Agentic GP: A Theoretical Framework for the Development of Genetic Programming Systems via Agentic AI

Anil Kumar Saini*, Jose Guadalupe Hernandez*, Jason H. Moore

Abstract Genetic Programming (GP) systems are composed of several core components, such as solution representation, parent selection strategies, and variation operators. While many GP systems exist, they are often tailored to specific problem domains. In cases where the existing systems do not generalize to novel tasks, users must assemble new GP systems. This typically involves evaluating multiple alternatives for each element of the algorithm, and if no suitable option exists for a particular component, implementing a custom one from scratch. In this chapter, we propose a theoretical framework for automating this design process. We introduce Agentic GP, a framework in which multiple AI agents, specializing in different GP components, collaborate to build GP systems from scratch based on user-defined requirements. We first describe a general version of the framework applicable to any class of AI agents, and then focus on how that framework can be instantiated using Large Language Model (LLM) agents. We also implemented a simple prototype of the framework using an LLM agent designed to generate selection methods for a given GP system. We evaluated this prototype using a tree-based GP system on a suite of symbolic regression tasks. While it frequently generated selection methods that resulted in run-time errors, the valid methods it produced consistently performed better than random selection but worse than established benchmark methods such as lexicase and tournament selection. This indicates the need for further enhancements in this simple design of the agentic framework.

Anil Kumar Saini
Cedars-Sinai Medical Center, Los Angeles, CA, USA e-mail: anil.saini@cshs.org

Jose Guadalupe Hernandez
Cedars-Sinai Medical Center, Los Angeles, CA, USA e-mail: jose.hernandez8@cshs.org

Jason H. Moore
Cedars-Sinai Medical Center, Los Angeles, CA, USA e-mail: jason.moore@csmc.edu

*These authors contributed equally to this work.

1 Introduction

Genetic Programming (GP), a branch of evolutionary algorithms (EAs), evolves a population of candidate solutions through biologically inspired processes to solve a given problem. GP has demonstrated efficacy across a wide range of domains, including program synthesis (Sobania et al., 2023), automated machine learning (AutoML) (Banzhaf et al., 2024), and symbolic regression (La Cava et al., 2021). One of GP's core strengths is its adaptability, allowing GP systems to be customized for specific problem domains. For example, GP evolves computer programs in program synthesis, constructs machine learning pipelines in AutoML, and discovers mathematical expressions in symbolic regression. However, this domain-specific customization presents a tradeoff: while it enhances performance and domain alignment, it also diminishes generalizability. Consequently, the complexity of designing tailored GP systems for novel tasks can be a barrier to entry. Automating the development and configuration of GP systems offers a promising approach to mitigating this tradeoff and improving accessibility.

Although many GP systems are tailored to specific problems, they typically share a common set of core components (Eiben and Smith, 2015; Hernandez, 2023): solution representation, selection strategies (including both parent and survival selection), variation operators, and a fitness function. Each of these components must be carefully defined for the target problem, with the consideration that their interactions can significantly influence the likelihood of achieving successful outcomes. For instance, in program synthesis, the solution representation corresponds to executable programs. While the high-level structure of this representation is understood, numerous implementation paradigms exist, such as tree-based GP (Koza, 1992) and linear GP (Brameier and Banzhaf, 2007). Once a representation is chosen, suitable variation operators such as Uniform Mutation by Addition and Deletion (UMAD) (Helmuth et al., 2018) must be developed to effectively explore the space of potential programs. Selection strategies are then used to identify high-performing solutions based on their fitness. Among these, lexicase selection (Helmuth et al., 2014) has emerged as one of the most effective parent selection techniques in the context of program synthesis. After specifying all components, practitioners *compose* them into a cohesive GP system, with the final step involving the configuration of hyperparameters at both the system (e.g., the population size) and component levels (e.g., the tournament size for tournament selection).

This overview of constructing a GP system for program synthesis highlights the complexity in the associated design decisions. It also highlights the importance of domain expertise in selecting and configuring system components identified through the literature review (e.g., Sobania et al. (2023)). In the absence of such expertise, the design process becomes more challenging mainly because of two factors (Yang, 2020; Bartz-Beielstein et al., 2014; Yang, 2014): (1) the selection of suitable GP components and (2) the tuning of hyperparameters. Component selection is particularly difficult given the wide range of available alternatives and their context-dependent performance characteristics. Specifically, a component that performs well in one domain may not generalize effectively to another. For example,

both lexicase and tournament selection are commonly used methods for parent selection. Although lexicase often yields superior results in program synthesis, it has demonstrated lower effectiveness than tournament selection in certain AutoML applications (Hernandez et al., 2025b). With respect to hyperparameter tuning, both Karafotias et al. (2015) and Eiben and Smit (2011) present a conceptual framework that delineates the relationships among various tuning strategies for EAs, along with a comprehensive survey of existing techniques.

