

Towards No-Code Programming of Cobots: Experiments with Code Synthesis by Large Code Models for Conversational Programming

Chalamalasetti Kranti¹, Sherzod Hakimov¹, and David Schlangen^{1,2}

¹Computational Linguistics, Department of Linguistics
University of Potsdam, Germany

²German Research Center for Artificial Intelligence (DFKI), Berlin, Germany
{kranti.chalamalasetti, sherzod.hakimov, david.schlangen}@uni-potsdam.de

Abstract: While there has been a lot of research recently on robots in household environments, at the present time, most robots in existence can be found on shop floors, and most interactions between humans and robots happen there. “Collaborative robots” (cobots) designed to work alongside humans on assembly lines traditionally require expert programming, limiting ability to make changes, or manual guidance, limiting expressivity of the resulting programs. To address these limitations, we explore using Large Language Models (LLMs), and in particular, their abilities of doing in-context learning, for conversational code generation. As a first step, we define RATS, the “Repetitive Assembly Task”, a 2D building task designed to lay the foundation for simulating industry assembly scenarios. In this task, a ‘programmer’ instructs a cobot, using natural language, on how a certain assembly is to be built; that is, the programmer induces a program, through natural language. We create a dataset that pairs target structures with various example instructions (human-authored, template-based, and model-generated) and example code. With this, we systematically evaluate the capabilities of state-of-the-art LLMs for synthesising this kind of code, given in-context examples. Evaluating in a simulated environment, we find that LLMs are capable of generating accurate ‘first order code’ (instruction sequences), but have problems producing ‘higher-order code’ (abstractions such as functions, or use of loops).

Keywords: Program Synthesis, Cobots, Repetitive Assembly Tasks

1 Introduction

Collaborative robots (cobots), designed to work safely alongside humans, have traditionally required expert programming [1, 2], hindering wider accessibility for novice workers. To bridge this gap, conversational programming is emerging as a possible solution, demanding systems that can parse human input, grasp context, and craft corresponding programs [3] interactively. While historically methods have relied on domain-specific model training and learning by demonstration, these techniques often fall short due to their extensive data needs and inability to handle complex instructions [4, 5, 6, 7]. This necessitates exploring efficient alternative approaches for cobot programming, such as program synthesis.

Program synthesis from natural language instructions (NL2Code) is an active research area in Natural Language Processing, leveraging LLMs for tasks like code completion, debugging, and generating programs from natural language descriptions [8, 9, 10, 11]. In robotics, LLMs have been increasingly used for task planning [12, 13, 14, 15], grounding [16, 17, 18], and instruction following, focusing primarily on immediate, specific actions [19, 20, 21, 22] for “here and now” scenarios.

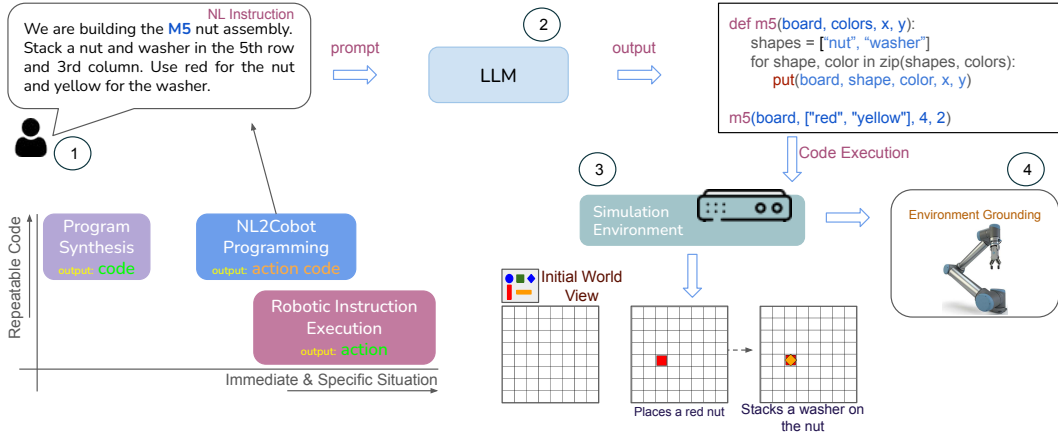


Figure 1: Overview of NL2Cobot Programming, demonstrating the abstraction of user instructions into a high-level executable function and its execution in a simulated environment.

However, replicating these instructions in new environments require generating a new set of actions each time, which can be inefficient and cumbersome. Instead, abstracting robot actions as higher-order programs makes them adaptable, repeatable, and generalizable to novel environments, significantly improving flexibility and efficiency, which is essential for industrial contexts. Our research (see Figure 1) sits between traditional program synthesis (generating code for programmers who understand it) and robot instruction following (generating robot actions, not repeatable programs).

The proposed work focuses on the step from natural language to programs for building assemblies in a construction setting. As such, it is related to that of Paetzel-Prüsmann et al. [23]; however, we abstract away questions of the physical execution of the programs on a robot and instead simulate the setting and the effects of pick and place operations.

Our key contributions are as follows:

- A component assembly dataset with different categories of objects and styles of instructions (Section 3)
- An evaluation methodology to assess program synthesis capabilities of LLMs (Section 4)
- An experimental setup and comprehensive analysis of differences in program synthesis behavior between commercial and open-access code generation LLMs (also Section 4)

2 Related Work

Program Synthesis: There exist several natural-language-to-code (NL2Code) datasets [24, 25, 26, 27] for training machine learning models [8, 9, 28, 29, 10, 11]. Numerous studies have utilized these datasets to explore the capabilities of LLMs, as surveyed by [30, 31, 32]. While these studies demonstrate the ability of LLMs to generate and complete code, they predominantly target users familiar with programming concepts and terminologies. In contrast, our work focuses on code generation in an industrial context, where the end-users are novice workers without programming backgrounds. This raises challenges in dealing with ambiguous instructions, domain-specific knowledge that LLMs might not be aware of, and insufficient context, making it a necessary area for exploration.

Building Tasks: In the virtual block world of Minecraft, datasets have emerged from structure-building tasks [33, 34, 35], linking instructions with actions in a 3D grid. These datasets resemble industry-assembly tasks involving detailed instructions and actions. However, these tasks’ high complexity, lack of spatial constraints and intricate instructions challenge even the best current LLMs [36], limiting their applicability to such collaborative 3D building tasks. Furthermore, these datasets lack tasks involving repeating target structures, which is common in cobot applications. On

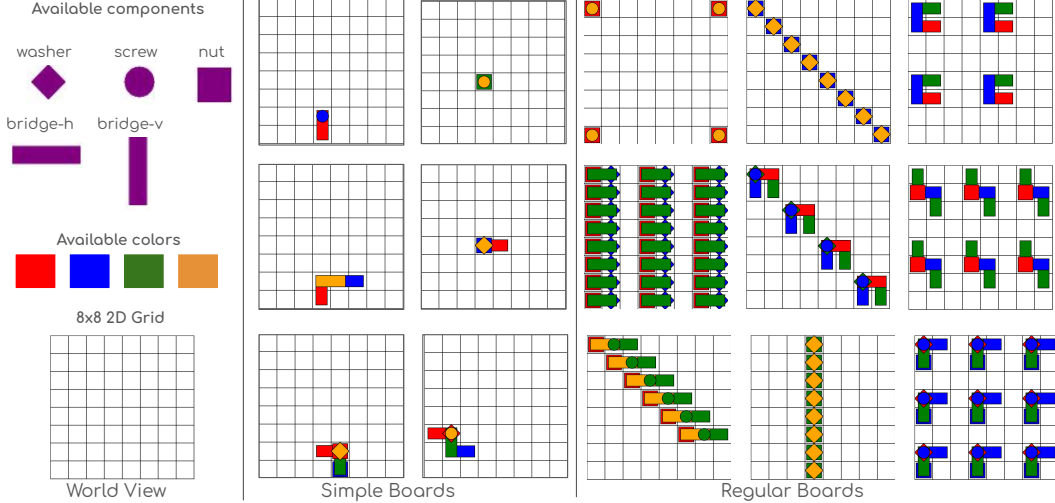


Figure 2: Overview of the environment with available components, and samples of simple and regular boards.

the other hand, LLMs have shown promising performance in abstract spatial tasks [37, 38] in 2D controlled environment, highlighting their abstraction, reasoning and execution capabilities. This discrepancy inspired us to propose a 2D building task that is tailored for controlled industrial environments and includes repetitive tasks and giving us an opportunity to measure different aspects of LLMs abilities in synthesizing code.

Instruction-Following Tasks: A favored domain in this space is human-agent interaction for household tasks [39, 40, 41], where the focus is on a single execution from a given starting point of command. This is typically framed as generating a sequence of actions (and hence a straightforward form of program which is not intended to be repeated). Similar to our setting, *HEXAGONS* dataset [42] aims to translate natural language instructions into programs potentially using higher-level constructs such as loops. However, this dataset focuses more on simulating drawing rather than constructions; as we will see, our dataset puts more weight on introducing assemblies that, in turn, can be built out of smaller assemblies.

3 Simulating Industry-Style Assembly with 2D Building Tasks

Our proposed 2D building task takes inspiration from *HEXAGONS* [42] and *MINECRAFT Collaborative Building* [33]. The goal, just as in those datasets, is to reconstruct a given target structure; here, however, the structures strongly suggest a recursive analysis (e.g., “A being built out of two Bs; with the board in turn being assembled out of repeated As”), with meaning given to the sub-components (e.g., “now D is built out of Bs again, but this time we need three Bs”).

The simulated environment features a 2D grid where each cell can hold configurations of elementary components. Specifically, we have modeled common industrial components: *washers*, *nuts*, *screws*, and *bridges* (horizontal or vertical), available in colors: *red*, *green*, *blue*, and *yellow*. Bridges span two cells (either horizontally or vertically), while washers, nuts, and screws occupy a single cell. There are additional rules about what configurations are legal (see Section 7.1 in Appendix 7).

A target structure is a specific arrangement of components on the 2D grid. If components are connected (imagine that this would allow electricity to flow through the components), we refer to the configuration as constituting a single *object*. *Simple boards* contain only one single object, whereas *regular boards* consist of regular patterns of a single object type. Examples are shown in Figure 2.

3.1 Target Structure Generation

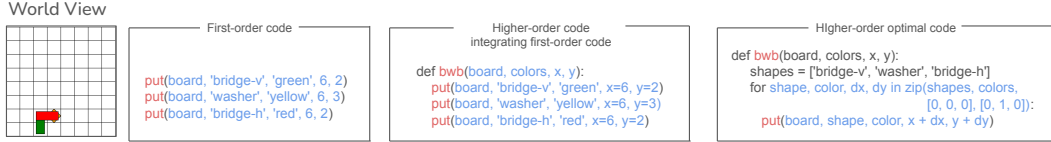


Figure 3: Various gold standard code styles represent the target structure: (a) features first-order code snippet using primitive controls, (b) code includes higher-order functions built from this first-order code, and (c) code featuring optimal higher-order functions.

To facilitate the creation of objects, we represent the 2D grid as an array. This array-based representation allows for precise control and manipulation of shape placements. By abstracting the arrangement of components on the grid into a Python program, we can systematically generate numerous objects. To control task complexity, we defined a limited number of such python programs (referred to as seeds, and expressed using *Jinja2*¹ templates) on an 8x8 grid and extrapolated them with all the possible combinations of components, colors and locations. Table 1 details these seeds and the number of possible objects and boards across the four quadrants of the grid. The code snippet representing each board serves as the gold-standard code during the evaluation. We further expand these programs to generate two additional forms using *Jinja2* templates (see Figure 3): (a) *first-order code*, involving a series of *put* commands, and (b) a *higher-order function* that integrates these first-order code sequences.

3.2 Instruction Generation

At this step, we have pairs of target boards and code (in three variations) that generates them. Next, we add appropriate natural language instructions that verbalise the code and describe the target board, in three different ways. Figure 4 showcases the different types of instructions.

Template-based: To generate natural language instructions automatically, we use templates (see Section 7.2 in Appendix 7) that include grammar entries defined in *Jinja2* format (inspired by work on data synthesis in other domains, e.g. [43, 44]). These templates generate detailed, unambiguous instructions for the target board reconstruction.

Human-written: In addition to template-based instructions, we curated human-written instructions to add natural and varied linguistic style for the target boards in the test set (see Table 2). Using the *slurk* [45] framework, we implemented an interface to collect human-written instructions for the target boards. We recruited one human participant for this task. The participant’s task was to prepare instructions to reconstruct a given target board. Specifics of the setup is available in Section 7.3 in Appendix 7.

