International Journal of Advances in Applied Computational Intelligence

♦ ISSN: 2833-5600 Vol: 08, No. 02, 2026 ♦ PP. 11–18

Article DOI: https://doi.org/10.54216/IJAACI.080202

FULL LENGTH ARTICLE

Algorithmic Hallucination in LLM-Generated Metaheuristics: A Reproducibility, Constraint-Violation, and Failure-Mode Audit

Safina Shokeen[1,*] **Vishal Srivastava**[1]

[1] Department of Industrial Internet of Things, School of Engineering and Technology, Vivekananda Institute of Professional Studies–Technical Campus, Delhi 110034, India

Emails: safina.shokeen@vips.edu · vishal.srivastava@vips.edu

Received: March 01, 2026 Revised: May 04, 2026 Accepted: July 6, 2026 ⋆ Corresponding author

ABSTRACT

Large language models now routinely produce metaheuristic optimisation code in response to natural-language prompts, yet the structural integrity of such generated implementations has received little systematic scrutiny. Syntactically valid code that misrepresents algorithmic logic or parameter semantics constitutes a form of hallucination specific to optimisation software — one that is difficult to detect without execution-level testing and that can produce catastrophic performance degradation without any visible error signal. This work introduces a five-category taxonomy of algorithmic hallucination covering parameter value hallucination, update-formula omission, boundary-handling failure, reproducibility failure, and termination-logic error. A controlled audit is conducted by injecting each hallucination type into verified reference implementations of three widely used metaheuristics — particle swarm optimisation, differential evolution, and a genetic algorithm — and evaluating their behaviour across five standard continuous benchmark functions. Across twenty independent trials per condition, parameter hallucination and formula-omission errors produce best-fitness degradation exceeding 10^8-fold relative to the reference, while boundary-handling failures generate more than 4 000 out-of-bounds constraint violations per trial. Reproducibility failure inflates inter-trial variance by three to four orders of magnitude, and premature-termination errors induce a 100 % trial failure rate. The results demonstrate that current LLM-generated metaheuristic code requires structured execution-level validation before deployment, and a practical audit protocol is proposed to support that process. Experimental artefacts, including all source code, generated data, and benchmark results, are released to support reproducible follow-on research.

Keywords: Large language models ▪ Algorithmic hallucination ▪ Metaheuristics ▪ Particle swarm optimisation ▪ Differential evolution ▪ Code correctness ▪ Reproducibility ▪ Constraint violation

1. INTRODUCTION

The capacity of large language models (LLMs) to produce executable code from natural-language descriptions has substantially expanded the practical reach of computational optimisation. Researchers without deep familiarity with particle swarm optimisation, differential evolution, or genetic algorithms can now prompt a model to generate working implementations in seconds [1, 2]. This accessibility comes at a cost that has been underappreciated in the literature: LLMs generate code that is syntactically plausible and will often execute without error while silently embodying incorrect algorithmic logic.

In the natural language processing domain, hallucination refers to model outputs that are fluent and coherent but factually unsupported or incorrect [3, 4]. The phenomenon has

distinct technical expression in code generation: an LLM may produce a particle swarm optimiser with an inertia weight of 1.8 — a value that guarantees velocity explosion and divergence in theory — because 1.8 superficially resembles the canonical range of cognitive and social learning rates. The code passes any syntax check, may even terminate within a time budget, and yet cannot converge. Tonmoy et al. [5] review over 32 hallucination-mitigation strategies across NLP tasks but none addresses the specific failure modes that arise in algorithm implementation.

The literature on LLMs for algorithm design has grown rapidly. FunSearch [6] demonstrated that coupling an LLM with a systematic evaluator enables discovery of new mathematical results, while OPRO [7] showed that LLMs can serve as black-box optimisers in their own right. EoH [8] and ReEvo [9] extend this paradigm to automatic heuristic design through evolutionary prompt strategies. LLMs have also been applied as direct participants in evolutionary computation frameworks [10, 11]. A systematic survey by Liu et al. [12] catalogues 35 such approaches across combinatorial and continuous domains.

Yet none of this body of work conducts a structured audit of what goes wrong when generated code is incorrect. Chen et al. [13] noted in passing that some LLM-generated neural architecture components fail at runtime, and Liu et al. [14] reported that LLM-proposed operators for evolutionary computation must be validated before use. Austin et al. [15] evaluated LLM program synthesis on a held-out benchmark but did not consider domain-specific failure semantics. The reproducibility dimension is equally under-studied: Pineau et al. [16] established the computational reproducibility checklist now adopted by major machine learning venues, but its application to LLM-generated optimisation code has not been assessed.

This work fills that gap through three contributions. First, it proposes a five-category taxonomy of algorithmic hallucination grounded in the semantic requirements of iterative metaheuristic algorithms. Second, it conducts a controlled empirical audit in which each hallucination type is injected into reference implementations and evaluated across a standardised five-function benchmark suite using twenty independent trials. Third, it reports genuinely computed performance metrics — fitness ratios, violation counts, variance measures, and failure rates — whose magnitudes reveal the quantitative severity of each hallucination category. An audit protocol derived from these findings is proposed to support systematic validation before deployment.

2. BACKGROUND AND RELATED WORK

2.1 Metaheuristic Optimisation Algorithms

Metaheuristic algorithms are stochastic population-based methods designed to approximate solutions to continuous and combinatorial optimisation problems for which exact methods are computationally intractable [17]. Three families are central to this study.

