Mutation Without Variation: Convergence Dynamics in LLM-Driven Program Evolution

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

Forrest Stonedahl
Augustana College
Rock Island, IL, USA

Uri Wilensky
Northwestern University
Evanston, IL, USA

Abstract

When an LLM repeatedly mutates a program, does it explore new forms or circle back to the same ones? We study this question by analyzing LLM-driven mutation chains in the absence of selection pressure within a domain-specific language, varying prompt design, model family, and stochastic replication. We find that LLM-based mutation consistently converges toward restricted attractor regions in program space. Convergence is especially severe at the structural level: in 87% of chains, over 93% of mutations revisit a previously seen structural form, with most variation confined to terminal substitutions within recurring templates. Cycle analysis reveals short cycles and self-loops dominating the transition structure. The rate of convergence varies with prompt wording and model choice, but the phenomenon is robust across conditions. A classical GP subtree mutation operator does not exhibit comparable convergence, suggesting that the effect is intrinsic to the LLM mutation pipeline. These findings reveal a tension at the heart of LLM-driven program evolution: the same capabilities that enable semantics-aware program transformation also carry a systematic bias toward structural homogeneity that must be accounted for if such systems are to sustain open-ended exploration. Source code is available at https://github.com/can-gurkan/lmca.

CCS Concepts

• **Computing methodologies** → **Natural language generation**; *Heuristic function construction*; *Discrete space search*; Intelligent agents; Agent / discrete models; **Genetic programming**; **Genetic algorithms**; **Generative and developmental approaches**; • **Theory of computation** → *Tree languages.*

Keywords

Large Language Models, Genetic Programming, Evolutionary Computation, Dynamical Systems

ACM Reference Format:
Can Gurkan, Forrest Stonedahl, and Uri Wilensky. 2026. Mutation Without Variation: Convergence Dynamics in LLM-Driven Program Evolution. In *Genetic and Evolutionary Computation Conference (GECCO Companion '26), July 13–17, 2026, San Jose, Costa Rica.* ACM, New York, NY, USA, 16 pages. https://doi.org/10.1145/3795101.3814735

*Corresponding Author

GECCO Companion '26, San Jose, Costa Rica

ACM ISBN 979-8-4007-2488-6/2026/07
https://doi.org/10.1145/3795101.3814735

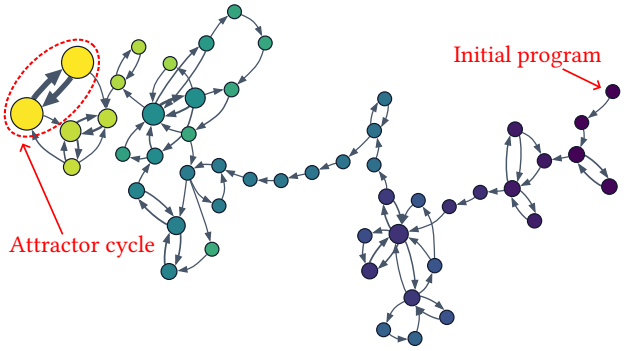


Figure 1: Example transition graph from a single LLM mutation chain. Nodes represent unique program states, with size proportional to visit frequency and color indicating discovery order. The trajectory passes through a transient sequence before settling into a 2-cycle attractor.

1 Introduction

In the conclusion of *On the Origin of Species* [5], Charles Darwin wrote, "...from so simple a beginning endless forms most beautiful and most wonderful have been, and are being, evolved". This vision of open-ended innovation has long served as both inspiration and aspiration for evolutionary computation. Evolutionary algorithms (EAs) rely on the premise that simple variation operators, applied iteratively, can give rise to a vast and diverse space of candidate solutions. Among these operators, mutation plays a central role, continually injecting new genetic material and enabling populations to explore beyond their current configurations. Both the rate at which variation is introduced and the magnitude of individual changes shape the trajectory of the search, and maintaining this balance has always been of central importance to a successful EA [2].

Recently, advances in large language models (LLMs) have enabled a new paradigm of LLM-driven discovery, in which models iteratively generate and refine diverse candidate solutions [9]. Ushered in by the seminal work of Lehman et al. [20], this line of research embeds LLMs within iterative optimization loops, where they generate, modify, and recombine solutions across successive generations [15, 18, 38, 47]. Such systems have shown promise in domains ranging from program synthesis to scientific discovery, at times producing solutions that appear to extend beyond the scope of their training distributions [4, 6, 13, 14, 19, 21–24, 29, 34, 39, 49].

A particularly active application of this paradigm is in genetic programming (GP) [16], where LLMs are employed to generate syntactically valid code, enhancing the capabilities of evolutionary

operators [3, 8, 11, 20, 26]. This integration has given rise to LLM-driven genetic programming (LLM-GP), in which generative models serve as variation operators within the evolutionary loop [10].

A central component of the LLM-GP framework is the use of LLMs as mutation operators for programs. Unlike traditional syntactic perturbations, LLM-based mutations are semantics-aware and capable of producing coherent, structured transformations that are often directly applicable to modern programming languages [8, 11, 20]. These properties suggest that LLMs provide a qualitatively different mechanism for traversing program space, one that is more guided and potentially more effective than conventional mutation operators.

However, this promise raises a more fundamental question that remains largely unexamined. When an LLM is used repeatedly as a mutation operator, how does it traverse the space of programs? Does it continue to generate novel forms in the spirit of Darwin's observation, or does it instead exhibit a tendency to converge toward particular regions of the space, gradually limiting the diversity of representations? If LLM-driven mutation introduces implicit biases that steer programs toward canonical structures favored by the model's training distribution, such biases could shape evolutionary search trajectories in ways that are not immediately apparent, potentially restricting exploration even in the absence of explicit constraints.

Existing work has largely examined LLMs within optimization loops, where their effectiveness is measured in terms of solution quality [3, 4, 6, 8, 10, 11, 13–15, 17–24, 26, 29, 34, 38, 39, 42, 43, 47, 49]. This perspective obscures the intrinsic behavior of the mutation operator itself: the presence of selection pressure makes it difficult to disentangle whether observed convergence arises from the fitness landscape or from biases inherent to the LLM. To understand the role of the mutation operator in isolation, it is therefore necessary to study its behavior independently of selection, focusing on how it transforms programs over repeated application.

The question of how repeated application shapes LLM outputs connects to a broader class of iterative LLM systems, including chain-of-thought reasoning [46], self-refinement pipelines [25], and autonomous agents [14, 44, 48], in which model outputs are recursively fed back as inputs. Prior work has identified drift toward high-probability outputs and loss of diversity in text [27, 32, 40, 41, 45] and image [12, 28] domains, but analogous analyses in program space remain limited, despite code providing a structured domain in which convergence dynamics can be rigorously measured.

In this work, we study LLM-driven mutation as a stochastic, semantics-aware rewriting process over program space. We analyze mutation chains in the absence of selection pressure in order to isolate the intrinsic dynamics of the operator. Our central objective is to determine whether repeated application of LLM-based mutation promotes structural diversity through the generation of novel genetic material, or instead induces convergence toward restricted regions of the program space, potentially exhibiting attractor-like behavior. To enable precise structural analysis, we conduct these experiments within a constrained, strongly typed domain-specific language (DSL) that bounds the space of valid programs. Our investigation is purely genotypic: we analyze the structural properties of programs independent of their behavior, ensuring that the dynamics we observe reflect the operator's intrinsic biases rather than any property of the task environment or fitness landscape. By characterizing these dynamics across multiple models, prompts, and initial conditions, we aim to provide a quantitative account of how LLMs shape the evolution of program representations.

1.1 Research Questions & Contributions

Guided by these objectives, we address the following research questions:

(1) Do LLM-driven mutation chains, in the absence of selection pressure, converge toward attractor regions in program space?
(2) To what extent do these mutation processes reduce the diversity of program representations, and do patterns of convergence differ between high-level structural form and surface-level variation?
(3) How sensitive are these dynamics to factors such as prompt design, initial program structure, and model family?

To answer these questions, we make the following contributions:

- We introduce a framework for studying LLM-driven mutation as a dynamical system over program space, enabling the analysis of mutation chains independent of optimization objectives.
- We provide empirical evidence of convergence and diversity stagnation in neutral LLM-driven mutation chains, identifying recurring patterns of change at both the structural and surface level, as well as oscillatory behaviors, across multiple experimental conditions.
- We conduct a systematic analysis of how these dynamics vary across models, prompts, and initial conditions within a constrained DSL setting, revealing the extent to which convergence is a robust property of the LLM mutation pipeline rather than an artifact of any single configuration.

2 Related Work

We situate this work at the intersection of three lines of research: the integration of LLMs into evolutionary computation, the study of convergence dynamics in iterative LLM systems, and the role of drift in program evolution.

2.1 LLMs in Evolutionary Computation

The emergence of large language models as components of evolutionary search has reshaped evolutionary computation, enabling a new class of systems in which generative models augment or replace traditional variation operators [10, 47]. Lehman et al. [20] demonstrated that code-generating LLMs can serve as effective mutation operators for genetic programming by approximating semantics-aware program transformations. This insight has since produced systems for mathematical program discovery [34], large-scale algorithmic optimization [19, 29], heuristic design [21, 39, 49], control policy synthesis [7], and GP-based code evolution [8, 11]. Code Evolution Graphs [42, 43] have further provided tools for analyzing how these systems traverse algorithm space.

Despite the successes of LLM-driven evolutionary systems, a recurring observation is that LLM-driven mutation operators appear to exert systematic pressure toward particular program structures.