Model-generated: We employed an LLM to generate the required instructions to reconstruct the target board. The rationale behind using LLMs for this process includes their adaptability to a) generate instructions with varying nuances, b) maintain a standard and consistent format, and c) support multiple languages. Furthermore, experiments evaluating the effectiveness of such instructions for code generation offer opportunities for automation. Similar to human-written instructions, we generated these instructions for the target boards in the test set (see Table 2). Figure 4 showcases the instructions generated by three code-generation LLMs for reconstructing the board. Particulars on the input representation (ASCII representation of the target board is used in probing the model), prompt settings etc. are discussed in detail in Section 7.4 in Appendix 7.

Board Type	Object Type	NS	NO	NB
Regular	Simple	18	68	152,352
	Complex	10	68	156,948

Table 1: Distribution of generated boards across all categories; NS: number of seeds, used to generate the boards, NO: number of distinct objects, NB: number of boards generated using all color and shape combinations

¹<https://palletsprojects.com/p/jinja/>

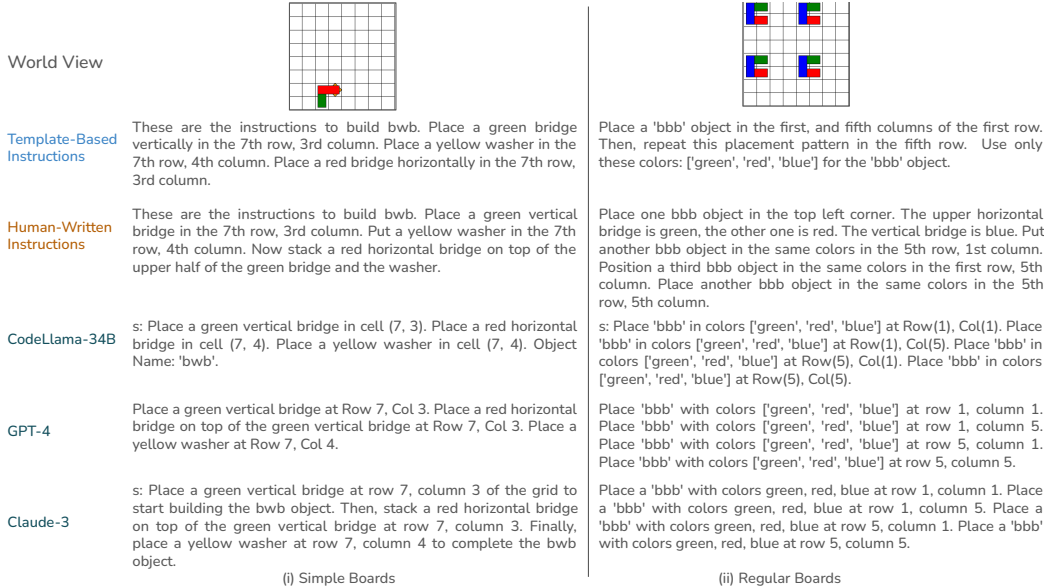


Figure 4: Three types of instructions pair with the *simple* and *regular* boards. Template-based instructions are generated using a template grammar. Human-author instructions are prepared by a human instructor. The remaining instructions are generated by three LLMs that describe the target.

4 Experimental Setup: From Language to Robot Programs

At this point, we now have tuples consisting of a) code and b) natural language expressions, c) both of which describe the same target structure. With this setup, we can investigate to what extent LLMs can realise a function that takes NL expressions into code, with the meaning (the target structure) as invariant; where the function retrieval is possibly being helped by in-context learning from examples.

We design our experiments to delve deeply into the generalizations [46] these models can make. We are interested in the following aspects of the process: a) *Property Compositionality* - can models generalize from in-context examples that are similar yet vary in properties such as shape, color, and location? By curating examples that differ from the test instructions, we assess whether the model is merely copying or generalizing. b) *Function Compositionality* - do the semantic understanding and pattern matching abilities of LLMs aid in generating higher-order code? and c) *Function Repeatability* - can LLMs detect and reason how to repeat higher-order functions as per input instructions? This aspect tests LLMs’ on two fronts: understanding abstract instructions and optimizing code for repetitions. Such an approach is particularly relevant in cobot programming, where repetitive tasks demand optimal handling.

4.1 Setup

Following previously reported prompting approaches [47, 12, 19], we constructed a multi-part prompt, which we validated through an ablation study (see Figure 7a, Table 4f in Appendix 7). We used instruction-tuned code generation LLMs such as *GPT-4* (version 1106-preview), *CodeLlama-34b-instruct* [29] and *Claude-3* (version *opus*), with a *temperature* of 0 and a *max_new_tokens* limit of 250.

For evaluation purposes, we randomly selected 130 target boards (see Table 2) modeled on a 8x8 grid (see Figure 2). Training split samples featured

Board Type	Object Type	Total Boards		
		Train	Val	Test
Simple	Simple	1072	130	130
Regular	Simple	1168	130	130
	Complex	2944	130	130

Table 2: Breakdown of board and object types for training, validation, and test splits. Out of all the possible boards available (see Table 1), these samples were randomly selected.

target boards in the grid’s top-left quadrant, validation samples in the top-right, and test samples in either bottom quadrant, ensuring coverage of all grid areas. The training split is used exclusively for in-context samples (see Figure 6 in Appendix 7.5). Our ablation study (detailed in Appendix 7.5.1) indicated that using *five* in-context examples yields the best results across various LLMs. We ensured the in-context examples did not share combinations of component types, and locations with the test instruction, which offers an opportunity to measure the LLMs responses for un-seen attributes and instructions. In real-world application, we lack control over in-context examples, and related examples are not necessarily detrimental. Thus, our results provide a lower bound on performance. The validation set is used for ablation studies, and the test set is used for evaluation.

4.2 Evaluation Metrics

Compared to program synthesis and machine translation, our proposed task benefits from a known target configuration (i.e., a fully specified intended semantics), enabling a more nuanced evaluation of the generated output. First, we use exact match (EM) to compare the generated code with the gold-standard code (of the target structure) at the token level, assessing the LLMs’ ability to produce semantically identical outputs. Second, the CodeBLEU score [48] evaluates the structural and functional quality of the generated code, testing the LLMs’ capability to generate lexically and semantically correct code. Finally, we use execution success (ES) to compare the reconstructed structure with the gold-standard structure in terms of type, color, and location of elements to measure the LLMs’ success in accurately reconstructing the target. This comprehensive evaluation (see Figure 11, Section 7.7 in Appendix 7) examines semantic understanding, code quality, and precise execution abilities of the LLMs.

5 Results and Analysis

As discussed in Section 4, we investigate, how well LLMs benefit from in-context learning. Table 3 shows LLMs performance on various aspects of program synthesis and the impact of component arrangement complexity. Overall these models perform better on template-based instructions and the performance degraded for human-written and model-generated instructions. Detailed error analysis is provided in Appendix (Table 5 in Section 7.6).

Property Compositionality: We assess if LLMs can synthesize input instructions into first-order code for unseen instructions and attributes. High scores across all measurements (EM, CB, and ES) show that LLMs are good at interpreting the instruction, extracting the attributes such as color, shape, and location, mapping the locations to the current environment (an 8x8 grid), and generating semantically identical code. This demonstrates that domain specific first-order code generation is achievable with few-shot prompting in both open-source and closed API-based LLMs.

Function Compositionality: We further investigate if LLMs show high performance on functional compositionality, assessing their ability to generate higher-order functions by abstracting the functionality into reusable programs. Zero scores for EM metric indicate that the generated code is not semantically identical to the ground truth and may have formatting discrepancies. Lower ES scores indicate that LLM-generated code struggles to accurately reconstruct the target board, highlighting the challenges in complex scenarios.

Manual analysis of model responses (as illustrated in Figure 5), revealed factors reducing the execution scores. CodeLlama-34b had location-related errors in 89% of template-based and 95% of human-written instructions, causing depth mismatches and incorrect placements. GPT-4 and Claude-3 also had location-related errors, with all errors in GPT-4 and 95% in Claude-3 due to incorrect locations. This indicates that in-context learning is less-effective for complex function generation and requires further exploration.

Function Repeatability: We continue our analysis of LLMs abilities in generating loops, including nested ones for function repeatability. Zero EM scores result from strict matching criteria. While models perform well with simple objects, they struggle with complex objects, due to intricate pat-

Board Type	Object Type	Task	Model	EM	CB	ES
Simple	Simple	Property Compositionality	CodeLlama	0.97	0.99	0.97
			GPT-4	1.00	1.00	1.00
			Claude-3	1.00	1.00	1.00
		Function Compositionality (Using sequences of first-order code)	CodeLlama	0	1.00	0.96
			GPT-4	0	1.00	1.00
			Claude-3	0	0.75	0.93
		Function Compositionality (Using optimal higher-order code)	CodeLlama	0	0.95	0.52
			GPT-4	0	0.95	0.56
			Claude-3	0	0.97	0.87
Regular	Simple	Function Repeatability	CodeLlama	0	0.99	0.75
			GPT-4	0	0.96	1.00
			Claude-3	0	0.95	0.86
	Complex		CodeLlama	0	0.22	0.09
			GPT-4	0	0.49	0.30
			Claude-3	0	0.51	0.10

(a) Template-based Instructions

Board Type	Object Type	Task	Model	EM	CB	ES
Simple	Simple	Function Compositionality (Using optimal higher-order code)	CodeLlama	0	0.95	0.28
			GPT-4	0	1.00	0.17
			Claude-3	0	0.79	0.43
Regular	Simple	Function Repeatability	CodeLlama	0	0.34	0.07
			GPT-4	0	0.49	0.31
			Claude-3	0	0.69	0.25
	Complex		CodeLlama	0	0.08	0.07
			GPT-4	0	0.09	0.28
			Claude-3	0	0.11	0.04

(b) Human-written Instructions

Board Type	Object Type	Task	Model	EM	CB	ES
Simple	Simple	Function Compositionality (Using optimal higher-order code)	CodeLlama	0	0.64	0.04
			GPT-4	0	0.85	0.22
			Claude-3	0	0.92	0.23
Regular	Simple	Function Repeatability	CodeLlama	0	0.19	0
			GPT-4	0	0.19	0.13
			Claude-3	0	0.09	0.33

(c) Model-generated Instructions

Table 3: Code generation LLMs’ performance across aspects of program synthesis. Evaluation includes atomic component placement (first-order code) and sequence arrangement (higher-order function generation), assessing the LLMs’ ability to translate natural language input into executable code; EM - Exact Match, CB - Code BLEU, ES - Execution Success.

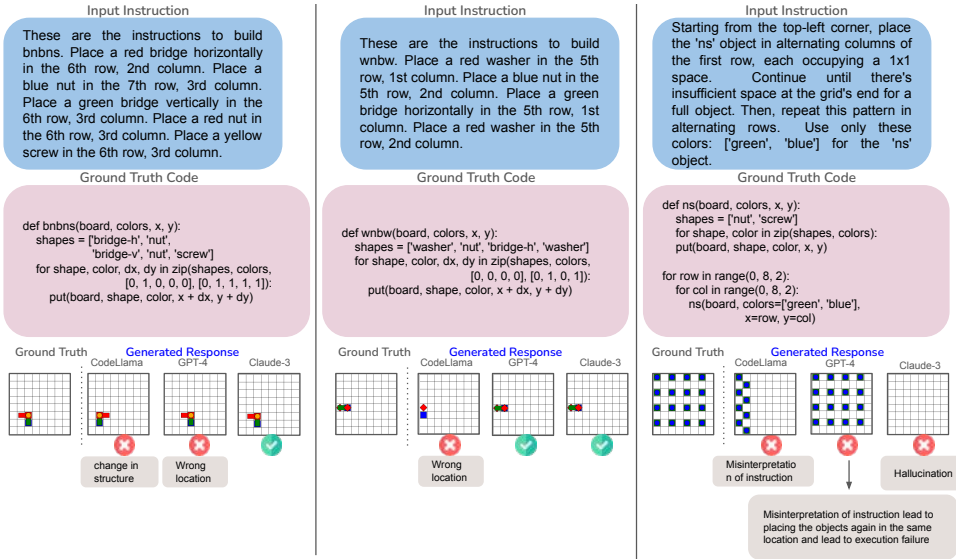


Figure 5: Execution response of the code generated by models. It highlights the types of errors encountered, including incorrect object placement, structural distortions, and hallucinations.

terms requiring overlap management. GPT-4 was effective when objects don't overlap. Claude-3 often hallucinated, with about 79% of errors linked to this problem. CodeLlama had about 92% of errors resulting from failure to consider overlaps. These results highlight the challenges of learning repetitions and the difficulty of incorporating this ability through in-context learning alone.

Our comprehensive analysis reveals that in-context learning is beneficial for LLMs in generating first-order code and handling simple loops, but underscore the need for more advanced techniques in generating code for complex and nested functions.

