LEAR: LLM-Driven Evolution of Agent-Based Rules

Can Gurkan*
gurkan@u.northwestern.edu
Northwestern University
Evanston, IL, USA

Narasimha Karthik Jwalapuram
Northwestern University
Evanston, IL, USA

Kevin Wang
Northwestern University
Evanston, IL, USA

Rudy Danda
Northwestern University
Evanston, IL, USA

John Chen
Northwestern University
Evanston, IL, USA

Leif Rasmussen
College of DuPage
Glen Ellyn, IL, USA

Uri Wilensky
Northwestern University
Evanston, IL, USA

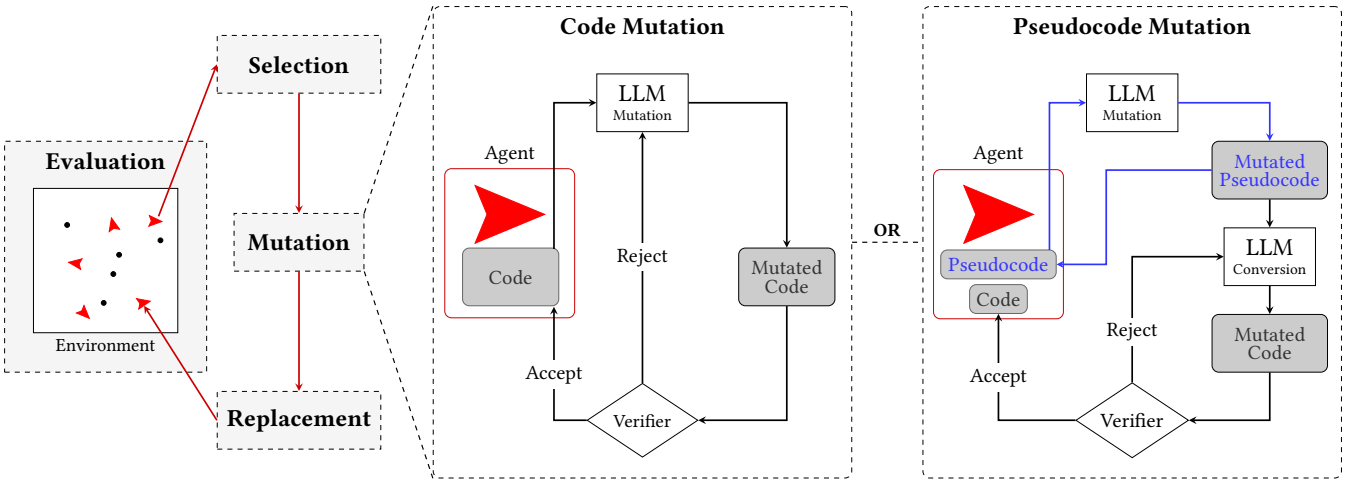


Figure 1: Overview of the LEAR framework. The evolutionary operators acting on agents in a multi-agent environment (Left) and the details of the two LLM-driven mutation paradigms for mutating code (Middle) or pseudocode (Right).

Abstract

This study investigates the feasibility and effectiveness of integrating Large Language Models (LLMs) as mutation operators within Genetic Programming (GP) frameworks so as to evolve agent behaviors in multi-agent systems (MAS) and provide benchmarks that evaluate the efficacy of LLM-generated code in multi-agent domains. Our approach leverages the sophisticated code-generation capabilities of LLMs to introduce semantically meaningful variations during the evolutionary process. Specifically, we explore and systematically compare these different prompting strategies: zero-shot, one-shot, and two-shot prompting as well as prompting the generation of commented code to assess their impact on the quality of evolved agent behaviors. Additionally, we propose a novel methodology where evolution operates at a higher abstraction level by mutating pseudocode representations of agent behaviors, subsequently converting them into executable code through another LLM-mediated step. This strategy capitalizes on the extensive natural language training data of LLMs, potentially enabling the discovery of more innovative solutions. Our results indicate that LLM-driven mutation with comment generation enhances agent performance while mutating pseudocode representations yields reduced performance. This research contributes valuable insights regarding the integration of LLM-driven GP techniques into MAS, highlighting both the potential and limitations of these approaches. All code is open-sourced at https://github.com/can-gurkan/LEAR.

*Corresponding Author

GECCO '25 Companion, July 14–18, 2025, Malaga, Spain

ACM ISBN 979-8-4007-1464-1/2025/07
https://doi.org/10.1145/3712255.3734368

CCS Concepts

• **Computing methodologies** → **Multi-agent systems**; *Intelligent agents*; **Natural language generation**; **Agent / discrete models**; *Simulation environments*; *Modeling methodologies*; **Genetic programming**; **Genetic algorithms**; **Generative and developmental approaches**.

Keywords

Large Language Models, Genetic Programming, Evolutionary Computation, Multi-Agent Systems, Agent-Based Modeling

ACM Reference Format:
Can Gurkan, Narasimha Karthik Jwalapuram, Kevin Wang, Rudy Danda, John Chen, Leif Rasmussen, and Uri Wilensky. 2025. LEAR: LLM-Driven Evolution of Agent-Based Rules. In *Genetic and Evolutionary Computation Conference (GECCO '25 Companion), July 14–18, 2025, Malaga, Spain.* ACM, New York, NY, USA, 18 pages. https://doi.org/10.1145/3712255.3734368

1 Introduction

Recent advances in Large Language Models (LLMs) have demonstrated their remarkable capabilities in comprehending and generating complex textual data, including computer code [5, 16, 21]. LLMs, trained on extensive corpora comprising both natural language and code repositories, exhibit a robust capacity for in-context learning and generating contextually coherent outputs. This proficiency facilitates the emergence of a novel paradigm of LLM-driven discovery, in which LLMs iteratively enhance diverse computational solutions, such as optimized prompts, improved algorithmic strategies, and refined code across various domains and problem sets [6, 11, 20, 25, 27–30, 41, 42]. A particularly promising direction, pioneered in the seminal work by Lehman et al. [26], involves the integration of LLMs within evolutionary algorithms, synergistically combining evolutionary principles with the generative power of large language models [48].

Evolutionary algorithms (EAs) consist of population-based optimization methods inspired by the principles of biological evolution [2]. These algorithms iteratively improve candidate solutions by employing mechanisms analogous to natural selection, genetic variation, and inheritance. Central to the operation of an EA is the systematic application of evolutionary operators, such as evaluation, selection, mutation, and replacement, which enable the exploration and exploitation of the search space. A subset of EAs specialized in evolving executable code constitutes the field of Genetic Programming (GP) [23, 33, 34, 45].

Traditional GP approaches have been employed to tackle a wide range of problems, including the evolution of agent behaviors by applying evolutionary operators to programmatic representations. However, the efficacy of these operators is often constrained by their reliance on syntactic manipulations, which may not effectively capture the semantic nuances necessary for generating innovative and functional behaviors. To compensate for this inadequacy, an exciting new approach [4, 19, 26, 31] has emerged to utilize LLMs in producing syntactically correct code, thus empowering evolutionary operators in GP. Inspired by the nomenclature in [19], we will refer to this approach as LLM-GP. Despite recent interest in the LLM-GP approach [18], many avenues of interest that could benefit from the synergistic integration between LLMs and GP remain to be explored.

The following sections outline the primary contributions of our study and explain how our research addresses existing gaps within these promising avenues.

1.1 Evolving Agent Rules with LLM-GP in Multi-Agent Systems

One compelling avenue is the application of LLM-powered GP techniques to evolve agent controllers in multi-agent systems (MAS), where many interacting agents can specialize into distinct roles and co-evolve complex behaviors akin to those observed in nature. LLMs in multi-agent systems have recently received considerable attention [12, 14]; however, this study is the first to employ an LLM-GP approach in MAS. This approach could be a promising candidate for optimizing agent behaviors to achieve desired collective outcomes—a significant and ongoing challenge within the field of artificial intelligence. Furthermore, the inherently stochastic and black-box nature of LLMs and their sensitivity to prompts and training data warrant the need for benchmarks. Hence, we introduce a collection of three multi-agent environments, implemented in NetLogo—the most widely used agent-based modeling platform [46]—that can serve as benchmarks for evaluating LLM-GP in MAS. This initial collection represents the first installment in a planned suite of LLM-driven NetLogo simulations that we aim to develop in future work.

1.2 Evaluating Prompting Strategies

Effective utilization of LLMs as evolutionary operators necessitates systematic exploration of prompting strategies to guide the LLM's output. Prompting strategies such as zero-shot and multi-shot prompting have been shown to influence the relevance, diversity, and quality of LLM-generated content [5]. In zero-shot prompting, the model generates outputs without explicit examples, relying solely on the prompt and its pre-trained knowledge. In contrast, one-shot and two-shot prompting involve providing one or two examples, respectively, to guide the model's output generation.

Another prompting strategy that we consider involves instructing the LLM to generate code accompanied by explicit comments that articulate the strategy or behavior the LLM intended to implement. We hypothesize that generating commented code can enable the LLM to reason with semantic information. This enriched representation, which would otherwise be difficult to infer from the code alone, may leverage the powerful contextual understanding and semantic representation capabilities of LLMs, thereby enhancing the potential to propose more intelligent modifications in future code mutations.

This study systematically evaluates these prompting strategies within the context of evolving agent behaviors in multi-agent simulations.

1.3 Evolving Pseudocode Representations

A novel aspect of this study is the proposition of using pseudocode representations to evolve code. Pseudocode can be defined as simplified, human-readable text that uses natural language to represent the logical structure of an algorithm or program without adhering strictly to the syntax of any specific programming language. Thus, pseudocode descriptions can offer a high-level semantic representation of agent behaviors. In our framework, agent behavior can be expressed with pseudocode, which is then mutated using an LLM-powered operator and is subsequently translated into executable code using a separate LLM operator.

This approach of using pseudocode representations is motivated by the observation that LLMs have been trained extensively on both natural language and pseudocode when compared to domain-specific programming languages such as NetLogo. As a result, using pseudocode representations may potentially enable LLMs to take

advantage of the semantically rich nature of high-level pseudocode descriptions to generate more innovative mutations and ultimately more creative code. By constraining program representations to a different latent manifold in the representation space of LLMs, compared to that of code representations, this method may allow for higher-level abstraction in the evolutionary process that could simplify the search space and enhance the efficiency of evolution.