Analyses of the LLaMEA framework's behavior space [43] find that independent runs tend to cluster toward structurally similar solutions. Digital Red Queen [17] similarly observes convergence toward generalist strategies across adversarially evolving populations despite diverse initializations. Indeed, diversity-preserving mechanisms have accompanied LLM-driven mutation from the outset: the ELM framework of Lehman et al. [20] itself employed a quality-diversity algorithm, and subsequent systems have similarly incorporated diversity-preserving archives [17], novelty-based rejection sampling [19], and niching [13]. While these mechanisms treat convergence as a practical failure mode to be corrected, the intrinsic dynamics of the mutation operator in the absence of such mechanisms remain unexplored. This leaves open a fundamental question: what intrinsic dynamics does the LLM mutation operator induce in program space when decoupled from selection, and do these dynamics persist robustly across models and prompting strategies?

2.2 Convergence Dynamics in Iterative LLM Systems

Beyond evolutionary computation, convergence dynamics have been observed in iterative LLM systems across multiple domains and modalities. Perez et al. [32] study iterated LLM-to-LLM transmission chains modeled on telephone-game dynamics, showing that textual properties such as toxicity, positivity, and difficulty converge toward equilibrium values over successive rewrites. They introduce an attractor estimation method based on regression between initial and final property values, providing a quantitative notion of attractor position and strength. Mohamed et al. [27] report similar cumulative distortion effects in iterative translation chains, where degradation depends on chain length and intermediate representations. Related studies on iterative LLM paraphrasing have documented similar cumulative transformation effects [35, 41]. In the image domain, Hintze et al. [12] show that autonomous text-to-image-to-text loops converge to just 12 dominant visual motifs across 700 independent trajectories and seven temperature settings, providing striking evidence that convergence toward generic outputs is a robust property of iterative generative systems regardless of modality.

Building on these observations, several studies have formalized iterative LLM generation explicitly as a dynamical system. Wang et al. [45] formalize successive paraphrasing as a discrete dynamical system and demonstrate that trajectories frequently converge not to fixed points but to 2-period limit cycles, introducing a 2-periodicity degree metric that captures alternating-cluster structure across iterations. Tacheny [40] analyzes agentic loops in which LLM outputs are recursively fed back as inputs, distinguishing between contractive dynamics that lead to attractors and divergent dynamics that promote exploration. Notably, the regime depends on prompt design, suggesting that convergence reflects a geometric property of the induced transformation that can be modulated through operator specification. Related theoretical work on model collapse [36, 37] shows that contraction toward typicality and loss of distributional support can arise under recursive generation even at the training level, reinforcing the importance of understanding convergence dynamics at inference time as well. This dynamical systems perspective, treating iterative LLM generation as trajectories in a representational space with measurable contraction and periodicity, provides the conceptual framework adopted in the present work.

In the code domain specifically, Peitek et al. [31] provide one of the most direct empirical studies, analyzing multi-step code refactoring trajectories under repeated LLM application. Their results reveal a characteristic two-phase dynamic: an initial restructuring phase marked by substantial syntactic changes, followed by stabilization in which successive versions become increasingly similar. They also document prompt-dependent oscillatory behavior, in which code alternates between a small number of variants rather than converging to a fixed point. While this work targets code readability improvement rather than mutation in a genetic programming context, the observed dynamics offer compelling evidence that iterative LLM rewriting of code exhibits structured convergence patterns. However, a systematic investigation of convergence under LLM-driven mutation in program space, particularly without selection pressure and across variation in model and prompt design, has not yet been conducted.

2.3 Drift in Program Evolution

The concept of genetic drift has a long history in genetic programming, where neutral drift refers to sequences of mutations that preserve semantic behavior while allowing syntactic variation [30, 33, 50]. Atkinson et al. [1] investigate this idea through equivalence-preserving transformations over program graphs, demonstrating that designed neutral moves can reshape evolutionary trajectories and alter the accessibility of regions in program space without invoking fitness-based selection. Their work utilizes a methodological principle we adopt here: by removing selection pressure, one can isolate and study the intrinsic properties of a mutation operator itself. The present work adopts this principle but shifts the setting: rather than rule-based, semantics-preserving rewrites, we apply LLM-driven mutations that are syntactically constrained but not restricted to preserve behavior. This allows us to study how trajectories move through program space under the operator's bias, and to assess the extent to which genotypic diversity contracts in the absence of selection.

2.4 This Work

The present work sits at the intersection of these three lines of research. Prior studies of LLM-driven evolutionary systems have observed convergence but have not disentangled the contributions of selection pressure from the intrinsic limitations of the LLM-based mutation operator itself. Convergence dynamics have been formally studied in iterative text and image generation, but not in the context of LLM-driven mutation over program space. By isolating the genotypic-level dynamics of iterated LLM-driven mutation in the absence of selection pressure, and applying the dynamical systems framing developed for iterative LLM generation, this work provides a systematic empirical investigation of how LLM mutation operators shape trajectories through program space across multiple models and prompting strategies.

```
PROGN(IF(IS_WALL(FORWARD),TURN_LEFT,MOVE_FORWARD),WHILE(NOT(IS_RESOURCE(FORWARD)),TURN_RIGHT))

PROGN(IF( PRED  (  DIR  ),  ACTION,  ACTION  ),  WHILE(BOOL  (  PRED  (  DIR  ) ), ACTION ))
```

Figure 2: Example program in the DSL (top) and its corresponding skeleton representation (bottom). The skeleton abstracts terminal-level details while preserving control-flow structure.

3 Methods

In this section, we describe the program representation, mutation operators, and trajectory analysis measures used throughout the experiments. The full source code is available at https://github.com/can-gurkan/lmca.

3.1 Program Representation and DSL

To facilitate direct comparison to classical tree-based GP, programs are expressed in a strongly typed, Lisp-like domain-specific language designed for gridworld agent control. The language comprises five categories of primitives, summarized in Table 1. Control flow operators support conditional branching, bounded looping, and sequential composition. Predicates are sensor queries, each parameterized by a relative direction.

Table 1: DSL primitive categories.

Category	Primitives
Control flow	IF, WHILE, PROGN
Booleans	AND, OR, NOT
Predicates	IS_WALL, IS_EMPTY, IS_HAZARD, IS_RESOURCE
Directions	FORWARD, LEFT, RIGHT
Actions	MOVE_FORWARD, TURN_LEFT, TURN_RIGHT, NO_OP

Every program is maintained as both a full program string, preserving all terminals and arguments, and a *skeleton* representation that abstracts terminal-level details while preserving structural form. Specifically, action terminals are replaced with ACTION, direction terminals with DIR, boolean operators with BOOL, and predicates with PRED, while control-flow operators and tree structure are retained. Figure 2 depicts an example program and its skeleton; abstract syntax trees are shown in Appendix A.1. This dual representation distinguishes structural changes from terminal-level substitutions.

3.2 Mutation Operators

LLM Mutation. Each mutation step prompts an LLM with the parent program and a specification of the allowed DSL primitives, instructing it to produce a single mutated variant. The full prompt templates are provided in Appendix A.2. After generation, the candidate is validated for syntactic well-formedness, primitive-set membership, and valid typing. If validation fails, the model is re-prompted with the failure reason and the invalid candidate for up to five retries. If all retries are exhausted, a fallback model is invoked with the same retry budget. Each chain step corresponds to one accepted, validated mutation. If both models exhaust their retry budgets the chain terminates, though this did not occur in any experiment reported here.

Classical GP Mutation. As a baseline, we run the same neutral chain procedure using a standard subtree mutation operator. At each step, a random node is selected in the program tree and replaced with a randomly generated subtree. A maximum tree depth of four is enforced to keep program sizes comparable to those produced by LLM mutation, preventing the count of unique programs from being inflated by unconstrained growth in program size.

3.3 Trajectory Analysis Measures

Our primary measure of convergence is the cumulative count of unique states visited over the course of a chain. At each step t, we record the number of distinct programs observed up to that point, as well as the number of distinct skeletons. By tracking both levels, we distinguish convergence in exact program identity from convergence in structural form. A chain may continue producing lexically distinct programs while visiting the same few skeletons, with variation limited to terminal substitutions (e.g., switching LEFT/RIGHT/FORWARD or swapping AND/OR) within a fixed structural template.

To characterize the structure of convergence, we also construct directed transition graphs from each chain, where nodes represent unique states (programs or skeletons) and directed edges represent observed mutations. We detect all simple cycles in these graphs and report their length distributions. Additionally, we compute the mean degree entropy across nodes, which captures how evenly transitions are distributed across successor states: low entropy indicates that each node tends to transition to a single successor, while higher entropy indicates more varied transitions.

As a complementary measure, we compute pairwise normalized token-level Levenshtein distances between successive programs, capturing the magnitude of change introduced by individual mutations.

4 Experimental Design

We conduct three experiments to assess the robustness of convergence dynamics under neutral LLM-driven mutation, varying prompt design, stochastic replication, and model family respectively. All experiments accept every valid mutation (one candidate per step, no selection pressure) with a chain length of 300 steps (accepted mutations). A classical GP subtree mutation baseline provides a point of comparison. Table 2 summarizes the shared experimental parameters.

4.1 Experiment 1: Prompt Sensitivity

To assess the effect of prompt wording on trajectory dynamics, we sweep over 50 distinct prompt variants. Each prompt shares the same structure, with only the instruction line varying across conditions. The 50 unique instruction variants were generated by

Table 2: Shared experimental parameters. Experiment 3 uses different primary and fallback models for each condition; details are provided in Appendix B.

Parameter	Value
Primary model	Gemini 3.1 Flash Lite
Fallback model	Gemini 3.1 Flash
Temperature	1.0
Max tokens	1024
Max retries (per model)	5
Chain length	300

prompting ChatGPT to produce diverse paraphrasings of the core mutation command; the generation prompt and full list of variants are provided in Appendix A.3. For each of the 50 prompts, we collect mutation chains starting from three different initial programs of varying size (small, medium, and large) drawn from a random initial population, yielding a total of 150 chains.