6 Conclusion and Limitations

This paper explores the program synthesis capabilities of code-generating LLMs, whose understanding can significantly enhance conversational programming for cobots. To this end, we developed a rapid prototyping simulator environment that evaluates these LLMs using specific input instructions, simulate their execution, and analyzes the output. This approach establishes a strong baseline for future studies. Our findings reveal promising results for template-based instructions that generate first-order code and functions composed of sequences of first-order code. However, performance significantly declines with optimal higher-order functions and complex patterns that require reasoning about object overlap in repetitions. Such a finding indicates a pressing need for further investigation. Similarly the LLMs performance for human-written and model-generated instructions is considerably lower compared to template-based instructions, calls for a detailed exploration. In future work, we plan to expand the setup to handle more complex sequence arrangements added with dialogue management. Additionally, we will explore fine-tuning the code generation LLMs to see if it improves their overall performance.

Though LLMs have shown promising performance for the 2D building task, there are limitations. The seeds for target board generation focus only on simple objects; we plan to include complex objects to better assess capabilities. The template-based instructions were designed to be clear and unambiguous, establishing a performance baseline. Future work will include environmental variability, collaborative scenarios, and dialogue management to enhance real-world simulation. Despite careful prompt curation, models sometimes produce hallucinations. Refining prompts based on feedback from execution errors can help. Instruction-tuned LLMs for code generation may struggle with multi-turn conversations, and applying this work to real-world assembly tasks will require adherence to safety principles.

Acknowledgments

The work reported here has been funded by the Bundesministerium für Bildung und Forschung (BMBF, German Federal Ministry of Research), project "COCOBOTS" (01IS21102A).

References

- [1] S. E. Zaatari, M. Marei, W. Li, and Z. Usman. Cobot programming for collaborative industrial tasks: An overview. *Robotics Auton. Syst.*, 116:162–180, 2019. doi:10.1016/J.ROBOT.2019.03.003. URL <https://doi.org/10.1016/j.robot.2019.03.003>.
- [2] G. Giannopoulou, E. Borrelli, and F. McMaster. "programming - it's not for normal people": A qualitative study on user-empowering interfaces for programming collaborative robots. In *30th IEEE International Conference on Robot & Human Interactive Communication, RO-MAN 2021, Vancouver, BC, Canada, August 8-12, 2021*, pages 37–44. IEEE, 2021. doi:10.1109/RO-MAN50785.2021.9515535. URL <https://doi.org/10.1109/RO-MAN50785.2021.9515535>.
- [3] J. V. Brummelen, K. Weng, P. Lin, and C. Yeo. Convo: What does conversational programming need? an exploration of machine learning interface design. *CoRR*, abs/2003.01318, 2020. URL <https://arxiv.org/abs/2003.01318>.
- [4] A. M. Bauer, D. Wollherr, and M. Buss. Human-robot collaboration: a survey. *Int. J. Humanoid Robotics*, 5(1):47–66, 2008. doi:10.1142/S0219843608001303. URL <https://doi.org/10.1142/S0219843608001303>.
- [5] D. Mukherjee, K. Gupta, L. H. Chang, and H. Najjaran. A survey of robot learning strategies for human-robot collaboration in industrial settings. *Robotics Comput. Integr. Manuf.*, 73:102231, 2022. doi:10.1016/J.RCIM.2021.102231. URL <https://doi.org/10.1016/j.rcim.2021.102231>.
- [6] L. D. Rozo, S. Calinon, D. G. Caldwell, P. Jiménez, and C. Torras. Learning physical collaborative robot behaviors from human demonstrations. *IEEE Trans. Robotics*, 32(3):513–527, 2016. doi:10.1109/TRO.2016.2540623. URL <https://doi.org/10.1109/TRO.2016.2540623>.
- [7] J. Berg and S. Lu. Review of interfaces for industrial human-robot interaction. *Current Robotics Reports*, 1:27–34, 2020.
- [8] M. Chen, J. Tworek, H. Jun, Q. Yuan, H. P. de Oliveira Pinto, J. Kaplan, and et al. Evaluating large language models trained on code. *CoRR*, abs/2107.03374, 2021. URL <https://arxiv.org/abs/2107.03374>.
- [9] E. Nijkamp, B. Pang, H. Hayashi, L. Tu, H. Wang, Y. Zhou, S. Savarese, and C. Xiong. Codegen: An open large language model for code with multi-turn program synthesis. In *The Eleventh International Conference on Learning Representations, ICLR 2023, Kigali, Rwanda, May 1-5, 2023*. OpenReview.net, 2023. URL https://openreview.net/pdf?id=iaYcJKpY2B_.
- [10] Y. Wei, Z. Wang, J. Liu, Y. Ding, and L. Zhang. Magicoder: Source code is all you need. *arXiv preprint arXiv:2312.02120*, 2023.
- [11] D. Guo, Q. Zhu, D. Yang, Z. Xie, K. Dong, W. Zhang, G. Chen, X. Bi, Y. Wu, Y. K. Li, F. Luo, Y. Xiong, and W. Liang. Deepseek-coder: When the large language model meets programming - the rise of code intelligence. *CoRR*, abs/2401.14196, 2024. doi:10.48550/ARXIV.2401.14196. URL <https://doi.org/10.48550/arXiv.2401.14196>.
- [12] B. Ichter, A. Brohan, Y. Chebotar, C. Finn, K. Hausman, A. Herzog, and et al. Do as I can, not as I say: Grounding language in robotic affordances. In *Conference on Robot Learning*,

- CoRL 2022, 14-18 December 2022, Auckland, New Zealand*, volume 205 of *Proceedings of Machine Learning Research*, pages 287–318. PMLR, 2022. URL <https://proceedings.mlr.press/v205/ichter23a.html>.
- [13] W. Huang, P. Abbeel, D. Pathak, and I. Mordatch. Language models as zero-shot planners: Extracting actionable knowledge for embodied agents. In *International Conference on Machine Learning, ICML 2022, 17-23 July 2022, Baltimore, Maryland, USA*, volume 162 of *Proceedings of Machine Learning Research*, pages 9118–9147. PMLR, 2022. URL <https://proceedings.mlr.press/v162/huang22a.html>.
- [14] A. Zeng, M. Attarian, B. Ichter, K. M. Choromanski, A. Wong, S. Welker, F. Tombari, A. Purohit, M. S. Ryoo, V. Sindhwani, J. Lee, V. Vanhoucke, and P. Florence. Socratic models: Composing zero-shot multimodal reasoning with language. In *The Eleventh International Conference on Learning Representations, ICLR 2023, Kigali, Rwanda, May 1-5, 2023*. OpenReview.net, 2023. URL <https://openreview.net/pdf?id=G2Q2Mh3avow>.
- [15] W. Huang, F. Xia, D. Shah, D. Driess, A. Zeng, Y. Lu, P. Florence, I. Mordatch, S. Levine, K. Hausman, and B. Ichter. Grounded decoding: Guiding text generation with grounded models for robot control. *CoRR*, abs/2303.00855, 2023. doi:10.48550/ARXIV.2303.00855. URL <https://doi.org/10.48550/arXiv.2303.00855>.
- [16] Y. Jiang, A. Gupta, Z. Zhang, G. Wang, Y. Dou, Y. Chen, L. Fei-Fei, A. Anandkumar, Y. Zhu, and L. Fan. VIMA: general robot manipulation with multimodal prompts. *CoRR*, abs/2210.03094, 2022. doi:10.48550/ARXIV.2210.03094. URL <https://doi.org/10.48550/arXiv.2210.03094>.
- [17] S. Huang, L. Dong, W. Wang, Y. Hao, S. Singhal, S. Ma, T. Lv, L. Cui, O. K. Mohammed, B. Patra, Q. Liu, K. Aggarwal, Z. Chi, N. J. B. Bjorck, V. Chaudhary, S. Som, X. Song, and F. Wei. Language is not all you need: Aligning perception with language models. In A. Oh, T. Naumann, A. Globerson, K. Saenko, M. Hardt, and S. Levine, editors, *Advances in Neural Information Processing Systems 36: Annual Conference on Neural Information Processing Systems 2023, NeurIPS 2023, New Orleans, LA, USA, December 10 - 16, 2023*, 2023. URL http://papers.nips.cc/paper_files/paper/2023/hash/e425b75bac5742a008d643826428787c-Abstract-Conference.html.
- [18] D. Driess, F. Xia, M. S. M. Sajjadi, C. Lynch, A. Chowdhery, B. Ichter, A. Wahid, J. Tompson, Q. Vuong, T. Yu, W. Huang, Y. Chebotar, P. Sermanet, D. Duckworth, S. Levine, V. Vanhoucke, K. Hausman, M. Toussaint, K. Greff, A. Zeng, I. Mordatch, and P. Florence. Palm-e: An embodied multimodal language model. In A. Krause, E. Brunskill, K. Cho, B. Engelhardt, S. Sabato, and J. Scarlett, editors, *International Conference on Machine Learning, ICML 2023, 23-29 July 2023, Honolulu, Hawaii, USA*, volume 202 of *Proceedings of Machine Learning Research*, pages 8469–8488. PMLR, 2023. URL <https://proceedings.mlr.press/v202/driess23a.html>.
- [19] J. Liang, W. Huang, F. Xia, P. Xu, K. Hausman, B. Ichter, P. Florence, and A. Zeng. Code as policies: Language model programs for embodied control. In *IEEE International Conference on Robotics and Automation, ICRA 2023, London, UK, May 29 - June 2, 2023*, pages 9493–9500. IEEE, 2023. doi:10.1109/ICRA48891.2023.10160591. URL <https://doi.org/10.1109/ICRA48891.2023.10160591>.
- [20] I. Singh, V. Blukis, A. Mousavian, A. Goyal, D. Xu, J. Tremblay, D. Fox, J. Thomason, and A. Garg. Progprompt: program generation for situated robot task planning using large language models. *Auton. Robots*, 47(8):999–1012, 2023. doi:10.1007/S10514-023-10135-3. URL <https://doi.org/10.1007/s10514-023-10135-3>.
- [21] B. Li, P. Wu, P. Abbeel, and J. Malik. Interactive task planning with language models. *CoRR*, abs/2310.10645, 2023. doi:10.48550/ARXIV.2310.10645. URL <https://doi.org/10.48550/arXiv.2310.10645>.

- [22] J. X. Liu, Z. Yang, I. Idrees, S. Liang, B. Schornstein, S. Tellex, and A. Shah. Grounding complex natural language commands for temporal tasks in unseen environments. In *Conference on Robot Learning, CoRL 2023, 6-9 November 2023, Atlanta, GA, USA*, volume 229 of *Proceedings of Machine Learning Research*, pages 1084–1110. PMLR, 2023. URL <https://proceedings.mlr.press/v229/liu23d.html>.
- [23] M. Paetzel-Prüsmann, J. Hunter, K. Chalamalasetti, K. Thompson, A. Nicolaou, O. Güngör, D. Schlangen, and N. Asher. Conversational programming for collaborative robots. In *ICRA Workshop on Collaborative Robots and Work of the Future (ICRA 2022 CoR-WotF)*, pages 1–5, 2022.
- [24] V. Zhong, C. Xiong, and R. Socher. Seq2sql: Generating structured queries from natural language using reinforcement learning. *CoRR*, abs/1709.00103, 2017. URL <http://arxiv.org/abs/1709.00103>.
- [25] S. Lu, D. Guo, S. Ren, J. Huang, A. Svyatkovskiy, A. Blanco, and et al. Codexglue: A machine learning benchmark dataset for code understanding and generation. In *Proceedings of the Neural Information Processing Systems Track on Datasets and Benchmarks 1, NeurIPS Datasets and Benchmarks 2021, December 2021, virtual*, 2021. URL <https://datasets-benchmarks-proceedings.neurips.cc/paper/2021/hash/c16a5320fa475530d9583c34fd356ef5-Abstract-round1.html>.
- [26] J. Austin, A. Odena, M. I. Nye, M. Bosma, H. Michalewski, D. Dohan, E. Jiang, C. J. Cai, M. Terry, Q. V. Le, and C. Sutton. Program synthesis with large language models. *CoRR*, abs/2108.07732, 2021. URL <https://arxiv.org/abs/2108.07732>.
- [27] D. Kocetkov, R. Li, L. B. Allal, J. Li, C. Mou, C. M. Ferrandis, Y. Jernite, M. Mitchell, S. Hughes, T. Wolf, D. Bahdanau, L. von Werra, and H. de Vries. The stack: 3 TB of permissively licensed source code. *CoRR*, abs/2211.15533, 2022. doi:10.48550/ARXIV.2211.15533. URL <https://doi.org/10.48550/arXiv.2211.15533>.
- [28] H. Touvron, L. Martin, K. Stone, P. Albert, A. Almahairi, Y. Babaei, N. Bashlykov, and et al. Llama 2: Open foundation and fine-tuned chat models. *CoRR*, abs/2307.09288, 2023. doi:10.48550/ARXIV.2307.09288. URL <https://doi.org/10.48550/arXiv.2307.09288>.
- [29] B. Rozière, J. Gehring, F. Gloeckle, S. Sootla, I. Gat, X. E. Tan, Y. Adi, J. Liu, T. Remez, J. Rapin, A. Kozhevnikov, I. Evtimov, J. Bitton, M. Bhatt, C. Canton-Ferrer, A. Grattafiori, W. Xiong, A. Défossez, J. Copet, F. Azhar, H. Touvron, L. Martin, N. Usunier, T. Scialom, and G. Synnaeve. Code llama: Open foundation models for code. *CoRR*, abs/2308.12950, 2023. doi:10.48550/ARXIV.2308.12950. URL <https://doi.org/10.48550/arXiv.2308.12950>.
- [30] X. Hou, Y. Zhao, Y. Liu, Z. Yang, K. Wang, L. Li, X. Luo, D. Lo, J. C. Grundy, and H. Wang. Large language models for software engineering: A systematic literature review. *CoRR*, abs/2308.10620, 2023. doi:10.48550/ARXIV.2308.10620. URL <https://doi.org/10.48550/arXiv.2308.10620>.
- [31] M. Wong, S. Guo, C. N. Hang, S. Ho, and C. Tan. Natural language generation and understanding of big code for ai-assisted programming: A review. *Entropy*, 25(6):888, 2023. doi:10.3390/E25060888. URL <https://doi.org/10.3390/e25060888>.
- [32] D. Zan, B. Chen, F. Zhang, D. Lu, B. Wu, B. Guan, Y. Wang, and J. Lou. Large language models meet nl2code: A survey. In A. Rogers, J. L. Boyd-Graber, and N. Okazaki, editors, *Proceedings of the 61st Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers), ACL 2023, Toronto, Canada, July 9-14, 2023*, pages 7443–7464. Association for Computational Linguistics, 2023. doi:10.18653/V1/2023.ACL-LONG.411. URL <https://doi.org/10.18653/v1/2023.acl-long.411>.