Ideally, a GP system should be implemented by an expert with comprehensive knowledge of the target problem domain and the intricacies of GP methodologies. However, attaining such expertise is increasingly difficult due to the vast and growing body of literature on these topics. Conversely, a practitioner lacking domain knowledge or GP experience must navigate an exponentially large configuration space, a challenge exacerbated by the high computational cost of fitness evaluations and the absence of guarantees for converging to optimal solutions (Bartz-Beielstein et al., 2014). The development of technologies capable of automating the construction of GP systems from scratch—by emulating the decision-making processes of domain experts—would substantially alleviate these challenges and enhance the accessibility of GP. In this chapter, we propose a theoretical framework for automating the construction of GP systems using Agentic AI.

2 Background

2.1 Genetic Programming Fundamental Building Blocks

As mentioned earlier, a given genetic programming (GP) system, irrespective of the type of problems it attempts to solve, comprises certain fundamental building blocks. We discuss each of those components briefly here.

1. **Representation:** It refers to the data structure that is used to represent a possible solution to the problem. Multiple options exist in this area, including but not limited to expression trees (Koza, 1992), a series of production rules used to construct valid programs using a context-free grammar (Whigham, 1995), linear sequences of datatype-specific instructions and constants (Spector et al., 2005).
2. **Fitness Function:** The quality or fitness of an individual during evolution is given by a fitness function. For each test case used to evaluate a given program, the error can be computed through multiple methods: error of 0 if the expected output matches the individual program output and 1 otherwise, actual difference (e.g., Levenshtein distance in the case of strings) between the expected output and the program output, etc. The fitness of an individual can therefore be a vector of these errors, or a single value aggregated (e.g., averaged) across all test cases. Additionally, non-performance related metrics may also be computed by a fitness function (e.g., size and runtime).

3. **Selection Method:** In GP, the process of evolving solutions across generations involves selecting high-performing individuals from the current population to propagate into future generations. Typically, a subset of individuals is chosen to form a pool of parents, which are then subjected to modification to generate the next generation. This process is referred to as *parent selection*, and several strategies are commonly used, including tournament selection (Brindle, 1980), fitness sharing (Goldberg et al., 1987), and lexicase selection (Helmuth et al., 2014). In some GP systems, an additional step known as survival selection is applied, where the original population and the newly generated offspring are combined, and only the most fit individuals are retained for the next generation. A well-known example of this approach is the NSGA-II (Deb et al., 2002) algorithm, which performs survival selection based on Pareto dominance, among other metrics.
4. **Variation Operators:** In order to introduce variation in the population, parents are modified to produce the next generation of individuals. Parts of one parent can be modified, in a set of methods called mutation operators, or parts of two parents can be combined, in a process called crossover, to produce child individuals. In case of tree-based GP, for example, mutation can involve adding or deleting a node, whereas crossover can involve combining subtrees from two different trees to produce a new tree.

2.2 Agentic Paradigm

Recent advancements in artificial intelligence (AI) and machine learning (ML) have enabled the automation of increasingly complex tasks that traditionally required human-level intelligence (Russell and Norvig, 2016). This progress is driving a transformation in the workforce, with many organizations investing in tools designed to replicate aspects of human labor (Argenti, 2025; Shavit et al., 2023; Shein, 2025). Among these technologies are **AI agents**. The concept of an "agent" is not new and has already been discussed in the context of AI (Russell et al., 1995) in general and Reinforcement Learning (Sutton et al., 1998) in particular, among many other fields. In this work, however, we define AI agents more broadly as the systems that operate with a degree of autonomy to pursue goals defined by humans or other agents within specified environments, often in collaboration with human users (Shavit et al., 2023).

Expanding upon this concept, **Agentic AI** refers to a problem-solving framework composed of a single or multiple AI agents, in which each agent manages a subtask contributing to a broader objective. The internal architectures of these agents can vary, incorporating supervised learning, reinforcement learning, or other ML techniques. For example, Insa-Cabrera et al. (2011) utilized a Q-learning-based agent to benchmark performance against human players. More recently, Large Language Models (LLMs)—such as GPT-4 models (Achiam et al., 2023)—have sparked in-

terest in their application as the computational backbone of AI agents (Luo et al., 2025).

In this work, we adopt the multidimensional definition of 'agency' introduced by Shavit et al. (2023) to inform the design of an Agentic AI to build GP systems—referred to as the Agentic GP framework hereafter. This framework starts by processing a user's specification as input and autonomously constructs a GP system tailored to the given task. The framework comprises multiple AI agents that collaborate in the construction and execution of the GP system, collectively working toward a viable solution. The multidimensional definition of agency is central to our approach, as it enables a nuanced representation of agentic behavior that a binary classification (agentic vs. non-agentic) cannot adequately capture. In the following sections, we describe the four dimensions of agency defined by Shavit et al. (2023) and detail how each dimension informs both the overall framework and the behavior of the individual AI agents within it.