An additional advantage of evolving pseudocode representations, as well as generating commented code as discussed in the previous section, is their inherent interpretability, which aligns closely with the principles underlying explainable artificial intelligence (XAI) [36, 49]. Unlike executable code, which often involves syntactically complex language-specific constructs, pseudocode is intentionally structured in a human-readable format. This transparency is valuable for facilitating a clearer understanding of algorithmic logic and agent behaviors, which is becoming increasingly rare in the age of neural network-driven agent controllers. Moreover, this approach aligns with the newly popularized term of "vibe coding" where rather than writing code directly, programmers increasingly rely on LLMs to generate code, an innovation highlighting the relevance of studying natural language-based representations of code in generative frameworks.

1.4 Research Objectives and Contributions

To summarize, the primary objectives of this research are as follows:

(1) To evaluate the feasibility and effectiveness of using LLMs as evolutionary operators in evolving agent code in multi-agent systems and introduce benchmarks implemented in NetLogo that evaluate the efficacy of LLM-generated code in multi-agent domains.
(2) To systematically assess the impact of different prompting strategies that consist of zero-shot, one-shot, and two-shot prompting as well as prompting the generation of commented code on the quality of evolved agent behaviors.
(3) To investigate the potential benefits of using LLMs to evolve pseudocode representations and translating them to executable code, in enhancing the innovation and functionality of agent behaviors.

By addressing these objectives, this study aims to contribute to the advancement of LLM-driven genetic programming methodologies and the development of more sophisticated and interpretable agent behaviors in multi-agent systems.

2 Related Work

In this section, we summarize relevant studies that use LLM-based evolutionary operators to evolve code for agent-controllers or employ natural language abstractions to enhance code generation. Additional background on large language models, genetic programming, and multi-agent systems is provided in Appendix A.

Lehman et al. [26] are the first to integrate LLMs into genetic programming to evolve computer programs in an approach called Evolution through Large Models (ELM). The authors use LLMs as mutation operators within the MAP-Elites algorithm [35], an evolutionary quality-diversity algorithm, to generate a diverse set of Python programs that describe the morphology of ambulatory robots in a 2D terrain, known as the Sodarace domain. The generated programs are then utilized to train a new conditional language model capable of producing appropriate robotic walkers tailored to specific terrains. This methodology paves the way for the LLM-GP approach [18]. Although the Sodarace domain provides a rich visual description, the programs that encode the morphology of the robots are difficult to interpret since they do not directly describe agent behavior.

Meyerson et al. [31] introduce an approach that builds on the work of [26] by leveraging LLMs as an intelligent crossover operator, called Language Model Crossover (LMX). This approach highlights the flexibility and suitability of using LLMs within evolutionary algorithms across diverse domains. The LMX mechanism produces variations in solutions by considering a few parent text-based genotypes—such as code snippets, plain-text sentences, or mathematical expressions. These parents are provided as prompts to an LLM, which then generates offspring that incorporate and recombine features from the parents. This method demonstrates versatility across various domains, including the evolution of binary bit-strings, sentences, equations, text-to-image prompts, as well as Python code for agent controllers in the sodarace domain.

Bradley et al. [4] introduce OpenELM, an open-source Python library designed to integrate LLMs into evolutionary algorithms using the approaches introduced by [26, 31]. The authors democratize the use of ELM and LMX methods by providing a user-friendly library along with four benchmark domains: Sodarace, image generation, prompts, and programming puzzles.

Liu et al. [27] apply LLMs to produce heuristics evaluated on combinatorial optimization problems. Their approach, termed Evolution of Heuristics (EoH), employs LLMs to generate and evolve heuristic strategies articulated in both high-level natural language descriptions and executable code simultaneously. The authors demonstrate that EoH outperforms traditional handcrafted heuristics as well as other LLM-based code generation methods on several benchmark optimization tasks while requiring fewer LLM queries.

3 Methods

Our methodology consists of several key components, including evolutionary operators, particularly the LLM-driven mutation operators, along with agents and environments. In this section, we provide a detailed explanation of these evolutionary operators and their interaction with agents and environments. Section 3.1 provides an overview of how these evolutionary operators collectively facilitate the evolution of agent rules. Section 3.2 describes our specific implementation of LLMs as mutation operators. Section 3.3 details our use of pseudocode representations for mutating agent code. Section 3.4 discusses the various prompting strategies employed to guide LLM outputs.

3.1 Overview

This section outlines the evolutionary process of agents within a given multi-agent environment. The evolution process is structured into four stages that correspond to the main evolutionary operators: evaluation, selection, mutation, and replacement. Figure 1 (Left) illustrates the interaction between these operators, and Algorithm 1 details the evolution procedure.

Algorithm 1 Main Evolution Loop

```
Given simulation environment S, number of agents |A|, number of generations N, simulation steps T, fitness function f, number of parents m
Initialize |A| agents with an initial rule r_i ← r_init
for generation = 0 to N do
    Initialize environment S
    Set the fitness of each agent f_i ← 0
    for iteration = 0 to T do
        for each agent A_i do
            Get observation
            Execute rule r_i
            Update fitness f_i using f
        end for
        Update environment state
    end for
    Choose m parents using tournament selection
    Kill m agents with the lowest fitness
    for each parent do
        Produce offspring with rule r_i ← mutate(r_i)
    end for
    Add all offspring to population
end for
```

3.1.1 Initialization. The first step is to initialize a given environment and a population of agents. Each agent is initialized with the same simple initial rule. The initial code and pseudocode rules are provided in Section 5.

3.1.2 Evaluation. Following initialization, the first generation of agents undergoes evaluation. At each generation's start, agents reset their fitness scores to zero. The simulation environment then runs for a defined number of time steps so that the agents' rules can be evaluated. At each simulation timestep, the agents' fitness scores are updated based on a fitness function tailored to the specific environment. The environments are detailed further in Section 4.

3.1.3 Selection. Following the evaluation of the agents, we employ tournament selection to select parent agents that will produce the next generation of offspring. Tournament selection, a widely used selection mechanism in evolutionary algorithms, probabilistically favors individuals with higher fitness while preserving population diversity [32]. This method randomly selects a subset of individuals from the population (with a size defined by `tournament size`). Within this subset, individuals are compared based on fitness, and the best-performing individual is chosen with a probability defined by `selection pressure`. This probabilistic approach allows for the occasional selection of lower-ranked individuals in order to prevent premature convergence. The selection process is repeated until the required number of parents is selected, noting that the same agent can be selected multiple times.

3.1.4 Mutation. Once the parent agents are selected, each parent produces an offspring. The offspring inherits a mutated version of its parent's rule. The only LLM-powered operator that we consider is the mutation operator. We implement two distinct mutation schemes: one that directly mutates agent code, and another that uses pseudocode representations to mutate agent code, as explained in detail in Sections 3.2 and 3.3 respectively. While LLMs can be incorporated into other evolutionary operators, such as selection, we observed that their integration into variation operators such as mutation yields the greatest benefit for evolving agent controllers for the environments we consider while maintaining a low computational and economic cost.

3.1.5 Replacement. Newly generated offspring replace agents with the lowest fitness scores in the existing population. Notably, the offspring's fitness scores remain unknown at the time of replacement; thus, it is possible for new offspring to perform worse than the agents they replace, reflecting natural evolutionary processes.

3.2 LLM as Mutation Operators

To integrate LLMs as mutation operators within a GP framework, we employ a structured process in which agent rules, represented either as NetLogo code or pseudocode, are provided as input to the LLM with specific prompts designed to elicit semantically meaningful mutations. Figure 1 (middle) and Algorithm 2 (in Appendix B) outline the mutation process that operates directly on code representations. The LLM generates modified rules that are then evaluated within the simulation environment based on predefined fitness criteria. However, before the generated code is accepted, it is subjected to a rigorous verification process to ensure that the code is error-free and relevant to execute.

3.2.1 Verifier. The verification process ensures that the mutated code satisfies three criteria: (1) adherence to NetLogo syntax, (2) absence of undeclared variables, and (3) avoidance of modifications to the agent's own variables, interference with other agents, or disruption of the environment in an undesirable way—for example, preventing mutations in the code that would eliminate other agents. If a mutated rule fails verification, the rejected code is sent back to the LLM with an appropriate prompt that contains an error message and corresponding line numbers from the verifier to facilitate targeted corrections.

3.2.2 Retry Mechanism. In cases where verification fails, the rejected code undergoes a retry process. The failed code, along with a detailed error message including line numbers, is collated into a retry prompt. The LLM then generates new code using this prompt designed explicitly to address the error messages. Rather than generating entirely new logic, the retry mechanism directs the LLM to revise only the problematic elements highlighted by the verifier, thus allowing for a focused correction process that efficiently refines the mutated rules. Despite this verification process, faulty code can still slip through, in which case the agents do not perform any actions.

3.3 Pseudocode Evolution

In this study, we introduce a novel mutation approach employing pseudocode representations. Initially, each agent is assigned an initial pseudocode along with its corresponding executable NetLogo code. During mutation, the parent's pseudocode is supplied to the LLM along with a tailored prompt to generate a mutated version. This mutated pseudocode is then inputted into the LLM a second time using a different prompt that instructs the model to translate

the high-level pseudocode into executable NetLogo code. The resulting NetLogo code subsequently undergoes a verification process analogous to the one outlined in Section 3.2.1. Should the verifier detect errors, an iterative process ensues: the mutated prompt and the faulty code with corresponding error messages are returned to the LLM with an appropriate prompt to guide targeted corrections until the code passes verification or reaches a predefined retry limit. Figure 1 (Right) and Algorithm 3 (in Appendix B) outline this mutation procedure that uses pseudocode representations.

3.4 Prompts

Since every LLM call is made using prompts, there are several different types of prompts we use throughout our system. We outline them here.

3.4.1 Prompts for Mutation. The mutation operators are supplied with prompts that describe the context of the simulation environment and the specific objectives the agent is expected to achieve. Using this information, the LLM generates modified code based on the code it receives as input.