- [33] P. Jayannavar, A. Narayan-Chen, and J. Hockenmaier. Learning to execute instructions in a minecraft dialogue. In *Proceedings of the 58th Annual Meeting of the Association for Computational Linguistics, ACL 2020, Online, July 5-10, 2020*, pages 2589–2602. Association for Computational Linguistics, 2020. doi:10.18653/V1/2020.ACL-MAIN.232. URL <https://doi.org/10.18653/v1/2020.acl-main.232>.
- [34] C. Bara, S. CH-Wang, and J. Chai. Mindcraft: Theory of mind modeling for situated dialogue in collaborative tasks. In *Proceedings of the 2021 Conference on Empirical Methods in Natural Language Processing, EMNLP 2021, Virtual Event / Punta Cana, Dominican Republic, 7-11 November, 2021*, pages 1112–1125. Association for Computational Linguistics, 2021. doi:10.18653/V1/2021.EMNLP-MAIN.85. URL <https://doi.org/10.18653/v1/2021.emnlp-main.85>.
- [35] A. Skrynnik, Z. Volovikova, M. Côté, A. Voronov, A. Zholus, N. Arabzadeh, S. Mohanty, M. Teruel, A. Awadallah, A. Panov, M. Burtsev, and J. Kiseleva. Learning to solve voxel building embodied tasks from pixels and natural language instructions. *CoRR*, abs/2211.00688, 2022. doi:10.48550/ARXIV.2211.00688. URL <https://doi.org/10.48550/arXiv.2211.00688>.
- [36] C. Madge and M. Poesio. Large language models as minecraft agents. *CoRR*, abs/2402.08392, 2024. doi:10.48550/ARXIV.2402.08392. URL <https://doi.org/10.48550/arXiv.2402.08392>.
- [37] S. Mirchandani, F. Xia, P. Florence, B. Ichter, D. Driess, M. G. Arenas, K. Rao, D. Sadigh, and A. Zeng. Large language models as general pattern machines. In J. Tan, M. Toussaint, and K. Darvish, editors, *Conference on Robot Learning, CoRL 2023, 6-9 November 2023, Atlanta, GA, USA*, volume 229 of *Proceedings of Machine Learning Research*, pages 2498–2518. PMLR, 2023. URL <https://proceedings.mlr.press/v229/mirchandani23a.html>.
- [38] Y. Xu, W. Li, P. Vaezipoor, S. Sanner, and E. B. Khalil. Llms and the abstraction and reasoning corpus: Successes, failures, and the importance of object-based representations. *CoRR*, abs/2305.18354, 2023. doi:10.48550/ARXIV.2305.18354. URL <https://doi.org/10.48550/arXiv.2305.18354>.
- [39] M. Shridhar, J. Thomason, D. Gordon, Y. Bisk, W. Han, R. Mottaghi, L. Zettlemoyer, and D. Fox. ALFRED: A benchmark for interpreting grounded instructions for everyday tasks. In *2020 IEEE/CVF Conference on Computer Vision and Pattern Recognition, CVPR 2020, Seattle, WA, USA, June 13-19, 2020*, pages 10737–10746. Computer Vision Foundation / IEEE, 2020. doi:10.1109/CVPR42600.2020.01075. URL https://openaccess.thecvf.com/content_CVPR_2020/html/Shridhar_ALFRED_A_Benchmark_for_Interpreting_Grounded_Instructions_for_Everyday_Tasks_CVPR_2020_paper.html.
- [40] A. Padmakumar, J. Thomason, A. Shrivastava, P. Lange, A. Narayan-Chen, S. Gella, R. Pirmuthu, G. Tür, and D. Hakkani-Tür. Teach: Task-driven embodied agents that chat. In *Thirty-Sixth AAAI Conference on Artificial Intelligence, AAAI 2022, Thirty-Fourth Conference on Innovative Applications of Artificial Intelligence, IAAI 2022, The Twelveth Symposium on Educational Advances in Artificial Intelligence, EAAI 2022 Virtual Event, February 22 - March 1, 2022*, pages 2017–2025. AAAI Press, 2022. doi:10.1609/AAAI.V36I2.20097. URL <https://doi.org/10.1609/aaai.v36i2.20097>.
- [41] S. Shrestha, Y. Zha, S. Banagiri, G. Gao, Y. Aloimonos, and C. Fermüller. Natsgd: A dataset with speech, gestures, and demonstrations for robot learning in natural human-robot interaction. *CoRR*, abs/2403.02274, 2024. doi:10.48550/ARXIV.2403.02274. URL <https://doi.org/10.48550/arXiv.2403.02274>.

- [42] R. Lachmy, V. Pyatkin, A. Manevich, and R. Tsarfaty. Draw me a flower: Processing and grounding abstraction in natural language. *Trans. Assoc. Comput. Linguistics*, 10:1341–1356, 2022. URL <https://transacl.org/ojs/index.php/tacl/article/view/3961>.
- [43] A. Rastogi, X. Zang, S. Sunkara, R. Gupta, and P. Khaitan. Towards scalable multi-domain conversational agents: The schema-guided dialogue dataset. In *The Thirty-Fourth AAAI Conference on Artificial Intelligence, AAAI 2020, The Thirty-Second Innovative Applications of Artificial Intelligence Conference, IAAI 2020, The Tenth AAAI Symposium on Educational Advances in Artificial Intelligence, EAAI 2020, New York, NY, USA, February 7-12, 2020*, pages 8689–8696. AAAI Press, 2020. doi:10.1609/AAAI.V34I05.6394. URL <https://doi.org/10.1609/aaai.v34i05.6394>.
- [44] T. Aksu, Z. Liu, M. Kan, and N. F. Chen. Velocidapter: Task-oriented dialogue comprehension modeling pairing synthetic text generation with domain adaptation. In *Proceedings of the 22nd Annual Meeting of the Special Interest Group on Discourse and Dialogue, SIGdial 2021, Singapore and Online, July 29-31, 2021*, pages 133–143. Association for Computational Linguistics, 2021. URL <https://aclanthology.org/2021.sigdial-1.14>.
- [45] J. Götze, M. Paetzel-Prüsmann, W. Liermann, T. Diekmann, and D. Schlangen. The slurk interaction server framework: Better data for better dialog models. In *Proceedings of the Thirteenth Language Resources and Evaluation Conference, LREC 2022, Marseille, France, 20-25 June 2022*, pages 4069–4078. European Language Resources Association, 2022. URL <https://aclanthology.org/2022.lrec-1.433>.
- [46] D. Hupkes, V. Dankers, M. Mul, and E. Bruni. Compositionality decomposed: How do neural networks generalise? *J. Artif. Intell. Res.*, 67:757–795, 2020. doi:10.1613/JAIR.1.11674. URL <https://doi.org/10.1613/jair.1.11674>.
- [47] T. B. Brown, B. Mann, N. Ryder, M. Subbiah, J. Kaplan, and et al. Language models are few-shot learners. In *Advances in Neural Information Processing Systems 33: Annual Conference on Neural Information Processing Systems 2020, NeurIPS 2020, December 6-12, 2020, virtual*, 2020. URL <https://proceedings.neurips.cc/paper/2020/hash/1457c0d6bfc4967418bfb8ac142f64a-Abstract.html>.
- [48] S. Ren, D. Guo, S. Lu, L. Zhou, S. Liu, D. Tang, N. Sundaresan, M. Zhou, A. Blanco, and S. Ma. Codebleu: a method for automatic evaluation of code synthesis. *CoRR*, abs/2009.10297, 2020. URL <https://arxiv.org/abs/2009.10297>.
- [49] K. Chalamalasetti, J. Götze, S. Hakimov, B. Madureira, P. Sadler, and D. Schlangen. clem-bench: Using game play to evaluate chat-optimized language models as conversational agents. In *Proceedings of the 2023 Conference on Empirical Methods in Natural Language Processing, EMNLP 2023, Singapore, December 6-10, 2023*, pages 11174–11219. Association for Computational Linguistics, 2023. URL <https://aclanthology.org/2023.emnlp-main.689>.

7 Appendix

7.1 Simulated Environment

As mentioned in Section 3, our proposed simulated environment uses a 2D grid of size 8x8, where each cell of the grid can hold a component. Any specific arrangement of components on the grid should adhere to the rules such as a) components of the same type or color cannot be stacked on each other and b) components can be placed atop one another as long as the depths (the number of cells occupied by the components) match, with the sole exception that no components can be placed on a *screw*. These rules allow us to construct component arrangements that resemble real-world style assemblies. Each such arrangement, where the components are connected, in such a way that electricity flows through them is referred to as an object. These objects are categorized as simple and complex based on their configurations. *Simple objects* are basic configurations of up to five components and no more than three stacks.

7.2 Template-Based Instructions

Instruction Templates: We employed a standard template grammar for generating instructions based on templates. Simple boards are organized into two categories: a) multi-turn and b) single-turn. As illustrated in Figure 16a, multi-turn instructions consist of sequential steps described over 'n' turns. In contrast, single-turn instructions, depicted in Figure 16b, compile all steps necessary to construct a target structure within a single turn. For regular boards, the template grammar applied to simple and complex objects is presented in Figure 17 and Figure 18, respectively. Both templates articulate the specific arrangements utilized. Moreover, the complex object templates include placeholders for each object's space, which is critical for the instruction follower when generating the construction code.

Code Templates: Figures 19-25 showcase the Jinja2 templates used for generating the code for higher-order functions for the manually curated code seeds (see Section 3.1). The placeholders for shapes, colors, and locations are replaced with appropriate values during the board generation process.

7.3 Human-Written Instructions

As mentioned in Section 3.1, we curated human instructions for the test set (see Table 2) to measure how well the LLMs grasp human abstractions and communication styles for the code generation. We developed a bot using *stark* [45] interface, which features a target board on the left and a text input area on the right for writing the instructions to reconstruct it. A student assistant from our department was recruited for this task, which took approximately 20 hours to complete.

7.4 Model-Generated Instructions

As described in Section 3.2, LLMs are used to describe the target structure. These descriptions are later used as part of the component assembly task for generating the associated code. Using the *clembench* framework [49], we setup the process of describing the target structure as a *clemgame*. In this game, the game master probes the player (a code generation LLM) with the grid details represented in a textual format as shown in the Figure 8 and Figure 9 using zero-shot prompting techniques [47, 12, 19] to generate the textual instructions.