2.2.1 Goal Complexity

Broadly speaking, this dimension captures the level of difficulty humans would face in solving a given problem. That difficulty would inform the breadth of goals that a system would potentially be required to achieve. Developing GP systems from scratch is a non-trivial task, comparable in complexity to challenges found within the CASH (Combined Algorithm Selection and Hyperparameter optimization) problem domain. Ideally, a team of experts would collaborate to explore various GP configurations, with each expert contributing specialized knowledge related to specific aspects of the GP development process. However, identifying and assembling such a group of experts is itself a challenging endeavor, underscoring the complexity of the problem that the Agentic GP framework aims to address. The overarching objective of the framework is to autonomously generate GP systems in response to user-defined tasks, with the assumption that it can adapt to any problem domain. Within this framework, individual AI agents are responsible for solving distinct subproblems commonly encountered in the design of GP systems.

2.2.2 Environmental Complexity

This dimension captures the complexity of the environment in which a framework must operate to address a given problem. The Agentic GP framework is designed to function in such complex environments, as evidenced by the diversity of problems it addresses and the variety of tools it employs. Each user request typically specifies a particular problem type or instance, inherently tied to a domain-specific context. Moreover, the tools integrated within a GP system often span multiple disciplines and may require the incorporation of cross-domain knowledge. For instance, a geneticist developing a GP system for epistasis detection would integrate domain-specific expertise from genetics alongside machine learning techniques (Hernandez

et al., 2025a). Ideally, the Agentic GP framework would leverage existing knowledge on a wide range of topics to deal with any kind of problem.

2.2.3 Adaptability

This dimension assesses the framework's ability to handle new or unforeseen scenarios. Specifically, we interpret this as the case in which the Agentic GP framework is confronted with a problem for which no prior guidance or domain-specific knowledge is available. Such open-ended problems are well-suited for GP, as its inherent flexibility permits an initial exploration using a minimalist setup. Although no predefined strategy exists, the Agentic GP framework should iteratively refine its approach by analyzing results from previous runs. The GP literature offers numerous strategies for enhancing performance, such as introducing alternative selection mechanisms to promote population diversity or transitioning from single-objective to multi-objective optimization. Nonetheless, if the system fails to make meaningful progress, user intervention—such as providing additional heuristics or domain hints—may be necessary to facilitate further improvement.

2.2.4 Independent Execution

This dimension assesses the ability of agents to operate autonomously, without reliance on external intervention. Under this criterion, the current framework does not exhibit full autonomy, as it requires user-initiated prompts to initiate execution and follows a predetermined flow of execution. However, the framework can itself be embedded within a higher-level system that can refine it independently without much input from the user. Within the Agentic GP framework, AI agents are not active unless explicitly invoked; nonetheless, once initiated, these agents should exhibit a degree of autonomy by operating independently and coordinating to construct a GP system tailored to the given problem.

3 Agentic GP

The central idea behind employing Agentic AI in this chapter is to leverage a team of specialized agents—each with expertise in a different aspect of the Genetic Programming (GP) algorithm—to collaboratively design a GP system from scratch. This multi-agent approach is hypothesized to yield more effective systems than those constructed by a single, general-purpose agent. The inspiration comes partly from how humans collaborate in teams to accomplish tasks (Specht and Crowston, 2022; Woolley et al., 2010). Imagine if a group of GP experts is given the task of designing a GP system from scratch. Assuming there is a wide variety of expertise in the group, there would be some experts who specialize in the solution represen-

tation, some experts who are more experienced with selection algorithms, etc. Such a collaborative effort is likely to result in a more robust and well-tuned GP system than one designed by a single researcher (Surowiecki, 2005).

Given that an AI agent may have access to the internet and a large body of existing literature, it can potentially explore a broader range of options for each GP component than a human designer. Therefore, a well-designed Agentic framework enables the exploration and testing of a significantly larger number of more diverse GP systems than would be feasible using human expert teams alone. However, the effectiveness of the GP system ultimately depends on the domain knowledge and expertise embedded by human designers or agentic AI developers during its construction. There can be multiple frameworks for using AI agents to build GP systems from scratch for given domains. While designing any such framework, one must determine the number of agents and the tasks they would be assigned to perform. A given goal—in this case, building a GP system—needs to be decomposed into different tasks that different AI agents can work on. Based on how the overall goal is decomposed, the framework might need more or fewer agents. For example, while proposing an agentic framework, we can choose between having one agent that specializes in variation operators or two agents specializing in mutation and crossover operators, respectively.

Another issue to consider while designing a framework is what parts of the GP algorithm we want to automatically generate. In the most expansive scenario, we can have the agentic system generate all the components of the GP system. This type of framework is especially useful when we are building a GP system for an entirely new domain. In many situations, however, we may want to use an existing GP system and optimize only a few components of the system. In these situations, we can come up with a framework that generates one component at a time while keeping the other components fixed. Such an approach is likely to be less complex than trying to build all the components at once.

