Proceedings of the 2024 Winter Simulation Conference
H. Lam, E. Azar, D. Batur, S. Gao, W. Xie, S. R. Hunter, and M. D. Rossetti, eds.

ENHANCING GPT-3.5'S PROFICIENCY IN NETLOGO THROUGH FEW-SHOT PROMPTING AND RETRIEVAL-AUGMENTED GENERATION

Joseph Martínez[1], Brian Llinas[1], Jhon G. Botello[1], Jose J. Padilla[1], and Erika Frydenlund[1]

[1]Virginia Modeling, Analysis, and Simulation Center, Old Dominion University, Suffolk, VA, USA

ABSTRACT

Recognizing the limited research on Large Language Models (LLMs) capabilities with low-resource languages, this study evaluates and increases the proficiency of the LLM GPT-3.5 in generating interface and procedural code elements for NetLogo, a multi-agent programming language and modeling environment. To achieve this, we employed "few-shot" prompting and Retrieval-Augmented Generation (RAG) methodologies using two manually created datasets, *NetLogoEvalCode* and *NetLogoEvalInterface*. The results demonstrate that GPT-3.5 can generate NetLogo elements and code procedures more effectively when provided with additional examples to learn from, highlighting the potential of LLMs in aiding the development of agent-based models (ABMs). On the other hand, the RAG model obtained a poor performance. We listed possible reasons for this result, which were aligned with RAG's common challenges identified by the state-of-the-art. We propose future research directions for leveraging LLMs for simulation development and instructional purposes in the context of ABMs.

1 INTRODUCTION

Large Language Models (LLMs) emerged from the computer science field and are particularly good at widely adopted programming languages (e.g. Python, JavaScript, C, C++), benefitting programmers and early learners of these languages. LLMs such as ChatGPT make this technology accessible to a broader range of people through conversational interaction in an interface that allows users to formulate questions and generate responses without coding. This opens the doors for people without coding experience to engage with computer science or code-based resources through a conversational interface that does not require code, potentially assisting them in building Simulation models.

Several studies have investigated techniques to measure and improve the performance of LLMs in various tasks typically faced by programmers such as program repair and code generation (Murr et al. 2023; Weyssow et al. 2023; Haluptzok et al. 2022). However, most research has focused on high-resource general-purpose programming languages like Python or JavaScript (Chen et al. 2021; Tian et al. 2023; Denny et al. 2023) where the methods rely on large amounts of open-source, public code (Roziere et al. 2023) overlooking domain-specific programming languages lacking significant open-source code or models (Tarassow 2023).

One low-resource programming language is NetLogo, a multi-agent programming language and modeling environment for simulating complex phenomena (Wilensky 1999) whose number of open-source codes and models is reduced compared to high-resource programming languages. NetLogo is highly popular due to its user-friendly interface (Wilensky 1999), extensive documentation, and the ability to incorporate real-world data (Walker and Johnson 2019). Which combined with the ability to handle sophisticated modeling have made it widely used across various fields (Sklar 2007). Research focusing on the potential of LLMs to generate code of NetLogo could have a high impact on modelers in such fields.

Agent-Based Models (ABMs) are used to model complex systems, particularly those with autonomous, interacting agents (Macal and North 2005) in an array of domains, such as ecology, social science, economics, biology, and game theory (Allan 2010). They are particularly useful in gaining insight into the emergent behavior of complex systems (Garcia 2005). As LLMs have been adopted by a wider range of researchers, modelers have started to investigate their potential applications for ABMs. Most of these efforts are through the concept of "intelligent agents" (Junprung 2023; Park et al. 2023; Li et al. 2023), where an LLM gives the decision-making and behaviors of the agents with human-like characteristics. Other research has explored their use for interpreting the agents' actions in an ABM (Lynch et al. 2024). However, generating executable code for ABM models with LLMs has not been widely explored, and our work builds on the only study studying it at present (Frydenlund et al. 2024), which found that generating code from a narrative is not possible at the moment. However, our paper achieves promising results by implementing LLM learning techniques such as few-shot prompting and Retrieval-Augmented Generation (RAG).

For this study, we quantitatively measured the performance of GPT-3.5 to produce interface elements and procedural code procedures from ABM models in NetLogo and then implemented learning techniques to improve such knowledge. To achieve this, we first examine previous efforts to integrate LLMs into NetLogo interfaces and code generation (Section 2.1). We then introduce benchmarking methodologies for evaluating LLM performance in coding tasks and develop specific evaluation datasets (Sections 2.2). Subsequently, we explore prompt engineering techniques and few-shot prompting methods to enhance LLM performance (Section 2.3), along with the implementation of RAG for context-based responses (Section 2.4). After this, we present our methodology (Section 3) and evaluation results (Section 4), highlighting improvements in LLM performance with additional examples and identifying challenges in executability and contextual integration. Finally, we summarize the findings, discuss the potential of LLMs in NetLogo model development, and propose future research directions aimed at integrating interface and code components through collaborative LLM interaction (Section 5).

2 BACKGROUND

2.1 Incorporation of LLMs in NetLogo

Some efforts to incorporate LLMs into NetLogo have been made, such as the introduction of ChatLogo by Chen and Wilensky (2023), a hybrid natural programming language interface for agent-based modeling and programming. With ChatLogo, users can have conversations with an LLM within the NetLogo interface in a mix of natural and programming languages, providing a more user-friendly interface for novice learners. While it achieved good results when requested to create code, it was only tested by creating simple commands like moving agents. There is still a need for an automated approach to evaluate whether the tool can generate code correctly for more complex requests.