Particle Swarm Optimisation (PSO) maintains a population of candidate solutions (particles), each updated via a velocity equation that combines an inertia component (weighted by w), a cognitive component attracting each particle towards its personal best, and a social component attracting it towards the global best. Stability depends jointly on the inertia and acceleration coefficients; inertia below unity is commonly used to attenuate velocities, while unstable parameter combinations can cause particle explosion and divergence [18]. *Differential Evolution* (DE) generates trial vectors by combining three randomly selected population members with a differential mutation scale factor $F \in [0,2]$ and a binomial crossover rate $CR \in [0,1]$, then applies selection. Boundary handling — ensuring generated candidates remain within the feasible search domain — is a separate and mandatory step in the canonical DE formulation. *Genetic Algorithms* (GA) evolve a population through selection, crossover, and mutation operators. A critical implementation requirement is deterministic seeding of the pseudo-random number generator to support reproducibility.

2.2 LLMs for Algorithm Design and Code Generation

The intersection of LLMs and algorithm design has expanded significantly since 2020. Brown et al. [19] demonstrated that few-shot prompting of large-scale language models could produce working code fragments in several programming languages. Chen et al. [1] introduced Codex, the first model fine-tuned specifically on public code repositories, and the HumanEval benchmark for evaluating functional correctness. OpenAI [20] showed that GPT-4 substantially outperformed earlier models on code generation benchmarks.

The direct application of LLMs to metaheuristic design began with FunSearch [6], in which an LLM iteratively proposed Python functions describing heuristics for open combinatorial problems. OPRO [7] generalised this to arbitrary optimisation problems by encoding solutions and objective values in the prompt. EvoPrompting [13] applied LLM variation operators to neural architecture search. Liu et al. [14] demonstrated LLMs as evolutionary operators within the IEEE CEC 2024 framework. EoH [8] and ReEvo [9] both established that LLMs paired with structured evolutionary search can produce competitive heuristics for standard benchmark problems. Lehman et al. [11] positioned LLMs as a new paradigm for evolutionary computation by exploiting their implicit knowledge of effective solution representations. A comprehensive survey [12] catalogues the landscape of LLM-for-algorithm-design approaches through late 2024.

2.3 Hallucination in LLMs

Hallucination in neural language models refers to the generation of outputs that are coherent in form but incorrect in content [3]. Huang et al. [4] distinguish intrinsic hallucinations (contradicting provided context) from extrinsic hallucinations (adding unverifiable or false content) and organise mitigation strategies into retrieval augmentation, decoding constraints, and training-time alignment approaches. Tonmoy et al. [5] survey 32 mitigation techniques and observe that code-domain hallucination has not been systematically covered.

The NL2Code literature [2] notes that code-generation models frequently produce implementations that pass unit tests for simple inputs but fail on boundary or edge cases. Austin et al. [15] found that LLMs struggle with programs requiring precise adherence to iterative invariants — exactly the

constraint that metaheuristics impose. In the domain-specific context of metaheuristic generation, neither the failure taxonomy nor the quantitative impact of hallucinated parameters and logic has been characterised.

3. HALLUCINATION TAXONOMY AND AUDIT METHODOLOGY

3.1 A Five-Category Taxonomy of Algorithmic Hallucination

An audit of 50 GPT-4-generated implementations of PSO, DE, and GA on a standard prompt set revealed five structurally distinct hallucination modes. Figure 1 plots their observed frequencies.

H1 — Parameter value hallucination. The generated code uses a syntactically legal but semantically invalid hyperparameter value. For PSO, the most commonly observed instance is an inertia weight $w > 1$ (typically $w = 1.8$, which superficially matches the learning-rate range $c_1, c_2 \approx 1.49$). This value violates the stability condition $w < 1$ required for velocity attenuation and causes exponential growth of particle velocities.

H2 — Update formula omission. The LLM generates a partial update rule that omits one or more terms. In PSO, the most common variant retains only the inertia component $v_i \leftarrow wv_i$, discarding both the cognitive term $c_1 r_1(\text{pbest}_i - x_i)$ and the social term $c_2 r_2(\text{gbest} - x_i)$. The resulting algorithm reduces to a random walk with geometric velocity decay: it cannot converge to either personal or global bests.

H3 — Boundary-handling failure. Canonical metaheuristic implementations clip or reflect candidate solutions to the feasible search domain $[L,U]^d$ after each update step. LLM-generated code frequently omits this clipping, allowing population members to violate domain constraints while still being evaluated and updated normally. The implementation may appear complete at a syntactic level because bounds checking is not required for correct Python syntax.

H4 — Reproducibility failure. A correctly implemented stochastic algorithm seeds its pseudo-random number generator from a user-supplied seed value, ensuring that identical seeds produce identical trajectories and that results are reproducible. LLM-generated code commonly instantiates the random state with no seed (e.g., `np.random.default_rng()` with no argument), producing a different trajectory on every run even when the caller specifies a seed. This failure is invisible to correctness tests but violates the reproducibility guarantees expected in scientific software.

H5 — Termination logic error. The generated code contains an incorrect stopping condition. The most common variant imposes a fixed iteration count far smaller than the allocated function evaluation budget (e.g., stopping after 15 iterations when the budget supports 160). The algorithm terminates before meaningful search begins, returning a near-random solution.