3.1 Proposed Agentic GP Framework

Figure 1 presents our proposal for a framework for building a complete GP system from scratch. We first discuss various components of this framework and then describe how they work together to build GP systems.

The whole framework is composed of two types of elements: Agents and static programs (depicted as red and blue boxes, respectively, in the figure). Each agent specializes in and is responsible for one part of the overall pipeline for building the GP system. The process of developing a GP system proceeds in the following way.

Step 1 (User Prompt): The process begins with the user forming a detailed prompt to be sent to the agentic AI framework. The prompt should include a description of the particular domain we are trying to build the GP system for, the benchmark problems the synthesized system would be evaluated on, among other things.

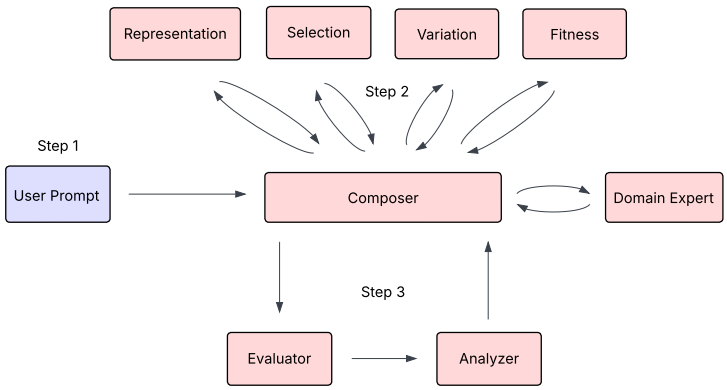

Fig. 1 Agentic GP framework for using agents for building a given GP system. The red boxes represent agents, while the blue boxes contain code that remains unchanged throughout the procedure.

Step 2 (Composition): The prompt is then sent to an agent we term as the *Composer*. This agent acts as a coordinator, responsible for delegating tasks to specialized agents, collecting their outputs, and integrating the results into a complete GP system. The process is largely synchronous: the composer sends out specific instructions to each agent and waits for all responses before assembling the components into the main GP loop. Drawing on the earlier analogy of human teams building GP systems, the composer is analogous to a project lead who oversees development, assigns responsibilities based on expertise, and ensures that the individual contributions form a valid GP system.

In Figure 1, we use four types of agents responsible for fundemental building blocks of the GP algorithm: *Representation, Selection, Variation, Fitness.* While assigning tasks to these agents, the Composer can also 'consult' with an agent we term here as the *Domain Expert*. This agent is specialized to provide domain-specific information to the Composer that can be passed to the agents working on different parts of the GP system. For example, if the problem domain is symbolic regression, the Domain Expert can recommend expression trees as the solution representation, tournament or lexicase as the selection algorithm, etc. This recommendation can be passed to the other agents as additional context. The agents working on building components of GP get the relevant instructions from the Composer and return their respective implementations to the Composer.

Step 3 (Evaluation): After assembling the outputs from different agents into a fully functioning GP system, the next step involves running the GP system on benchmark problems that are representative of the domain that the user mentioned in their prompt. Because of the presence of inherent randomness in the GP algorithm, the *Evaluator* agent should ideally attempt to solve every problem multiple times. The performance of the returned GP system can be measured in multiple ways. The simplest way is to count the number of benchmark problems correctly solved

by the system within a given budget (maximum number of generations, maximum number of evaluations, etc.). Another option is to calculate the success rate in terms of the proportion of runs launched for a problem that resulted in a solution, and then average this across the benchmark problems. Other options include looking at the average number of generations it took the system to find a solution and the average size of the solution programs. The next step is to collate all these performance data and send it to the *Analyzer* agent. This agent analyses the results and sends out the instructions to the Composer to improve the GP system on the performance metrics. This whole process (Steps 2-3 in Figure 1) repeats for a certain number of iterations. The final GP system so obtained is returned at the end.

Note that the above-mentioned framework is one such agentic framework. One can add more agents to the framework, or even make some modules, such as the Evaluator and Analysis modules, non-agentic.

3.2 Agentic GP for Piecewise Development

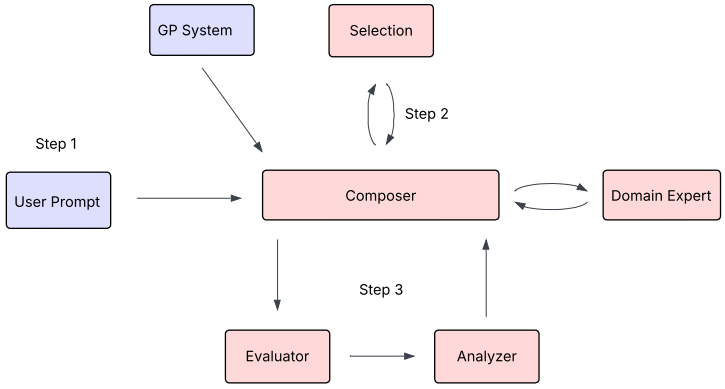

Fig. 2 Framework for using Agentic AI for building parts of a given GP system. The example shows the procedure for synthesizing a parent selection method for a given GP system.