By incorporating the specific contextual details of the environment within the prompt, the LLM can leverage its reasoning capabilities to produce code that is grounded in the context of the simulation. There are specific goals and caveats of the environment that the LLM must consider to ensure relevance and validity before generating the code. Section 5 and Appendix G further describe the details pertaining to mutation prompts.

3.4.2 Prompts for Fixing Code Errors. Our methodology involves a verification process as described in Section 3.2.1. As part of this process, we attempt to fix any generated code that fails to be verified. To do so, we construct a prompt that instructs the LLM to fix the errors in the code. This prompt includes the specific error message the verifier produces and the line numbers in the faulty code. In the case of pseudocode-based mutation, the prompt also contains the mutated pseudocode. See Appendix G for specific examples.

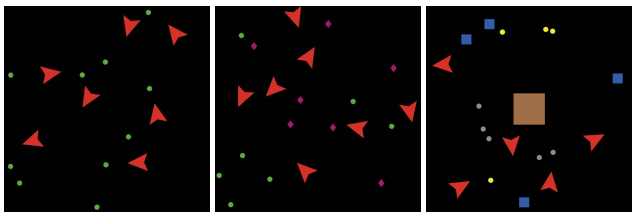
4 Environments

We introduce three multi-agent environments implemented in NetLogo, each designed to facilitate the evolution of creative strategies while minimizing constraints on agent behavior. Figure 2 provides a depiction of each environment. In these environments, agents are tasked with collecting various types of resources using strategies that must effectively balance exploration and exploitation. The environments are arranged in order of increasing difficulty, with each successive setting requiring more sophisticated agent behaviors. We propose these environments as a benchmark for evaluating LLM-GP approaches in the context of multi-agent systems.

4.1 Agents

Each environment contains agents with rules that can be mutated using our methodology. Each agent has the following attributes:

- **Fitness score:** A metric that records the performance of the agent in a given environment.
- **Code Rule:** NetLogo code that determines the main behavior of the agent. *Can be mutated by LLM-driven mutation operators.*



(a) Simple (b) Hazardous (c) Resources

Figure 2: Depictions of the multi-agent NetLogo environments. In each environment, red arrows depict the agents whose behavior is evolved using LLMs (a): The collection simple environment. (b): The collection hazardous environment. (c): The collection resources environment. See Section 4 for more details on each environment.

- **Pseudocode Rule:** (If Applicable) Pseudocode that determines the code rule of the agent. *Can be mutated by LLM-driven mutation operators.*
- **Observation Vector:** List or lists that contain information about the agent's local environment. See Section 4.1.1 for more information.

4.1.1 Agent Observation. The agents have access to only local information about their environment. This information is encoded in lists. Each agent has a field of vision consisting of n_{sector} circular sectors with angle α and radius r_{sector}. In our environments, we used $n_{sector} = 3, \alpha = 20°, r_{sector} = 7$, as depicted in Figure 4 (in Appendix C). Each sector corresponds to an entry in a list that the agent can access. In each sector, the distance to the closest item of a given type is recorded. If the sector is empty, the entry for that sector is zero. If the environment contains multiple items of interest, for example, food and poison sources, then the agent has multiple lists in its observation vector.

4.2 Collection Simple

The Collection Simple environment consists of agents that can move around freely and food sources that these agents can consume. In this environment, the LLM-generated agent code is meant to explore the space and efficiently pursue food sources. Once an agent approaches a food source, the food is automatically consumed and the fitness of the agent increases. Hence, the fitness of the agents is measured by how many food sources they can collect in a given period of time steps. Once a food source is collected, another one spawns at a random location so that the total number of food sources remains constant. In our experiments, we used 30 food sources. This simple environment is designed to evaluate the extent to which agents can utilize their limited observations to accurately locate food sources while exploring the available toroidal space.

4.3 Collection Hazardous

The Collection Hazardous environment is similar to the Collection Simple environment, except the agents have an additional task of avoiding poison as they are collecting food sources. Again, an agent automatically consumes food or ingests poison if it is nearby. While collecting food items adds to agent fitness, collecting poison

Table 1: Parameter Values for Experiments

Parameter	Value
Number of Agents	10
Number of Generations	300
Simulation Time Steps	500
Number of Parents	1
Tournament Size	8
Selection Pressure	0.8
LLM Provider	Groq
LLM Model	llama-3.3-70b-versatile
Temperature	0.65
Max Tokens	1024
Number of Retries	2

detracts from it. Our experiments used 30 food sources and 50 poison items. This environment design requires agents to carefully assess the location of both food and poison items and to effectively navigate the environment to maximize fitness.

4.4 Collection Resources

The Collection Resources environment is designed to evaluate agents based on their ability to efficiently collect and manage resources with varying values and weights. Specifically, the environment includes three different types of resources: silver coins, gold coins, and crystals, which are worth 1, 2, and 4 points, respectively. These resources spawn randomly across the map. Each agent seeks to maximize their cumulative point total, defined as the fitness score in this environment, by collecting and storing these resources in their inventory. However, higher-value resources are also heavier, imposing greater weight penalties on agents that carry them. The agents' loss of points at a rate proportional to the current total weight of their inventory implies that holding onto a large quantity of high-value resources will incur an accelerated point loss over time. To mitigate this loss, agents can deposit their collected resources in a chest located at the center of the map. This action resets their weight totals as well as the rate at which they lose points while preserving their accumulated point total. The motivation underlying this environmental design is to encourage agents to develop efficient and adaptive behaviors that balance resource collection with effective weight management. In particular, agents must learn to optimize the timing of their deposits to the central chest and their routes in order to minimize point loss while maximizing resource acquisition.

5 Experiments

In this section, we describe the experiments conducted to evaluate the different prompting and mutation strategies we consider. Our methodology relies heavily on guiding the LLM through carefully structured prompts. We use a base prompt structure that provides the LLM with essential context about the simulation environment, the agent's task (e.g., resource collection), available sensory inputs (observation vectors), valid NetLogo commands, constraints (e.g., forbidden actions), and strategic goals. The specific code or pseudocode to be mutated is dynamically inserted into this base prompt. Full details and examples of the prompts are available in Appendix G.

The experiments explore three main variations of how the LLM is prompted and utilized for mutation:

5.1 Few-Shot Prompting

We investigate the impact of providing examples within the prompt for directing NetLogo code mutation.

5.1.1 Zero-Shot. The LLM receives the base prompt containing instructions and the parent agent's code but no examples of desired mutations.

5.1.2 One-Shot. In addition to the base prompt and parent code, the LLM receives one example pair showing an input code snippet and a corresponding desired mutated output with a brief explanation.

5.1.3 Two-Shot. Adding onto the previous one-shot example, we add another distinct example-explanation pair to further guide the LLM.

5.2 Code-Based Mutation with Comment Generation

This set of experiments lies on top of the zero-shot, one-shot, and two-shot setups with an added instruction in the prompt explicitly asking the LLM to generate NetLogo comments explaining the strategy and logic of the mutated code it produces. This experiment tests whether prompting for self-explanation improves the coherence and strategy of the mutations.

5.3 Pseudocode-Based Mutation

This approach uses a two-step LLM process operating at a higher level of abstraction.

- **Step 1 (Pseudocode Mutation):** A dedicated prompt instructs the LLM to mutate the *pseudocode* representation of the parent agent's rule. This prompt contains similar contextual information as the direct code mutation prompt but focuses on evolving the high-level logic described in pseudocode. We test zero-shot, one-shot, and two-shot variations for this step as well, providing examples of pseudocode mutations where applicable.
- **Step 2 (Code Translation):** A second, distinct prompt instructs the LLM to translate the *newly mutated pseudocode* from Step 1 into executable NetLogo code. This translation prompt also includes context, constraints, and valid syntax, focusing on faithful implementation. Again, zero-shot, one-shot, and two-shot variations are tested, providing examples of pseudocode-to-code translations.

For all experiments, we used the simulation parameters and LLM configuration detailed in Table 1. These parameter values were empirically tuned to ensure consistent results while maintaining low computational overhead. The chosen LLM was selected for its versatility in handling both code and natural language, as well as its high rate limit for API calls and relatively low operational cost. The initial code rule provided to all agents at the start of evolution

was simply `rt random 20 lt random 20 fd 1`. This initial rule instructs the agent to turn right a random amount and then turn left a random amount and move forward one step. The corresponding initial pseudocode rule (used in pseudocode evolution experiments) was `Take left turn randomly within 0-20 degrees, then take right turn randomly within 0-20 degrees and move forward 1`. We conducted 5 trial repetitions with different seeds for each experimental condition across the three environments described in Section 4 (Collection Simple, Collection Hazardous, Collection Resources).

6 Results & Discussion

In this section, we discuss the results of our experiments. In Figure 3, we compare our three mutation approaches: mutation with code representations, mutation with comment generation, and mutation with pseudocode representations on the mean fitness of the agents. Each mutation scheme was tested with zero-shot, one-shot, and two-shot prompting for each of the three environments.

6.1 Few-Shot Prompting

The effects of few-shot prompting vary across the different environments. In all cases, the difference between one-shot and two-shot prompting are negligible. In the Collection Simple and Collection Resources environments, few-shot prompting leads to faster convergence to higher fitness values compared to zero-shot prompting when using direct code mutation. However, for mutation with comments, few-shot prompting strategies have minimal impact. In the Collection Hazardous environment, zero-shot prompting performs significantly worse than few-shot prompting under direct code mutation, and slightly worse under the commented code mutation scheme. For pseudocode mutation, zero-shot prompting underperforms relative to multi-shot prompting in both the Collection Simple and Collection Hazardous environments. Notably, in the Collection Hazardous setting, zero-shot pseudocode prompting leads to a degenerate strategy in which agents avoid both food and poison, resulting in a fitness of zero. In contrast, few-shot prompting has a negligible effect in the Collection Resources environment. In summary, while two-shot prompting offers no observable advantage over one-shot prompting, providing at least one example in the prompt generally improves performance across most conditions.