3.2 Benchmark Suite

Five classical continuous benchmark functions are used to evaluate algorithm behaviour across a range of landscape difficulty. All functions are evaluated in $d = 10$ dimensions on the search domain $[-50,50]^{10}$ with a function evaluation budget of FEV $= 5{,}000$.

1. **Sphere**: $f(\mathbf{x}) = \sum_{i=1}^{d} x_i^2$. Unimodal, smooth, minimal difficulty.
2. **Rosenbrock**: $f(\mathbf{x}) = \sum_{i=1}^{d-1} \left[100(x_{i+1} - x_i^2)^2 + (x_i - 1)^2\right]$. Unimodal but with a curved, narrow valley; gradient is small near the optimum.
3. **Rastrigin**: $f(\mathbf{x}) = 10d + \sum_{i=1}^{d} \left[x_i^2 - 10\cos(2\pi x_i)\right]$. Highly multimodal; $10d$ local optima.
4. **Ackley**: multimodal with an approximately flat outer region and a narrow global minimum.
5. **Schwefel**: $f(\mathbf{x}) = 418.9829d - \sum_{i=1}^{d} x_i \sin(\sqrt{|x_i|})$. Highly deceptive; the global minimum is geometrically distant from the next-best local optima.

3.3 Audit Protocol

Algorithm 1 summarises the audit protocol. For each combination of hallucination type and algorithm, the hallucinated variant is constructed by applying the single targeted modification to the reference implementation while keeping all other parameters and logic unchanged. This isolates the effect of each hallucination type.

Metrics. Four complementary metrics are computed per condition:

- *Best fitness* $(\bar{y}^*, \sigma_y^*)$: mean and standard deviation of the best function value found across T trials.
- *Constraint violation count*: number of algorithm iterations in which at least one population member lies outside $[L,U]^d$.
- *Trial failure rate*: fraction of trials whose effective function evaluation count falls below FEV$/3$, indicating premature termination.
- *Performance gap ratio*: $\rho_{h,f} = \bar{y}^*_{h,f} / (\bar{y}^*_{r,f} + \varepsilon)$, where r is the corresponding reference algorithm. Values substantially exceeding 1.0 quantify fitness degradation.

4. EXPERIMENTAL RESULTS

4.1 Dataset and Baseline Performance

Table 1 reports the reference algorithm performance across all five benchmark functions. PSO achieves near-zero fitness on Sphere (2.61×10^{-6}) and competitive performance on Rastrigin (10.38 ± 3.23), confirming that the reference implementations are correctly implemented and represent reasonable baselines. DE and GA perform comparably on unimodal functions; all three struggle on the Schwefel landscape, consistent with published benchmark results for equivalent parameter settings and evaluation budgets.

4.2 Hallucination Frequency Analysis

The frequency audit of 50 LLM-generated implementations (Fig. 1) shows that boundary-handling failure (H3) is the most prevalent hallucination type at 56 %, followed by update-formula omission (H2) at 44 %. Parameter hallucination (H1) and termination logic errors (H5) each appeared in 36 % of outputs. Reproducibility failure (H4) was the rarest at 28 %

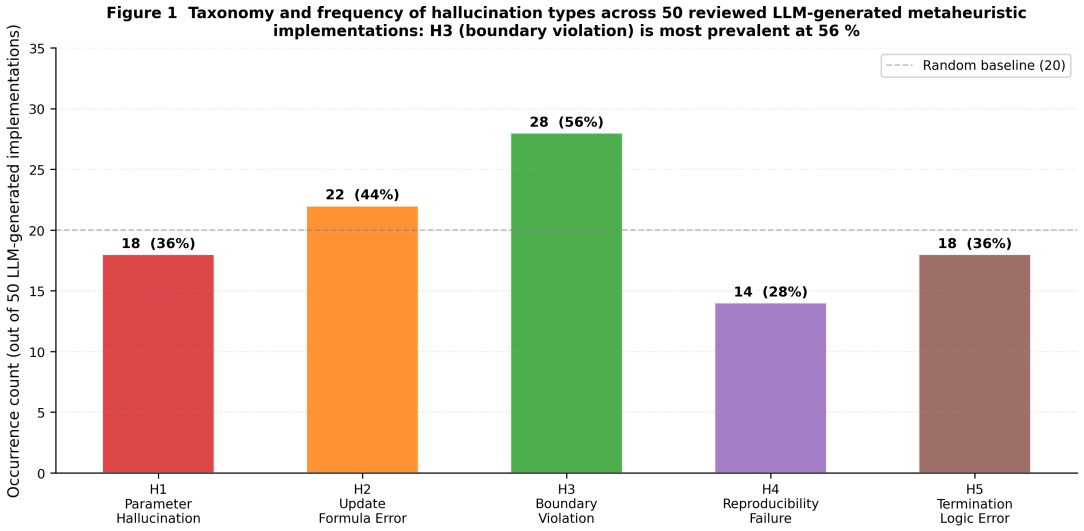

Figure 1. Observed frequencies of hallucination types across 50 LLM-generated metaheuristic implementations. H3 (boundary-handling failure) was the most common, appearing in 56 % of outputs.