Sometimes the user does not need to develop an entire GP system from scratch and instead just wants to improve some parts of the system. Figure 2 shows one possible framework for building a single component (a parent selection method) of a GP system, provided all other components are already determined. The process of building a GP component proceeds in a similar way to the one described in the previous section. In the first step, the user forms a detailed prompt to be sent to the agentic AI framework. The prompt, along with an already existing GP system, is sent to the *Composer*. This agent, after 'consulting' with the *Domain Expert* agent,

sends the detailed instruction to the *Selection* agent. The output from the Selection agent is added to the given GP system, which is then run on the benchmark problems. The *Analysis* agent gets the performance data and form a summary with detailed instruction to be sent to the Composer to improve the selection method. This whole process (Steps 2-3 in Figure 2) repeats for a certain number of iterations, and the final GP system is returned at the end.

4 LLM Agentic GP

In the previous section, we discussed a framework for Agentic GP that can use any implementation of an agent. Here, we discuss the framework in the context of the AI agents implementing LLM as the underlying mechanism.

4.1 Why LLM Agents?

In recent year, large language models (LLMs) have greatly improved in their capacity to synthesize programs when given user intent in the natural language (Ramírez-Rueda et al., 2024). Let's take GPT-4 as an example. OpenAI's GPT-4, along with its more cost-efficient variant GPT-4o, builds upon earlier architectures by increasing model scale to improve performance (OpenAI et al., 2024). GPT-4 has demonstrated strong capabilities in program synthesis, solving 67% of tasks from the HumanEval benchmark—a suite of 164 programming problems, each defined by a function signature, a natural language docstring, and associated unit tests. Furthermore, GPT-4 successfully solved 31 out of 41 problems categorized as easy on LeetCode (OpenAI et al., 2024), a popular platform for algorithmic coding challenges used in competitive programming and technical interviews.

LLMs agents can easily be made specialized by giving them access to existing research papers or other knowledge bases (discussed in the next subsection). They can even access internet to get the most up-to-date information on the subject. Additionally, LLM can inherently give reasoning for their actions, including the sources they relied on for making the decision. This allows for a more 'natural' communication between agents: they can pass information using natural language that humans can understand. It also makes it easier for humans to inspect the output in case the framework does not work as expected.

Using LLMs also makes is easier for the user to specify their requirements in the form of a prompt. For example, if the user wants to build a program synthesis GP system, part of the prompt can be along the lines of "You are a Genetic Programming (GP) expert. Can you build a GP system for program synthesis for the following set of tasks? The system should accept a problem where the test cases are in the form of input-output examples, and the system should attempt to evolve a program written in a programming language. You can choose an appropriate solution

representation, parent selection method, and variation operators." Similar descriptions can be formed for other domains, such as evolving quantum circuits (Spector et al., 1999) and designing receive antennas (O'Neill, 2009).

4.2 LLM Specialization

In Figure 1, each LLM agent can specialize in one part of the overall pipeline for building the GP system. Depending on the available time, compute, and other resources, different levels of specialization can be introduced into the LLM agents. We discuss three types of specializations in order of increasing complexity.

1. **Prompt Engineering:** Certain prompt engineering strategies can make a given LLM more specialized. For example, one can give the instructions to the LLM on how to process any prompt—called system-level instruction—in the form of a role that the LLM should play. For our case, if we want an agent to specialize in selection methods, the instruction can be, "You are a Genetic Programming (GP) expert that specializes in Parent Selection Methods." Such a tuning can improve the performance of the LLM with respect to the task at hand (Zhang et al., 2024).
2. **Retrieval-Augmented Generation (RAG)**: It is an architecture that improves the performance of LLMs by allowing them to access external knowledge during inference (Lewis et al., 2020). Specifically, the RAG module retrieves relevant documents or their parts from a knowledge base and uses them as context for the prompt to generate more informed and accurate responses. For our use case, we can provide an LLM agent access to a corresponding RAG module. For example, for the Representation agent, the RAG module would contain research papers and books that deal with different types of solution representation in the GP literature.
3. **Fine-tuning**: One of the most computationally intensive process of making an LLM agent more specialized would be to fine-tune the underlying LLM with the code and documents relevant to the task as hand (e.g., variation operators). Fine-tuning essentially involves retraining the final layers of the LLM with a smaller and more focused dataset.

5 Prototype

As a proof of concept, we implemented a pared-down version of the framework presented in Figure 2. Specifically, we do not use the Domain Expert module, and the Composer, Evaluator, and Analyzer modules contain static code and therefore are non-agents. The Selection module is not specialized and is a general-purpose LLM agent. This implementation serves as a baseline for more sophisticated frameworks in the future. We use symbolic regression as the problem domain, and try to build a selection method for an existing GP implementation. Figure 3 shows the implemented framework. The particulars of different components are as follows.