7.5 Prompt Structure

The principal objective of the proposed work is to investigate the capability of the LLM in capturing the procedural steps involved in arranging components in a specific sequence, generating an abstract code representation that is both execution-ready and applicable within the simulation environment. To achieve execution readiness, the LLM must extract requisite details from instructions and transform them into an intended structural format expressed through context information and in-context

samples. Building on this, we construct a multi-part prompt (shown in Figure 7a) to probe the LLM. Each section of this prompt is designed to convey distinct pieces of information, clearly defining the task’s goals and expected response for the LLM.

System Information guides the LLM toward the overall desired outcome. The simulation setup uses distinct names for components based on their orientation properties, which is crucial for the LLM to identify and utilize the correct names accurately. In our simulation setup, columns increase along the x-axis, and rows along the y-axis. It is important to note that the numbering begins at the top, unlike the conventional approach, which starts at the bottom. This places the top-left corner as the first row and first column. Understanding this unique orientation is essential for the LLM to translate spatial information from instructions into code correctly. Therefore, these specifics are included as part of the environment information.

Following this, the context information specifies the functions available in the environment, eliminating the need for the LLM to generate new code for these functions. After the context information, task information provides the labels the LLM should use in its responses. This assists the parser in identifying and extracting the relevant responses. In-context samples, which come next, illustrate the types of instructions and associated actions the LLM encounters. In adherence to prompt engineering best practices, we have also included explicit instructions to prevent the generation of explanations or instructions other than the responses labeled as specified in the task information. All the experiments for property compositionality, function compositionality, and function repeatability follow the same prompt structure but differ in terms of the in-context samples they use. The structure of in-context samples used for each of these experiments is shown in Figure 6).

7.5.1 Ablation Study

We conducted an ablation study to investigate the impact of various parts of the prompt on overall task performance. We used the validation split of the dataset to perform the study. Table 4f provides a comprehensive overview of the ablation study results, presenting the performance metrics for different prompt configurations. The analysis focuses on determining optimal prompts for capturing procedural steps and generating execution-ready, simulation-applicable code representations. Each row in the table corresponds to a specific prompt structure element, and the columns include relevant performance metrics, such as the execution score for each code LLM, to quantify the impact of these elements on the overall effectiveness in code generation. The results demonstrate that the structure shown in Figure 7a is optimal for all the tasks and LLMs used in our experiments. Overall, the prompting structure lacking task information and in-context samples deteriorated the performance, while adding environment and context information improved the performance slightly. System details helped guiding the desired outcome from LLM. Therefore, we use a prompt structure comprising *system*, *environment*, *task*, *context*, *five in-context samples* and *other information* for the remaining experiments.

7.5.2 Selection of In-context Samples

The training split is used to select in-context samples for prompts. Our ablation study described above shows that using *five* in-context samples yields the best results across various LLMs. We dynamically prepare examples for each sample in the test split, ensuring they do not share the test instruction’s component combination to avoid overlap and bias. From this pool, five random samples are chosen for each test sample, maintaining their uniqueness and relevance. Figure 6 and Figure 15 displays examples illustrating *simple* boards and *regular* boards highlighting the tailored in-context samples for each scenario.

7.6 Detailed Error Analysis

The detailed error analysis focuses on issues that lead to reduced execution scores for the LLM-generated response, as showcased in Table 5. These errors are broadly classified into two categories: board placement errors and element mismatch errors. Board placement errors refer to instances

Prompt Header	System + Env + Task + Context + Other Information ...	System + Env + Task + Context + Other Information ...	System + Env + Task + Context + Other Information ...
In-Context Samples	<p>Instruction place a yellow bridge vertically in the 6th row, 4th column</p> <p>Output <code>put(board, shape='bridge-v', color='yellow', x=5, y=3)</code></p> <p>Instruction place a green nut in the 5th row, 2nd column</p> <p>Output <code>put(board, shape='nut', color='green', x=4, y=1)</code></p>	<p>Instruction These are the instructions to build bws. Place a green horizontal bridge in the 7th row, 5th column. Place a yellow washer in the 7th row, 5th column. Place a red screw in the 7th row, 6th column.</p> <p>Function <code>def bws (board, colors, x, y): put(board, "bridge-h", 'green', x=6, y=4) put(board, "washer", 'yellow', x=6, y=4) put(board, "screw", 'red', x=6, y=5)</code></p> <p>Usage <code>board = bws(board, ['green', 'yellow', 'red'], 6, 4)</code></p> <p>Instruction These are the instructions to build bns. Place a blue horizontal bridge in the 1st row, 1st column. Place a yellow nut in the 1st row, 2nd column. Place a green washer in the 1st row, 2nd column.</p> <p>Function <code>def bns (board, colors, x, y): put(board, "bridge-h", 'green', x=0, y=0) put(board, "nut", 'blue', x=0, y=1) put(board, "washer", 'yellow', x=0, y=1)</code></p> <p>Usage <code>board = bns(board, ['green', 'blue', 'yellow'], 5, 1)</code></p>	<p>Instruction These are the instructions to build bws. Place a green horizontal bridge in the 7th row, 5th column. Place a yellow washer in the 7th row, 6th column. Place a red screw in the 7th row, 6th column.</p> <p>Function <code>def bws (board, colors, x, y): shapes = ['bridge-h', 'washer', 'screw'] for shape, color, dx, dy in zip(shapes, colors, [0, 0, 0], [0, 1, 1]): put(board, shape, color, x + dx, y + dy)</code></p> <p>Usage <code>board = bws(board, ['green', 'yellow', 'red'], 6, 4)</code></p> <p>Instruction These are the instructions to build bns. Place a blue horizontal bridge in the 1st row, 1st column. Place a yellow nut in the 1st row, 2nd column. Place a green washer in the 1st row, 2nd column.</p> <p>Function <code>def bns (board, colors, x, y): shapes = ['bridge-h', 'nut', 'green'] for shape, color, dx, dy in zip(shapes, colors, [0, 0, 0], [0, 1, 1]): put(board, shape, color, x + dx, y + dy)</code></p> <p>Usage <code>board = bns(board, ['blue', 'yellow', 'green'], 0, 0)</code></p>
Test Instruction	<p>Instruction place a red washer in the 7th row, 7th column</p>	<p>Instruction These are the instructions to build nwb. Place a red nut in the 4th row, 4th column. Place a green washer in the 5th row, 4th column. Place a yellow vertical bridge in the 4th row, 4th column.</p>	<p>Instruction These are the instructions to build nwb. Place a red nut in the 4th row, 4th column. Place a green washer in the 5th row, 4th column. Place a yellow vertical bridge in the 4th row, 4th column.</p>
	(i) Property Compositionality	(ii) Function Compositionality (series of first-order code)	(iii) Function Compositionality (higher-order code)

Figure 6: Illustration of prompt for evaluating property and function compositionality. **Left:** first-order code examples with varied component properties (name, color, location) for property compositionality testing. **Middle:** the construction of higher-order functions from first-order code snippets for function compositionality evaluation. **Right:** the construction of higher-order functions with optimal code for function compositionality evaluation.

where the generated response cannot be executed, while element mismatch errors occur when the execution has discrepancies in location, color, or component.

The following list highlights the different scenarios where board placement errors can occur.

1. **Syntax Error:** Incorrect indentation or inclusion of non-Python code, such as hallucinations.
2. **Key Error:** Use of colors or components not supported in the environment.
3. **Name Error:** References to unsupported functions or random function calls.
4. **Value Error:** Placement of bridges at grid boundaries, specifically the 7th row for vertical bridges and the 7th column for horizontal bridges.
5. **Dimensions Mismatch Error:** Locations specified outside the grid boundaries, exceeding an 8x8 grid.
6. **Depth Mismatch Error:** Stacking of horizontal or vertical bridges without proper depth alignment.
7. **Bridge Placement Error:** Stacking of bridges at height-3 or higher.
8. **Same Shape Stacking Error, Same Shape At Alternate Levels Error:** Occur when identical shapes are stacked incorrectly.
9. **Not On Top Of Screw Error:** Placement of components on top of a *screw*.

It is important to note that the target structures used in the test split Table 2 do not contain any invalid scenarios that may cause these errors. Ideally, if the information is accurately extracted from the instructions, none of these errors would occur. The quantitative analysis described in Table 3 demonstrate that models struggle to interpret the instructions and extract relevant information, often making mistakes in associating the correct order, color, count, and location of components.

7.7 Evaluation Metrics

We have followed three different metrics to evaluate the LLM generated code. First, we use exact matching to compare the generated code with the ground truth code. Figure 12 showcases how this measurement uses a strict equality criterion for matching. The generated code (examples indicated by generated code-1, 2, 3, 4, 5 in Figure 12) is compared with the ground truth string and any mismatches (formatting, logical, or functional) in the code are treated as a failure. This score is useful to measure the perfect accuracy, but is inefficient in capturing semantic logic. Second, we use Code BLEU score to measure the syntactic accuracy of the generated code. The Code BLEU score [48] (CB) uses weighted combination of n-gram match, syntactic AST match and semantic data-flow match. We use the python package, `codebleu`² to compute the score. `calc_codebleu()` API takes the ground truth code and generated code as inputs and returns the overall Code BLEU score. Figure 13 shows how the CB score varies for differences in the generated code w.r.t the ground truth code. This score is useful to measure the syntactic correctness and is unreliable in capturing the overall accuracy. Lastly, we use execution success to measure how accurately the target board is reconstructed. Figure 14 demonstrates how the current grid state changes after the execution of the generated code. The example generated code-1, 2, and 3 share the same world state (row:5, column: 3 has two components: nut:red and washer:yellow) as the ground truth and are treated as successes. Although the example generated code-4 shares similar location and component types with the ground truth code, they differ in the colors associated with each component type (nut:yellow and washer:red) and is marked as a failure. Similarly, the example generated code-5 has a different component (screw:red) at the location(row:5, column: 3) and is marked as a failure. Thus, the execution success score captures semantic identicalness, functional correctness, making it reliable for target board reconstruction tasks.

²<https://github.com/k4black/codebleu>

TEMPLATE 7.7.1

System Info

You are a helpful assistant who is designed to interpret and translate natural language instructions into python executable code snippets.

Environment Info

The environment is an 8x8 grid allowing shape placement and stacking. A shape can be placed in any cell, while stacking involves adding multiple shapes to the same cell, increasing its depth. Shapes typically occupy a single cell, except for the "bridge," which spans two cells and requires two other shapes for stacking. Horizontal bridges span adjacent columns (left and right), and vertical ones span consecutive rows (top and bottom). Stacking is only possible if the shapes have matching depths.

In the grid, columns align with the x-axis and rows with the y-axis. Python indexing is used to identify each cell. The cell in the top-left corner is in the first row and first column, corresponding to x and y values of 0, 0. Similarly, the top-right corner cell is in the first row and eighth column, with x and y values of 0, 7.

- Use the shape name 'bridge-h' if a bridge is placed horizontally - Use the shape name 'bridge-v' if a bridge is placed vertically

Context Info

The following functions are already defined; therefore, do not generate additional code for it

- Use 'put(board: np.ndarray, shape: string, color: string, x: int, y: int)' to place a shape on the board

TEMPLATE 7.7.2

TASK INFO

The TASK INFO varies for each of the sub-tasks (property compositionality and function compositionality).

For each instruction labeled \$INSTRUCTION_LABEL please respond with code under the label \$OUTPUT_LABEL followed by a newline.

INCONTEXT_SAMPLES

The INCONTEXT_SAMPLES varies for each of the sub-tasks (property compositionality and function compositionality). Refer to Figure 6 and Figure 15

OTHER DETAILS

Do not generate any other text/explanations.