6.2 Commented Code and Pseudocode Mutation

Code generation with comments consistently produced the highest fitness scores across all three environments, while the pseudocode-based mutation strategy demonstrated the poorest performance. Although pseudocode representations did not yield high-performing agents, the observed benefits of including comments suggest that alternative ways of leveraging pseudocode may still offer potential for guiding evolution. In Appendix D, we provide additional results which demonstrate that using a different LLM results in a similar performance difference among the code and pseudocode-based mutation schemes. The primary limitation of the pseudocode-based approach stemmed from the difficulty of accurately translating high-level instructions into executable NetLogo code. Pseudocode often included abstract directives such as "`If food is detected to the left`"; however, the LLM frequently failed to map these concepts correctly to relevant NetLogo code using the agent's observation vector. For example, instead of producing valid checks to see if the observation vector detects food in the left cone (e.g., `if item 0 input > 0`), the LLM often hallucinated non-existent variables, such as `if-food-left`. Representative examples of both generated code and pseudocode are provided in Appendix F.

6.3 Interpretability of Commented Code and Pseudocode

Generating commented code and pseudocode representations offers an additional benefit of improved interpretability compared to code alone. In this section, we discuss the extent to which our approach was effective in making generated code artifacts more interpretable by incorporating comments and generating pseudocode representations. We provide illustrative examples for both commented code and pseudocode.

An examination of the highest-fitness agents across all environments revealed that the generated comments were relevant and accurately reflected the corresponding code logic, thereby improving comprehensibility. For instance, in Figure 6 (in Appendix F), the comments "`Check if there is food in front`" and "`Move towards food`" clearly clarify the intent of the line `if item 2 input != 0 [fd item 2 input]`.

Pseudocode representations similarly contributed to increased interpretability by explicitly conveying the agent behavior intended by the LLM. For example, the part of the pseudocode shown in Appendix F "`If food is detected ahead and a random number between 0 and 1 is less than 0.8, move forward 1`" is more intuitive than its NetLogo equivalent: `if (item 2 input != 0) and (random-float 1 < 0.8) [fd 1]`. However, in some instances, the intended behavior expressed in the pseudocode was not accurately translated into syntactically correct or semantically valid NetLogo code, highlighting a limitation in the translation process.

7 Limitations & Future Work

Although our methodology and investigation yield valuable insights into the use of LLMs for evolving agent behaviors in multi-agent systems, several limitations remain that point to promising directions for future work:

(1) Further research is necessary to fully assess the potential of leveraging high-level abstractions for guiding the evolution of code artifacts.
(2) Our approach is highly sensitive to the design of prompts. More systematic prompt optimization is required to improve the consistency and quality of generated mutations.
(3) Further experimentation is needed to investigate the shortcomings of the pseudocode-to-code translation process. Additionally, comparative analyses across different LLMs and experimental parameters—such as LLM temperature, population sizes, and the number of generations—are necessary to better understand their effects on performance.
(4) A broader set of multi-agent environments is needed to test the generality of our approach, particularly those involving more complex dynamics that might enable the emergence of diverse competitive or cooperative behaviors.

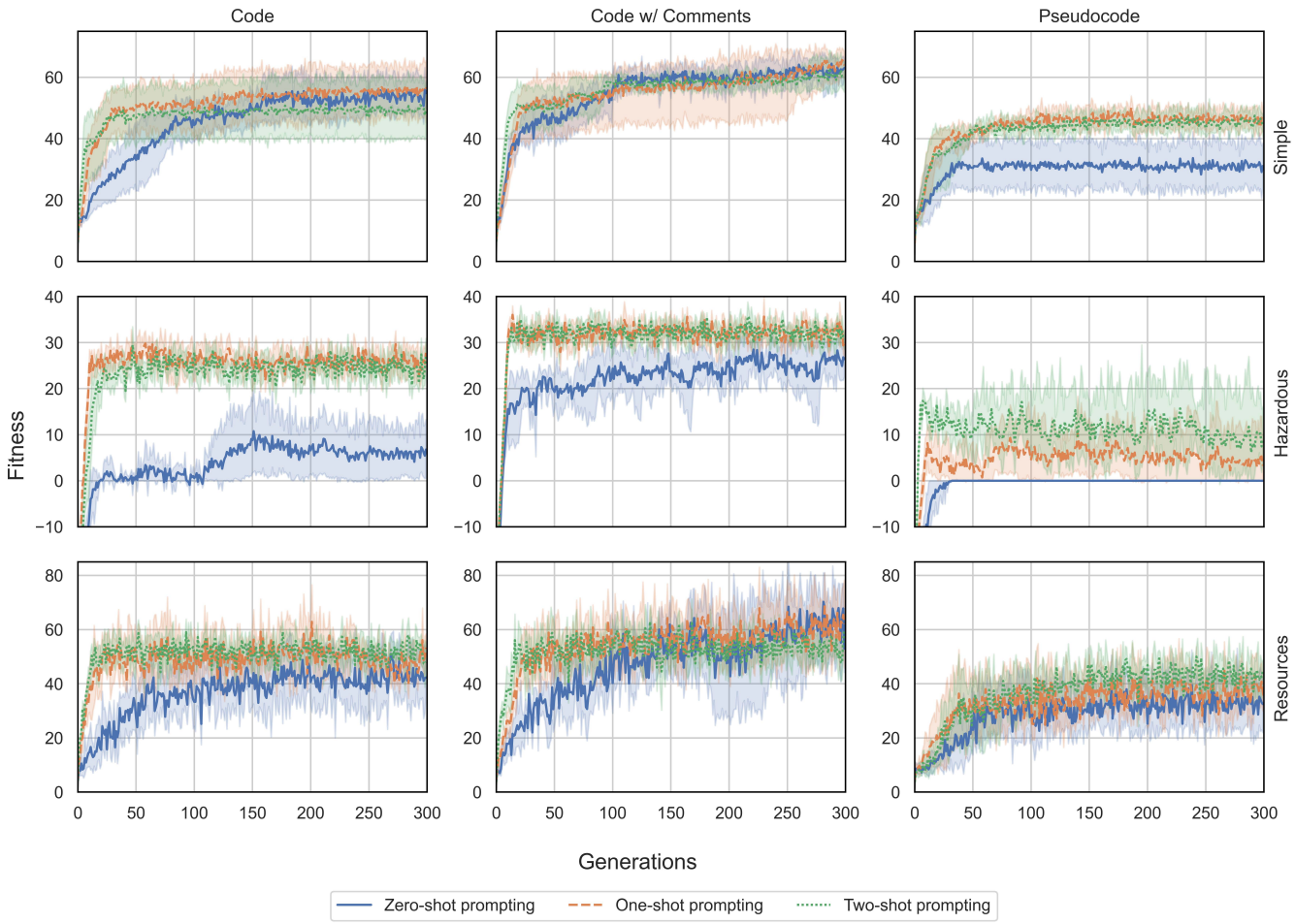


Figure 3: Mean fitness of agents over generations. The columns correspond to using code generation directly, code generation along with comments, and using pseudocode representations respectively. The rows correspond to the collection simple hazardous, and resources environments respectively.

(5) Our current framework evaluates and mutates agents individually. However, in many multi-agent contexts, performance depends on collective behavior. Future work should explore evaluating and evolving agents as teams, though this will likely incur greater computational cost.

8 Conclusion

In this study, we investigated the integration of LLM-driven mutation operators within genetic programming frameworks to evolve agent behaviors in NetLogo-based multi-agent simulations. Our experiments systematically evaluated different prompting strategies, zero-shot, one-shot, and two-shot prompting, on the fitness of agents across various multi-agent environments focused on resource collection tasks. The results demonstrated that while different prompting strategies can have some impact on the fitness of agents, it is not always the case.

Additionally, we introduced and explored a novel methodology involving code mutation using pseudocode representations, which were subsequently translated into executable NetLogo code by leveraging another LLM-mediated step. Our findings revealed that pseudocode-based mutations perform worse compared to direct code-based mutation. However, mutation with comment generation increases the performance of the agents.

Overall, our research provides evidence supporting the feasibility and effectiveness of using LLMs in genetic programming for multi-agent systems. These findings open exciting avenues for future research, particularly in further refining prompting techniques, exploring additional levels of abstraction in representation evolution, and extending applications to other complex domains requiring transparent, interpretable, and sophisticated agent behaviors.

Acknowledgments

This work was partially supported by the NSF Grant POSE: Phase II: Cultivating Modeling Literacy and Practice through a NetLogo Open Source Ecosystem. We thank Joel Lehman for insightful discussions and feedback.

References

[1] R. Axelrod. 1987. The evolution of strategies in the Iterated Prisoner's Dilemma. *Genetic Algorithms and Simulated Annealing* (1987), 32–41. https://ci.nii.ac.jp/naid/10000082922/en/

[2] Wolfgang Banzhaf, Guillaume Beslon, Steffen Christensen, James A. Foster, François Képès, Virginie Lefort, Julian F. Miller, Miroslav Radman, and Jeremy J. Ramsden. 2006. Guidelines: From artificial evolution to computational evolution: A research agenda. *Nature Reviews Genetics* 7 (2006). Issue 9. doi:10.1038/nrg1921

[3] Eric Bonabeau. 2002. Agent-based modeling: Methods and techniques for simulating human systems. *Proceedings of the National Academy of Sciences* 99, suppl 3 (2002), 7280–7287. doi:10.1073/pnas.082080899 arXiv:"https://www.pnas.org/content/99/suppl_3/7280.full.pdf"

[4] Herbie Bradley, Honglu Fan, Theodoros Galanos, Ryan Zhou, Daniel Scott, and Joel Lehman. 2024. *The OpenELM Library: Leveraging Progress in Language Models for Novel Evolutionary Algorithms.* Springer Nature Singapore, Singapore, 177–201. doi:10.1007/978-981-99-8413-8_10

[5] Tom Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared D Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, Sandhini Agarwal, Ariel Herbert-Voss, Gretchen Krueger, Tom Henighan, Rewon Child, Aditya Ramesh, Daniel Ziegler, Jeffrey Wu, Clemens Winter, Chris Hesse, Mark Chen, Eric Sigler, Mateusz Litwin, Scott Gray, Benjamin Chess, Jack Clark, Christopher Berner, Sam McCandlish, Alec Radford, Ilya Sutskever, and Dario Amodei. 2020. Language Models are Few-Shot Learners. In *Advances in Neural Information Processing Systems*, H. Larochelle, M. Ranzato, R. Hadsell, M.F. Balcan, and H. Lin (Eds.), Vol. 33. Curran Associates, Inc., 1877–1901. https://proceedings.neurips.cc/paper_files/paper/2020/file/1457c0d6bfcb4967418bfb8ac142f64a-Paper.pdf