In addition, Chen et al. (2024) conducted an interview study with 30 adult participants to understand how functional LLM-based interfaces for learning and practicing NetLogo were for them. Their results show that experts reported more perceived benefits than novices. Therefore, despite the possibility of beginners benefitting from LLM-aided tools for NetLogo, the most significant benefit is for the experts because they already know what they need to do and which functions or commands to use.

2.2 Benchmarking LLMs

The most common way to measure the performance of the LLMs in certain tasks is with benchmarks. Typical benchmarks include AGIEval (Zhong et al. 2023) for evaluating their ability of tackling human-level tasks, BIG-Bench (Srivastava et al. 2022) for measuring and extrapolating their capabilities (measured for 204 tasks at this moment), and MMLU (Hendrycks et al. 2020) for quantifying their multitask language understanding. For LLMs that generate code, most benchmarks follow the format using an evaluation dataset as introduced by the "HumanEval" dataset (Chen et al. 2021). This dataset is usually a compilation of problems or tasks that contain a prompt for the LLM with a primary function, and its expected results. However, no dataset has been created for evaluating the generation of NetLogo interface elements or code, limiting the ability to measure the performance of LLMs for that task. This gap highlights the need for new benchmarks tailored to the unique requirements of agent-based modeling environments like NetLogo.

2.3 Prompt Engineering and "Few-Shot" Prompting

Prompt engineering is a method that improves the ability of a language model to perform predictions based solely on adding task-specific instructions, known as prompts, that guide its behavior and generate desired outputs without updating the model's parameters. Since even minor modifications in natural language can yield better results for a task (Denny et al. 2023), it is crucial to write the tentative prompts for the datasets accurately, as they will significantly impact the results. Furthermore, the performance of LLM-based tools increases when the tasks are decomposed into well-defined microtasks Denny et al. (Denny et al. 2023). Two techniques commonly used in prompt engineering are the usage of *instructive prompts*, which guide the LLM to solve specific tasks, and *contextual prompts*, which include additional context to guide the LLM within a constrained knowledge base (Giray 2023). Aiming to obtain better responses from the LLM with effective prompts, we created the datasets by writing instructive prompts specifying all the parameters and variables of the expected interface elements and code procedures.

Few-shot prompting is an approach to enhance the performance of language models solving tasks, on which a number of examples (*shots*) of queries along with their expected outputs as additional input to the model (Reynolds and McDonell 2021). For instance, "three-shot" prompting includes three examples in the prompt, while the baseline "zero-shot" prompting uses no additional examples. Our research utilized zero-shot prompting to evaluate how well GPT-3.5 performs without any examples, and few-shot prompting to improve the model's performance by providing useful examples. Additionally, to implement the contextual prompting technique, we enriched the prompts by incorporating a model summary alongside the examples. This summary, taken from the description of the model in the NetLogo library, explains what the model is about and how it works.

2.4 Retrieval-Augmented Generation (RAG)

RAG is a method that enhances language models by retrieving information from external sources to generate a response (Lewis et al. 2020). It follows a similar principle to the technique of *contextual prompts,* comparable to a few-shot approach, but in RAG the context comes from an external data source retrieved based on its relevance. It is a promising solution for enhancing the accuracy and credibility of LLMs (Gao et al. 2023), it is also traceable as it allows for the inclusion of a source document (Jiang et al. 2023b). It has been applied to various knowledge-intensive NLP tasks, like retrieving unstructured text documents from Wikipedia (Yu 2022), financial documents (Setty et al. 2024), and biomedical texts (Ji et al. 2024). In the context of software development, a RAG framework has been proposed to supplement code generation and summarization models (Parvez et al. 2021). However, a RAG framework has not been implemented for NetLogo.

3 METHODOLOGY

When constructing a model with NetLogo, two components are commonly used: elements in the interface that allow interaction between the user and the model; and code procedures that combine a series of NetLogo commands that will perform the tasks of the model such as making turtles move, eat, reproduce, and die. Consequently, our study evaluated the proficiency of the model GPT-3.5 in generating both the NetLogo interface elements and code procedures. For this purpose, we employed the gpt-3.5-turbo-0125 version as of March 2024, accessible via the OpenAI API. This specific version was selected due to its enhancements in dialog-oriented tasks, making it an optimal choice for our assessment.

We created two datasets to measure GPT-3.5's understanding of NetLogo for the interface and the code. These datasets were manually created by writing prompts intended to generate such pieces when used for inference in an LLM. The first dataset, *NetLogoEvalInterface*, contains an array of interface elements with prompts expected to generate them when running inference on an LLM. The second dataset, *NetLogoEvalCode*, contains an array of code procedures of a NetLogo model with prompts expected to generate them when running inference on an LLM. More details of these two datasets are in the sections 3.1 and 3.2.

We assessed the performance of the LLM generation capabilities in two learning strategies: using few-shot prompting and combining few-shot prompting with RAG. We intended to use RAG to improve the responses of the LLM with few-shot by adding a retrieval of relevant code implementation examples from an external database containing the Programming Guide from the NetLogo User Manual. Three metrics were used to evaluate the LLM response: (1) whether the code can be executed or not, (2) the number of correct elements (or code pieces) generated, and (3) how similar the generated code is to the desired code using the Bilingual Evaluation Understudy (BLEU) metric. More details in section 3.3 on the metrics used.