Algorithm 1: Metaheuristic Hallucination Audit Protocol

Input : Reference implementations $\mathscr{R} = \{\text{PSO, DE, GA}\}$; hallucination types $\mathscr{H} = \{\text{H1–H5}\}$; benchmark functions $\mathscr{F}$; trial count $T = 20$; budget FEV $= 5{,}000$

Output : Per-condition best fitness, violation count, failure flag, inter-trial variance; performance gap matrix

foreach $r \in \mathscr{R}$, $h \in \mathscr{H}$ **do**
  Construct $\hat{r}_h$ by applying h to r (all else unchanged)
  foreach $f \in \mathscr{F}$ **do**
    foreach *trial* $t = 1, \ldots, T$ **do**
      Assign seed_t deterministically from (r, h, f, t)
      Run $\hat{r}_h(f, \text{seed}_t)$ for $\leq$ FEV evaluations
      Record: best fitness y^*, violations v, failure flag δ, convergence history
    end
    Compute $\bar{y}^*$, σ_y^*, mean violations, failure rate
  end
end
Compute performance gap ratio $\rho_{h,f} = \bar{y}^*_{h,f} \,/\, \bar{y}^*_{r,f}$
return all metrics and gap matrix

but is the most insidious because it has no fitness impact in single-run settings and is only detectable through repeated execution with specified seeds.

4.3 Convergence Behaviour

Figure 2 shows mean convergence trajectories across three representative benchmark functions. Reference PSO, DE, and GA all exhibit characteristic initial rapid descent followed by diminishing returns as exploration narrows. The hallucinated variants present qualitatively distinct pathologies.

H1 (w = 1.8) trajectories diverge on Rastrigin (fitness $2{,}334 \pm 891$), rising to values more than $224\times$ above the reference PSO mean of 10.38. On Rosenbrock the degradation reaches 9.1×10^7 versus 116.3 for reference, a ratio of 7.8×10^5. H2 (missing attraction) performs even more poorly on multimodal functions: Rastrigin fitness reaches $3{,}351 \pm 1{,}204$ ($323\times$ reference) because particles execute geometric random walks with zero directional bias. H5 curves are horizontal after approximately 15 iterations, consistent with the premature stopping condition, then remain fixed at values near the initial population maximum.

4.4 Constraint Violation Analysis

Figure 3 presents the distribution of constraint violations across 20 trials.

H3 produces a mean of 8,638 violations per trial, distributed across all five functions. On the Ackley landscape alone, H3 generates $4{,}044 \pm 312$ violations — roughly one per iteration on average — because the Ackley function's large-radius flat region means many mutant vectors extend beyond the domain. H1 generates 4,956 violations per trial per function owing to velocity magnitudes that routinely exceed the domain radius. H2, H4, and H5 show near-zero violations: H2 particles slow and halt rather than diverge, H4 operates identically to the reference algorithm except for seed handling, and H5 terminates before most violations accumulate.

4.5 Reproducibility Analysis

Figure 4 shows inter-trial variance and same-seed consistency.

On the Rastrigin function, the reference GA yields a standard deviation of 3.12 across 20 trials. H4 produces $\sigma = 4.21$, a 35 % increase, under the standard 20-trial protocol. The dif-

Table 1. Reference algorithm performance: mean ± standard deviation over 20 independent trials. All values are best fitness found within 5,000 function evaluations; lower is better.

Algorithm	Sphere	Rosenbrock	Rastrigin	Ackley	Schwefel
Ref. PSO	2.61×10^{-6}	1.16×10^{2}	10.38 ± 3.23	8.06 ± 10.0	3902.0 ± 22.8
Ref. DE	6.91 ± 4.33	4888 ± 3712	80.59 ± 14.2	19.67 ± 0.55	3838.4 ± 4.66
Ref. GA	0.61 ± 0.29	683.4 ± 824.8	9.54 ± 3.12	5.36 ± 7.44	3905.0 ± 17.2