For comparison using the classical GP baseline, we also collect mutation chains using standard subtree mutation with a maximum depth of four, on the same three initial programs. We collect 50 independent chains per program, for a total of 150 baseline chains.

4.2 Experiment 2: Intrinsic Variability

To assess the stochastic variation inherent in the LLM mutation process itself, we run 30 independent trajectories for each of four prompts using the same medium-sized initial program and Gemini 3.1 Flash Lite, for a total of 120 chains. This experiment isolates the variability introduced by the model's sampling process, holding all other factors constant.

4.3 Experiment 3: Model Sensitivity

To assess whether the observed convergence dynamics are model-specific or reflect a broader property of LLM-driven mutation, we run chains across seven models: Gemini 3.1 Flash Lite, Gemini 3.1 Flash Lite with reasoning, Claude Haiku 4.5, Claude Sonnet 4, Claude Sonnet 4.5, GPT-5 Mini, and GPT-5 Mini with reasoning. Each model is tested with four prompts and one initial program, for a total of 28 chains. Further details about the experimental parameters for each model are provided in Appendix B.

5 Results

Across all three experiments, LLM-driven mutation chains without selection pressure exhibit strong convergence, revisiting a small number of programs and structural forms rather than continuing to explore new regions of program space. The extent varies substantially with prompt wording and model choice, but the phenomenon is robust: the majority of chains visit far fewer unique programs and skeletons than the 300-step chain length would permit under standard GP subtree mutation. We first examine sensitivity to prompt design (Section 5.1), then assess stochastic consistency (Section 5.2), compare dynamics across seven LLMs (Section 5.3), and analyze the cyclic structures underlying convergence (Section 5.4).

5.1 Experiment 1: Prompt Sensitivity

Figure 3 summarizes the distribution of cumulative unique programs and unique skeletons across all 150 LLM mutation chains in the prompt sensitivity sweep, grouped by initial program size.

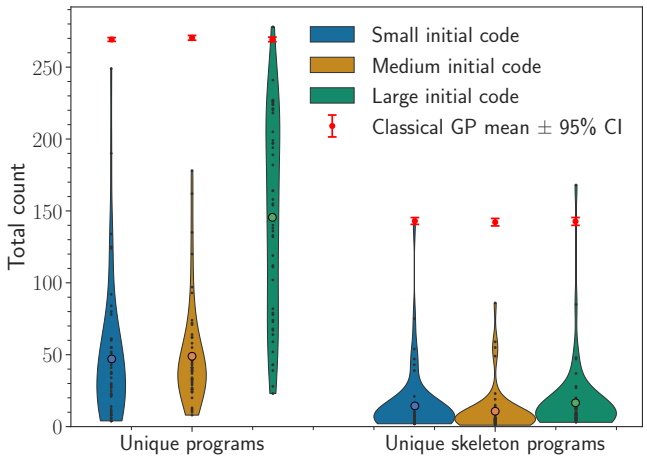


Figure 3: Distribution of cumulative unique programs and unique skeletons across 150 LLM mutation chains (50 prompts), grouped by three initial program sizes, with the classical GP subtree mutation baseline overlaid for comparison.

The wide variation in the resulting distributions demonstrates that the precise wording of the prompt plays a crucial role in the rate of convergence. One of the most convergent prompts produced fewer than 10 unique programs across a 300-step chain, whereas one of the least convergent produced over 250, comparable to the classical GP subtree mutation baseline. Initial program size, by contrast, has only a modest effect: chains initialized from the large program produce somewhat higher unique program counts, but the distributions across initialization sizes overlap substantially.

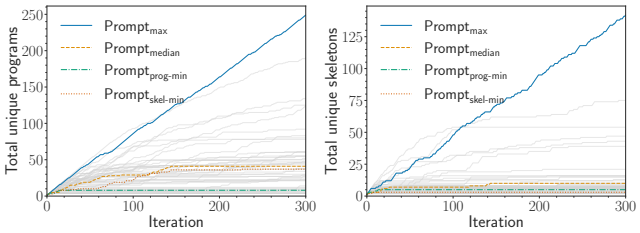
At the skeleton level, convergence is more severe. The median LLM chain visits approximately 10 unique skeletons over 300 steps, indicating that for a wide range of prompts, LLM-generated variation consists primarily of terminal-level substitutions within a small set of recurring structural templates rather than exploration of substantially new program architectures.

The classical GP baseline (overlaid in Figure 3) provides a direct comparison under identical conditions: the same DSL, the same initial programs, and the same chain length. GP chains visit approximately 270 unique programs and 143 unique skeletons per chain. This suggests that the size of the DSL is not a limiting factor: the space of reachable programs is large enough to sustain continued exploration under random subtree mutation. The convergence observed under LLM mutation reflects a property of the operator itself.

While the broader finding that prompt wording affects convergence might have been anticipated, two aspects of the results were unexpected:

(i) The distribution of outcomes across prompts is heavily skewed: only a small fraction of the 50 prompts sustained substantial exploration, while the majority led to rapid convergence in the production of new genetic variants.
(ii) The relationship between prompt wording and convergence behavior is difficult to predict: prompts that appear semantically similar to one another can produce markedly different convergence profiles. See Appendix C.1 for further examples.

Overall, of the 150 LLM-based mutation chains, 71% visit fewer than 100 unique programs and 87% visit fewer than 20 unique skeletons over 300 steps. Put differently, in 87% of chains, over 93% of individual mutations revisit a previously seen structural form. For most of the 50 prompts investigated, repeated LLM-based mutation tends to revisit the same programs (verbatim, or in common structural forms) far more frequently than classical GP subtree mutation.



(a) Programs **(b) Skeletons**

Figure 4: Cumulative unique programs (left) and unique skeletons (right) over 300 iterations for all 50 prompts (shown in faint gray) using the small initial program, with four representative prompts highlighted.

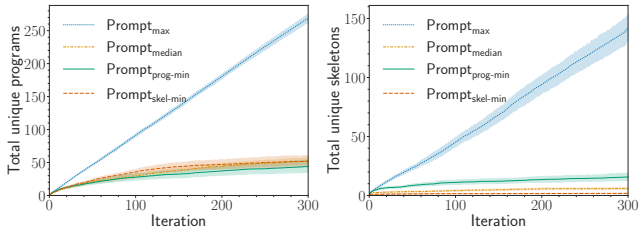
Figure 4 shows cumulative unique counts over 300 iterations for all 50 prompts, with four representative prompts highlighted. These four span the range of observed convergence behavior (Table 3): Prompt$_{max}$ (highest unique counts), Prompt$_{median}$ (median), Prompt$_{prog-min}$ (fewest unique programs), and Prompt$_{skel-min}$ (fewest unique skeletons). Because the latter two do not correlate, both are retained to capture distinct convergence dynamics. Most prompts show a characteristic flattening: unique counts rise in the first 50 to 100 steps before plateauing, while a small set (including Prompt$_{max}$) sustain near-linear growth throughout.

Table 3: Selected prompt instructions used in Experiments 2 and 3.

Label	Instruction
Prompt$_{max}$	`Generate a program that includes exploratory modifications.`
Prompt$_{median}$	`Create a program with minor but meaningful modifications.`
Prompt$_{prog-min}$	`Produce a program that has been slightly restructured.`
Prompt$_{skel-min}$	`Generate a program with a small mutation applied.`

5.2 Experiment 2: Intrinsic Variability

To assess whether the convergence patterns observed in Section 5.1 reflect stable properties of the prompt-model pairing or merely stochastic variation between runs, we replicate 30 independent chains for each of the four representative prompts identified in Table 3.



(a) Programs **(b) Skeletons**

Figure 5: Cumulative unique programs (left) and unique skeletons (right) over 300 iterations, averaged across 30 independent replications per prompt with 95% confidence bands.

Figure 5 plots the cumulative unique program and skeleton counts over 300 iterations, averaged across the 30 replications with 95% confidence bands. The trajectories separate into two distinct regimes. Under Prompt$_{max}$, unique program count grows near-linearly throughout the chain, reaching a mean of 269 unique programs and 140 unique skeletons by step 300. Under the remaining three prompts, the curves flatten within the first 50 to 100 steps: Prompt$_{median}$ reaches 52 unique programs and 6 unique skeletons, Prompt$_{prog-min}$ reaches 44 programs and 16 skeletons, and Prompt$_{skel-min}$ reaches 52 programs but only 1.7 unique skeletons.

The confidence bands are narrow relative to the separation between prompts, confirming that convergence behavior is highly consistent across independent runs of the same prompt. Prompt$_{skel-min}$ is a particularly striking case: across 30 independent chains, the mean number of unique skeletons visited is 1.7, meaning that most chains spend the entire 300-step trajectory cycling through variants of a single structural template.

5.3 Experiment 3: Model Sensitivity

Figure 6 compares cumulative unique programs and skeletons across seven LLMs, each tested with the same four prompts from Table 3. The results reveal substantial variation across model families that persists regardless of prompt.

At one extreme, Claude Sonnet 4 produced as few as 6 unique programs and 1 unique skeleton over 300 steps, collapsing to a single structural form almost immediately regardless of prompt. At the other extreme, GPT-5 Mini with reasoning produced up to 301 unique programs and 225 unique skeletons, sustaining near-complete exploration even under the prompt that produced the strongest convergence in the Gemini-based prompt sweep.