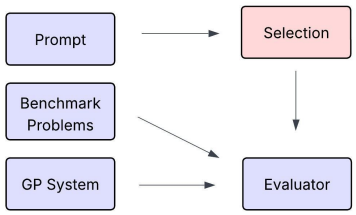

Fig. 3 A simple Agentic system to build parent selection methods.

We select a very simple GP system called TinyGP[1] and modify it for our experiments. This system implements a tree-based GP system for symbolic regression problems that allows for the following possible nodes: (a) 'add' (addition), 'sub' (subtraction), and 'mul'(multiplication) as non-terminals, (b) 'x', -2, -1, 0, 1, and 2 as terminals. We adapt TinyGP by removing only the parent selection mechanism, leaving the remaining components unchanged, to allow our agentic framework to design the missing part. As benchmark problems, we consider two symbolic regression tasks: (a) x^3+x+2, and (b) $x^4+x^3+x^2+x+1$.

For the LLM agent, we chose Llama 3.1 model with 8 billion parameters. We selected this model because, in addition to it being an open-source model from the latest family of models by Meta, it has comparable or better performance than other similar models[2]. To keep the terminology consistent and for the possibility of enhancements in the future, we will continue to refer to our system as an agentic framework even though we use a simple LLM without any specialization.

5.1 Experimental Setup

LLMs can sometimes generate code that is syntactically incorrect or produce runtime errors. Before experimenting with the selection methods generated by the LLM, we first evaluated how reliably the LLM could produce usable code. To do that, we prompted the LLM 100 times to generate selection methods for our GP system. For each generated response, we tested it by integrating it with the GP system and running the GP system on our benchmark problems, using a population size of 10 and a maximum of 50 generations. The results are as follows. Out of 100 independent prompts, the LLM produced a valid selection method 43 times.

Furthermore, we compared the selection methods generated by our agentic framework with existing selection methods. Specifically, we evaluated five experimental conditions with an equal number of runs for all of them: the first three correspond to widely used parent selection strategies—random selection, tourna-

[1] https://github.com/moshesipper/tiny_gp/

[2] https://ai.meta.com/blog/meta-llama-3-1/

ment selection, and lexicase selection—serving as baselines. The remaining two conditions represent different configurations of our agentic framework, referred to as AGP1 and AGP2, which are described in detail below.

When comparing the LLM-generated selection methods against baseline methods, we only use valid methods. To generate a valid method, we prompt the LLM a maximum of 500 times until we get a valid function. If we do not get a valid function at the end of 500 trials, we just use the function given at the 500th trial. For AGP1 and AGP2, we generate 10 and 20 valid methods, respectively, using the procedure described above.

For the 10 valid selection methods obtained for AGP1, we launch 10 runs for each of the generated methods, giving us a total of 100 runs. Similar to the AGP1 case, for AGP2, we obtain 20 valid selection methods, and launch 5 run for each of them, giving us a total of 100 runs. We compare AGP1 and AGP2 with three baseline methods with 100 replicates each: random, tournament, and lexicase. For each of the runs, we run the GP system with a population size of 100 for a maximum of 500 generations. To test for the significant differences in success rates under different experimental conditions, we used the pairwise chi-square test with Holm correction and a 0.05 significance level.

All software, data analysis code, and documentation related to the experiments are provided as supplementary materials on our GitHub repository[3].

5.2 Results

For each method, we calculate the success rate, defined as the number of runs out of 100 that produced a solution to the problem. A solution is defined as a program that leads to zero errors on all the test cases. We additionally report the average fitness scores of the best individual per replicate (based on the total error) across all 100 runs for each of the methods. The results are given in Table 1. For both tasks, tournament selection produced the largest number of successes, followed by lexicase. Importantly, both AGP1 and AGP2 significantly outperformed random selection ($p < 10^{-2}$ in both tasks). However, their performance was significantly worse than that of tournament and lexicase selection methods across both tasks ($p < 10^{-6}$ for all comparisons).

Further analysis of the valid functions used in AGP1 and AGP2 reveals that most of the returned functions implement Elitism Selection (i.e., select the individual with the lowest total error), followed by Fitness-proportionate selection. Specifically, in AGP1, 9 selection methods implemented elitism, and one implemented fitness-proportionate selection, and in AGP2, 18 selection methods implemented elitism, and one implemented fitness-proportionate selection (more details in the supplementary material). Note that even though all the valid functions used in the final experiments implement either elitism or fitness-proportionate selection, the

[3] https://github.com/theaksaini/GPTP-2025-Agentic-GP

Table 1 Success rate (more is better) and average fitness values (less is better) across 100 runs for each of the methods studied in this paper.