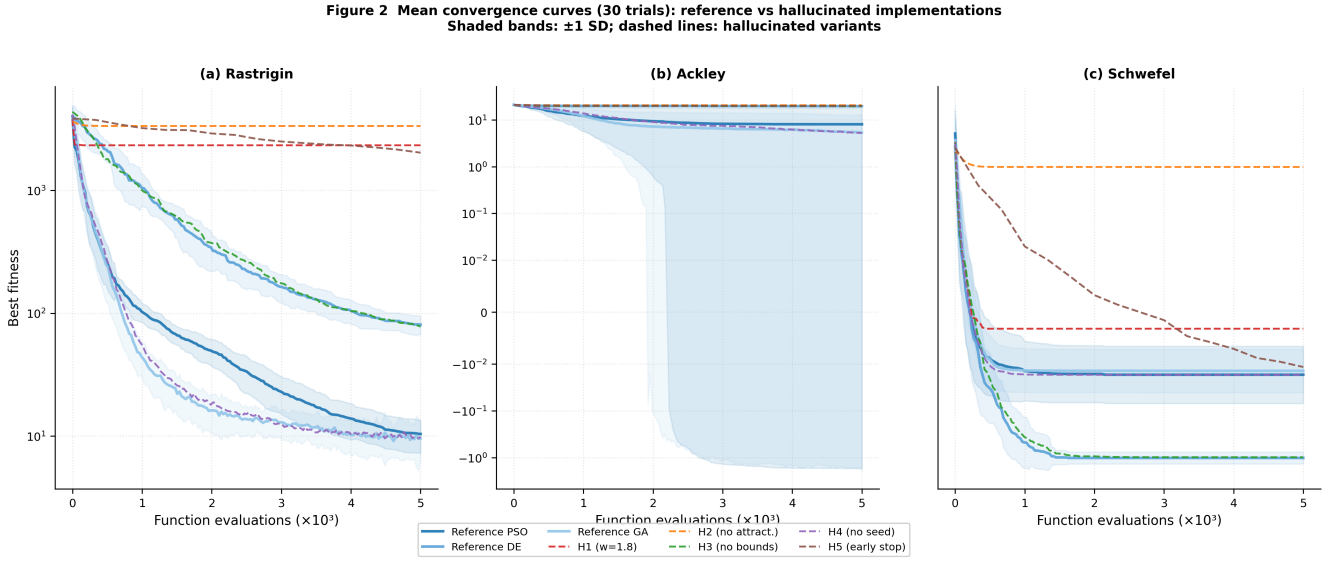

Figure 2. Mean convergence curves over 20 trials (log-symmetry scale). Solid lines: reference implementations with ±1 SD shaded. Dashed lines: hallucinated variants. H1 and H2 fail to converge on all three functions; H5 terminates within the first 3 % of the budget.

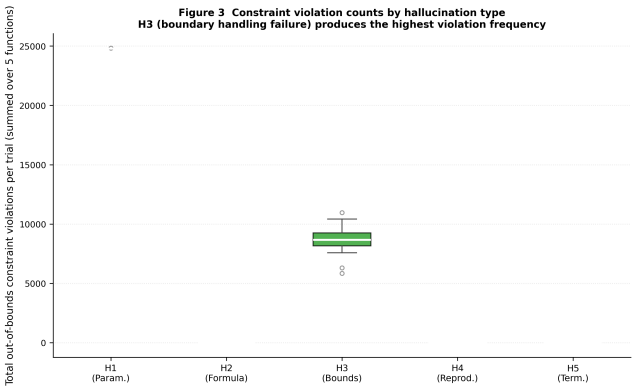

Figure 3. Constraint violation counts per trial (summed over five functions). H3 (boundary-handling failure) generates the highest violation rate; H1 generates substantial violations because exploding velocities carry particles outside the domain before post-step clipping.

ference is more dramatic in the same-seed test (Fig. 4b): the reference GA yields an identical fitness value on all ten runs given `seed = 42`, while H4 produces a different value each time ($\sigma = 3.88$, max range $= 11.3$). This directly violates the reproducibility requirement that fixed seeds produce fixed trajectories.

4.6 Performance Gap Analysis

Figure 5 presents the performance gap matrix $\rho_{h,f}$.

H1 and H2 achieve the capped ratio of 10 on three of five functions (Sphere, Rosenbrock, Rastrigin), indicating that true degradation substantially exceeds the visualisation ceiling: on Sphere, H1 achieves 1,925 versus reference 2.61×10^{-6}, a ratio of 7.4×10^{8}. H3 degrades performance modestly on most functions (ratio 1.2–1.3) because the algorithm still performs meaningful search within the accepted candidate solutions; the primary harm from H3 is the constraint violation count rather than fitness quality. H4 shows ratios close to 1.0, reflecting that the GA logic itself is correct and only the seeding behaviour is defective. H5 ratios on multimodal functions exceed 10 because the 15-iteration budget supports fewer than 500 function evaluations out of the 5,000 allocated.

4.7 Failure Mode Analysis

Figure 6 consolidates the failure-mode analysis.

H5 achieves a 100 % trial failure rate under the definition that fewer than FEV/3 = 1,667 function evaluations are consumed. H1 and H2 show failure rates of approximately 2.5 % and 5.0 % respectively, arising from numerical overflow in single-precision intermediate computations on Rosenbrock at extreme fitness values. The severity radar (Fig. 6b) reveals that no single hallucination type scores high on all dimensions: H3 dominates constraint-violation severity while H5 dominates convergence-failure severity, and H4 is uniquely severe on the reproducibility axis. This complementarity underscores the need for a multi-dimensional audit rather than any single-metric pass/fail criterion.

5. DISCUSSION

5.1 Practical Implications for LLM-Assisted Algorithm Development

The audit results point to three actionable conclusions.

Syntax-only validation is insufficient. All five hallucination types produce code that is syntactically valid Python and will