While some models sustain greater program diversity, the majority of the seven models tested still exhibit convergent behavior across most or all prompts. Two further patterns emerge. First, the ranking is largely prompt-invariant: models that converge under Prompt$_{median}$ also converge under Prompt$_{max}$, and models that

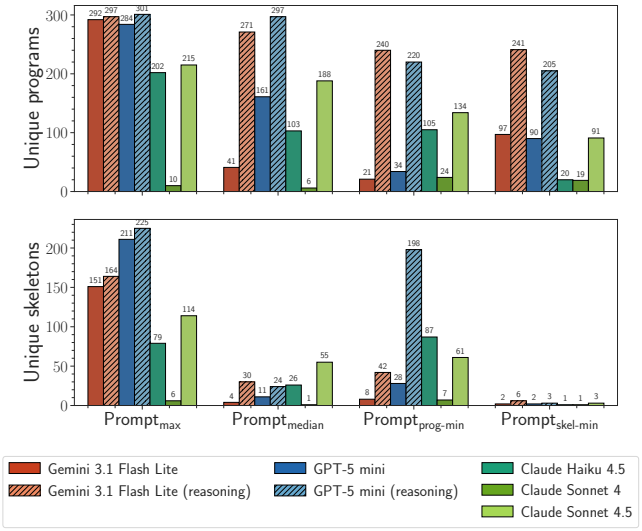
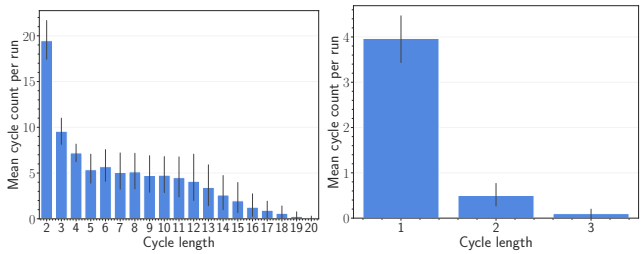


Figure 6: Cumulative unique programs and unique skeletons across seven LLMs, each tested with four representative prompts.



(a) Programs **(b) Skeletons**

Figure 7: Cycle length distributions for Prompt$_{median}$ at the program level (left) and skeleton level (right), averaged over 30 replications.

maintain a high rate of exploration under Prompt$_{max}$ also explore under more convergent prompts. This suggests that the tendency to converge reflects a persistent property of the model rather than an interaction with specific prompt wording. Second, reasoning-enabled variants consistently produce higher diversity than their base counterparts: GPT-5 Mini with reasoning substantially outperforms GPT-5 Mini, and Gemini 3.1 Flash Lite with reasoning outperforms the base Gemini model across all four prompts. Whether this reflects the additional computation afforded by reasoning or a difference in how these models represent and transform code is beyond the scope of this study.

5.4 Attractor Structure and Cycle Analysis

The preceding sections establish that, across a range of prompts, initial programs, and models, LLM mutation chains tend to "converge", visiting a restricted set of programs and skeletons. To characterize the behavior of this convergence, we construct directed transition graphs inspired by [42], where nodes are unique states and directed edges represent observed mutations (an example is shown in Figure 1). We build these graphs from the 30-replication experiment (Section 5.2) and analyze the cycles within them.

Table 4 reports graph-level statistics at the program and skeleton level, averaged over 30 replications per prompt condition.

Short cycles dominate at both levels. At the program level, the minimum cycle length is 2 for three of the four prompts, indicating that 2-cycles are a consistent feature of the transition structure. Mean cycle lengths range from 3.4 to 5.6 across conditions. At the skeleton level, the distribution is more extreme: under the more convergent prompts (Prompt$_{skel-min}$, Prompt$_{median}$), mean cycle lengths are close to 1 (1.04 and 1.10 respectively), meaning nearly all skeleton-level cycles are self-loops. Figure 7 shows these distributions for Prompt$_{median}$.

Degree entropy captures how evenly transitions are distributed across successor states. Convergent prompts show higher program-level degree entropy (0.085–0.104) than Prompt$_{max}$ (0.030), reflecting the fact that chains revisiting a small set of nodes accumulate multiple outgoing edges per node, while exploratory chains visit most nodes only once.

Prompt$_{max}$ is a notable exception at the skeleton level, averaging 73 cycles with a mean length of 3.5 and a maximum of 32.5. This indicates that even chains sustaining structural exploration develop recurrent patterns at longer scales.

Figure 1 shows a representative program-level transition graph from a single chain using Prompt$_{median}$ and the medium initial program (C_{medium}). The trajectory passes through a transient sequence of states before settling into a 2-cycle: two program states that alternate indefinitely, with the majority of the 300 steps spent oscillating between them. This 2-cycle structure is consistent with the minimum cycle lengths reported in Table 4 and parallels the 2-period attractor cycles identified by Wang et al. [45] in successive paraphrasing. The corresponding skeleton-level transition graph for this same run is provided in Appendix C. Pairwise normalized Levenshtein distance heatmaps, which visualize the temporal fine structure of these attractor dynamics at the individual-chain level, are presented in Appendix C.2.

6 Discussion

The results provide clear answers to each of the three research questions posed in Section 1.1. First, LLM-driven mutation chains do converge toward attractor regions: the majority of chains plateau in their cumulative unique counts within the first 50 to 100 steps, and cycle analysis reveals structured revisitation patterns, with 2-cycles dominating at the program level and self-loops dominating at the skeleton level. Second, diversity stagnation is pronounced and operates at two distinct levels: while chains may continue producing lexically distinct programs (especially when mutating larger programs), structural diversity (measured at the skeleton level) stagnates far more severely, with some chains visiting only one or two unique skeletons across 300 steps. Third, these dynamics are sensitive to both prompt design and model choice but robust in aggregate: convergence is the default outcome across the majority of prompts and models tested, with prompt wording and model family modulating the severity but not eliminating the phenomenon.

Table 4: Program and skeleton-level transition graph statistics, averaged over 30 replications per prompt condition. Values are reported as mean ± standard deviation.

Graph Type	Prompt	Mean Cycle Length	Min Cycle Length	Max Cycle Length	Mean Degree Entropy
Program	$\text{Prompt}_{\text{skel-min}}$	3.41 ± 1.92	2.00 ± 0.00	8.87 ± 5.73	0.085 ± 0.019
	$\text{Prompt}_{\text{prog-min}}$	3.42 ± 1.65	1.77 ± 0.43	8.00 ± 5.13	0.104 ± 0.032
	$\text{Prompt}_{\text{median}}$	5.60 ± 2.63	2.00 ± 0.00	13.63 ± 5.49	0.091 ± 0.014
	$\text{Prompt}_{\text{max}}$	4.03 ± 4.34	1.03 ± 0.18	21.33 ± 32.28	0.030 ± 0.001
Skeleton	$\text{Prompt}_{\text{skel-min}}$	1.04 ± 0.09	1.00 ± 0.00	1.13 ± 0.35	0.226 ± 0.198
	$\text{Prompt}_{\text{prog-min}}$	1.71 ± 0.53	1.00 ± 0.00	3.90 ± 2.32	0.122 ± 0.036
	$\text{Prompt}_{\text{median}}$	1.10 ± 0.15	1.00 ± 0.00	1.50 ± 0.68	0.173 ± 0.076
	$\text{Prompt}_{\text{max}}$	3.54 ± 3.28	1.00 ± 0.00	32.50 ± 25.36	0.049 ± 0.009

These findings align with and extend convergence results reported in other iterative LLM domains. The dominance of short cycles in program-level transition graphs parallels the 2-period limit cycles identified by Wang et al. [45] in successive paraphrasing. The collapse toward a small number of recurring structural forms is analogous to the 12 dominant visual motifs observed by Hintze et al. [12] in text-to-image-to-text loops. That these patterns emerge across text, image, and now program space suggests that convergence under iterative application may be a general property of current LLM architectures rather than a domain-specific artifact. Our results also complement the findings of the Digital Red Queen study [17], which reported convergence in behavior space but not in syntactic program space under adversarial co-evolution with selection pressure. In our setting, without selection pressure, convergence manifests directly in the program space itself, suggesting that in some instances selection may mask rather than prevent structural convergence.

For practitioners building systems that employ LLMs as code mutation or generation operators, these results warrant careful attention. The tendency of LLM mutation operators toward structural convergence is more pronounced for smaller models and for models without reasoning capabilities, but it is difficult to predict in advance how a given prompt-model pairing will affect the search space. The finding that prompt design substantially modulates convergence severity is encouraging, but the relationship between prompt wording and exploration behavior can be unintuitive: prompts that appear semantically similar can produce very different convergence profiles. Practitioners may therefore benefit from conducting mutation-chain analysis in the absence of selection pressure to identify prompts that sustain exploration. More broadly, systems that require sustained structural diversity may benefit from explicit diversity maintenance mechanisms, as several recent systems have already recognized. Selection pressure itself may also help drive the LLM beyond attractors in program space, though the interaction between selection and the operator's intrinsic convergence dynamics remains an open question.

Several limitations should be noted. The experiments use a single constrained Lisp-like DSL, and the extent to which these dynamics generalize to richer programming languages, which are more strongly represented in LLM pretraining data, remains an open question. The analysis is purely genotypic: programs that are structurally identical may differ in behavior, and programs that converge structurally may still provide useful behavioral variation. Due to computational constraints, the model sensitivity experiment used only four prompts per model with no replication, limiting the strength of model-level claims. Additionally, for models with high retry rates, the error-correction process may contribute to observed variation, though retries correct syntax errors rather than produce independent mutations.

Future work should extend this analysis to broader program representations and incorporate behavioral evaluation. A particularly promising direction is investigating how diversity preservation mechanisms and selection pressure interact with the operator's intrinsic convergence dynamics, and whether such mechanisms can effectively drive LLM-based mutation beyond the attractor regions identified here. Further formalization of the dynamical systems framework, including characterization of attractor basins, contraction rates, and embedding-space representations of mutation trajectories, may deepen our understanding of the mechanisms driving convergence.