Ensure the response can be executed by Python 'exec()', e.g.: no trailing commas, no periods, etc. Lets begin

(a) base structure of prompt

(b) task and in-context samples

Figure 7: Prompt templates used for code generation for the tasks of Property Compositionality and Function Compositionality. The system information specifies system level behavior, the environment information indicates the environment details of the simulation framework, the context information describes the available functions that can be reused, task information indicates the specific response format to follow based on task type (differs for property compositionality and function compositionality)

Model	Number of In-Context Samples										
	0	1	2	3	4	5	6	7	8	9	10
CodeLlama-7B	0.53	0.90	0.92	0.95	0.95	0.95	0.95	0.97	0.98	0.99	0.98
CodeLlama-13B	0.92	0.89	0.94	0.95	0.95	0.97	0.97	0.97	0.96	0.98	0.96
CodeLlama-34B	1.00	0.95	0.98	0.99	0.98	1.00	1.00	0.99	1.00	1.00	1.00
Mistral-7B-v0.1	0.05	0.11	0.10	0.09	0.13	0.15	0.12	0.12	0.11	0.12	0.10
Mistral-7B-v0.2	0.02	0.07	0.10	0.11	0.09	0.10	0.12	0.11	0.12	0.13	0.12
StabilityAI-3B	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00

(a) Optimal number of in-context samples for the property compositionality task

Model	Number of In-Context Samples										
	0	1	2	3	4	5	6	7	8	9	10
CodeLlama-7B	0.01	0.80	0.89	0.92	0.95	0.94	0.89	0.93	0.89	0.92	0.85
CodeLlama-13B	0.52	0.73	0.77	0.80	0.81	0.82	0.94	0.93	0.94	0.93	0.91
CodeLlama-34B	0.75	0.67	0.89	0.92	0.95	0.99	1.00	1.00	0.99	1.00	1.00
Mistral-7B-v0.1	0.00	0.27	0.45	0.48	0.42	0.34	0.37	0.34	0.40	0.35	0.32
Mistral-7B-v0.2	0.02	0.24	0.25	0.23	0.25	0.20	0.32	0.35	0.35	0.29	0.30
StabilityAI-3B	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00

(b) Optimal number of in-context samples for the function compositionality task (higher-order code from sequences of first-order code)

Model	Number of In-Context Samples										
	0	1	2	3	4	5	6	7	8	9	10
CodeLlama-7B	0.01	0.16	0.22	0.23	0.29	0.25	0.27	0.29	0.829	0.31	0.34
CodeLlama-13B	0.52	0.18	0.28	0.37	0.42	0.39	0.46	0.43	0.45	0.46	0.48
CodeLlama-34B	0.75	0.13	0.30	0.35	0.43	0.41	0.45	0.44	0.42	0.46	0.49
Mistral-7B-v0.1	0.00	0.11	0.10	0.09	0.13	0.15	0.12	0.12	0.11	0.12	0.10
Mistral-7B-v0.2	0.02	0.07	0.10	0.11	0.09	0.10	0.12	0.11	0.12	0.13	0.12
StabilityAI-3B	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00

(c) Optimal number of in-context samples for the function compositionality task (higher-order optimal code)

Model	Number of In-Context Samples										
	0	1	2	3	4	5	6	7	8	9	10
CodeLlama-34B	0.01	0.89	0.63	0.68	0.70	0.75	0.82	0.82	0.82	0.83	0.80

(d) Optimal number of in-context samples for the function repeatability task for the simple objects

Model	Number of In-Context Samples										
	0	1	2	3	4	5	6	7	8	9	10
CodeLlama-34B	0.00	0.01	0.04	0.02	0.05	0.05	0.07	0.07	0.05	0.05	0.06

(e) Optimal number of in-context samples for the function repeatability task (complex objects)

Prompt Structure	CodeLlama-34b
S + E + C + T + O + I*	0.41
E + C + T + O + I*	0.35
S + C + T + O + I*	0.39
S + E + T + O + I*	0.37
S + E + C + O + I*	0.33
S + E + C + T + I*	0.30

(f) Impact of prompt structure elements on code generation LLM performance for the functional compositionality task. S: System Information, E: Environment Information, C: Context Information, T: Task Information, I: In-context Samples, I*: 5 In-context Samples, O: Other Information.

Table 4: Overview of ablation study experiments for the optimal number of in-context samples and the prompt structure

Board Type	Object Type	Task	Error Category	CodeLlama	GPT-4	Claude-3
Simple	Simple	Property Compositionality	Total	8	0	0
			Syntax Error	3	0	0
			Mismatch Location	3	0	0
			Mismatch Shape	2	0	0
		Function Compositionality (Using sequences of first-order code)	Total	5	0	9
			Syntax Error	0	0	9
			Mismatch Count	2	0	0
			Mismatch Location	2	0	0
		Function Compositionality (Using higher-order optimal code)	Mismatch Shape	1	0	0
			Total	62	57	17
			Depth Mismatch	16	34	11
			Dimensions Mismatch	3	5	4
			Bridge Placement	6	3	0
			Value Error	0	0	1
			Mismatch Location	6	5	0
			Mismatch Count	26	10	1
Mismatch Shape	1	0	0			
Same Shape Stacking	2	0	0			
Same Shape At Alternate Levels	2	0	0			

(a) Detailed error analysis for the template-based instructions

Board Type	Object Type	Task	Error Category	CodeLlama	GPT-4	Claude-3
Simple	Simple	Function Compositionality (Using higher-order optimal code)	Total	94	108	74
			Depth Mismatch	37	59	39
			Dimensions Mismatch	3	9	5
			Bridge Placement	15	4	5
			Value Error	2	5	4
			Syntax Error	0	0	4
			Mismatch Location	9	6	1
			Mismatch Color	1	0	0
			Mismatch Count	23	24	16
			Same Shape Stacking	2	0	0
			Same Shape At Alternate Levels	2	1	0

(b) Detailed error analysis for the human-authored instructions

Board Type	Object Type	Task	Error Category	CodeLlama	GPT-4	Claude-3
Simple	Simple	Function Compositionality (Using higher-order optimal code)	Total	125	101	100
			Depth Mismatch	43	88	82
			Dimensions Mismatch	5	1	0
			Bridge Placement	14	0	1
			Value Error	11	3	4
			Name Error	1	0	0
			Syntax Error	8	0	7
			Not On Top Of Screw	11	0	0
			Mismatch Location	13	0	0
			Mismatch Count	8	9	4
			Mismatch Shape	3	0	0
			Same Shape Stacking	4	0	2
Same Shape At Alternate Levels	4	0	0			

(c) Detailed error analysis for the model-generated instructions

Table 5: Overview of detailed error analysis for *simple boards* across all the tasks

Board Type	Object Type	Task	Error Category	CodeLlama-34b	GPT-4	Claude-3
Regular	Simple	Function Repeatability	Total	33	0	18
			Depth Mismatch	4	0	1
			Bridge Placement	5	0	0
			Mismatch Location	18	0	0
			Syntax Error	0	0	16
			Name Error	2	0	0
			Same Shape At Alternate Levels	3	0	1
			Same Shape Stacking	1	0	0
			Total	119	91	117
			Complex	Depth Mismatch	4	1
	Dimensions Mismatch			3	0	7
	Bridge Placement			7	4	2
	Value Error			11	10	5
	Syntax Error			2	0	92
	Name Error			2	0	0
	Mismatch Location			72	43	6
	Mismatch Count			9	25	2
	Mismatch Color			0	1	0
	Mismatch Shape			1	5	1
	Not On Top Of Screw		2	1	0	
Same Shape At Alternate Levels	6	1	0			

(a) Detailed error analysis for the template-based instructions

Board Type	Object Type	Task	Error Category	CodeLlama-34b	GPT-4	Claude-3
Regular	Simple	Function Repeatability	Total	121	90	98
			Depth Mismatch	5	0	2
			Dimensions Mismatch	7	5	0
			Bridge Placement	6	2	2
			Value Error	8	3	2
			Syntax Error	6	0	72
			Name Error	2	0	1
			Key Error	12	0	1
			Type Error	4	0	0
			Mismatch Location	32	18	2
			Mismatch Count	14	6	2
			Mismatch Color	24	55	14
			Same Shape At Alternate Levels	1	1	0
			Total	121	94	125
	Complex		Depth Mismatch	1	0	2
			Dimensions Mismatch	13	6	3
			Bridge Placement	3	0	3
			Value Error	23	9	2
			Syntax Error	6	6	103
			Name Error	7	0	0
			Key Error	22	0	1
			Mismatch Location	19	17	6
			Mismatch Color	20	53	4
			Mismatch Count	7	3	0
			Same Shape Stacking	0	0	1

(b) Detailed error analysis for the human-authored instructions

Board Type	Object Type	Task	Error Category	CodeLlama-34b	GPT-4	Claude-3
Regular	Simple	Function Repeatability	Total	130	113	87
			Depth Mismatch	2	0	0
			Dimensions Mismatch	41	13	1
			Bridge Placement	0	2	0
			Mismatch Location	66	85	33
			Mismatch Count	0	2	0
			Mismatch Shape	0	1	0
			Name Error	2	0	0
			Value Error	11	2	0
			Syntax Error	5	8	51
			Not On Top Of Screw	1	0	1
			Same Shape At Alternate Levels	2	0	1

(c) Detailed error analysis for the model-generated instructions

Table 6: Overview of detailed error analysis for *regular boards* across all the tasks

TEMPLATE 7.7.3

System Info

You are an expert annotator who generates sequential instructions for populating a grid with the given shapes.

Environment Info

The environment is an 8x8 grid allowing shape placement and stacking. A shape can be placed in any cell, while stacking involves adding multiple shapes to the same cell, increasing its depth. Shapes typically occupy a single cell, except for the "bridge," which spans two cells and requires two other shapes for stacking. Horizontal bridges span adjacent columns (left and right), and vertical ones span consecutive rows (top and bottom). Stacking is only possible if the shapes have matching depths.

In the grid, columns align with the x-axis and rows with the y-axis. The cell in the top-left corner is the first row and first column, corresponding to row and column values of 1, 1. Similarly, the top-right corner cell is the first row and eighth column, with row and column values of 1, 8.

Some of the cells in the grid are filled with shapes, and the current status of the grid is labeled under 'Current Grid Status'. If multiple shapes are placed in the same cell, they are mentioned in the order from bottom to top. All the shapes combined are referred to as an 'object', and the name of the object is labeled under 'Object Name'. Each filled cell in the grid contains a list of tuples, where each tuple indicates the name of the shape and its color. Empty cells are indicated by '█'.

The elaboration about the grid is labeled under 'Grid Explanation'.

Task Info

Your task is to respond with the sequential instructions under the label Instruction followed by a newline.

Generate the instructions to fill the grid with given shapes, listing all steps in a continuous format without numbering or bullet points. Also ensure to mention the object name in the instructions. Assume the grid starts empty and only describe actions for placing shapes. The order of colors, x, y matters, as these are assigned to the shapes in the same sequence.

Other Info

Do not generate any other text/explanations.
Lets begin

\$CURRENT_GRID_STATUS

\$GRID_EXPLANATION

Figure 8: prompt template for probing LLM to generate instruction for simple boards

TEMPLATE 7.7.4

System Info

You are an expert annotator who generates sequential instructions for populating a grid with the given shapes.

Environment Info

The environment is an 8x8 grid allowing shape placement and stacking. A shape can be placed in any cell, while stacking involves adding multiple shapes to the same cell, increasing its depth. Shapes typically occupy a single cell, except for the "bridge," which spans two cells and requires two other shapes for stacking. Horizontal bridges span adjacent columns (left and right), and vertical ones span consecutive rows (top and bottom). Stacking is only possible if the shapes have matching depths.

In the grid, columns align with the x-axis and rows with the y-axis. The cell in the top-left corner is the first row and first column, corresponding to row and column values of 1, 1. Similarly, the top-right corner cell is the first row and eighth column, with row and column values of 1, 8.

Some of the cells in the grid are filled with objects, and the current status of the grid is labeled under 'Current Grid Status'. Each filled cell in the grid contains a list of tuples, where each tuple indicates the name of the object and its colors. Empty cells are indicated by '█'.

The elaboration about the grid is labeled under 'Grid Explanation'.

Task Info

Your task is to respond with the sequential instructions under the label Instruction followed by a newline. Generate the instructions to fill the grid with the given object, in a continuous format without numbering or bullet points. Assume the grid starts empty and only describe actions for placing the object. The order of colors, x, y matters, as these are assigned to the object in the same sequence.

Other Info

Do not generate any other text/explanations.
Lets begin

\$CURRENT_GRID_STATUS

\$GRID_EXPLANATION

Figure 9: prompt template for probing LLM to generate instruction for regular boards

```

TEMPLATE 7.7.5

CURRENT_GRID_STATUS

CURRENT_GRID_STATUS varies for each of the target boards

[ ['█', '█', '█', '█', '█', '█', '█', '█'],
  ['█', '█', '█', '█', '█', '█', '█', '█'],
  ['█', '█', '█', '█', '█', '█', '█', '█'],
  ['█', '█', '█', '█', '█', '█', '█', '█'],
  ['█', '█', '█', '█', '█', '█', '█', '█'],
  ['█', '█', '█', '█', '█', '█', '█', '█'],
  ['█', '█', ['(washer', 'red'), ('screw', 'blue')], '█', '█', '█', '█',
  '█'],
  ['█', '█', '█', '█', '█', '█', '█', '█']

Object Name 'ws'.

Grid Explanation:

Row(7), Col(3) contains red washer, blue screw.