3.1 NetLogoEvalInterface: Dataset for evaluation of knowledge of the NetLogo Interface

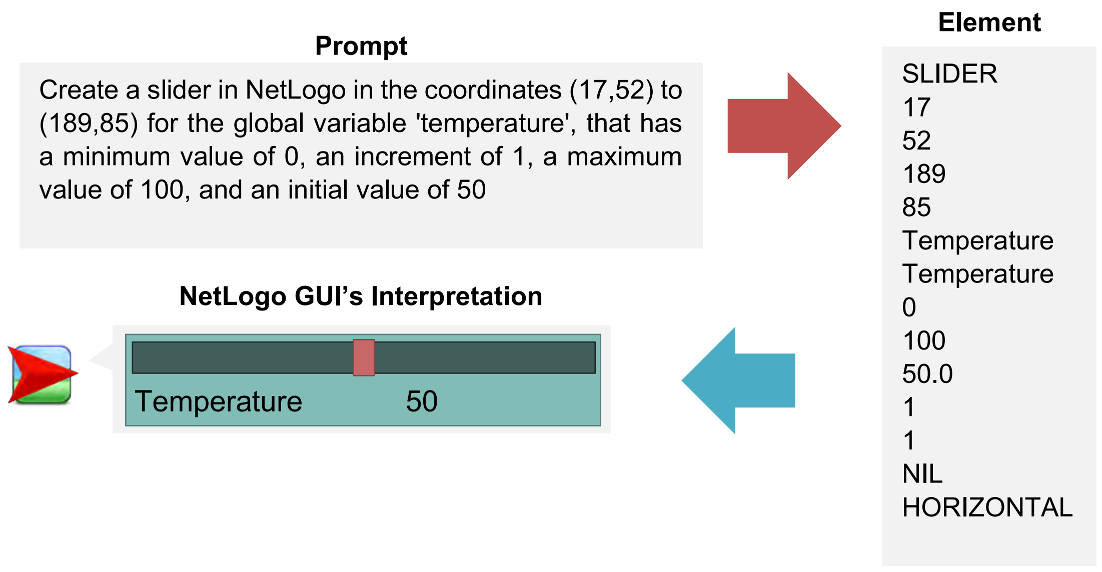
In the interface tab of NetLogo, nine elements can be included: *button, slider, switch, chooser, input, monitor, plot, output*, and *note*. Their descriptions are in Table 1. To build our dataset, we only considered seven of them, excluding *note* and *output* because they do not influence a model behavior and are not associated with global variables.

Table 1: Description of all interface elements in NetLogo.

Element	Description
Button	Executes a set of instructions once or forever.
Slider	Sliders are global variables accessible by all agents. To change the value of a variable the user moves the slider to a value without having to recode the procedure every time.
Switch	Visual representation for a true/false global variable. The user can set the variable to either on (true) or off (false) by flipping the switch.
Chooser	Allows the user to choose a value for a global variable from a list of choices, presented in a drop-down menu. The choices may be strings, numbers, booleans, or lists.
Input	Global variables that contain strings or numbers whose values can be typed by the user inside a text box.
Monitor	Displays the value of any reporter. The reporter could be a variable, a complex reporter, or a call to a reporter procedure
Plot	Shows data that the model is generating.
Output	Scrolling area of text which can be used to create a log of activity in the model
Note	Informative text labels whose contents do not change as the model runs nor affect the model behavior.

In this dataset, we included 28 out of the 67 interface elements from the Garbage Can model (Cohen et al. 1972) implemented by Fioretti (2001) in the NetLogo's user community models library. This particular model implementation was chosen because it contains a large number of elements, allowing the creation of a dataset with a single large model and avoiding contradictions that might arise from using multiple models. For each one, we manually wrote prompts that can generate them when running inference on an LLM. The prompts were written including various details to prevent the LLM from generating incorrect information. These include variable and procedure names, values, and assumptions (initial values assumed by the modeler). We manually inspected the prompts to assess whether they generated accurate answers, and they were refined around two times when they generated better responses. This assessment did not involve using a metric, its purpose was to obtain refined prompts after a few trials.

The elements were represented in the format used in the *.nlogo* files, structured such that each row delineates a different parameter. Figure 1 illustrates an example containing a *slider* element, a prompt that guides the LLM to generate it, and how it would be interpreted within the NetLogo graphical user interface (GUI). The 28 examples, each consisting of an element and its corresponding prompt, were distributed evenly across the seven selected useful elements, with four examples each. This distribution ensures that the LLM encounters a diverse set of elements during training, promoting a balanced understanding to create a robust and reliable dataset that accurately reflects the nuances of the model's interface elements.



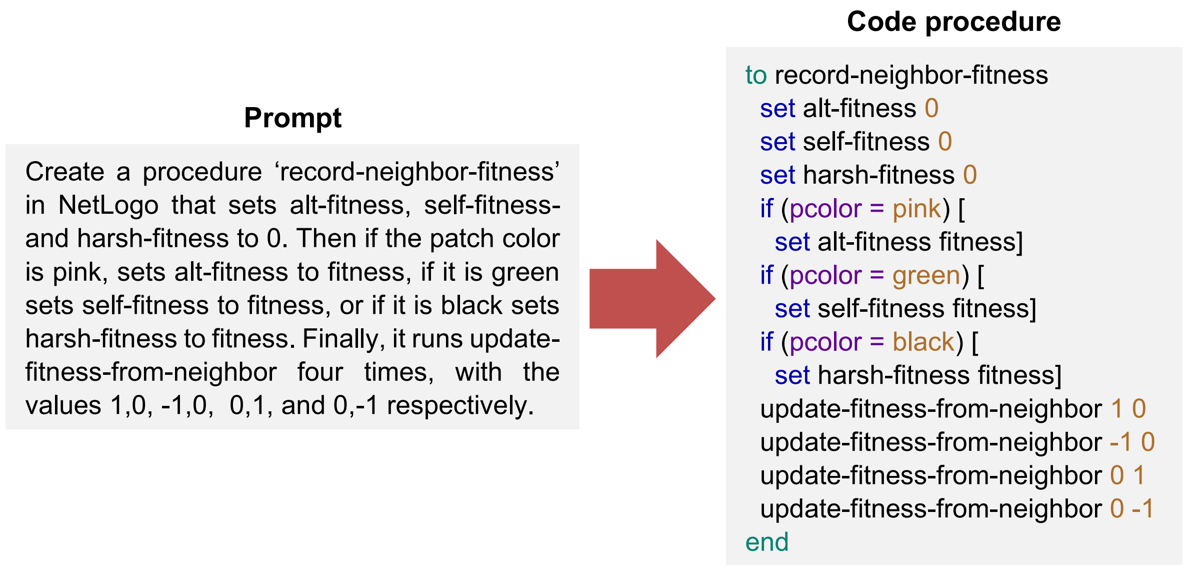
Create a slider in NetLogo in the coordinates (17,52) to (189,85) for the global variable 'temperature', that has a minimum value of 0, an increment of 1, a maximum value of 100, and an initial value of 50

```
SLIDER
17
52
189
85
Temperature
Temperature
0
100
50.0
1
1
NIL
HORIZONTAL
```