7 Conclusion

This work provides a systematic empirical investigation of LLM-driven mutation as a dynamical process over program space. By studying mutation chains in the absence of selection pressure across 50 prompt variants, seven models, and 30-fold replications, we demonstrate that convergence toward restricted structural regions is a robust property of LLM-based mutation operators in this constrained DSL setting. The rate and severity of convergence depend on prompt design and model choice, but the phenomenon itself is pervasive. These findings have direct implications for any system that relies on iterative LLM-driven code transformation: the mutation operator that enables semantics-aware program transformation also carries a systematic bias toward structural homogeneity that must be understood and accounted for if LLM-driven systems are to sustain the open-ended exploration on which evolutionary computation, automated scientific discovery, and iterative program synthesis increasingly depend.

Acknowledgments

The authors used generative AI tools for coding assistance and to improve the readability of this manuscript.

References

[1] Timothy Atkinson, Detlef Plump, and Susan Stepney. 2018. Evolving Graphs by Graph Programming. In *Genetic Programming*, Mauro Castelli, Lukas Sekanina, Mengjie Zhang, Stefano Cagnoni, and Pablo García-Sánchez (Eds.). Springer International Publishing, Cham, 35–51.

[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] 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

[4] 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

[5] Charles Darwin. 1859. *On the Origin of Species by Means of Natural Selection.* Murray, London. or the Preservation of Favored Races in the Struggle for Life.

[6] 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

[7] Ping Guo, Chao Li, Yinglan Feng, and Chaoning Zhang. 2026. Code Evolution for Control: Synthesizing Policies via LLM-Driven Evolutionary Search. arXiv:2601.06845 [cs.AI] https://arxiv.org/abs/2601.06845

[8] Can Gurkan, Narasimha Karthik Jwalapuram, Kevin Wang, Rudy Danda, Leif Rasmussen, John Chen, and Uri Wilensky. 2025. LEAR: LLM-Driven Evolution of Agent-Based Rules. In *Proceedings of the Genetic and Evolutionary Computation Conference Companion* (NH Malaga Hotel, Malaga, Spain) *(GECCO '25 Companion)*. Association for Computing Machinery, New York, NY, USA, 2309–2326. doi:10.1145/3712255.3734368

[9] 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

[10] 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

[11] 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

[12] Arend Hintze, Frida Proschinger Åström, and Jory Schossau. 2026. Autonomous language-image generation loops converge to generic visual motifs. *Patterns* 7, 1 (09 Jan 2026). doi:10.1016/j.patter.2025.101451

[13] Qinglong Hu and Qingfu Zhang. 2025. Partition to Evolve: Niching-enhanced Evolution with LLMs for Automated Algorithm Discovery. In *The Thirty-ninth Annual Conference on Neural Information Processing Systems.* https://openreview.net/forum?id=OEawM2coNT

[14] 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

[15] 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

[16] 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

[17] Akarsh Kumar, Ryan Bahlous-Boldi, Prafull Sharma, Phillip Isola, Sebastian Risi, Yujin Tang, and David Ha. 2026. Digital Red Queen: Adversarial Program Evolution in Core War with LLMs. arXiv:2601.03335 [cs.AI] https://arxiv.org/abs/2601.03335

[18] 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

[19] Robert Tjarko Lange, Yuki Imajuku, and Edoardo Cetin. 2026. ShinkaEvolve: Towards Open-Ended and Sample-Efficient Program Evolution. In *The Fourteenth International Conference on Learning Representations.* https://openreview.net/forum?id=lKEdGCoDNC

[20] 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

[21] 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.

[22] 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

[23] 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

[24] 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

[25] Aman Madaan, Niket Tandon, Prakhar Gupta, Skyler Hallinan, Luyu Gao, Sarah Wiegreffe, Uri Alon, Nouha Dziri, Shrimai Prabhumoye, Yiming Yang, Shashank Gupta, Bodhisattwa Prasad Majumder, Katherine Hermann, Sean Welleck, Amir Yazdanbakhsh, and Peter Clark. 2023. Self-Refine: Iterative Refinement with Self-Feedback. In *Thirty-seventh Conference on Neural Information Processing Systems.* https://openreview.net/forum?id=S37hOerQLB

[26] 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

[27] Amr Mohamed, Mingmeng Geng, Michalis Vazirgiannis, and Guokan Shang. 2025. LLM as a Broken Telephone: Iterative Generation Distorts Information. In *Proceedings of the 63rd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, Wanxiang Che, Joyce Nabende, Ekaterina Shutova, and Mohammad Taher Pilehvar (Eds.). Association for Computational Linguistics, Vienna, Austria, 7493–7509. doi:10.18653/v1/2025.acl-long.371

[28] Sabbir Mollah, Rohit Gupta, Sirnam Swetha, Qingyang Liu, Ahnaf Munir, and Mubarak Shah. 2025. The Telephone Game: Evaluating Semantic Drift in Unified Models. arXiv:2509.04438 [cs.CV] https://arxiv.org/abs/2509.04438

[29] Alexander Novikov, Ngân Vu, Marvin Eisenberger, Emilien Dupont, Po-Sen Huang, Adam Zsolt Wagner, Sergey Shirobokov, Borislav Kozlovskii, Francisco J. R. Ruiz, Abbas Mehrabian, M. Pawan Kumar, Abigail See, Swarat Chaudhuri, George Holland, Alex Davies, Sebastian Nowozin, Pushmeet Kohli, and Matej Balog. 2025. AlphaEvolve: A coding agent for scientific and algorithmic discovery. *CoRR* abs/2506.13131 (2025). arXiv:2506.13131 doi:10.48550/ARXIV.2506.13131

[30] U.-M. O'Reilly. 1997. Using a distance metric on genetic programs to understand genetic operators. In *1997 IEEE International Conference on Systems, Man, and Cybernetics. Computational Cybernetics and Simulation*, Vol. 5. 4092–4097 vol.5. doi:10.1109/ICSMC.1997.637337

[31] Norman Peitek, Julia Hess, and Sven Apel. 2026. From Restructuring to Stabilization: A Large-Scale Experiment on Iterative Code Readability Refactoring with Large Language Models. arXiv:2602.21833 [cs.SE] https://arxiv.org/abs/2602.21833

[32] Jeremy Perez, Corentin Leger, Marcela Ovando-Tellez, Chris Foulon, Joan Dussauld, Pierre-Yves Oudeyer, and Clement Moulin-Frier. 2024. Cultural evolution in populations of Large Language Models. arXiv:2403.08882 [cs.MA] https://arxiv.org/abs/2403.08882

[33] Riccardo Poli, William B. Langdon, and Nicholas Freitag McPhee. 2008. *A Field Guide to Genetic Programming.* Lulu Press. http://www.gp-field-guide.org.uk With contributions by John R. Koza.

[34] 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

[35] Vinu Sankar Sadasivan, Aounon Kumar, Sriram Balasubramanian, Wenxiao Wang, and Soheil Feizi. 2024. Can AI-Generated Text be Reliably Detected? https://openreview.net/forum?id=NvSwR4IvLO

[36] Ilia Shumailov, Zakhar Shumaylov, Yiren Zhao, Yarin Gal, Nicolas Papernot, and Ross Anderson. 2024. The Curse of Recursion: Training on Generated Data Makes Models Forget. arXiv:2305.17493 [cs.LG] https://arxiv.org/abs/2305.17493

[37] Ilia Shumailov, Zakhar Shumaylov, Yiren Zhao, Nicolas Papernot, Ross Anderson, and Yarin Gal. 2024. AI models collapse when trained on recursively generated data. *Nature* 631, 8022 (01 Jul 2024), 755–759. doi:10.1038/s41586-024-07566-y