[6] Leonardo Lucio Custode, Fabio Caraffini, Anil Yaman, and Giovanni Iacca. 2024. An investigation on the use of Large Language Models for hyperparameter tuning in Evolutionary Algorithms. In *Proceedings of the Genetic and Evolutionary Computation Conference Companion* (Melbourne, VIC, Australia) *(GECCO '24 Companion)*. Association for Computing Machinery, New York, NY, USA, 1838–1845. doi:10.1145/3638530.3664163

[7] Leonardo Lucio Custode, Chiara Camilla Migliore Rambaldi, Marco Roveri, and Giovanni Iacca. 2024. Comparing Large Language Models and Grammatical Evolution for Code Generation. In *Proceedings of the Genetic and Evolutionary Computation Conference Companion* (Melbourne, VIC, Australia) *(GECCO '24 Companion)*. Association for Computing Machinery, New York, NY, USA, 1830–1837. doi:10.1145/3638530.3664162

[8] Kerstin Dautenhahn. 2007. Socially intelligent robots: dimensions of human and robot interaction. *Philosophical Transactions of the Royal Society B: Biological Sciences* 362, 1480 (2007), 679–704. doi:10.1098/rstb.2006.2004 arXiv:https://royalsocietypublishing.org/doi/pdf/10.1098/rstb.2006.2004

[9] Joshua M. Epstein. 1999. Agent-based computational models and generative social science. *Complexity* 4, 5 (1999), 41–60. doi:10.1002/(SICI)1099-0526(199905/06)4:5<41::AID-CPLX9>3.0.CO;2-F arXiv:https://onlinelibrary.wiley.com/doi/pdf/10.1002/

[10] Joshua M. Epstein. 2007. *Generative Social Science.* Princeton University Press, Princeton. doi:doi:10.1515/9781400842872

[11] Chrisantha Fernando, Dylan Sunil Banarse, Henryk Michalewski, Simon Osindero, and Tim Rocktäschel. 2024. Promptbreeder: Self-Referential Self-Improvement via Prompt Evolution. In *Proceedings of the 41st International Conference on Machine Learning (Proceedings of Machine Learning Research, Vol. 235)*, Ruslan Salakhutdinov, Zico Kolter, Katherine Heller, Adrian Weller, Nuria Oliver, Jonathan Scarlett, and Felix Berkenkamp (Eds.). PMLR, 13481–13544. https://proceedings.mlr.press/v235/fernando24a.html

[12] Chen Gao, Xiaochong Lan, Nian Li, Yuan Yuan, Jingtao Ding, Zhilun Zhou, Fengli Xu, and Yong Li. 2024. Large language models empowered agent-based modeling and simulation: a survey and perspectives. *Humanities and Social Sciences Communications* 11, 1 (27 Sep 2024), 1259. doi:10.1057/s41599-024-03611-3

[13] Volker Grimm and Steven F. Railsback. 2005. *Individual-based Modeling and Ecology.* Princeton University Press, Princeton. doi:doi:10.1515/9781400850624

[14] Taicheng Guo, Xiuying Chen, Yaqi Wang, Ruidi Chang, Shichao Pei, Nitesh V. Chawla, Olaf Wiest, and Xiangliang Zhang. 2024. Large language model based multi-agents: a survey of progress and challenges. In *Proceedings of the Thirty-Third International Joint Conference on Artificial Intelligence* (Jeju, Korea) *(IJCAI '24)*. Article 890, 10 pages. doi:10.24963/ijcai.2024/890

[15] Can Gurkan, Leif Rasmussen, and Uri Wilensky. 2019. Effects of Visual Sensory Range on the Emergence of Cognition in Early Terrestrial Vertebrates: An Agent-Based Modeling Approach *(ALIFE 2020: The 2020 Conference on Artificial Life, Vol. ALIFE 2019: The 2019 Conference on Artificial Life)*. 475–476. doi:10.1162/isal_a_00206 arXiv:https://direct.mit.edu/isal/proceedings-pdf/isal2019/31/475/1903463/isal_00206.pdf

[16] Desta Haileselassie Hagos, Rick Battle, and Danda B. Rawat. 2024. Recent Advances in Generative AI and Large Language Models: Current Status, Challenges, and Perspectives. *IEEE Transactions on Artificial Intelligence* 5, 12 (2024), 5873–5893. doi:10.1109/TAI.2024.3444742

[17] Bryan Head and Uri Wilensky. 2018. *Agent Cognition Through Micro-simulations: Adaptive and Tunable Intelligence with NetLogo LevelSpace: Proceedings of the Ninth International Conference on Complex Systems.* 71–81. doi:10.1007/978-3-319-96661-8_7

[18] Erik Hemberg, Steven Jorgensen, and Una-May O'Reilly. 2025. *Survey of Genetic Programming and Large Language Models.* Springer Nature Singapore, Singapore, 67–86. doi:10.1007/978-981-96-0077-9_4

[19] Erik Hemberg, Stephen Moskal, and Una-May O'Reilly. 2024. Evolving code with a large language model. *Genetic Programming and Evolvable Machines* 25, 2 (12 Sep 2024), 21. doi:10.1007/s10710-024-09494-2

[20] Shengran Hu, Cong Lu, and Jeff Clune. 2025. Automated Design of Agentic Systems. In *The Thirteenth International Conference on Learning Representations.* https://openreview.net/forum?id=t9U3LW7JVX

[21] Juyong Jiang, Fan Wang, Jiasi Shen, Sungju Kim, and Sunghun Kim. 2024. A Survey on Large Language Models for Code Generation. arXiv:2406.00515 [cs.CL] https://arxiv.org/abs/2406.00515

[22] Hamdi Kavak, Jose J. Padilla, Christopher J. Lynch, and Saikou Y. Diallo. 2018. Big Data, Agents, and Machine Learning: Towards a Data-Driven Agent-Based Modeling Approach. In *Proceedings of the Annual Simulation Symposium* (Baltimore, Maryland) *(ANSS '18)*. Society for Computer Simulation International, San Diego, CA, USA, Article 12, 12 pages.

[23] John R. Koza. 1992. *Genetic Programming: On the Programming of Computers by Means of Natural Selection.* MIT Press, Cambridge, MA, USA. http://mitpress.mit.edu/books/genetic-programming

[24] W. B. Langdon and W. Banzhaf. 2008. Repeated patterns in genetic programming. *Natural Computing* 7, 4 (01 Dec 2008), 589–613. doi:10.1007/s11047-007-9038-8

[25] Robert Lange, Yingtao Tian, and Yujin Tang. 2024. Large Language Models As Evolution Strategies. In *Proceedings of the Genetic and Evolutionary Computation Conference Companion* (Melbourne, VIC, Australia) *(GECCO '24 Companion)*. Association for Computing Machinery, New York, NY, USA, 579–582. doi:10.1145/3638530.3654238

[26] Joel Lehman, Jonathan Gordon, Shawn Jain, Kamal Ndousse, Cathy Yeh, and Kenneth O. Stanley. 2024. *Evolution Through Large Models.* Springer Nature Singapore, Singapore, 331–366. doi:10.1007/978-981-99-3814-8_11

[27] Fei Liu, Xialiang Tong, Mingxuan Yuan, Xi Lin, Fu Luo, Zhenkun Wang, Zhichao Lu, and Qingfu Zhang. 2024. Evolution of heuristics: towards efficient automatic algorithm design using large language model. In *Proceedings of the 41st International Conference on Machine Learning* (Vienna, Austria) *(ICML'24)*. JMLR.org, Article 1304, 23 pages.

[28] Chris Lu, Samuel Holt, Claudio Fanconi, Alex J. Chan, Jakob Foerster, Mihaela van der Schaar, and Robert Tjarko Lange. 2024. Discovering Preference Optimization Algorithms with and for Large Language Models. In *Advances in Neural Information Processing Systems*, A. Globerson, L. Mackey, D. Belgrave, A. Fan, U. Paquet, J. Tomczak, and C. Zhang (Eds.), Vol. 37. Curran Associates, Inc., 86528–86573. https://proceedings.neurips.cc/paper_files/paper/2024/file/9d88b87b31986f8293bb0067a841579e-Paper-Conference.pdf

[29] Chris Lu, Cong Lu, Robert Tjarko Lange, Jakob Foerster, Jeff Clune, and David Ha. 2024. The AI Scientist: Towards Fully Automated Open-Ended Scientific Discovery. arXiv:2408.06292 [cs.AI] https://arxiv.org/abs/2408.06292

[30] Yecheng Jason Ma, William Liang, Guanzhi Wang, De-An Huang, Osbert Bastani, Dinesh Jayaraman, Yuke Zhu, Linxi Fan, and Anima Anandkumar. 2024. Eureka: Human-Level Reward Design via Coding Large Language Models. In *The Twelfth International Conference on Learning Representations.* https://openreview.net/forum?id=IEduRUO55F

[31] Elliot Meyerson, Mark J. Nelson, Herbie Bradley, Adam Gaier, Arash Moradi, Amy K. Hoover, and Joel Lehman. 2024. Language Model Crossover: Variation through Few-Shot Prompting. *ACM Trans. Evol. Learn. Optim.* 4, 4, Article 27 (Nov. 2024), 40 pages. doi:10.1145/3694791

[32] Brad L Miller, David E Goldberg, et al. 1995. Genetic algorithms, tournament selection, and the effects of noise. *Complex systems* 9, 3 (1995), 193–212.

[33] Julian F. Miller. 2011. *Cartesian Genetic Programming.* Springer Berlin Heidelberg, Berlin, Heidelberg, 17–34. doi:10.1007/978-3-642-17310-3_2

[34] Melanie Mitchell. 1998. *An Introduction to Genetic Algorithms.* MIT Press, Cambridge, MA, USA.