Figure 1: Example of a *slider* element, its guiding prompt, and its interpretation within the NetLogo GUI.

3.2 NetLogoEvalCode: Dataset for Evaluation of Knowledge of the NetLogo Code

From the Social Science model library in NetLogo, we selected eight models and divided their code into pieces, each corresponding to a procedure. We also included the primitives *turtles-own*, *patches-own*, and *globals* as pieces because they are essential for defining characteristics of agents, patches, and setting global variables, respectively. Whereas other primitives (e.g. *forward*, *to*, *color,* and arithmetic operators) were not included as pieces because they are usually contained inside procedures. We manually wrote prompts with the potential for generating these code pieces, obtaining a dataset that encompasses both prompts and code sections. Figure 2 presents an example from the dataset, showcasing an example prompt alongside its corresponding code procedure. The entire dataset comprises 93 code pieces from the eight models, with their respective prompts. The number of code pieces taken from each model is detailed in Table 2.



Create a procedure 'record-neighbor-fitness' in NetLogo that sets alt-fitness, self-fitness- and harsh-fitness to 0. Then if the patch color is pink, sets alt-fitness to fitness, if it is green sets self-fitness to fitness, or if it is black sets harsh-fitness to fitness. Finally, it runs update-fitness-from-neighbor four times, with the values 1,0, -1,0, 0,1, and 0,-1 respectively.

```
to record-neighbor-fitness
  set alt-fitness 0
  set self-fitness 0
  set harsh-fitness 0
  if (pcolor = pink) [
    set alt-fitness fitness]
  if (pcolor = green) [
    set self-fitness fitness]
  if (pcolor = black) [
    set harsh-fitness fitness]
  update-fitness-from-neighbor 1 0
  update-fitness-from-neighbor -1 0
  update-fitness-from-neighbor 0 1
  update-fitness-from-neighbor 0 -1
end
```

Figure 2: Example of prompt and expected code procedure.

Table 2: Number of sections per model of the NetLogo code.

Model	**Number of sections**
Altruism	11
Cooperation	12
Divide the Cake	7
HIV	16
Minority Game	18
PD Basic Evolutionary	9
Scatter	12
Segregation	8
Total	**93**

3.3 Metrics

We used three metrics to evaluate the performance of GPT-3.5 generating the interface elements and code procedures: (1) executability, or whether the generated code can be executed or not; (2) proportion of correct portions; and (3) BLEU, which assesses textual similarity between a generated output and a reference. For interface elements, executability was manually labeled as a binary variable with a score of 1 assigned if the element ran correctly when added to a *nlogo* file and 0 if it failed. Similarly, for procedural code, executability was rated 1 when the model was executable after replacing the original procedure with the generated procedure and 0 otherwise. The proportion of correct code pieces was quantified manually by the ratio of correct lines of code (which include parameters, variable names, and commands) divided by the total number of lines. The BLEU score, introduced by Papineni et al. (2002), serves as a metric for evaluating the fidelity of machine-generated text against a human-generated reference by comparing word groupings, or n-grams, between the two texts. It produces a score ranging from 0 to 1, where 1 denotes a perfect match between the generated code and its expected form. This metric can be calculated automatically, offering the advantage of rapid assessment, and facilitating exploratory analyses on its potential for future applications without extensive reliance on manual evaluation. Furthermore, including an executability metric enables the identification of cases where the code, despite being operational, fails to fulfill the specified requirements.

Our experiments explored the effectiveness of the LLM under different instructional conditions, including zero-shot scenarios to benchmark the model's inherent capabilities. To enhance interface performance, we employed few-shot prompting techniques, while procedural code improvement utilized both few-shot prompts and a combination of the same technique with RAG.

3.4 Few-shot Prompting

The prompt templates utilized for the generation of the interface elements and code procedures are *instructive prompts* and *contextual prompts.* These templates comprise a base text that guides the model's functionality through three main components: (1) the question, consisting of the manually written prompts; (2) the examples, which are randomly chosen prompts and not utilized as questions; and (3) the summary of the model description. Figure 3 illustrates the prompt templates utilized for both the interface and code, for the zero-shot evaluation, the template excludes the last sentence and the examples (text in red).

3.5 RAG and Few-Shot Prompting

A RAG model involves two parts: a retriever and a generator. The retriever uses an input sequence to retrieve the *k* most relevant passages from the external documents, and the generator uses the passages as additional context to generate a response (Lewis et al. 2020). For both parts, we used the LLM GPT-3.5. Within the retriever, dividing the documents into smaller sections or passages (chunking) is a key step, and one of the main chunking strategies is paragraph-level chunking which splits the text into sections of the same size (Yepes et al. 2024). We followed the same strategy, dividing the Programming Guide into different sections of the same size. In Figure 4 is presented the framework used for the RAG model.