Exp. Condition	Task 1		Task 2	
	Succ. Rate	Avg. Fitness	Succ. Rate	Avg. Fitness
AGP1	46	968.34	10	327236680.81
AGP2	47	748.1	12	304286787.19
Random	20	868.69	0	28800072.03
Tournament	95	109.11	53	89986710.1
Lexicase	93	16.16	46	60494389.31

LLM might still be generating tournament or other selection methods, but those generated methods might be syntactically incorrect or produce runtime errors and hence were not included in the final experiments.

These results indicate that while LLMs are capable of generating valid selection methods (albeit in less than 50% of attempts) and those valid methods can surpass random selection, the reliability and quality of generation remain insufficient. Further enhancements are needed to increase the frequency of valid outputs and to enable LLM-generated methods to match or exceed the performance of strong baseline approaches such as tournament and lexicase selection. The future work can look into making the LLMs more specialized using the strategies described in Section 4.2.

6 Conclusions

In this chapter, we presented a framework called Agentic GP for using Agentic AI to develop GP systems composed of fundamental building blocks from scratch. Agentic AI is composed of agents that plan and execute their actions but work towards achieving goals set by human users, which may prove useful in developing specialized agents tasked with building complete GP systems. We first provided a general framework that can be developed with any type of AI agents, and later provided detailed instructions on how the framework can be actualized using LLM agents. As a proof of concept, we implemented an Agentic GP framework to build selection methods for a simple GP system. We found that just using an out-of-the-box LLM produced syntactically incorrect code most of the time. Although the executable selection methods performed better than random selection, they still performed worse than tournament and lexicase selection methods. Consequently, more specialized LLM agents are needed in Agentic GP to build effective GP systems.

Acknowledgments

We thank members of the Department of Computational Biomedicine at Cedars-Sinai Medical Center for their helpful comments on this work. Cedars-Sinai Medical Center provided computational resources through its High Performance Computing clusters. The work was supported by NIH grants R01 LM010098, R01 LM014572, and U01 AG066833.

References

Achiam, J., Adler, S., Agarwal, S., Ahmad, L., Akkaya, I., Aleman, F. L., Almeida, D., Altenschmidt, J., Altman, S., Anadkat, S., et al. (2023). Gpt-4 technical report. *arXiv preprint arXiv:2303.08774*.

Argenti, M. (2025). What to expect from ai in 2025: Hybrid workers, robotics, expert models. https://www.goldmansachs.com/insights/articles/what-to-expect-from-ai-in-2025-hybrid-workers-robotics-expert-models. Accessed: 2025-05-08.

Banzhaf, W., Machado, P., and Zhang, M. (2024). *Handbook of Evolutionary Machine Learning*. Springer Singapore.

Bartz-Beielstein, T., Branke, J., Mehnen, J., and Mersmann, O. (2014). Evolutionary algorithms. *WIREs Data Mining and Knowledge Discovery*, 4(3):178–195.

Brameier, M. F. and Banzhaf, W. (2007). *Linear Genetic Programming*. Genetic and Evolutionary Computation. Springer, New York, NY.

Brindle, A. (1980). *Genetic algorithms for function optimization*. PhD dissertation, University of Alberta.

Deb, K., Pratap, A., Agarwal, S., and Meyarivan, T. (2002). A fast and elitist multiobjective genetic algorithm: Nsga-ii. *IEEE transactions on evolutionary computation*, 6(2):182–197.

Eiben, A. and Smit, S. (2011). Parameter tuning for configuring and analyzing evolutionary algorithms. *Swarm and Evolutionary Computation*, 1(1):19–31.

Eiben, A. E. and Smith, J. E. (2015). *Introduction to evolutionary computing*. Springer Berlin, Heidelberg.

Goldberg, D. E., Richardson, J., et al. (1987). Genetic algorithms with sharing for multimodal function optimization. In *Genetic algorithms and their applications: Proceedings of the Second International Conference on Genetic Algorithms*, pages 41–49. Hillsdale, NJ: Lawrence Erlbaum.

Helmuth, T., McPhee, N. F., and Spector, L. (2018). Program synthesis using uniform mutation by addition and deletion. In *Proceedings of the Genetic and Evolutionary Computation Conference*, pages 1127–1134.

Helmuth, T., Spector, L., and Matheson, J. (2014). Solving uncompromising problems with lexicase selection. *IEEE Transactions on Evolutionary Computation*, 19(5):630–643.

Hernandez, J. G. (2023). *Beyond Benchmarks Suites: Engineering Diagnostic Tools to Characterize Selection Schemes*. PhD dissertation, Michigan State University.

Hernandez, J. G., Ghosh, A., Freda, P. J., Meng, Y., Matsumoto, N., and Moore, J. H. (2025a). Starbase-gp: Biologically-guided automated machine learning for genotype-to-phenotype association analysis.

Hernandez, J. G., Saini, A. K., Gupta, A., and Moore, J. (2025b). Evaluating the generalizability of machine learning pipelines when using lexicase or tournament selection. In *Companion of the Genetic and Evolutionary Computation Conference (GECCO '25 Companion)*, New York, NY, USA. ACM.