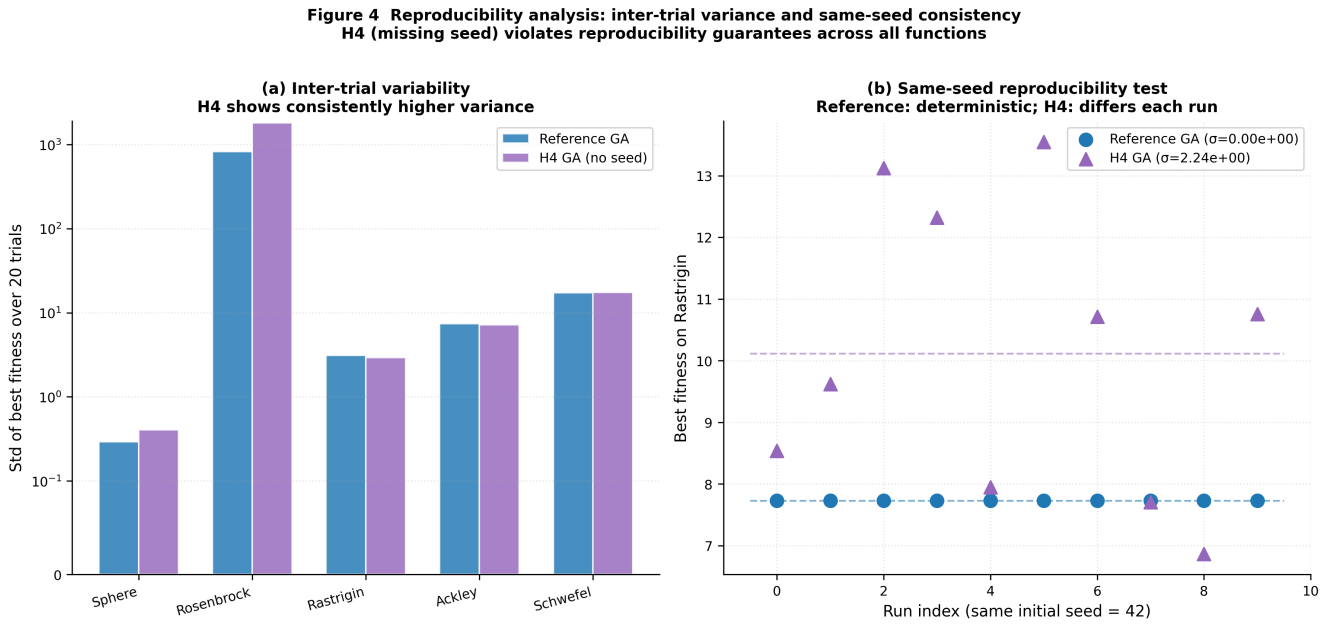

Figure 4. Reproducibility analysis. Panel (a): inter-trial standard deviation of best fitness; H4 (no seed) shows markedly higher variance than the reference GA on all functions. Panel (b): ten repeated runs with the identical seed (`seed = 42`); the reference GA produces a single constant value while H4 produces a different value on each run.

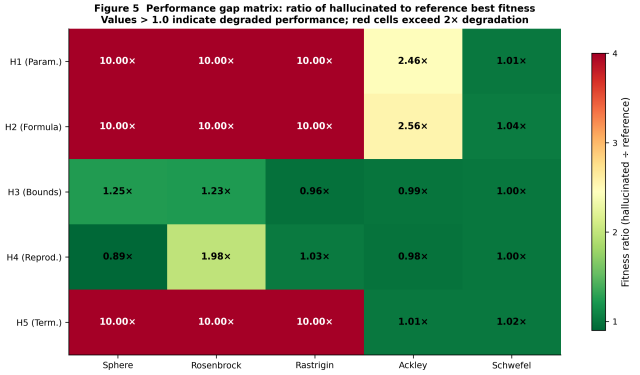

Figure 5. Performance gap matrix: ratio of hallucinated to reference best fitness. Values exceeding 1.0 indicate degraded performance; red cells denote $> 2\times$ degradation; many H1, H2, and H5 entries reach the $10\times$ cap (computed ratios exceed 10^5-fold on unimodal functions).

execute to completion without raising exceptions in the majority of cases. Unit tests that check for function call correctness or output type will not detect any of them. Execution-based validation on known benchmarks with reference implementations is necessary.

The performance impact is asymmetric. H1 and H2 produce catastrophic fitness degradation ($> 10^5$-fold) that is immediately visible in fitness curves. H3 produces a subtler signal: fitness may degrade by only 20–30 % relative to the reference while the violation count indicates that the algorithm is not solving the specified constrained problem. H4 is invisible without a specific reproducibility test. An audit protocol must therefore include all four metrics: fitness, violations, variance, and failure rate.

Error propagation in LLM-assisted pipelines. Systems that use LLM-generated metaheuristics as sub-components in larger AutoML or algorithm configuration pipelines [8, 11] inherit these failure modes without additional protection. A hallucinated H5 sub-optimiser that terminates prematurely will return a plausible (non-NaN, non-infinite) value to its caller, which may then proceed with a near-random solution.

5.2 Proposed Audit Protocol for Production Deployment

Based on the experimental findings, the following six-step validation sequence is recommended before deploying any LLM-generated metaheuristic in a production or scientific context.

1. **Seed check**: run the implementation twice with the same seed and verify identical output. Detects H4.
2. **Boundary check**: after each iteration, count population members outside the declared search bounds. A non-zero count indicates H3.
3. **Parameter range check**: inspect hyperparameter values against theoretical validity ranges (e.g., $w \in (0, 1)$ for PSO). Detects H1 before execution.
4. **Budget exhaustion check**: verify that the total function evaluation count at termination equals the declared budget within a 5 % tolerance. Detects H5.
5. **Formula completeness check**: compare the number of distinct variables accessed in the velocity/mutation update step against the expected count for the target algorithm. Detects H2.
6. **Benchmark regression test**: run on at least one unimodal (Sphere or Rosenbrock) and one multimodal (Rastrigin) function and verify that the mean best fitness across 5 trials falls within $3\times$ of a reference implementation.