Interface

You are a helpful assistant that answers the questions provided to you. Assume any additional information you need based on your knowledge, like variables, etc., and avoid using any variable names like X or Y. Only use the format of the examples provided. Use examples of questions and answers to help you answer the question:
Question: {question}
Examples: {examples}
Model description: {model description}

Code

You are a helpful assistant that answers the questions provided to you. Assume any additional information you need based on your knowledge, like variables, etc. Just provide the code, no additional information, descriptions, or explanations. Include comments in the code. Use examples of questions and answers to help you answer the question:
Question: {question}
Examples: {examples}
Model description: {model description}

Figure 3: Prompt templates for interface and code. Text in red is excluded in zero-shot.

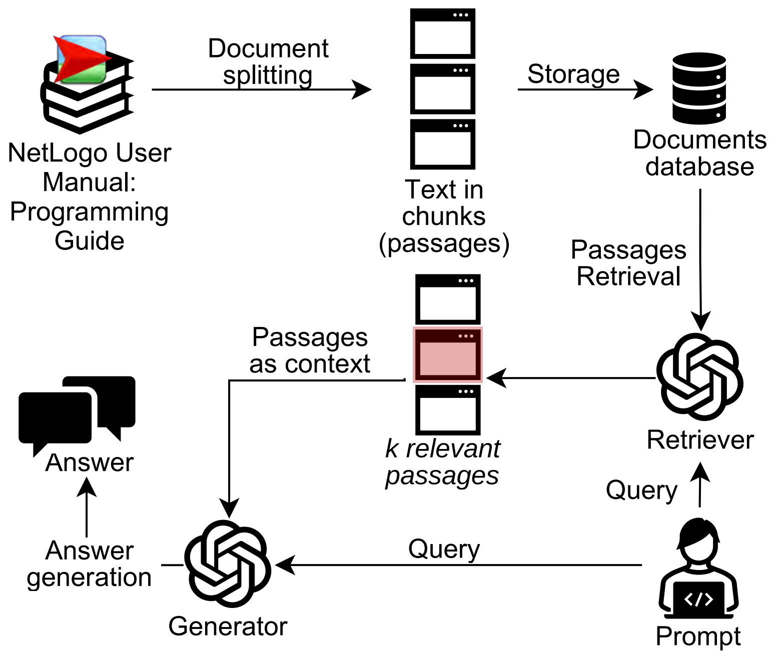


Figure 4: Retrieval-Augmented Generation framework.

While the NetLogo interface has several tutorials online, all the information on the elements is presented textual and does not include the format used in the *.nlogo* files. Because of this, we could not create an external database of useful context for the interface, so we only implemented RAG for the code. In the prompt template for the RAG, we followed a format similar to the template used in few-shot prompting but modified it to incorporate instructions to use the NetLogo Programming Guide as contextual data. Figure 5 illustrates the prompt template used for the RAG, showing the modifications made as compared to the prompt engineering (text in blue). The number of examples (shots) included in the RAG was based on the optimal number identified during the few-shot prompting phase that yielded the best performance. This assessment aimed to determine whether adding "useful" code implementations as in-context information could improve the LLM's performance. Ultimately, our goal was to enhance the model's capability to generate accurate code snippets by leveraging contextual data from the NetLogo Programming Guide.

```
You are a helpful assistant that answers the questions
provided to you. Use the following pieces of context from
the NetLogo Programming Guide to generate code for
the question at the end. Assume any additional
information you need based on your knowledge, like
variables, etc. Just provide the code, no additional
information, descriptions, or explanations. Include
comments in the code. Use examples of questions and
answers to help you answer the question:
Question: {question}
Examples: {examples}
Model description: {model description}
Context from programming guide: {retrieved documents}
```

Figure 5: Prompt template for code RAG. The text in blue represents the additional commands compared to few-shot prompting.

4 RESULTS AND DISCUSSION

To evaluate the LLMs' knowledge of the NetLogo interface we selected seven instances from the dataset *NetLogoEvalInterface* for evaluation, one for each element type. Additionally, several examples were selected randomly and included based on the number of shots. For the code, we chose 16 instances from the dataset *NetLogoEvalCode*, allocating two per model. Additional examples were again selected randomly based on the number of shots. It is important to note that we omitted comments during the code evaluation process. This decision was made to avoid potential reductions in the BLEU metric that could arise from comments written differently. The evaluation results are presented in Table 3 and in Figure 6.

When evaluating the interface with zero-shot prompting, all outputs were incorrect. This outcome was anticipated, as the LLM was not pre-informed about the expected format of the interface elements. Notably, the inclusion of additional examples consistently improved performance across all metrics, with the three-shot learning approach yielding the most favorable outcomes. This is evidenced by both the BLEU scores and the average proportion of correct elements exceeding 0.9, indicating the model's high proficiency in the assigned tasks. However, the average executability score of 0.714 suggests a discrepancy; even minor errors in a number or parameter made the interface elements inexecutable. This highlights a critical vulnerability: Despite the model's high accuracy in generating elements, it remains prone to minor errors that necessitate expert intervention for correction.

Similarly, in the code generation, the use of three examples also achieved the highest metrics. The average proportion of correct code pieces reached 0.826, indicating a solid performance. However, the executability rate was only 0.625 because of minor errors, such as incorrect variable names, procedures, or primitives, prevented the code from running. This equals the interface evaluation, where the model's capability to generate accurate code is damaged by its susceptibility to minor errors demanding expert resolution. The BLEU scores reflected the performance improvement with additional examples. However, the disparity between the BLEU scores and the average proportion of correct pieces was more pronounced in code generation than in interface tasks. This discrepancy may be attributed to extraneous code lines or minor variations in the code implementation. Despite these challenges, the metric BLEU remains a valuable tool for assessing the model's understanding of the NetLogo interface and code generation.