[35] Jean-Baptiste Mouret and Jeff Clune. 2015. Illuminating search spaces by mapping elites. arXiv:1504.04909 [cs.AI]

[36] W. James Murdoch, Chandan Singh, Karl Kumbier, Reza Abbasi-Asl, and Bin Yu. 2019. Definitions, methods, and applications in interpretable machine learning. *Proceedings of the National Academy of Sciences* 116, 44 (2019), 22071–22080. doi:10.1073/pnas.1900654116 arXiv:https://www.pnas.org/doi/pdf/10.1073/pnas.1900654116

[37] Thanh Thi Nguyen, Ngoc Duy Nguyen, and Saeid Nahavandi. 2020. Deep Reinforcement Learning for Multiagent Systems: A Review of Challenges, Solutions, and Applications. *IEEE Transactions on Cybernetics* 50, 9 (2020), 3826–3839. doi:10.1109/TCYB.2020.2977374

[38] Michael O'Neill, Leonardo Vanneschi, Steven Gustafson, and Wolfgang Banzhaf. 2010. Open issues in genetic programming. *Genetic Programming and Evolvable Machines* 11, 3–4 (Sept. 2010), 339–363. doi:10.1007/s10710-010-9113-2
[39] Riccardo Poli, William Langdon, and Nicholas Mcphee. 2008. *A Field Guide to Genetic Programming.*
[40] Steven F Railsback and Volker Grimm. 2019. *Agent-based and individual-based modeling: a practical introduction.* Princeton university press.
[41] Bernardino Romera-Paredes, Mohammadamin Barekatain, Alexander Novikov, Matej Balog, M. Pawan Kumar, Emilien Dupont, Francisco J. R. Ruiz, Jordan S. Ellenberg, Pengming Wang, Omar Fawzi, Pushmeet Kohli, and Alhussein Fawzi. 2024. Mathematical discoveries from program search with large language models. *Nature* 625, 7995 (01 Jan 2024), 468–475. doi:10.1038/s41586-023-06924-6
[42] Xingyou Song, Yingtao Tian, Robert Tjarko Lange, Chansoo Lee, Yujin Tang, and Yutian Chen. 2024. Position: leverage foundational models for black-box optimization. In *Proceedings of the 41st International Conference on Machine Learning* (Vienna, Austria) *(ICML'24)*. JMLR.org, Article 1878, 13 pages.
[43] Wouter Vermeer, Can Gurkan, Arthur Hjorth, Nanette Benbow, Brian M. Mustanski, David Kern, C. Hendricks Brown, and Uri Wilensky. 2022. Agent-based model projections for reducing HIV infection among MSM: Prevention and care pathways to end the HIV epidemic in Chicago, Illinois. *PLOS ONE* 17, 10 (10 2022), 1–27. doi:10.1371/journal.pone.0274288
[44] Aymeric Vie. 2021. Qualities, challenges and future of genetic algorithms: a literature review. arXiv:2011.05277 [cs.NE]
[45] Darrell Whitley. 1998. A Genetic Algorithm Tutorial. *Statistics and Computing* 4 (10 1998). doi:10.1007/BF00175354
[46] Uri Wilensky. 1999. *NetLogo.* http://ccl.northwestern.edu/netlogo/. Northwestern University, Evanston, IL. http://ccl.northwestern.edu/netlogo/
[47] Uri Wilensky and William Rand. 2015. *An Introduction to Agent-Based Modeling: Modeling Natural, Social, and Engineered Complex Systems with NetLogo.* The MIT Press.
[48] Xingyu Wu, Sheng-Hao Wu, Jibin Wu, Liang Feng, and Kay Chen Tan. 2024. Evolutionary Computation in the Era of Large Language Model: Survey and Roadmap. *IEEE Transactions on Evolutionary Computation* (2024), 1–1. doi:10.1109/TEVC.2024.3506731
[49] Ryan Zhou, Jaume Bacardit, Alexander E. I. Brownlee, Stefano Cagnoni, Martin Fyvie, Giovanni Iacca, John McCall, Niki van Stein, David J. Walker, and Ting Hu. 2024. Evolutionary Computation and Explainable AI: A Roadmap to Understandable Intelligent Systems. *IEEE Transactions on Evolutionary Computation* (2024), 1–1. doi:10.1109/TEVC.2024.3476443

A Background

In this section, we briefly describe the relevant concepts of large language models, genetic programming, and multi-agent systems.

A.1 Large Language Models

Large Language Models (LLMs) are typically transformer-based deep neural network architectures comprised of billions of parameters, designed to process and generate coherent and contextually relevant text through auto-regressive training on vast, internet-scale corpora of data [5]. Transformer models leverage attention mechanisms, notably self-attention, which allow the model to dynamically weigh the importance of input elements relative to one another. This mechanism enables the model to effectively capture complex contextual relationships within textual data. Training of LLMs typically involves self-supervised learning, specifically through an auto-regressive sequence completion task, where models are optimized to probabilistically predict subsequent words or tokens when given a preceding context. This training process enables the acquisition of robust representations of syntactic and semantic relationships embedded within extensive text corpora. Commercial uses of LLMs typically involve an interface that can be queried using an input sequence of text called a prompt to generate an output sequence of text that is intended to be the optimal response to the prompt. Within the context of genetic programming, the generative power and flexibility of LLMs make them ideal candidates for evolving programmatic solutions and agent behaviors [7, 18].

A.2 Genetic Programming

Genetic Programming (GP) is an evolutionary algorithm designed to produce computer programs using the principles of biological evolution, specifically the concepts of variation, selection, and inheritance [2, 23]. GP uses various evolutionary operators on a population of candidate solutions to iteratively improve them. The evolutionary process begins with an initialization operator that configures a population of candidate programs, which are evaluated based on their performance or fitness for accomplishing a given task using an evaluation operator. Then a selection operator determines high-performing programs to serve as selected candidates, referred to as parents, that will undergo subsequent variation based on the performance of each candidate. The selected candidates or parents produce offspring using variation operators that typically consist of crossover and mutation operators that combine and alter aspects of parent solutions, respectively. A replacement operator subsequently replaces a portion of the population with the newly produced offspring solutions that are integrated into the next generation. Each of these evolutionary operators can benefit from the complementary use of LLMs [19]. In this study, we specifically focus on using LLMs as mutation operators. Given the versatility GP has demonstrated across a wide range of complex problem domains, including symbolic regression, automated design, control strategies, and multi-agent systems [24, 33, 38, 39, 44], the LLM-GP approach may hold significant potential to expand the capabilities of evolutionary computation beyond current applications.

A.3 Multi-Agent Systems & Agent-Based Modeling

Multi-agent systems (MAS) refer to computational frameworks composed of multiple autonomous agents that interact with each other, typically characterized by localized sensing, decision-making, and communication capabilities [14, 37]. Due to their suitability for modeling complex interactions and emergent phenomena, MAS have been extensively studied and successfully applied across diverse scientific domains, including ecology, economics, epidemiology, and robotics [1, 8, 10, 13, 15, 17, 22, 40, 43]. A related methodological approach, Agent-Based Modeling (ABM) is a form of computational modeling that leverages multi-agent systems as computational tools to simulate and analyze phenomena modeled in terms of agents and their interactions [47]. Defining appropriate behavioral rules for agents in these systems remains a fundamental challenge, as even relatively simple agent behaviors can give rise to complex, emergent patterns that are difficult to anticipate or analytically predict [3, 9]. In this context, the LLM-GP approach holds promise for evolving complex yet interpretable behavioral rules tailored to individual agents with distinct capabilities, thereby enabling nuanced interactions among heterogeneous agents. As a result, the LLM-GP framework has the potential to facilitate the effective simulation and analysis of intricate real-world scenarios within multi-agent systems.

B Additional Algorithms

In this section, we provide the algorithms for code and pseudocode mutation operators.

Algorithm 2 LLM-Driven Mutation with Code Representations

Given agent with code rule r, mutation prompt ρ_m, retry prompt ρ_r, max number of retries z, code verifier V, and LLM
$r' \leftarrow \text{LLM}(\rho_m + r)$
retry $\leftarrow 0$
while $V(r') \neq$ **true and** retry $< z$ **do**
  $w \leftarrow$ get error message from $V(r')$
  $r' \leftarrow \text{LLM}(\rho_r + w + r')$
  retry $\leftarrow$ retry +1
end while
if $V(r') =$ **true then**
  return r'
else
  return r
end if

Algorithm 3 LLM-Driven Mutation with Pseudocode Representations

Given agent with pseudocode rule r, code rule c, pseudocode mutation prompt ρ_m, code translation prompt ρ_c, retry prompt ρ_r, max number of retries z, code verifier V, and LLM
$r' \leftarrow \text{LLM}(\rho_m + r)$
$c' \leftarrow \text{LLM}(\rho_c + r')$
retry $\leftarrow 0$
while $V(c') \neq$ **true and** retry $< z$ **do**
  $w \leftarrow$ get error message from $V(c')$
  $c' \leftarrow \text{LLM}(\rho_r + r' + w + c')$
  retry $\leftarrow$ retry +1
end while
if $V(c) =$ **true then**
  return r', c'
else
  return r, c
end if

C Agent Observation Illustration

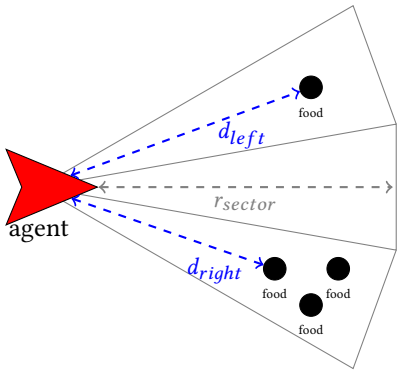


Figure 4: An example of encoding the information the agent can see in its field of vision. In this case, the observation vector would be $v = [d_{\text{left}}, 0, d_{\text{right}}]$.

D Additional Results

Here we provide the results of an additional experiment where we used Claude 3.5 Haiku on the Collection Simple environment with zero-shot prompting using the code, code with comments and pseudocode-based mutation schemes to assess the impact of a different LLM on our findings.