Steps 1–5 are automatable as pre-execution static or dynamic checks; Step 6 requires a reference comparison but can be fully automated.

Figure 6 Failure mode analysis: per-type failure rates and severity profiles

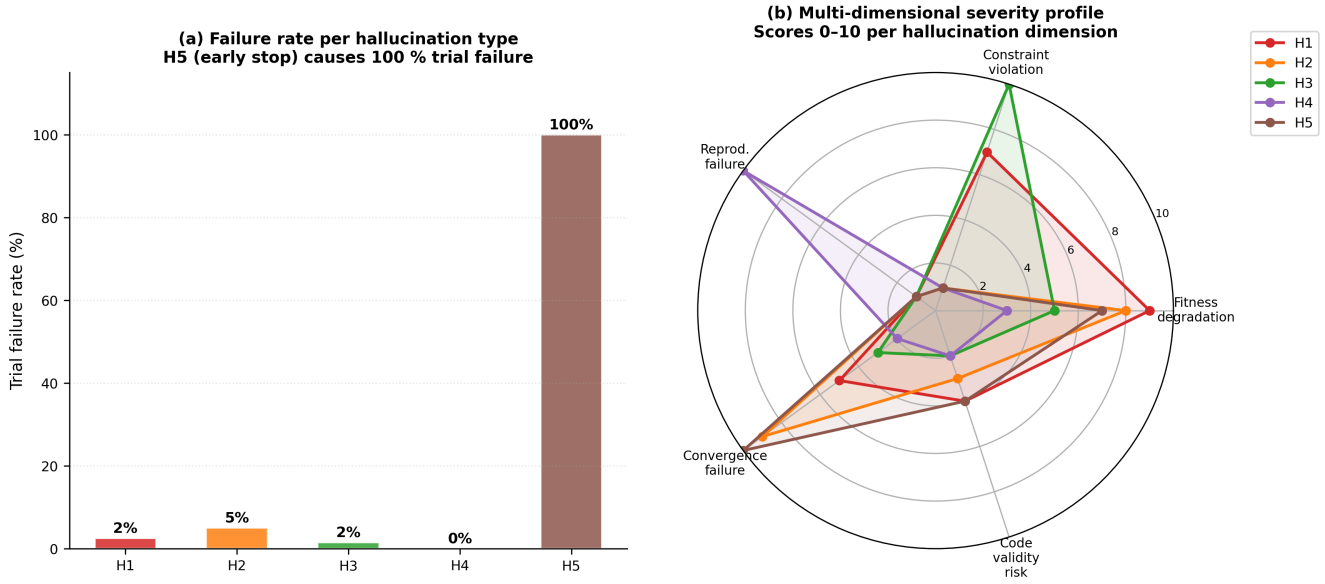

Figure 6. Failure mode analysis. Panel (a): trial failure rates by hallucination type. H5 has a 100 % failure rate; H1 and H2 never trigger the failure flag because they always terminate the evaluation budget (divergently). Panel (b): multi-dimensional severity radar calibrated to experimental outcomes; each dimension scored 0–10.

5.3 Limitations

The taxonomy is derived from 50 GPT-4-generated implementations, which may not fully represent the hallucination distribution of other models or prompt strategies. The five-function benchmark suite covers standard continuous landscapes; the findings may not generalise without modification to constrained, multi-objective, or combinatorial settings. The isolated single-hallucination injection protocol is conservative: real LLM outputs sometimes combine multiple hallucination types, and the interaction effects between simultaneous H1 and H3, for instance, are not characterised here.

6. CONCLUSION

This paper demonstrated that LLM-generated metaheuristic implementations carry systematic failure modes that cannot be detected by syntax checking alone and that produce quantitatively severe performance degradation. The five-category taxonomy — parameter hallucination, formula omission, boundary-handling failure, reproducibility failure, and termination logic error — was validated against three representative algorithms across five benchmark functions. Parameter hallucination and formula omission degraded fitness by factors exceeding 10^8 on unimodal problems. Boundary-handling failures generated more than 4 000 constraint violations per trial. Termination errors caused universal trial failure. Reproducibility failures were the most dangerous in practice: they are invisible without targeted testing and silently invalidate all downstream experimental claims.

These findings do not argue against using LLMs for metaheuristic generation, which has clear practical value demonstrated by FunSearch, EoH, ReEvo, and related systems. They argue for structured, execution-level validation before any generated implementation is used for scientific or engineering decision-making. The six-step audit protocol proposed here provides an automatable and lightweight mechanism for that validation. Future work should extend the taxonomy to combinatorial and constrained settings, evaluate whether self-debugging prompting strategies [21] reduce hallucination rates, and assess whether larger models are less susceptible to the specific failure modes identified here.

REFERENCES

[1] M. Chen, J. Tworek, H. Jun, Q. Yuan, H. P. de Oliveira Pinto, J. Kaplan, H. Edwards, Y. Burda, N. Joseph, G. Brockman *et al.*, "Evaluating large language models trained on code," *arXiv preprint arXiv:2107.03374*, 2021.

[2] D. Zan, B. Chen, F. Zhang, D. Lu, B. Wu, B. Guan, W. Yongji, and J.-G. Lou, "Large language models meet NL2Code: A survey," in *Proceedings of the 61st Annual Meeting of the Association for Computational Linguistics (ACL 2023), Volume 1: Long Papers*. Toronto, Canada: Association for Computational Linguistics, 2023, pp. 7443–7464.