Table 3: LLMs' performance by learning technique.

	Learning technique	**BLEU**	**Average executability**	**Average proportion of correct pieces**
Interface	zero-shot	0	0	0
	one-shot	0.804	0.142	0.824
	two-shot	0.886	0.714	0.905
	three-shot	**0.928**	**0.714**	**0.958**
Code	zero-shot	0.406	0.250	0.571
	one-shot	0.482	0.438	0.791
	two-shot	0.469	0.500	0.748
	three-shot	**0.622**	**0.625**	**0.826**
	RAG three-shot	0.437	0.313	0.518

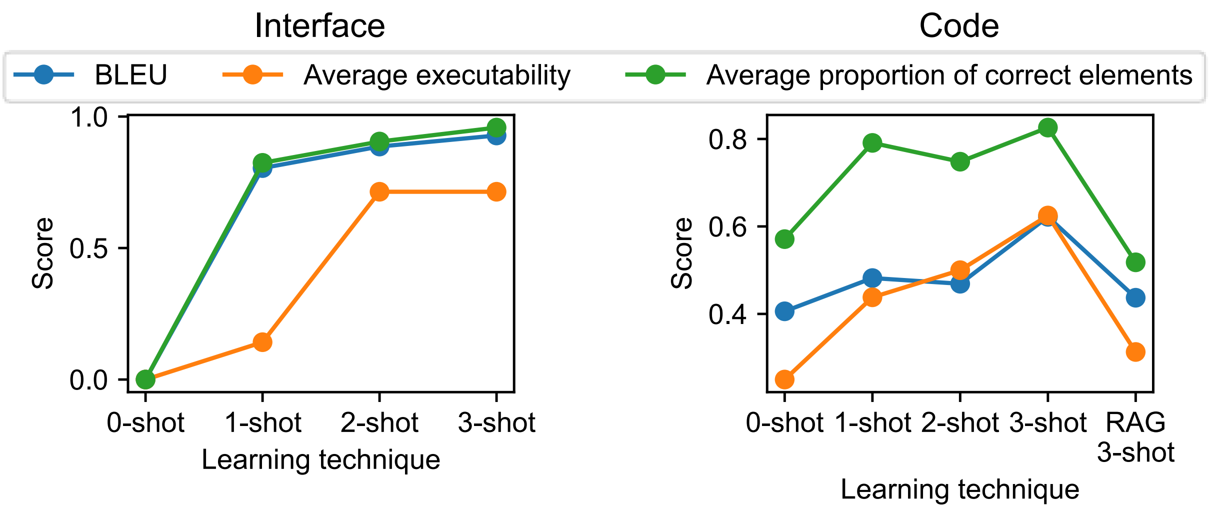


Figure 6. Performance of the learning techniques.

The RAG model exhibited inferior performance in code generation tasks. Upon manual inspection, it was found that the model inaccurately handled primitives and procedure names, where a three-shot learning approach had previously succeeded. Many of these inaccuracies were not derived from the retrieved documents but were instead fabrications by the model itself. Based on the several limitations of RAG models identified by Zhao et al. (2024) and other authors, we attributed the following three potential reasons for the poor performance of the RAG: (1) Noise in retrieval results, which can introduce critical errors despite the necessity of response diversity (Cuconasu et al. 2024). (2) Integration of a retrieval component, though potentially enhancing attribution accuracy, can adversely affect the fluency of outputs (Aksitov et al. 2023). And (3) the model's limited capacity to handle lengthy contexts often leads to bottlenecks. Although advancements have mitigated this issue such as long-context support (Han et al. 2023) and prompt compression techniques (Jiang et al. 2023a), our inclusion of examples, detailed model descriptions, and contextual documents may still exceed the optimal length for effective processing.

Based on this, we attribute that a possible fourth reason (and a critical one) is having a small corpus of documents. While corpora of RAG models are typically assumed to be large (Cuconasu et al. 2024), our corpus only contained approximately 1,600 lines of text, equivalent to a 74-page document. This small size might not only be insufficient but could also lead to counterproductive results when implementing the RAG model. All these factors collectively suggest that the integration of additional information deteriorates its performance in the tasks, aligning with common challenges encountered in the state-of-the-art.

5 CONCLUSION AND FUTURE WORK

Our study focused on assessing and improving the proficiency of the LLMs GPT-3.5, in generating NetLogo interface elements and code procedures. Through evaluation using various two learning techniques (few-shot prompting and RAG), and three metrics (BLEU, average executability, and average proportion of correct pieces) we gained insights into the model's performance and limitations in these tasks.

For interface element generation, we observed a significant improvement in performance by including additional examples, indicating the model's ability to learn from few-shot prompts. However, despite achieving high performance in the generation, the model showed vulnerability to minor errors, affecting executability. This highlights the need for expert intervention to rectify such errors. Similarly, in code generation, the model demonstrated improved performance with additional examples, but executability rates remained lower due to susceptibility to minor errors. Despite achieving high BLEU scores, the model's understanding of code pieces did not always align perfectly with executability.

Furthermore, the RAG model showed poor performance in code generation tasks compared to the few-shot learning, struggling with handling primitives and procedure names and often fabricating information. Key reasons for this include noise in retrieval results, the retrieval component affecting output fluency, and the limited capacity to manage lengthy contexts despite recent advancements. These factors collectively indicate that adding more documents as contextual information deteriorates the model's performance, upheld by common challenges in state-of-the-art research on RAG models.