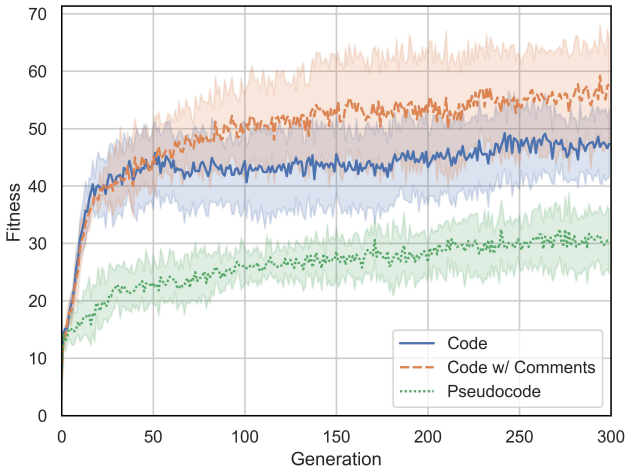


Figure 5: Mean fitness of agents over generations in the Collection Simple environment for code, code with comments and pseudocode-based mutation with zero-shot prompting.

E Cost of Experiments

On average a single experiment cost around $10. The total cost of all experiments was roughly $200.

F Generated Code and Pseudocode Examples

```
; Check if there is food in front
if item 2 input != 0 [
  ; Move forward towards food
  fd item 2 input
]

; If no food in front, compare left and right cones
if item 2 input = 0 [
  ; If left cone has food and it's closer than right cone
  if item 0 input > item 1 input [
    ; Turn left and move towards food
    lt 15
    fd item 0 input
  ]

  ; If right cone has food and it's closer than left cone
  if item 1 input > item 0 input [
    ; Turn right and move towards food
    rt 15
    fd item 1 input
  ]

  ; If both cones have food at the same distance
  if item 0 input = item 1 input [
    ; Move forward if there is food
    if item 0 input != 0 [
      fd item 0 input
    ]
    ; Randomly turn left or right
    if random 2 = 0 [
      lt 15
    ]
    if random 2 = 1 [
      rt 15
    ]
  ]

  ; If no food in either cone
  if item 0 input = 0 [
    if item 1 input = 0 [
      ; Randomly turn and move forward
      ifelse random 2 = 0 [
        rt 45
      ] [
        lt 45
      ]
      fd 5
    ]
  ]
]
```

Figure 6: An LLM generated agent code snippet.

An LLM generated pseudocode snippet.

```
If food is detected ahead and a random number between
0 and 1 is less than 0.8,
   move forward 1
Else if food is detected to the left,
   turn left by 10 degrees and move forward 1
Else if food is detected to the right,
   turn right by 10 degrees and move forward 1
Else if a random number between 0 and 1 is less than
0.3,
   turn left by a random angle between 5-15 degrees and
move forward 1
Else if a random number between 0 and 1 is less than
0.6,
    turn right by a random angle between 5-15 degrees
and move forward 1
Else
    turn left by a random angle between 10-20 degrees
and move forward 1
If a random number between 0 and 1 is less than 0.1,
   turn right by 90 degrees and move forward 1
```

G Prompts

This appendix details the structure and variations of prompts used to guide the LLM in generating NetLogo code or pseudocode for agent movement. The placeholder {} in the prompt templates represents the location where the parent agent's current code or pseudocode is inserted dynamically during the evolutionary run.

G.1 Direct Code Mutation Prompts

These prompts instruct the LLM to directly modify existing NetLogo code.

G.1.1 Base Prompt Structure (Zero-Shot, No Comments). The core structure provided context, constraints, valid commands, and goals. The LLM was asked to improve the input NetLogo code (inserted at {}) and return only the runnable code block.

````
You are an expert NetLogo coder.
You are trying to improve the code of a given turtle agent that is trying to collect as much food as possible.
Improve the given agent movement code following these precise specifications:

Here is the current code of the turtle agent:

```
{}
```

INPUT CONTEXT:
- You have access to a variable called input
- Input is a NetLogo list that contains three values representing distances to food
in three cone regions of 20 degrees each
- The first item in the input list is the distance to the nearest food in the left cone,
the second is the right cone, and the third is the front cone
- Each value encodes the distance to nearest food source where a value of 0 indicates no food
- Non-zero lower values indicate closer food
- Use the information in this variable to inform movement strategy
- Remember that you only have access to the variable named input and no other variables

SIMULATION ENVIRONMENT:
- The turtle agent is in a food collection simulation
- The goal is to collect as much food as possible
- The turtle agent can detect food in three cone regions encoded in the input list
- The food sources are randomly distributed in the environment

CONSTRAINTS:
1. Do not include code to kill or control any other agents
2. Do not include code to interact with the environment
3. Do not include code to change the environment
4. Do not include code to create new agents
5. Do not include code to create new food sources
6. Do not include code to change the rules of the simulation
7. Follow NetLogo syntax and constraints
8. Do not use any undefined variables or commands besides the input variable
9. Focus on movement strategies based on the input variable

VALID COMMANDS AND SYNTAX:
 - Use only these movement commands: fd, forward, rt, right, lt, left, bk, back
 - Use only these reporters: random, random-float, sin, cos, item, xcor, ycor, heading
 - The syntax of the if primitive is as follows: if boolean [ commands ]
 - The syntax of the ifelse primitive is as follows: ifelse boolean [ commands1 ] [ commands2 ]
 - An ifelse block that contains multiple boolean conditions must be enclosed in parentheses as follows:
 (ifelse boolean1 [ commands1 ] boolean2 [ commands2 ] ... [ elsecommands ])

ABSOLUTELY FORBIDDEN:
 - DO NOT use "of" primitives - will cause errors
 - DO NOT use "ask", "with", "turtles", or "patches"
 - DO NOT use any undefined variables
 - DO NOT use while loops or recursive constructs

ERROR PREVENTION:
 - Ensure all bracket pairs match
 - Make sure every movement command has a parameter
 - Keep values within reasonable ranges (-1000 to 1000)
 - Ensure at least one movement command is included
 - There is no such thing as an `else` statement in NetLogo

STRATEGIC GOALS:
1. Balance exploration and food-seeking behavior
2. Respond to sensor readings intelligently
3. Combine different movement patterns
4. Be creative in your movement strategy

The code must be runnable in NetLogo in the context of a turtle.
Do not write any procedures and assume that the code will be run in an ask turtles block.
Return ONLY the changed NetLogo code.
Do not include any explanations or outside the code block.

```
[Your changed NetLogo code goes here]
```
````

G.1.2 Few-Shot Prompting Variations. To provide examples, the following text blocks were inserted into the base prompt structure immediately before the final instructions ("The code must be runnable...").

One-Shot Example Addition.

````
EXAMPLES OF VALID CODE GENERATION:
Current Code: ```fd 1 rt random 45 fd 2 lt 30```
Changed Code: ```ifelse item 0 input != 0 [rt 15 fd 0.5] [rt random 30 lt random 30 fd 5]```
Why: This code uses the information in the input list to turn right and go forward a little to reach food if the first element of input list contains a non-zero value, else moves forward in big steps and turns randomly to explore
````

Two-Shot Example Addition. The one-shot example was included, followed immediately by this second example within the same `EXAMPLES OF VALID CODE GENERATION:`

````
Current Code:
```
ifelse item 0 input != 0 [
  lt 5
  fd 0.2
] [
  ifelse item 1 input != 0 [
    rt 5
    fd 0.2
  ] [
    ifelse item 2 input != 0 [
      fd 0.2
    ] [
      ifelse random 100 < 50 [
        fd 2
        rt random-float 45
      ] [
        rt random-float 30
        fd 5
      ]
    ]
  ]
]
```


Changed Code:
```
ifelse item 0 input < item 1 input [
  ifelse item 0 input != 0 [
    lt 15
    fd 0.5
  ] [
    ifelse item 1 input != 0 [
      rt 15
      fd 0.5
    ] [
      ifelse item 2 input > 0 [
        fd 1
      ] [
        rt random 30
        lt random 30
        fd 5
      ]
    ]
  ]
] [
  ifelse item 1 input != 0 [
    ifelse item 1 input < item 0 input [
      rt 15
      fd 0.5
    ] [
      ifelse item 2 input > 0 [
        lt 15
        fd 0.5
      ] [ fd 1 ]
    ]
  ] [
    ifelse item 2 input > 0 [
      lt 15
      fd 1
    ] [
      rt random 30
      lt random 30
      fd 5
    ]
  ]
]
```

Why: This code uses the information in the input list to make decisions based on the distances to food in different directions.
It checks the distances in the left, front, and right cones and adjusts the movement strategy accordingly.
````

G.1.3 Comment Generation Variation. For experiments requiring comments, the final instruction block in the base prompt (or the few-shot variations) starting with "The code must be runnable..." was appended with:

```
Detail your strategy in NetLogo code comments (;;) before you generate the implementation.
Include comments throughout the code to explain your strategy.
```

G.2 Pseudocode Mutation and Translation Prompts

These prompts were used for the two-step pseudocode evolution approach.