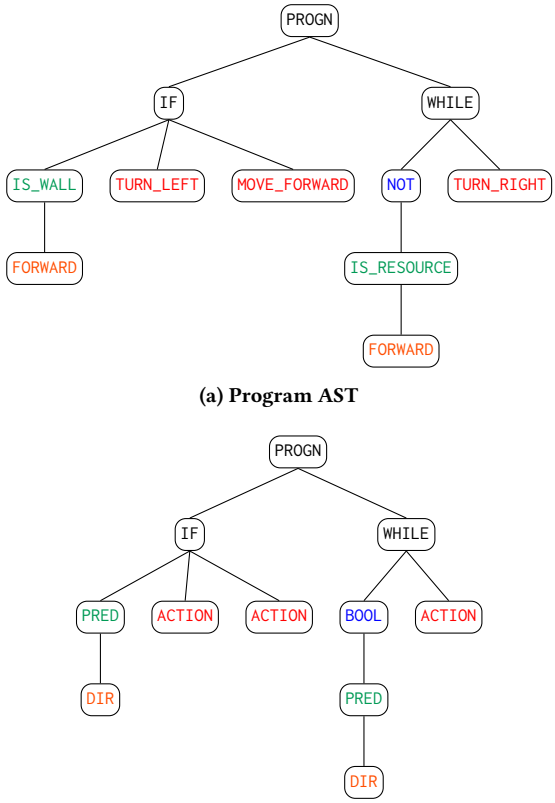
[38] Xingyou Song, Yingtao Tian, Robert Tjarko Lange, Chansoo Lee, Yujin Tang, and Yutian Chen. 2024. Position: leverage foundational models for black-box optimization. In *Proceedings of the 41st International Conference on Machine Learning* (Vienna, Austria) *(ICML'24)*. JMLR.org, Article 1878, 13 pages.
[39] Niki van Stein and Thomas Bäck. 2025. LLaMEA: A Large Language Model Evolutionary Algorithm for Automatically Generating Metaheuristics. *IEEE Transactions on Evolutionary Computation* 29, 2 (2025), 331–345. doi:10.1109/TEVC.2024.3497793
[40] Nicolas Tacheny. 2026. Geometric Dynamics of Agentic Loops in Large Language Models. arXiv:2512.10350 [cs.LG] https://arxiv.org/abs/2512.10350
[41] Nafis Irtiza Tripto, Saranya Venkatraman, Dominik Macko, Robert Moro, Ivan Srba, Adaku Uchendu, Thai Le, and Dongwon Lee. 2024. A Ship of Theseus: Curious Cases of Paraphrasing in LLM-Generated Texts. In *Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, Lun-Wei Ku, Andre Martins, and Vivek Srikumar (Eds.). Association for Computational Linguistics, Bangkok, Thailand, 6608–6625. doi:10.18653/v1/2024.acl-long.357
[42] Niki van Stein, Anna V. Kononova, Lars Kotthoff, and Thomas Bäck. 2025. Code Evolution Graphs: Understanding Large Language Model Driven Design of Algorithms. In *Proceedings of the Genetic and Evolutionary Computation Conference* (NH Malaga Hotel, Malaga, Spain) *(GECCO '25)*. Association for Computing Machinery, New York, NY, USA, 943–951. doi:10.1145/3712256.3726328
[43] Niki van Stein, Haoran Yin, Anna V. Kononova, Thomas Bäck, and Gabriela Ochoa. 2026. Behaviour Space Analysis of LLM-Driven Meta-Heuristic Discovery. In *Computational Intelligence*, Francesco Marcelloni, Kurosh Madani, Niki van Stein, and Joaquim Filipe (Eds.). Springer Nature Switzerland, Cham, 367–385.
[44] Guanzhi Wang, Yuqi Xie, Yunfan Jiang, Ajay Mandlekar, Chaowei Xiao, Yuke Zhu, Linxi Fan, and Anima Anandkumar. 2024. Voyager: An Open-Ended Embodied Agent with Large Language Models. *Transactions on Machine Learning Research* (2024). https://openreview.net/forum?id=ehfRiF0R3a
[45] Zhilin Wang, Yafu Li, Jianhao Yan, Yu Cheng, and Yue Zhang. 2025. Unveiling Attractor Cycles in Large Language Models: A Dynamical Systems View of Successive Paraphrasing. In *Proceedings of the 63rd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, Wanxiang Che, Joyce Nabende, Ekaterina Shutova, and Mohammad Taher Pilehvar (Eds.). Association for Computational Linguistics, Vienna, Austria, 12740–12755. doi:10.18653/v1/2025.acl-long.624
[46] Jason Wei, Xuezhi Wang, Dale Schuurmans, Maarten Bosma, Brian Ichter, Fei Xia, Ed H. Chi, Quoc V. Le, and Denny Zhou. 2022. Chain-of-thought prompting elicits reasoning in large language models. In *Proceedings of the 36th International Conference on Neural Information Processing Systems* (New Orleans, LA, USA) *(NIPS '22)*. Curran Associates Inc., Red Hook, NY, USA, Article 1800, 14 pages.
[47] 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
[48] Shunyu Yao, Jeffrey Zhao, Dian Yu, Nan Du, Izhak Shafran, Karthik R Narasimhan, and Yuan Cao. 2023. ReAct: Synergizing Reasoning and Acting in Language Models. In *The Eleventh International Conference on Learning Representations*. https://openreview.net/forum?id=WE_vluYUL-X
[49] Haoran Ye, Jiarui Wang, Zhiguang Cao, Federico Berto, Chuanbo Hua, Haeyeon Kim, Jinkyoo Park, and Guojie Song. 2024. ReEvo: Large Language Models as Hyper-Heuristics with Reflective Evolution. 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., 43571–43608. doi:10.52202/079017-1381
[50] Tina Yu and Julian F. Miller. 2001. Neutrality and the Evolvability of Boolean Function Landscape. In *Genetic Programming, 4th European Conference, EuroGP 2001, Lake Como, Italy, April 18-20, 2001, Proceedings (Lecture Notes in Computer Science)*, Julian F. Miller, Marco Tomassini, Pier Luca Lanzi, Conor Ryan, Andrea Tettamanzi, and William B. Langdon (Eds.). Springer, 204–217. doi:10.1007/3-540-45355-5_16

A Prompt and Representation Details

This section documents the program representations and prompt infrastructure used throughout the experiments.

A.1 AST Representations

Figure 8 illustrates the abstract syntax tree (AST) representations used in the analysis. The full program AST preserves all terminal symbols, while the skeleton AST replaces terminals with their type categories (`ACTION`, `DIR`, `PRED`, `BOOL`), retaining only control-flow structure. This dual representation underlies the distinction between program-level and skeleton-level convergence reported in the main text.



(a) Program AST

(b) Skeleton AST

Figure 8: AST representation of example program in the DSL (top) and its corresponding skeleton representation (bottom).

A.2 Mutation Prompt Templates

Each mutation step sends the LLM a single prompt containing the DSL constraints, the current parent program, and an instruction line directing the model to produce a mutated variant.

The base mutation prompt template is:

```
You are performing mutation on a genetic program for a
grid-world agent.

Shared constraints:
<shared_info>

Parent program:
<parent_program>

Task:
- <variant_instruction>
- The produced program must be syntactically valid.
- Use only the allowed primitives and directions above.
```

```
- The program must include at least one action leaf (MOVE_FORWARD, TURN_LEFT, TURN_RIGHT).
- Do not include trailing commas.
- Output only the mutated program string.
```

The *variant_instruction* placeholder is the only component that varies across the 50 prompt conditions used in Experiment 1; the full list of instruction variants is provided in Appendix A.3.

The *shared_info* block specifies the allowed DSL primitives and syntax rules. For the no_memory primitive configuration used in all experiments, it renders as:

```
Syntax: use uppercase function names with parentheses, e.g. IF(cond,then,else).
Each program must be a single expression. Output only the program string.

Allowed statements:
- IF(cond, then, else) - conditional branching.
- NO_OP - no operation (returns None).
- PROGN(stmt1, stmt2) - sequential evaluation.
- WHILE(cond, body) - loop with max-iteration guard.

Allowed action leaves:
- MOVE_FORWARD - action leaf.
- TURN_LEFT - action leaf.
- TURN_RIGHT - action leaf.

Allowed predicates:
- IS_EMPTY(DIR) - True if empty cell in DIR.
- IS_HAZARD(DIR) - True if a hazard in DIR.
- IS_RESOURCE(DIR) - True if a resource in DIR.
- IS_WALL(DIR) - True if a wall in DIR.

Allowed boolean ops:
- AND(a, b) - boolean conjunction.
- NOT(a) - boolean negation.
- OR(a, b) - boolean disjunction.

Allowed directions:
- FORWARD - relative direction.
- LEFT - relative direction.
- RIGHT - relative direction.
```

When a candidate program fails validation (e.g., uses a disallowed primitive or is syntactically malformed), the model is re-prompted with the following retry template:

```
You are fixing a previously generated program that failed validation.

Use the allowed primitives and constraints below. Return only the corrected program string.

Failure reason:
<failure_reason>

Program to fix:
<candidate_text>

Constraints and allowed primitives:
<shared_info>
```

A.3 Prompt Instruction Variants

The 50 mutation instruction variants used in Experiment 1 (Section 4.1) were generated by prompting ChatGPT to produce diverse paraphrasings of the core mutation command. Each variant replaces the *variant_instruction* placeholder in the base prompt template (Appendix A.2); all other prompt components remain identical across conditions. Table 6 lists all 50 variants. The four variants selected for Experiments 2 and 3 are indicated in bold, chosen based on their position in the distribution of unique program counts from Experiment 1: $\text{Prompt}_{\text{max}}$ produced the most unique programs, $\text{Prompt}_{\text{median}}$ fell at the median, $\text{Prompt}_{\text{prog-min}}$ produced the fewest unique programs, and $\text{Prompt}_{\text{skel-min}}$ produced the fewest unique skeletons. Below is the prompt that was used to produce the prompt variations.

```
You will be generating 50 different variations of a specific instruction line that is used in genetic programming mutation operations. Each variation should convey the same core meaning but use different wording, phrasing, and terminology.

Here is the original instruction line you need to create variations of:

<original_line>
Produce a mutated program
</original_line>

Your task is to create 50 distinct variations of this instruction line. Follow these requirements carefully:

**Core Requirements:**
1. Each variation must convey the same fundamental meaning and instruction as the original line
2. Each variation must use different wording and phrasing from the original and from other variations
3. Each variation must maintain a clear, instructional tone appropriate for prompting an LLM
4. Each variation must be suitable for use in genetic programming mutation contexts
5. Each variation should be a complete, standalone instruction that could directly replace the original line

**Vocabulary and Terminology:**
- Vary the terminology you use across variations. Instead of always using "mutation," consider alternatives such as:
  - "change"
  - "variation"
  - "modification"
```

```
  - "improvement"
  - "alteration"
  - "transformation"
  - "adjustment"
- Mix these terms across your variations to create
lexical diversity

**Change Characterization:**
- Different variations should characterize the type
or scope of changes differently. Consider including
descriptors such as:
  - "small changes" or "minor modifications"
  - "creative changes" or "innovative variations"
 - "substantial improvements" or "significant alterations"
  - "incremental adjustments"
  - "exploratory modifications"
- Not every variation needs to specify the type of
change, but several should to add useful diversity

**Output Format:**
After your planning, provide your 50 variations, each
enclosed in numbered XML tags as follows:

<variation_1>
[First variation here]
</variation_1>

<variation_2>
[Second variation here]
</variation_2>

[Continue through variation_50]
```

Table 5: Labels of chosen representative prompts and corresponding IDs

Prompt Label	Prompt ID
$\text{Prompt}_{\text{max}}$	27
$\text{Prompt}_{\text{median}}$	43
$\text{Prompt}_{\text{prog-min}}$	47
$\text{Prompt}_{\text{skel-min}}$	21

B Experimental Parameters

This section provides the initial programs and per-model configurations referenced in the experimental design.