```

Figure 10: Overview of grid explanation in the prompt for describing a target structure

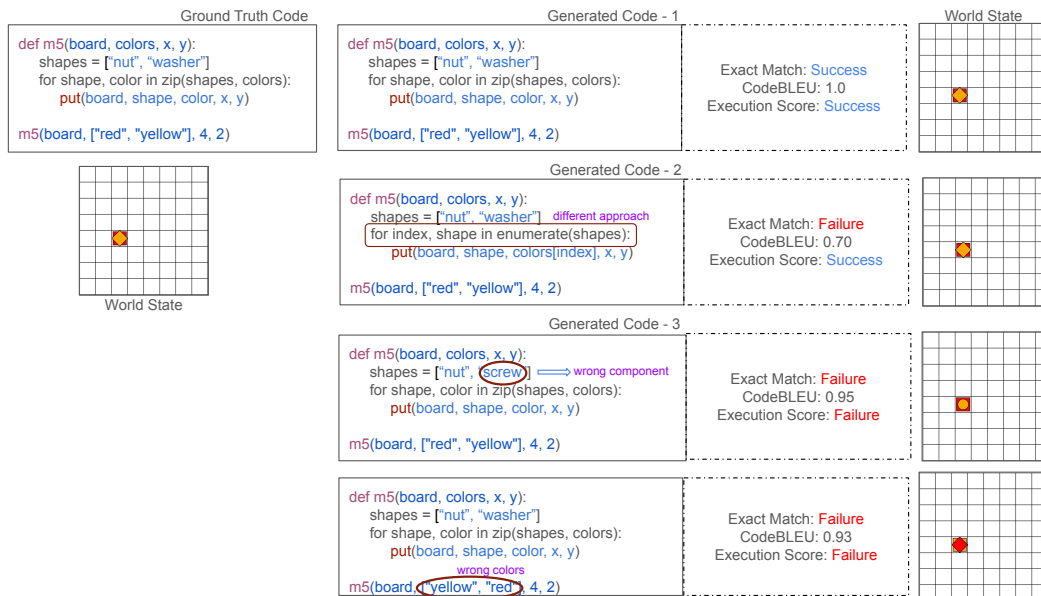


Figure 11: Overview of evaluation measurement for the generated code; Comparison of the generated code against ground truth using the metrics Exact Match (EM), Code BLEU score (CB), and Execution Success (ES); The ground truth code and its associated world state is shown on the left; Four examples of generated code are measured on the right; This highlights the strict criterion of EM, syntactic match criterion of CB and overall reconstruction accuracy of ES;

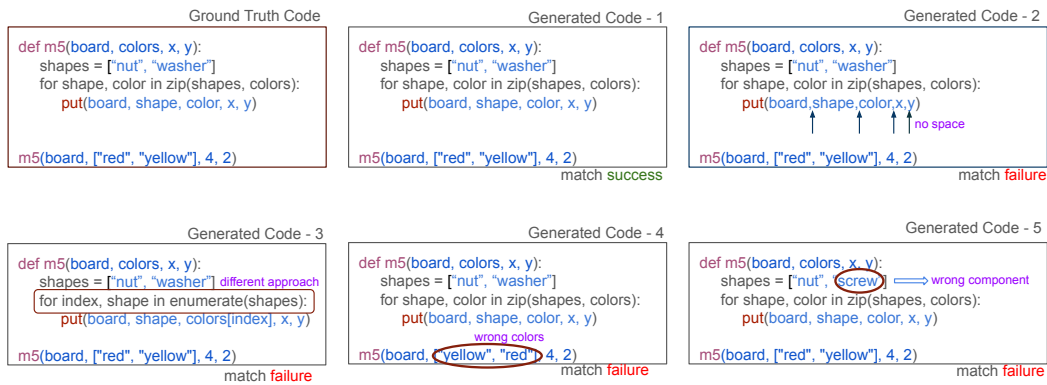


Figure 12: Overview of Exact Match measurement for the generated code; The ground truth code and its associated world state are shown on the left. Although exact match is able to detect mismatches in attributes (colors and components), it fails to detect the functional correctness of code with the same semantic logic.

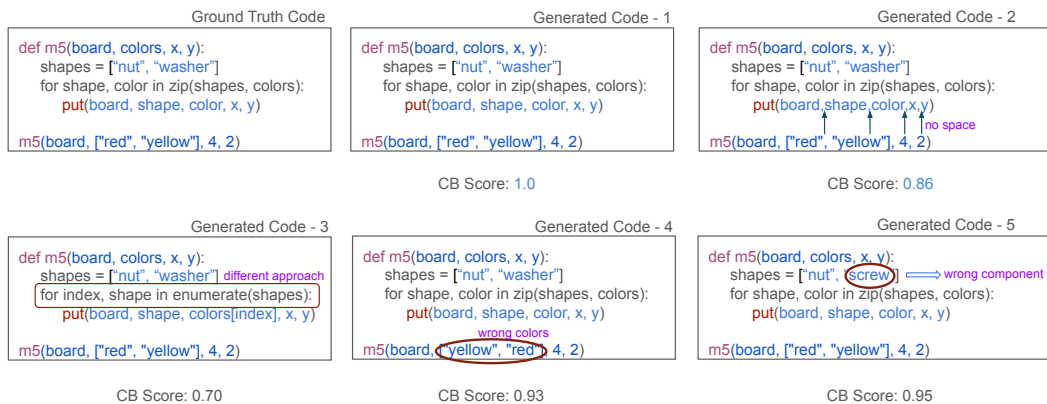


Figure 13: Overview of Code BLEU score measurement for the generated code; The ground truth code and its associated world state are shown on the left. The Code BLEU score is able to detect the syntactic correctness of the code (different styles), and also have high scores for the code where the similar style is followed but uses in-correct attributes. Since the values of attributes has no significance in Code BLEU score computation, it fails to detect mismatches in the attributes and cannot be relied upon completely for such reconstruction tasks.

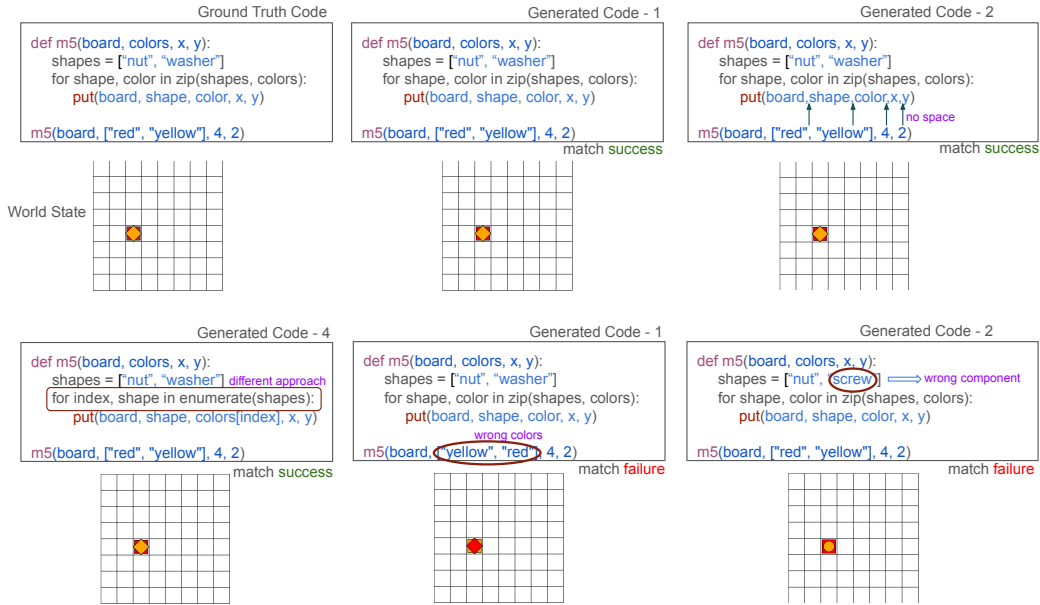


Figure 14: Overview of Execution Success measurement for the generated code; The ground truth code and its associated world state are shown on the left. Execution success compares the resultant 2D grid for its cell values (type, color and location of each component) against the ground truth grid state. As a result, the score reflects functional correctness along with mismatches in the attributes.

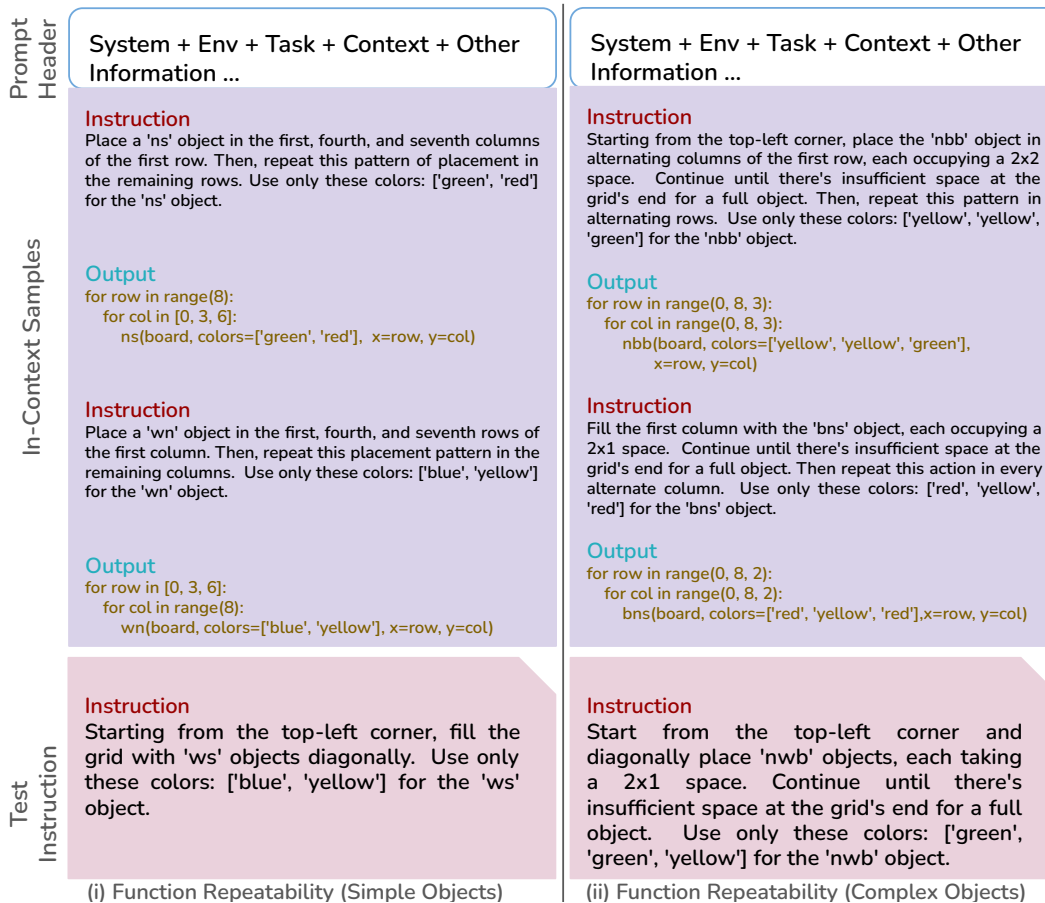


Figure 15: In-context samples for regular boards.

```

{%- if instruction_index == 0 -%}
These are the step-by-step instructions to build {{ data.
  ↳ combo_name }}.{{ " " }}
{%- endif -%}
{%- if orientation -%}
place a {{ color }} {{ shape }} {{ orientation }}ly in the {{ x }}
  ↳ row, {{ y }} column
{%- else -%}
place a {{ color }} {{ shape }} in the {{ x }} row, {{ y }} column
{%- endif -%}

```

(a) Multi-turn Instruction template

```

These are the instructions to build {{ data.combo_name }}.{{ " "
  ↳ }}
{%- for i in range(data.shapes|length) -%}
Place a {{ data.colors[i] }} {{ data.shapes[i] }} {%- if data.
  ↳ orientations[i] %} {{ data.orientations[i] }}ly {%- endif %}
  ↳ in the {{ x[i] }} row, {{ y[i] }} column.{{% if not loop.
  ↳ last %} {% endif %}
{%- endfor -%}

```

(b) Single-turn Instruction template

Figure 16: *Jinja2* templates for generating multi-turn and single-turn instructions for simple boards. **Top:** Defines the template grammar used for atomic placement of a component in each turn. **Bottom:** Defines the template grammar used for multiple component arrangement in a sequence.

Place a '{{combo_name}}' object in the first, fourth, and seventh
↪ columns of the first row. Then, repeat this pattern of
↪ placement in the remaining rows. Use only these colors: {{
↪ colors }} for the '{{combo_name}}' object.

(a) arrangement-1

Place a '{{combo_name}}' object in the first, fourth, and seventh
↪ rows of the first column. Then, repeat this placement
↪ pattern in the remaining columns. Use only these colors: {{
↪ colors }} for the '{{combo_name}}' object.

(b) arrangement-2

Starting from the top-left corner, fill the grid with '{{
↪ combo_name }}' objects diagonally. Use only these colors:
↪ {{ colors }} for the '{{combo_name}}' object.