G.2.1 Pseudocode Mutation Prompt (Zero-Shot). This prompt asks the LLM to evolve the input pseudocode (inserted at {}).

```
You are an expert NetLogo pseudocode creator specializing
in complex turtle agent movement.
Your are trying to improve the given pseudocode of a
given turtle
agent that is trying to collect as much food as possible.

Here is the current pseudocode of the turtle agent:

```
{}
```

Improve the given agent movement pseudocode following
these precise specifications:

INPUT CONTEXT:
- The agent has ONLY access to a variable called input
- When writing the pseudocode, you can only use the
variable named input
and no other variables
- Input is a NetLogo list that contains three values
representing distances
to food in three cone regions of 20 degrees each
- The first item in the input list is the distance to
the nearest food in the left cone,
the second is the right cone, and the third is the front
cone
- Each value encodes the distance to nearest food source
where a
value of 0 indicates no food
- Non-zero lower values indicate closer food
- Use the information in this variable to inform movement
strategy

SIMULATION ENVIRONMENT:
- The turtle agent is in a food collection simulation
- The goal is to collect as much food as possible
- The turtle agent can detect food in three cone regions
encoded in the input list
- The food sources are randomly distributed in the
environment

EVOLUTIONARY ADVANCEMENT OBJECTIVES:

1. PROGRESSIVE COMPLEXITY ENHANCEMENT:
- Build upon the existing pseudocode's core logic
- Add advanced movement concepts or conditional
behaviors
- Incorporate more sophisticated decision-making based
on food sensor inputs

2. INNOVATION GUIDELINES:
- Introduce adaptive movement that responds to changing
environments
- Create multi-stage movement sequences that balance
local and global exploration
- Develop intelligent turning behaviors that optimize
path trajectories
- Implement energy-efficient movement strategies that
minimize unnecessary actions
- Consider emergent swarm-like behaviors when multiple
agents use this rule

3. VALID MOVEMENT CONCEPTS ONLY:
- "Move forward" (will become fd or forward in NetLogo)
- "Turn right" (will become rt or right in NetLogo)
- "Turn left" (will become lt or left in NetLogo)
- "Move backward" (will become bk or back in NetLogo)
- Conditional movements based on food sensor readings
(the "input" list)

4. ABSOLUTELY FORBIDDEN CONCEPTS:
- DO NOT include any reference to "of" relationships
between agents
- DO NOT create or reference any variables that don't
exist
- DO NOT ask other agents to perform actions
- DO NOT create or kill any agents
- DO NOT change the environment environment or any
variables
- DO NOT use loops or recursive patterns

5. ALLOWED STRUCTURE:
- Do not use any variables other than "input". For
example, do not say
"if food is detected on the left" - use "if the first
item of input is
greater than 0"
- You may include "if/else" logic based on the "input"
list values
- You may combine multiple movement commands in sequence

6. FORMATTING:
- Keep the pseudocode readable and focused on movement
logic
- Use plain English descriptions of movement patterns
- Be specific about how food sensor readings influence
movement

Present your evolved pseudocode enclosed in triple
backticks.
You may include comments in the pseudocode detailing
your strategy.
Do not include any explanations outside the code block:
```

```
[Your evolved pseudocode here]
```

G.2.2 Code Translation Prompt (Zero-Shot). This prompt asks the LLM to translate the input pseudocode (inserted at {}) into NetLogo code.

```
You are an expert NetLogo programmer tasked with converting pseudocode into valid,
executable NetLogo code.
Your goal is to faithfully implement the pseudocode while ensuring
the code adheres to NetLogo syntax and execution constraints.

PSEUDOCODE TO TRANSLATE:

{}

TRANSLATION REQUIREMENTS:

1. UNDERSTANDING THE PSEUDOCODE:
- Focus on understanding the FUNCTIONALITY described in the pseudocode
- DO NOT use variable names from the pseudocode directly in your NetLogo code
- Translate conceptual descriptions into valid NetLogo syntax
- The pseudocode is a guideline for behavior, not a direct translation template

2. CONSTRAINTS:
- Do not include code to kill or control any other agents
- Do not include code to interact with the environment
- Do not include code to change the environment
- Do not include code to create new agents
- Do not include code to create new food sources
- Do not include code to change the rules of the simulation
- Follow NetLogo syntax and constraints
- Do not use any undefined variables or commands besides the input variable
- Focus on movement strategies based on the input variable

3. VALID COMMANDS AND SYNTAX:
- Use only these movement commands: fd, forward, rt, right, lt, left, bk, back
- Use only these reporters: random, random-float, sin, cos, item, xcor, ycor, heading
- The syntax of the if primitive is as follows: if boolean [ commands ]
- The syntax of the ifelse primitive is as follows: ifelse boolean [ commands1 ] [ commands2 ]
- An ifelse block that contains multiple boolean conditions must be enclosed in parentheses as follows: (ifelse boolean1 [ commands1 ] boolean2 [ commands2 ] ... [ elsecommands ])

4. COMPLEXITY IMPLEMENTATION:
- Accurately implement all described movement patterns
- Translate conditional logic to ifelse statements with proper brackets
- Implement sensor-responsive behavior using the "input" list only
- Do not use any variables other than "input" in your code
- Convert multi-stage movements into appropriate command sequences

4. ABSOLUTELY FORBIDDEN:
- DO NOT use the "of" primitive/reporter - this will cause errors
- DO NOT use any non-existent or undefined variables
- DO NOT use "ask", "with", "turtles", "patches" - these are not allowed
- DO NOT use "set", "let", or create any variables
- DO NOT include any infinite loops - avoid "while" or "loop" constructs
- DO NOT copy variable names from pseudocode and do not use any variables other than "input"

5. ALLOWED STRUCTURE:
- You may use "if/ifelse" statements with item checks on the "input" list
- For complex or nested conditions, ensure proper bracket nesting and balance
- Make sure every opening bracket '[' has a matching closing bracket ']'
- Remember "input" is the only valid variable you can reference

6. ERROR PREVENTION:
- Ensure each condition has both true and false branches in ifelse statements
- Verify that each command has a valid parameter
- Make sure bracket pairs are properly matched and nested
- Keep all numeric values between -1000 and 1000

7. ROBUST IMPLEMENTATION:
- Generate code that is resilient to edge cases
- If pseudocode mentions a variable that doesn't exist in NetLogo, translate its purpose
without using the variable name (e.g., if pseudocode contains "food ahead", use the value of the last item in the input list)
- Focus on capturing the intent and behavior, not the exact syntax
```

```
Your task is to carefully analyze the provided pseudocode
and translate it into well-formed NetLogo code that
represents the described movement strategy.
The code must be runnable in NetLogo in the context of
a turtle. Do not write any procedures and assume that
the code will be run in an ask turtles block.
Return ONLY the changed NetLogo code. Do not include
any explanations or outside the code block.

Present your generated NetLogo code enclosed in triple
backticks:

```
[Your generated NetLogo code here]
```
```

G.2.3 Few-Shot Variations for Pseudocode Prompts. For one-shot and two-shot prompting in the pseudocode mutation and code translation steps, example blocks were inserted into the respective base prompts (shown above) under a new section numbered `7. EXAMPLES:` (for the mutation prompt) or `8. EXAMPLES:` (for the translation prompt), just before the final instruction asking for the output.

One-Shot Example Additions.

- For Pseudocode Mutation Prompt:

```
7. EXAMPLES:
"If all values in the input list are 0, move
forward randomly.
Otherwise, identify the smallest value in the
input list and turn towards that direction."
```

- For Code Translation Prompt:

```
8. EXAMPLES:
If the given pseudocode says "If no food is detected,
move forward randomly.
Otherwise, identify the smallest value in the
input list
and turn towards that direction.",
you should implement this logic in NetLogo code
as follows:

```
ifelse (input = [0 0 0]) [
  lt random 20
  rt random 20
  fd 5
] [
  (ifelse min input = item 0 input [lt 20]
          min input = item 1 input [rt 20]
          [fd 3])
]
```
```

Two-Shot Example Additions. The respective one-shot examples were included, followed immediately by these second examples within the same `EXAMPLES:` section.

- For Pseudocode Mutation Prompt:

```
" If the third item of input is not zero, move
forward towards food. If the third item of input
is zero, and if first item of input is greater
than the second item, turn left and move towards
food. Otherwise, if the second item of input is
greater than the first item, turn right and move
towards food. Otherwise, move randomly."
```

- For Code Translation Prompt:

```
If the given pseudocode says "If the third item
of input is not zero, move forward towards food.
If the third item of input is zero, and if first
item of input is greater than the second item,
turn left and move towards food. Otherwise, if
the second item of input is greater than the first
item, turn right and move towards food. Otherwise,
move randomly.", you should implement this logic
in NetLogo code as follows:
```
if item 2 input != 0 [
  fd item 2 input
]
if item 2 input = 0 [
  if item 0 input > item 1 input [
    lt 15
    fd item 0 input
  ]
  if item 1 input > item 0 input [
    rt 15
    fd item 1 input
  ]
]
ifelse random 2 = 0 [
  rt 45
] [
  lt 45
]
fd 1
```
```

G.3 Retry Prompts

If the verifier detects that the code the LLM generated contains any syntax errors or undesired Netlogo code, as mentioned in Section 3.2.1 the error message generated by the verifier and the original code is sent to the LLM with a retry prompt. This prompt is structured such that the original meaning of the faulty code is preserved while the specific errors are fixed.

G.3.1 Retry Prompt without Psuedocode.

```
You are an expert NetLogo coder tasked with fixing a movement code error for a turtle agent. Your goal is to update the provided NetLogo movement code to fix the error message shown below.

Here is the current rule:
{original_code}

Here is the error message:
{error_message}

STRICT GUIDELINES FOR FIXING THE CODE:

1. VALID COMMANDS ONLY:
    - Use only these movement commands: fd, forward, rt, right, lt, left, bk, back
    - Use only these reporters: random, random-float, sin, cos, item, xcor, ycor, heading

2. ABSOLUTELY FORBIDDEN:
    - DO NOT use the "of" primitive/reporter - this will always cause errors
   - DO NOT use any non-existent or undefined variables
    - DO NOT use "ask", "with", "turtles", "patches" - these are not allowed
    - DO NOT use "set", "let", or create any variables
    - DO NOT use loops or recursion - these create infinite loops

3. ALLOWED STRUCTURE:
    - You may use "if/ifelse" statements with item checks on the "input" list
    - Basic example: ifelse item 0 input != 0 [fd 1] [rt 90 fd 2]
   - For complex or nested conditions, maintain proper bracket balance

4. FORMATTING:
    - Each command (fd/rt/lt) must be followed by a number or simple expression
    - All commands must be properly separated by spaces
    - Keep the code simple, focused only on movement
    - Ensure all brackets are properly paired and balanced

5. ERROR-SPECIFIC FIXES:
    - For "Dangerous primitives" errors: Remove ALL prohibited commands
   - For "Unclosed brackets" errors: Check and fix ALL bracket pairs
    - For "Invalid value" errors: Ensure all numeric values are valid and positive
    - For "No movement commands" errors: Include at least one movement command (fd, rt, lt)
    - For "Command needs a value" errors: Ensure every command has a parameter

Generate ONLY basic movement code that strictly avoids the error mentioned. The code must be runnable in NetLogo turtle context. Present your corrected NetLogo code enclosed in triple backticks:

```
[Your corrected NetLogo code here]
```

Do not include any explanations - the code itself should be the only output.
```

G.3.2 Retry Prompt with Psuedocode. This prompt is identical to Retry Prompt without Pseudocode with the addition of the lines:

```
**Guiding Pseudocode:**
```pseudocode
{pseudocode}
```
```