0.8573	0.9973	0.8507	0.9286	0.9091	0.9972
Finetuned Max	True	False	0.8573	0.9973	0.8507	0.9286	0.9091	0.9979

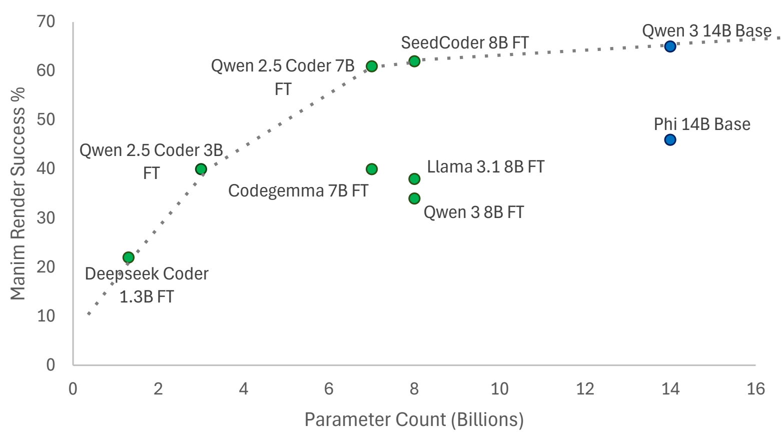

Fig. 7: Pass@1 performance comparison. Fine-tuned models are represented in green, and the base models are represented in blue.

indicate that LoRA FT models consistently outperformed their base counterparts in rendering success, except for a significantly smaller LLM with around 1 billion parameters. However, it was observed that, despite advancements in performance, all tested LLMs committed hallucination-related API errors in their generated Manim code, indicating a potential area for improvement in future work. These errors further emphasise the need for additional analysis and validation. It can also be noted that future work could utilise techniques such as Chain-of-Thought reasoning and Reinforcement Learning to improve this performance, incorporating API details into the prompt.

This work has explored the bridging of natural language to Manim Code, ultimately bridging natural language to educational visualisations and videos, thereby simplifying the creation of educational videos. Furthermore, the authors of this study advocate for further research into scalable, AI-powered solutions for educational content creation, which could potentially enhance traditional pedagogical methods.

References

1. L. Chen, P. Chen, and Z. Lin, "Artificial intelligence in education: A review," *Ieee Access*, vol. 8, pp. 75 264–75 278, 2020.
2. Q. Wen, J. Liang, C. Sierra, R. Luckin, R. Tong, Z. Liu, P. Cui, and J. Tang, "Ai for education (ai4edu): Advancing personalized education with llm and adaptive learning," in *Proceedings of the 30th ACM SIGKDD Conference on Knowledge Discovery and Data Mining*, 2024, pp. 6743–6744.
3. I. Adeshola and A. P. Adepoju, "The opportunities and challenges of chatgpt in education," *Interactive Learning Environments*, vol. 32, no. 10, pp. 6159–6172, 2024.
4. O. Weerakoon, V. Leppänen, and T. Mäkilä, "Enhancing pedagogy with generative ai: Video production from course descriptions," in *Proceedings of the International Conference on Computer Systems and Technologies 2024*, 2024, pp. 249–255.
5. M. Ku, T. Chong, J. Leung, K. Shah, A. Yu, and W. Chen, "Theoremexplainagent: Towards multimodal explanations for llm theorem understanding," *arXiv preprint arXiv:2502.19400*, 2025.
6. M. C. D. , "Manim community," www.manim.community, 2025. [Online]. Available: https://www.manim.community/
7. D. Zheng, Y. Wang, E. Shi, H. Zhang, and Z. Zheng, "How well do llms generate code for different application domains? benchmark and evaluation," *arXiv preprint arXiv:2412.18573*, 2024.
8. E. J. Hu, Y. Shen, P. Wallis, Z. Allen-Zhu, Y. Li, S. Wang, L. Wang, W. Chen *et al.*, "Lora: Low-rank adaptation of large language models." *ICLR*, vol. 1, no. 2, p. 3, 2022.
9. O. , "Introducing openai o3 and o4-mini," Openai.com, 2025. [Online]. Available: https://openai.com/index/introducing-o3-and-o4-mini/
10. J. Wang and Y. Chen, "A review on code generation with llms: Application and evaluation," in *2023 IEEE International Conference on Medical Artificial Intelligence (MedAI)*. IEEE, 2023, pp. 284–289.
11. A. Bucaioni, H. Ekedahl, V. Helander, and P. T. Nguyen, "Programming with chatgpt: How far can we go?" *Machine Learning with Applications*, vol. 15, p. 100526, 2024.