(c) arrangement-3

Place a '{{ combo_name }}' object at all the corners of the grid.
↪ Use only these colors: {{ colors }} for the '{{combo_name
↪ }}' object.

(d) arrangement-4

Place a '{{combo_name}}' object in the first, and fifth columns of
↪ the first row. Then, repeat this placement pattern in the
↪ fifth row. Use only these colors: {{ colors }} for the '{{
↪ combo_name}}' object.

(e) arrangement-5

Figure 17: *Jinja2* templates for the construction of simple objects in regular boards. Each template grammar refers to a particular sequence arrangement.

```
Start from the top-left corner and diagonally place '{{ combo_name
↳ }}' objects, each taking a {{ occupied_rows }}x{{
↳ occupied_columns }} space. Continue until there's
↳ insufficient space at the grid's end for a full object. Use
↳ only these colors: {{ colors }} for the '{{combo_name}}'
↳ object.
```

(a) arrangement-1

```
Starting from the top-left corner, place the '{{ combo_name }}'
↳ object in alternating columns of the first row, each
↳ occupying a {{ occupied_rows }}x{{ occupied_columns }} space
↳ . Continue until there's insufficient space at the grid's
↳ end for a full object. Then, repeat this pattern in
↳ alternating rows. Use only these colors: {{ colors }} for
↳ the '{{combo_name}}' object.
```

(b) arrangement-2

```
Fill the first column with the '{{ combo_name }}' object, each
↳ occupying a {{ occupied_rows }}x{{ occupied_columns }} space
↳ . Continue until there's insufficient space at the grid's
↳ end for a full object. Then repeat this action in every
↳ alternate column. Use only these colors: {{ colors }} for
↳ the '{{combo_name}}' object.
```

(c) arrangement-3

```
Fill the fourth column with the '{{ combo_name }}' object, each
↳ occupying a {{ occupied_rows }}x{{ occupied_columns }} space
↳ . Continue until there's insufficient space at the grid's
↳ end for a full object. Use only these colors: {{ colors }}
↳ for the '{{combo_name}}' object.
```

(d) arrangement-4

Figure 18: *Jinja2* templates for the construction of complex objects in regular boards.


```

def {{ combo_name }}(board, colors, x, y):
    shapes = ['{{ shapes[0] }}', '{{ shapes[1] }}']
    for shape, color in zip(shapes, colors):
        put(board, shape, color, x, y)

```

(a) Stack two shapes

```

def {{ combo_name }}(board, colors, x, y):
    shapes = ['{{ shapes[0] }}', '{{ shapes[1] }}', '{{ shapes[2]
    ↪ }}']
    for shape, color in zip(shapes, colors):
        put(board, shape, color, x, y)

```

(b) Stack three shapes

```

def {{ combo_name }}(board, colors, x, y):
    shapes = ['{{ shapes[0] }}', '{{ shapes[1] }}', 'bridge-h']
    for shape, color, dx, dy in zip(shapes, colors, [0, 0, 0], [0,
    ↪ 1, 0]):
        put(board, shape, color, x + dx, y + dy)

```

(c) Place two shapes adjacently on a row and stack a horizontal bridge on top of them

```

def {{ combo_name }}(board, colors, x, y):
    shapes = ['{{ shapes[0] }}', '{{ shapes[1] }}', 'bridge-v']
    for shape, color, dx, dy in zip(shapes, colors, [0, 1, 0], [0,
    ↪ 0, 0]):
        put(board, shape, color, x + dx, y + dy)

```

(d) Place two shapes vertically in a column and stack a vertical bridge on top of them

Figure 19: *Jinja2* templates for the code generation for the component arrangements

```

def {{ combo_name }}(board, colors, x, y):
    shapes = ['bridge-h', '{{ shapes[1] }}', '{{ shapes[2] }}']
    for shape, color, dx, dy in zip(shapes, colors, [0, 0, 0], [0,
        ↪ 1, 1]):
        put(board, shape, color, x + dx, y + dy)

```

(a) Place a horizontal bridge and stack two shapes on its right side

```

def {{ combo_name }}(board, colors, x, y):
    shapes = ['bridge-v', '{{ shapes[1] }}', '{{ shapes[2] }}']
    for shape, color, dx, dy in zip(shapes, colors, [0, 1, 1], [0,
        ↪ 0, 0]):
        put(board, shape, color, x + dx, y + dy)

```

(b) Place a vertical bridge and stack two shapes on the bottom side of it

```

def {{ combo_name }}(board, colors, x, y):
    shapes = ['{{ shapes[0] }}', '{{ shapes[1] }}', '{{ shapes[2] }}',
        ↪ '{{ shapes[3] }}']
    for shape, color in zip(shapes, colors):
        put(board, shape, color, x, y)

```

(c) Stack four shapes

```

def {{ combo_name }}(board, colors, x, y):
    shapes = ['{{ shapes[0] }}', '{{ shapes[1] }}', 'bridge-h',
        ↪ '{{ shapes[3] }}']
    for shape, color, dx, dy in zip(shapes, colors, [0, 0, 0, 0],
        ↪ [0, 1, 0, 0]):
        put(board, shape, color, x + dx, y + dy)

```

(d) Place two shapes side by side on a row, then stack a horizontal bridge and an additional shape on top of these two shapes

Figure 20: *Jinja2* templates for the code generation for the component arrangements

```

def {{ combo_name }}(board, colors, x, y):
    shapes = ['{{ shapes[0] }}', '{{ shapes[1] }}', 'bridge-h',
    ↪ '{{ shapes[3] }}']
    for shape, color, dx, dy in zip(shapes, colors, [0, 0, 0, 0],
    ↪ [0, 1, 0, 1]):
        put(board, shape, color, x + dx, y + dy)

```

(a) Place two shapes adjacently on a row and stack a horizontal bridge along with a fourth shape on top of these two shapes

```

def {{ combo_name }}(board, colors, x, y):
    shapes = ['{{ shapes[0] }}', '{{ shapes[1] }}', 'bridge-v',
    ↪ '{{ shapes[3] }}']
    for shape, color, dx, dy in zip(shapes, colors, [0, 1, 0, 0],
    ↪ [0, 0, 0, 0]):
        put(board, shape, color, x + dx, y + dy)

```

(b) Place two shapes vertically in a column and stack a vertical bridge and an additional shape on top of these two shapes

```

def {{ combo_name }}(board, colors, x, y):
    shapes = ['{{ shapes[0] }}', '{{ shapes[1] }}', 'bridge-v',
    ↪ '{{ shapes[3] }}']
    for shape, color, dx, dy in zip(shapes, colors, [0, 1, 0, 1],
    ↪ [0, 0, 0, 0]):
        put(board, shape, color, x + dx, y + dy)

```

(c) Place two shapes vertically in a column and stack a vertical bridge along with a fourth shape on top of these two shapes

Figure 21: *Jinja2* templates for the code generation for the component arrangements

```

def {{ combo_name }}(board, colors, x, y):
    shapes = ['bridge-h', '{{ shapes[1] }}', 'bridge-v', '{{
        ↪ shapes[3] }}']
    for shape, color, dx, dy in zip(shapes, colors, [0, 1, 0, 0],
        ↪ [0, 1, 1, 1]):
        put(board, shape, color, x + dx, y + dy)

```

(a) Placement and stacking of a combination that includes a horizontal bridge, a vertical bridge, and two other shapes

```

def {{ combo_name }}(board, colors, x, y):
    shapes = ['bridge-v', '{{ shapes[1] }}', 'bridge-h', '{{
        ↪ shapes[3] }}']
    for shape, color, dx, dy in zip(shapes, colors, [0, 1, 1, 1],
        ↪ [0, 1, 0, 0]):
        put(board, shape, color, x + dx, y + dy)

```

(b) Placement and stacking of a combination that includes a vertical bridge, a horizontal bridge, and two other shapes

```

def {{ combo_name }}(board, colors, x, y):
    shapes = ['bridge-h', 'bridge-h', 'bridge-v', 'bridge-v']
    for shape, color, dx, dy in zip(shapes, colors, [0, 1, 0, 0],
        ↪ [0, 0, 0, 1]):
        put(board, shape, color, x + dx, y + dy)

```

(c) Placement and stacking of combinations of horizontal and vertical bridges

```

def {{ combo_name }}(board, colors, x, y):
    shapes = ['bridge-v', 'bridge-v', 'bridge-h', 'bridge-h']
    for shape, color, dx, dy in zip(shapes, colors, [0, 0, 0, 1],
        ↪ [0, 1, 0, 0]):
        put(board, shape, color, x + dx, y + dy)

```

(d) Placement and stacking of combinations of vertical and horizontal bridges

Figure 22: Jinja2 templates for the code generation for the component arrangements

```

def {{ combo_name }}(board, colors, x, y):
    shapes = ['bridge-h', '{{ shapes[1] }}', 'bridge-v', '{{
    ↳ shapes[3] }}', '{{ shapes[4] }}']
    for shape, color, dx, dy in zip(shapes, colors, [0, 1, 0, 0,
    ↳ 0], [0, 1, 1, 1, 1]):
        put(board, shape, color, x + dx, y + dy)

```

(a) Placement and Stacking of horizontal bridge, vertical bridge and a combination of two other shapes

```

def {{ combo_name }}(board, colors, x, y):
    shapes = ['{{ shapes[0] }}', '{{ shapes[1] }}', 'bridge-h',
    ↳ '{{ shapes[3] }}', '{{ shapes[4] }}']
    for shape, color, dx, dy in zip(shapes, colors, [0, 0, 0, 0,
    ↳ 0], [0, 1, 0, 0, 0]):
        put(board, shape, color, x + dx, y + dy)

```

(b) Place two shapes adjacently on a row and stack a horizontal bridge along with two other shapes on top of these two shapes

```

def {{ combo_name }}(board, colors, x, y):
    shapes = ['{{ shapes[0] }}', '{{ shapes[1] }}', 'bridge-v',
    ↳ '{{ shapes[3] }}', '{{ shapes[4] }}']
    for shape, color, dx, dy in zip(shapes, colors, [0, 1, 0, 0,
    ↳ 0], [0, 0, 0, 0, 0]):
        put(board, shape, color, x + dx, y + dy)

```

(c) Place two shapes vertically in a row and stack a vertical bridge along with two additional shapes on top of these two shapes

```

for row in range('{{ num_rows }}'):
    for col in [0, 3, 6]:
        {{ combo_name }}(board, colors={{ colors }},x=row, y=col)

```

(d) Repeat a simple object in the first, fourth, and seventh columns across all the rows in the grid

Figure 23: Jinja2 templates for the code generation for the component arrangements

```
for row in [0, 3, 6]:
    for col in range({{ num_cols }}):
        {{ combo_name }}(board, colors={{ colors }},x=row, y=col)
```

(a) Repeat a simple object in the first, fourth, and seventh rows across all columns of the grid

```
for row in range({{ num_rows }}):
    for col in range({{ num_cols }}):
        if row == col:
            {{ combo_name }}(board, colors={{ colors }},x=row, y=
                ↪ col)
```

(b) Fill the grid by placing a simple object along the diagonal

```
for row, col in [[0,0], [0,{{ num_cols-1 }}], [{{ num_rows-1 }},
    ↪ 0], [{{ num_rows-1 }}, {{ num_cols-1 }}]]:
    {{ combo_name }}(board, colors={{ colors }}, x=row, y=col)
```

(c) Place a simple object at all the corners of the grid

```
for row in [0, 4]:
    for col in [0, 4]:
        {{ combo_name }}(board, colors={{ colors }}, x=row, y=col)
```

(d) Fill the entire fourth column and the entire fourth row of the grid with a simple object

Figure 24: *Jinja2* templates for the code generation for the component arrangements

```
for row in range(0, {{ num_rows }}, {{ 2+min_rows-1 }}):
    for col in range(0, {{ num_cols }}, {{ 2+min_cols-1 }}):
        {{ combo_name }}(board, colors={{ colors }},x=row, y=col)
```

(a) Place a complex object in alternating columns of the first row, and replicate this pattern in alternating rows throughout the grid.

```
for row in range(0, {{ num_rows }}, {{ min_rows }}):
    for col in range(0, {{ num_cols }}, {{ 2+min_cols-1 }}):
        {{ combo_name }}(board, colors={{ colors }},x=row, y=col)
```

(b) Place a complex object in alternating rows of the first column. Apply this pattern to alternating columns across the grid

```
for row in range(0, {{ num_rows }}, {{ min_rows }}):
    {{ combo_name }}(board, colors={{ colors }}, x=row, y=3)
```

(c) Repeat a complex object in the fourth column of the grid

Figure 25: *Jinja2* templates for the code generation for the component arrangements