In summary, while GPT-3.5 shows promise in understanding and generating NetLogo interface elements and code, they are susceptible to minor errors that need expert intervention for correction like incorrect primitive and procedure names. This highlights the high potential of LLMs in aiding the development of ABMs with minimum intervention of a modeler to check the generated code. Given that the most common errors like wrong primitives and procedure names can be easily fixed only if the user is familiar with the NetLogo language, the LLM's potential to aid non-modelers and users with no coding experience is not possible at the time. The future trajectory of research on LLMs and model generation should be oriented towards improving its capabilities for people with no coding experience, as well as the integration of the interface and code components, to construct a holistic model through the LLM interacting collaboratively with the user.

ACKNOWLEDGMENTS

This material, in part, was funded by the grant 22RT0286 through the Air Force Office of Scientific Research (AFOSR) under the Minerva Research Initiative. The views and conclusions contained herein are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of AFOSR.

REFERENCES

Aksitov, R., C.-C. Chang, D. Reitter, S. Shakeri, and Y. Sung. 2023. "Characterizing attribution and fluency tradeoffs for retrieval-augmented large language models". *arXiv preprint arXiv:2302.05578*.

Allan, R. 2010. "Survey of Agent Based Modelling and Simulation Tools". Technical Report DL-TR-2010-007, Science and Technology Facilities Council.

Chen, J., X. Lu, M. Rejtig, D. Du, R. Bagley, M. S. Horn et al. 2024. "Learning Agent-based Modeling with LLM Companions: Experiences of Novices and Experts Using ChatGPT & NetLogo Chat". *arXiv preprint arXiv:2401.17163*.

Chen, J., and U. Wilensky. 2023. "ChatLogo: A Large Language Model-Driven Hybrid Natural-Programming Language Interface for Agent-based Modeling and Programming". *arXiv preprint arXiv:2308.08102*.

Chen, M., J. Tworek, H. Jun, Q. Yuan, H. P. d. O. Pinto, J. Kaplan et al. 2021. "Evaluating large language models trained on code". *arXiv preprint arXiv:2107.03374*.

Cohen, M. D., J. G. March, and J. P. Olsen. 1972. "A Garbage Can Model of Organizational Choice". *Administrative Science Quarterly* 17(1):1-25.

Cuconasu, F., G. Trappolini, F. Siciliano, S. Filice, C. Campagnano, Y. Maarek et al. 2024. "The power of noise: Redefining retrieval for rag systems". *arXiv preprint arXiv:2401.14887*.

Denny, P., V. Kumar, and N. Giacaman. 2023. "Conversing with copilot: Exploring prompt engineering for solving cs1 problems using natural language". In *Proceedings of the 54th ACM Technical Symposium on Computer Science Education V. 1*, 1136-42.

Fioretti, G. 2001. GarbageCan NetLogo Community Model. https://ccl.northwestern.edu/netlogo/models/community/GarbageCan, accessed 12th March 2024.

Frydenlund, E., J. Martínez, J. J. Padilla, K. Palacio, and D. Shuttleworth. 2024. "Modeler in a box: how can large language models aid in the simulation modeling process?". *Simulation* 100(7):727-49.

Gao, Y., Y. Xiong, X. Gao, K. Jia, J. Pan, Y. Bi et al. 2023. "Retrieval-augmented generation for large language models: A survey". *arXiv preprint arXiv:2312.10997*.

Garcia, R. 2005. "Uses of Agent-Based Modeling in Innovation/New Product Development Research". *Journal of Product Innovation Management* 22(5):380-98.

Giray, L. 2023. "Prompt Engineering with ChatGPT: A Guide for Academic Writers". *Annals of Biomedical Engineering* 51(12):2629-33.

Haluptzok, P. M., M. Bowers, and A. T. Kalai. 2022. "Language Models Can Teach Themselves to Program Better". *arXiv preprint arXiv:2207.14502*.

Han, C., Q. Wang, W. Xiong, Y. Chen, H. Ji, and S. Wang. 2023. "Lm-infinite: Simple on-the-fly length generalization for large language models". *arXiv preprint arXiv:2308.16137*.

Hendrycks, D., C. Burns, S. Basart, A. Zou, M. Mazeika, D. Song et al. 2020. "Measuring massive multitask language understanding". *arXiv preprint arXiv:2009.03300*.

Ji, Y., Z. Li, R. Meng, S. Sivarajkumar, Y. Wang, Z. Yu et al. 2024. "RAG-RLRC-LaySum at BioLaySumm: Integrating Retrieval-Augmented Generation and Readability Control for Layman Summarization of Biomedical Texts". *arXiv preprint arXiv:2405.13179*.

Jiang, H., Q. Wu, C.-Y. Lin, Y. Yang, and L. Qiu. 2023a. "Llmlingua: Compressing prompts for accelerated inference of large language models". *arXiv preprint arXiv:2310.05736*.

Jiang, Z., F. F. Xu, L. Gao, Z. Sun, Q. Liu, J. Dwivedi-Yu et al. 2023b. "Active retrieval augmented generation". *arXiv preprint arXiv:2305.06983*.

Junprung, E. 2023. "Exploring the intersection of large language models and agent-based modeling via prompt engineering". *arXiv preprint arXiv:2308.07411*.

Lewis, P., E. Perez, A. Piktus, F. Petroni, V. Karpukhin, N. Goyal et al. 2020. "Retrieval-augmented generation for knowledge-intensive nlp tasks". *Advances in Neural Information Processing Systems* 33:9459-74.

Li, N., C. Gao, Y. Li, and Q. Liao. 2023. "Large language model-empowered agents for simulating macroeconomic activities". *arXiv preprint arXiv:2310.10436*.

Lynch, C. J., E. Jensen, M. H. Munro, V. Zamponi, J. Martinez, K. O'Brien et al. 2024. "GPT-4 Generated Narratives of Life Events using a Structured Narrative Prompt: A Validation Study". *arXiv preprint arXiv:2402.05435*.