B.1 Initial Programs

Three initial programs of varying complexity were generated using ramped half-and-half initialization with different random seeds. C_{medium} serves as the standard initial program across all three experiments; Experiment 1 additionally uses C_{small} and C_{large} to assess sensitivity to initial program size.

```
MOVE_FORWARD
```

(a) C_{small} (small initial program, seed 0).

```
WHILE(IS_EMPTY(LEFT),
  PROGN(
    PROGN(PROGN(MOVE_FORWARD, TURN_LEFT),
        TURN_LEFT),
    IF(AND(IS_RESOURCE(LEFT), IS_WALL(RIGHT))
        ,
       WHILE(IS_EMPTY(FORWARD), TURN_LEFT),
       IF(IS_EMPTY(FORWARD), TURN_RIGHT,
           MOVE_FORWARD))))
```

(b) C_{medium} (medium initial program, seed 5).

```
PROGN(
  IF(OR(OR(IS_HAZARD(RIGHT), IS_WALL(LEFT)),
        AND(IS_EMPTY(RIGHT), IS_HAZARD(LEFT))
           ),
     IF(OR(IS_HAZARD(RIGHT), IS_HAZARD(
         FORWARD)),
        WHILE(IS_HAZARD(LEFT), NO_OP),
        IF(IS_HAZARD(LEFT), MOVE_FORWARD,
            TURN_RIGHT)),
     PROGN(PROGN(TURN_RIGHT, TURN_LEFT),
           PROGN(TURN_LEFT, TURN_RIGHT))),
  IF(AND(OR(IS_RESOURCE(FORWARD), IS_EMPTY(
      FORWARD)),
         OR(IS_EMPTY(RIGHT), IS_WALL(RIGHT)))
           ,
     PROGN(PROGN(NO_OP, TURN_LEFT), TURN_LEFT
         ),
     PROGN(IF(IS_RESOURCE(RIGHT),
         MOVE_FORWARD, TURN_RIGHT),
           IF(IS_WALL(RIGHT), MOVE_FORWARD,
               MOVE_FORWARD))))
```

(c) C_{large} (large initial program, seed 10).

Figure 9: Initial programs used in the experiments, generated via ramped half-and-half initialization with varying random seeds.

B.2 Model Configurations

Table 7 lists the per-model parameters for Experiment 3 (Section 4.3). Each model was tested with a primary provider and a fallback provider invoked when the primary exhausts its retry budget on a given mutation step. All models share the parameters in Table 2 unless noted otherwise.

C Additional Results

This section presents supplementary analyses that support the findings in the main text.

Figure 10 plots total unique programs against total unique skeletons for each of the 120 individual chains. The endpoints of the trajectories stemming from these four prompts land in somewhat

Table 6: All 50 mutation instruction variants. Variants selected for Experiments 2 and 3 are shown in bold with their paper labels.

ID	Instruction	ID	Instruction
1	Generate a modified version of the program.	26	Produce a new program instance with minor alterations.
2	Create a new program that incorporates a mutation.	**27**	**Generate a program that includes exploratory modifications.**
3	Produce a slightly altered version of the given program.	28	Create a transformed variant of the program with adjustments.
4	Construct a variation of the program with minor changes.	29	Produce a program that has undergone a mutation operation.
5	Generate a transformed version of the program.	30	Generate a revised program incorporating small changes.
6	Create a program that results from mutating the original code.	31	Create a modified program with slight structural differences.
7	Produce an adjusted version of the program with incremental changes.	32	Produce a variation of the original program through alteration.
8	Generate a creatively modified program derived from the original.	33	Generate a program that reflects a subtle transformation.
9	Construct a revised version of the program with exploratory alterations.	34	Create a program with a controlled degree of modification.
10	Create a new variant of the program through modification.	35	Produce a program variant obtained via mutation.
11	Produce a slightly changed program based on the original.	36	Generate a modified program instance with incremental adjustments.
12	Generate a program that reflects a mutation of the input program.	37	Create a program that introduces small changes to the original.
13	Create an altered version of the program with small adjustments.	38	Produce a new program with slight deviations from the original.
14	Produce a modified program that differs from the original.	39	Generate a program reflecting a creative alteration of the original.
15	Generate a program variant that introduces a change to the existing structure.	40	Create a variation of the program with deliberate modifications.
16	Create a transformed program by applying a mutation to the original code.	41	Produce a program that has been adjusted through mutation.
17	Produce a revised program containing minor modifications.	42	Generate a program variant with refined changes.
18	Generate an alternative version of the program with structural changes.	**43**	**Create a program with minor but meaningful modifications.**
19	Create a program variation that modifies the original implementation.	44	Produce a transformed program reflecting a mutation step.
20	Produce a new version of the program that includes an alteration.	45	Generate a program that includes a set of controlled alterations.
21	**Generate a program with a small mutation applied.**	46	Create a modified version of the original program with small improvements.
22	Create a slightly modified instance of the program.	**47**	**Produce a program that has been slightly restructured.**
23	Produce a program that has been altered from its original form.	48	Generate a variant of the program through systematic modification.
24	Generate a variation of the program with controlled changes.	49	Create a program that reflects a mutation-driven change.
25	Create a program reflecting an incremental mutation step.	50	Produce a program incorporating an intentional alteration.

Table 7: Per-model experimental parameters for Experiment 3. All models use temperature 1.0. The "reasoning" variants enable provider-specific reasoning modes (thinking level medium for Gemini, reasoning effort medium for OpenAI). †GPT-5 Mini (non-reasoning) uses an elevated retry budget of 15 and its fallback uses temperature 0.5 with reasoning effort medium to be able to complete chains of 300 iterations without failing.

Model	Provider	Model ID	Tokens	Retries	Fallback
Gemini 3.1 Flash Lite	Google	`gemini-3.1-flash-lite-preview`	1024	5	Gemini 3.1 Flash
Gemini 3.1 Flash Lite (reasoning)	Google	`gemini-3.1-flash-lite-preview`	8192	5	Gemini 3.1 Flash
Claude Haiku 4.5	Anthropic	`claude-haiku-4-5-20251001`	1024	5	Claude Sonnet 4
Claude Sonnet 4	Anthropic	`claude-sonnet-4-20250514`	1024	5	Claude Sonnet 4
Claude Sonnet 4.5	Anthropic	`claude-sonnet-4-5-20250929`	1024	5	Claude Sonnet 4.5
GPT-5 Mini†	OpenAI	`gpt-5-mini-2025-08-07`	4096	15	GPT-5.1
GPT-5 Mini (reasoning)	OpenAI	`gpt-5-mini-2025-08-07`	12000	5	GPT-5.1

distinct clusters, with $\text{Prompt}_{\text{max}}$ occupying the upper-right region (high diversity on both axes) and the remaining prompts compressed into the lower-left. This separation reinforces the finding from Section 5.1: prompt wording determines not just the quantity of exploration but its character, controlling whether variation extends to program structure or remains confined primarily to terminal substitutions.

C.1 Prompt Sweep Full Results

Table 8 reports cumulative unique program and skeleton counts at step 300 for all 50 prompt variants and all three initial programs.

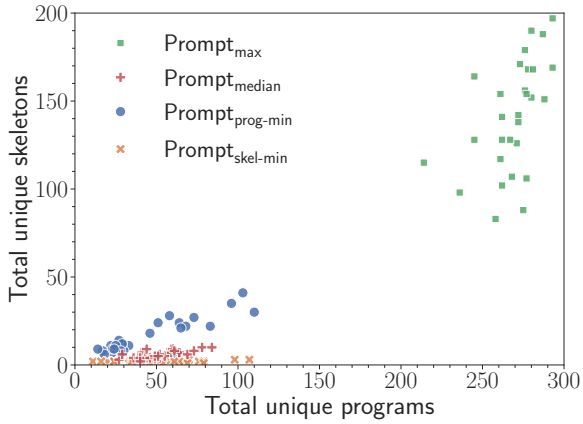


Figure 10: Total unique programs versus total unique skeletons for each of the 120 individual chains in Experiment 2, colored by prompt condition.

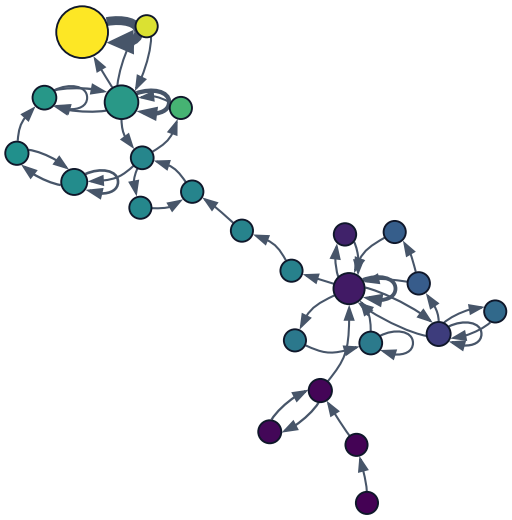


Figure 11: Skeleton-level transition graph corresponding to the same chain shown in Figure 1 (Prompt$_{median}$, C_{medium}). At the skeleton level, the graph collapses to fewer nodes, with self-loops dominating the structure. Compare with the program-level graph in Figure 1, where the same chain produces a richer set of distinct states but still converges to a 2-cycle.

Prompt variants selected for Experiments 2 and 3 are shown in bold.

Several entries in Table 8 illustrate how prompts that read as semantically similar can produce markedly different convergence profiles. Five prompts that request small changes using "slightly," "minor," or "small" (variants 3, 4, 11, 13, and 17) each produce exactly 4 unique programs on C_s, while prompt 43 ("Create a program with minor but meaningful modifications") produces 41; the qualifier "but meaningful" appears to shift behavior dramatically despite preserving the surface emphasis on small change. Similarly, prompts 15 ("introduces a change to the existing structure") and 18 ("with structural changes") both explicitly direct the operator toward structural variation, yet on C_s produce 28 and 124 unique programs respectively.