Insa-Cabrera, J., Dowe, D. L., Espana-Cubillo, S., Hernández-Lloreda, M. V., and Hernández-Orallo, J. (2011). Comparing humans and ai agents. In *Artificial General Intelligence: 4th International Conference, AGI 2011, Mountain View, CA, USA, August 3-6, 2011. Proceedings 4*, pages 122–132. Springer.

Karafotias, G., Hoogendoorn, M., and Eiben, A. E. (2015). Parameter control in evolutionary algorithms: Trends and challenges. *IEEE Transactions on Evolutionary Computation*, 19(2):167–187.

Koza, J. R. (1992). *Genetic programming: on the programming of computers by means of natural selection*. MIT Press, Cambridge, MA, USA.

La Cava, W., Burlacu, B., Virgolin, M., Kommenda, M., Orzechowski, P., de França, F. O., Jin, Y., and Moore, J. H. (2021). Contemporary symbolic regression methods and their relative performance. *Advances in neural information processing systems*, 2021(DB1):1.

Lewis, P., Perez, E., Piktus, A., Petroni, F., Karpukhin, V., Goyal, N., Küttler, H., Lewis, M., Yih, W.-t., Rocktäschel, T., et al. (2020). Retrieval-augmented generation for knowledge-intensive nlp tasks. *Advances in neural information processing systems*, 33:9459–9474.

Luo, J., Zhang, W., Yuan, Y., Zhao, Y., Yang, J., Gu, Y., Wu, B., Chen, B., Qiao, Z., Long, Q., et al. (2025). Large language model agent: A survey on methodology, applications and challenges. *arXiv preprint arXiv:2503.21460*.

OpenAI et al. (2024). Gpt-4 technical report.

O'Neill, M. (2009). Riccardo poli, william b. langdon, nicholas f. mcphee: A field guide to genetic programming: Lulu. com, 2008, 250 pp, isbn 978-1-4092-0073-4.

Ramírez-Rueda, R., Benítez-Guerrero, E., Mezura-Godoy, C., and Bárcenas, E. (2024). A systematic literature review of 10 years of research on program synthesis and natural language processing. *Programming and Computer Software*, 50(8):725–741.

Russell, S., Norvig, P., and Intelligence, A. (1995). A modern approach. *Artificial Intelligence. Prentice-Hall, Egnlewood Cliffs*, 25(27):79–80.

Russell, S. J. and Norvig, P. (2016). *Artificial intelligence: a modern approach*. pearson.

Shavit, Y., Agarwal, S., Brundage, M., Adler, S., O'Keefe, C., Campbell, R., Lee, T., Mishkin, P., Eloundou, T., Hickey, A., et al. (2023). Practices for governing agentic ai systems. *Research Paper, OpenAI*.

Shein, E. (2025). The outlook for programmers. *Commun. ACM*, 68(05):12–14.

Sobania, D., Schweim, D., and Rothlauf, F. (2023). A comprehensive survey on program synthesis with evolutionary algorithms. *IEEE Transactions on Evolutionary Computation*, 27(1):82–97.

Specht, A. and Crowston, K. (2022). Interdisciplinary collaboration from diverse science teams can produce significant outcomes. *PLOS ONE*, 17(11):1–25.

Spector, L., Barnum, H., Bernstein, H. J., and Swamy, N. (1999). Quantum computing applications of genetic programming. *Advances in genetic programming*, 3:135–160.

Spector, L., Klein, J., and Keijzer, M. (2005). The push3 execution stack and the evolution of control. In *Proceedings of the 7th annual conference on Genetic and evolutionary computation*, pages 1689–1696.

Surowiecki, J. (2005). *The Wisdom of Crowds*. Anchor.

Sutton, R. S., Barto, A. G., et al. (1998). *Reinforcement learning: An introduction*, volume 1. MIT press Cambridge.

Whigham, P. A. (1995). Grammatically-based genetic programming. In Rosca, J. P., editor, *Proceedings of the Workshop on Genetic Programming: From Theory to Real-World Applications*, pages 33–41, Tahoe City, California, USA.

Woolley, A. W., Chabris, C. F., Pentland, A., Hashmi, N., and Malone, T. W. (2010). Evidence for a collective intelligence factor in the performance of human groups. *Science*, 330(6004):686–688.

Yang, X.-S. (2014). Chapter 2 - analysis of algorithms. In Yang, X.-S., editor, *Nature-Inspired Optimization Algorithms*, pages 23–44. Elsevier, Oxford.

Yang, X.-S. (2020). Nature-inspired optimization algorithms: Challenges and open problems. *Journal of Computational Science*, 46:101104. 20 years of computational science.

Zhang, L., Ergen, T., Logeswaran, L., Lee, M., and Jurgens, D. (2024). Sprig: Improving large language model performance by system prompt optimization. *arXiv preprint arXiv:2410.14826*.