12. X. Gu, M. Chen, Y. Lin, Y. Hu, H. Zhang, C. Wan, Z. Wei, Y. Xu, and J. Wang, "On the effectiveness of large language models in domain-specific code generation," *ACM Transactions on Software Engineering and Methodology*, vol. 34, no. 3, pp. 1–22, 2025.
13. S. Joel, J. J. Wu, and F. H. Fard, "A survey on llm-based code generation for low-resource and domain-specific programming languages," *arXiv preprint arXiv:2410.03981*, 2024.
14. T. Galimzyanov, S. Titov, Y. Golubev, and E. Bogomolov, "Drawing Pandas: A Benchmark for LLMs in Generating Plotting Code," *arXiv preprint arXiv:2412.02764*, 2024.
15. Y. Tian, W. Cui, D. Deng, X. Yi, Y. Yang, H. Zhang, and Y. Wu, "ChartGPT: Leveraging LLMs to Generate Charts From Abstract Natural Language," *IEEE Transactions on Visualization and Computer Graphics*, vol. 31, no. 3, pp. 1731–1745, Mar. 2025.
16. P.-P. Vázquez, "Are LLMs ready for Visualization?" in *2024 IEEE 17th Pacific Visualization Conference (PacificVis)*. Tokyo, Japan: IEEE, Apr. 2024, pp. 343–352.
17. J. Lee, J. Lee, W. Feng, and A. Lan, "From Text to Visuals: Using LLMs to Generate Math Diagrams with Vector Graphics," 2025.
18. M. Weyssow, X. Zhou, K. Kim, D. Lo, and H. Sahraoui, "Exploring Parameter-Efficient Fine-Tuning Techniques for Code Generation with Large Language Models," *ACM Transactions on Software Engineering and Methodology*, p. 3714461, Jan. 2025.
19. P. Lewis, E. Perez, A. Piktus, F. Petroni, V. Karpukhin, N. Goyal, H. Küttler, M. Lewis, W.-t. Yih, T. Rocktäschel, S. Riedel, and D. Kiela, "Retrieval-Augmented Generation for Knowledge-Intensive NLP Tasks," in *Advances in Neural Information Processing Systems*, H. Larochelle, M. Ranzato, R. Hadsell, M. F. Balcan, and H. Lin, Eds., vol. 33. Curran Associates, Inc., 2020, pp. 9459–9474.
20. H. Hajipour, N. Yu, C.-A. Staicu, and M. Fritz, "SimSCOOD: Systematic Analysis of Out-of-Distribution Generalization in Fine-tuned Source Code Models," in *Findings of the Association for Computational Linguistics: NAACL 2024*. Mexico City, Mexico: Association for Computational Linguistics, 2024, pp. 1400–1416.
21. A. Esmaeili, I. Saberi, and F. H. Fard, "Empirical Studies of Parameter Efficient Methods for Large Language Models of Code and Knowledge Transfer to R," 2024.
22. S. Ayupov and N. Chirkova, "Parameter-Efficient Finetuning of Transformers for Source Code," 2022.
23. R. Taori, I. Gulrajani, T. Zhang, Y. Dubois, X. Li, C. Guestrin, P. Liang, and T. B. Hashimoto, "Alpaca: A strong, replicable instruction-following model," *Stanford Center for Research on Foundation Models. https://crfm.stanford.edu/2023/03/13/alpaca.html*, vol. 3, no. 6, p. 7, 2023.
24. T. Dettmers, A. Pagnoni, A. Holtzman, and L. Zettlemoyer, "QLoRA: Efficient Finetuning of Quantized LLMs," in *Advances in Neural Information Processing Systems*, A. Oh, T. Naumann, A. Globerson, K. Saenko, M. Hardt, and S. Levine, Eds., vol. 36. Curran Associates, Inc., 2023, pp. 10 088–10 115.