Table 8: Cumulative unique programs and unique skeletons at step 300 for all 50 prompt variants across three initial programs. Prompt variants selected for Experiments 2 and 3 are shown in bold.

	C_{small}		C_{medium}		C_{large}	
ID	Prog.	Skel.	Prog.	Skel.	Prog.	Skel.
1	43	10	29	9	67	17
2	11	6	72	8	197	10
3	4	2	41	3	221	5
4	4	2	24	5	155	5
5	29	12	40	11	74	18
6	51	8	43	5	164	12
7	21	7	12	3	111	12
8	190	75	178	59	218	85
9	125	39	93	23	112	37
10	38	13	30	9	43	17
11	4	2	33	3	221	5
12	60	9	41	4	226	12
13	4	2	10	2	241	7
14	14	5	13	2	146	6
15	28	7	26	4	227	7
16	25	7	26	3	199	10
17	4	2	11	2	224	4
18	124	47	135	55	133	48
19	48	7	25	4	43	8
20	22	6	47	5	154	5
21	**37**	**3**	**33**	**1**	**224**	**3**
22	6	3	41	2	182	5
23	23	7	39	6	132	6
24	46	7	38	5	78	8
25	10	4	37	4	189	12
26	6	3	62	2	225	6
27	**249**	**142**	**162**	**86**	**278**	**168**
28	37	7	20	6	52	13
29	55	8	45	3	158	20
30	12	3	8	4	205	6
31	47	15	26	4	73	27
32	61	6	27	3	226	8
33	8	4	71	4	82	5
34	30	5	62	5	194	13
35	84	10	24	4	220	9
36	27	8	46	6	79	12
37	22	6	64	5	227	4
38	9	4	32	2	164	6
39	134	43	120	49	140	47
40	55	15	33	11	23	9
41	61	8	97	14	138	9
42	55	16	44	12	39	13
43	**41**	**10**	**50**	**7**	**119**	**4**
44	45	7	34	6	68	9
45	92	21	74	9	197	8
46	78	54	31	15	64	28
47	**8**	**5**	**55**	**19**	**28**	**8**
48	52	7	46	5	102	14
49	80	16	58	11	59	10
50	34	6	39	6	136	7

C.2 Pairwise Levenshtein Distance Heatmaps

To visualize the temporal structure of convergence at the individual-chain level, we compute pairwise normalized token-level Levenshtein distances between all programs at each pair of steps in a

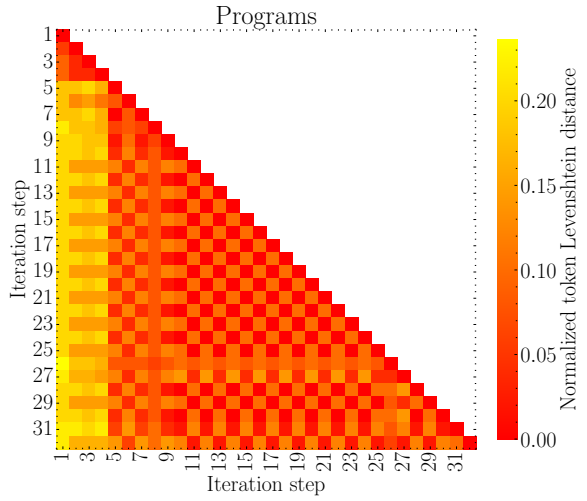


Figure 12: Pairwise normalized Levenshtein distance heatmap for a representative mutation chain, zoomed to the first 32 iterations. The checkered pattern along the diagonal indicates short-period cycling (primarily 2-cycles), while uniform dark blocks indicate intervals of stagnation where the chain revisited the same program repeatedly.

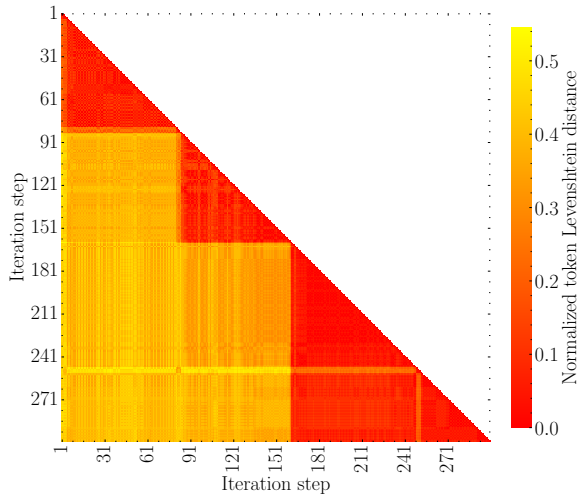


Figure 13: Full 300-iteration Levenshtein distance heatmap for the same chain as Figure 12. Large dark blocks correspond to extended periods of stagnation, and the lighter off-diagonal rectangles between blocks mark transitions where the chain jumped to a structurally different program. The overall structure shows that convergence proceeds through discrete attractor-hopping rather than gradual contraction.

chain. Each heatmap is lower-triangular with iteration steps on both axes. Red regions indicate low distance between programs at those steps (revisitation of similar forms), while yellow regions indicate greater divergence.

Two recurring visual motifs are apparent across chains. The first is a *checkered pattern*: alternating bands of low and high distance that produce a grid-like texture along the diagonal. This pattern arises when a chain oscillates between two (or a small number of) programs, so that even-numbered iterations are similar to one another and odd-numbered iterations form a separate cluster. The resulting structure is the spatial signature of short limit cycles, most commonly 2-cycles. Wang et al. [45] document an analogous pattern in their difference confusion matrices for successive paraphrasing, where alternating light and dark bands reveal 2-period attractor cycles in natural-language trajectories. The checkered regions in our heatmaps provide direct visual evidence that the same periodic dynamics occur in LLM-driven mutation over program space, not only in textual paraphrasing.

The second motif consists of *rectangular blocks* of uniform color. A dark (red) rectangle spanning iterations i through j on both axes indicates that the chain revisited the same program or a small set of nearly identical programs throughout that interval, a period of stagnation in which mutation produced no effective change. Conversely, a lighter (yellow) off-diagonal rectangle between two such blocks indicates a transition: the chain jumped from one attractor region to another, producing programs that differ substantially from those in the preceding block. Together, these block structures reveal that convergence is not always a smooth process. Chains can remain trapped in a narrow region of program space for dozens of iterations before abruptly shifting to a different region, only to stagnate again. The heatmaps thus complement the cumulative unique-count curves presented in Section 5.1 by exposing the temporal fine structure that aggregate statistics obscure.

Figure 12 shows a representative 32-iteration segment at high resolution, making the checkered and block structures clearly visible. Figure 13 shows the same chain over the full 300 iterations, where the large-scale block structure and transitions between attractor regions are prominent. Figure 14 presents nine independent chains side by side, illustrating both the consistency of these patterns across replications and the variation in block size and transition frequency.

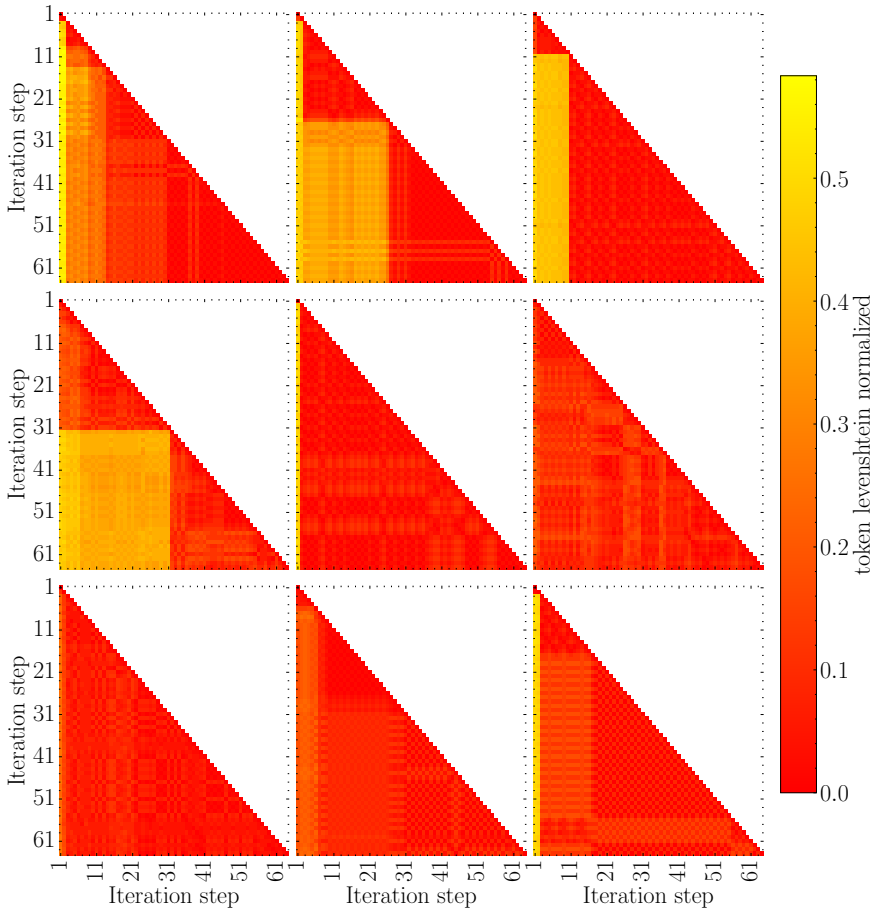


Figure 14: Levenshtein distance heatmaps for nine independent mutation chains (first 64 iterations each). Checkered patterns and block structures appear consistently across replications, though the specific block sizes and transition points vary. Some chains (e.g., top-left) enter stagnation almost immediately, while others (e.g., bottom-right) sustain more varied exploration before settling.