Macal, C., and M. North. 2005. "Tutorial on Agent-based Modeling and Simulation". In *2005 Winter Simulation Conference (WSC),* 2-15 https://doi.org/10.1109/WSC.2005.1574234.

Murr, L., M. Grainger, and D. Gao. 2023. "Testing LLMs on Code Generation with Varying Levels of Prompt Specificity". *arXiv preprint arXiv:2311.07599*.

Papineni, K., S. Roukos, T. Ward, and W.-J. Zhu. 2002. "Bleu: a Method for Automatic Evaluation of Machine Translation". In *Proceedings of the 40th Annual Meeting of the Association for Computational Linguistics*. July, Philadelphia, Pennsylvania, USA, 311-18.

Park, J. S., J. O'Brien, C. J. Cai, M. R. Morris, P. Liang, and M. S. Bernstein. 2023. "Generative Agents: Interactive Simulacra of Human Behavior". In *Proceedings of the 36th Annual ACM Symposium on User Interface Software and Technology*, San Francisco, CA, USA, 1-22.

Parvez, M. R., W. U. Ahmad, S. Chakraborty, B. Ray, and K.-W. Chang. 2021. "Retrieval augmented code generation and summarization". *arXiv preprint arXiv:2108.11601*.

Reynolds, L., and K. McDonell. 2021. "Prompt Programming for Large Language Models: Beyond the Few-Shot Paradigm". *Extended Abstracts of the 2021 CHI Conference on Human Factors in Computing Systems,* Yokohama, Japan, 1-7.

Roziere, B., J. Gehring, F. Gloeckle, S. Sootla, I. Gat, X. E. Tan et al. 2023. "Code llama: Open foundation models for code". *arXiv preprint arXiv:2308.12950*.

Setty, S., K. Jijo, E. Chung, and N. Vidra. 2024. "Improving Retrieval for RAG based Question Answering Models on Financial Documents". *arXiv preprint arXiv:2404.07221*.

Sklar, E. 2007. "NetLogo, a Multi-agent Simulation Environment". *Artificial Life* 13(3):303-11.

Srivastava, A., A. Rastogi, A. Rao, A. A. M. Shoeb, A. Abid, A. Fisch et al. 2022. "Beyond the imitation game: Quantifying and extrapolating the capabilities of language models". *arXiv preprint arXiv:2206.04615*.

Tarassow, A. 2023. "The potential of LLMs for coding with low-resource and domain-specific programming languages". *arXiv preprint arXiv:2307.13018*.

Tian, H., W. Lu, T. O. Li, X. Tang, S.-C. Cheung, J. Klein et al. 2023. "Is ChatGPT the ultimate programming assistant--how far is it?". *arXiv preprint arXiv:2304.11938*.

Walker, B., and T. Johnson. 2019. "NetLogo and GIS: A Powerful Combination". In *Proceedings of 34th International Conference on Computers and Their Applications*. March 18th-20th, Honolulu, Hawaii, USA, 257-64.

Weyssow, M., X. Zhou, K. Kim, D. Lo, and H. A. Sahraoui. 2023. "Exploring Parameter-Efficient Fine-Tuning Techniques for Code Generation with Large Language Models". *arXiv preprint arXiv:2308.10462*.

Wilensky, U. 1999. NetLogo (and NetLogo User Manual). https://ccl.northwestern.edu/netlogo/, accessed 12th March 2024.

Yepes, A. J., Y. You, J. Milczek, S. Laverde, and L. Li. 2024. "Financial Report Chunking for Effective Retrieval Augmented Generation". *arXiv preprint arXiv:2402.05131*.

Yu, W. 2022. "Retrieval-augmented generation across heterogeneous knowledge". In *Proceedings of the 2022 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies: Student Research Workshop*. July 10th-15th, Seattle, Washington, USA, 52-58.

Zhao, P., H. Zhang, Q. Yu, Z. Wang, Y. Geng, F. Fu et al. 2024. "Retrieval-Augmented Generation for AI-Generated Content: A Survey". *arXiv preprint arXiv:2402.19473*.

Zhong, W., R. Cui, Y. Guo, Y. Liang, S. Lu, Y. Wang et al. 2023. "Agieval: A human-centric benchmark for evaluating foundation models". *arXiv preprint arXiv:2304.06364*.

AUTHOR BIOGRAPHIES

JOSEPH MARTÍNEZ is a Graduate Research Assistant at the Virginia Modeling, Analysis, and Simulation Center (VMASC) at Old Dominion University. His research focuses on implementing Natural Language Processing (NLP) and Machine Learning (ML) models in simulation development, analysis of social media, and study of migration and refugee host communities. His email address is jmart130@odu.edu, and his web page is https://josephmars.github.io/.

BRIAN LLINAS is a Ph.D. student in Computer Science and a Graduate Research Assistant at VMASC. His research focuses on applying ML and NLP models in news articles and social media to analyze the perception of host communities toward migrants. His email address is bllin001@odu.edu.

JHON G BOTELLO is a Ph.D. student in Computer Science and a Graduate Research Assistant at VMASC. His research focuses on employing ML, NLP, and survey research to build and evaluate social models concerning migration and mobilities. His email address is jbote001@odu.edu, and his web page is https://jgbotello.github.io/.

JOSE J PADILLA is a Research Associate Professor at VMASC. His primary research focuses on advancing computational modeling methods toward increasing modeling accessibility across ages and disciplines. His research generates insight into topics ranging from forced migration to community resilience. His email address is jpadilla@odu.edu.

ERIKA FRYDENLUND is a Research Associate Professor at VMASC. Her primary research focus is on migration and mobility. Much of her work focuses on combining quantitative and qualitative data in simulations to understand the emergence, dynamics, and consequences of human migration and displacement. Her email address is efrydenl@odu.edu.