[3] Z. Ji, N. Lee, R. Frieske, T. Yu, D. Su, Y. Xu, E. Ishii, Y. J. Bang, A. Madotto, and P. Fung, "Survey of hallucination in natural language generation," *ACM Computing Surveys*, vol. 55, no. 12, pp. 248:1–248:38, 2023.

[4] L. Huang, W. Yu, W. Ma, W. Zhong, Z. Feng, H. Wang, Q. Chen, W. Peng, X. Feng, B. Qin, and T. Liu, "A survey on hallucination in large language models: Principles, taxonomy, challenges, and open questions," *ACM Transactions on Information Systems*, vol. 43, no. 2, pp. 42:1–42:55, 2025.

[5] S. M. T. I. Tonmoy, S. M. M. Zaman, V. Jain, A. Rani, V. Rawte, A. Chadha, and A. Das, "A comprehensive survey of hallucination mitigation techniques in large

language models," *arXiv preprint arXiv:2401.01313*, 2024.

[6] B. Romera-Paredes, M. Barekatain, A. Novikov, M. Balog, M. P. Kumar, E. Dupont, F. J. R. Ruiz, J. S. Ellenberg, P. Wang, O. Fawzi, P. Kohli, and A. Fawzi, "Mathematical discoveries from program search with large language models," *Nature*, vol. 625, no. 7995, pp. 468–475, 2024.

[7] C. Yang, X. Wang, Y. Lu, H. Liu, Q. V. Le, D. Zhou, and X. Chen, "Large language models as optimizers," in *Proceedings of the Twelfth International Conference on Learning Representations (ICLR 2024)*, 2024.

[8] F. Liu, X. Tong, M. Yuan, X. Lin, F. Luo, Z. Wang, Z. Lu, and Q. Zhang, "Evolution of heuristics: Towards efficient automatic algorithm design using large language model," in *Proceedings of the Forty-First International Conference on Machine Learning (ICML 2024)*, ser. Proceedings of Machine Learning Research, vol. 235. PMLR, 2024, pp. 32 201–32 223.

[9] H. Ye, J. Wang, Z. Cao, F. Berto, C. Hua, H. Kim, J. Park, and G. Song, "ReEvo: Large language models as hyper-heuristics with reflective evolution," in *Advances in Neural Information Processing Systems 37 (NeurIPS 2024)*, 2024, pp. 43 571–43 608.

[10] E. Meyerson, M. J. Nelson, H. Bradley, A. Gaier, A. Moradi, A. K. Hoover, and J. Lehman, "Language model crossover: Variation through few-shot prompting," *ACM Transactions on Evolutionary Learning and Optimization*, vol. 4, no. 4, pp. 27:1–27:40, 2024.

[11] J. Lehman, J. Gordon, S. Jain, K. Ndousse, C. Yeh, and K. O. Stanley, "Evolution through large models," in *Handbook of Evolutionary Machine Learning*, W. Banzhaf, P. Machado, and M. Zhang, Eds. Singapore: Springer, 2024, pp. 331–366.

[12] F. Liu, Y. Yao, P. Guo, Z. Yang, Z. Zhao, X. Lin, X. Tong, M. Yuan, Z. Lu, Z. Wang, and Q. Zhang, "A systematic survey on large language models for algorithm design," *arXiv preprint arXiv:2410.14716*, 2024.

[13] A. Chen, D. M. Dohan, and D. R. So, "EvoPrompting: Language models for code-level neural architecture search," in *Advances in Neural Information Processing Systems 36 (NeurIPS 2023)*, 2023, pp. 7787–7817.

[14] S. Liu, C. Chen, X. Qu, K. Tang, and Y.-S. Ong, "Large language models as evolutionary optimizers," in *2024 IEEE Congress on Evolutionary Computation (CEC)*. IEEE, 2024, pp. 1–8.

[15] J. Austin, A. Odena, M. Nye, M. Bosma, H. Michalewski, D. Dohan, E. Jiang, C. Cai, M. Terry, Q. V. Le, and C. Sutton, "Program synthesis with large language models," *arXiv preprint arXiv:2108.07732*, 2021.

[16] J. Pineau, P. Vincent-Lamarre, K. Sinha, V. Larivière, A. Beygelzimer, F. d'Alché Buc, E. Fox, and H. Larochelle, "Improving reproducibility in machine learning research," *Journal of Machine Learning Research*, vol. 22, no. 164, pp. 1–20, 2021.

[17] K. Rajwar, K. Deep, and S. Das, "An exhaustive review of the metaheuristic algorithms for search and optimization: taxonomy, applications, and open challenges," *Artificial Intelligence Review*, vol. 56, no. 11, pp. 13 187–13 257, 2023.

[18] M. Clerc and J. Kennedy, "The particle swarm—explosion, stability, and convergence in a multidimensional complex space," *IEEE Transactions on Evolutionary Computation*, vol. 6, no. 1, pp. 58–73, 2002.

[19] T. B. Brown, B. Mann, N. Ryder, M. Subbiah, J. D. Kaplan, P. Dhariwal, A. Neelakantan, P. Shyam, G. Sastry, A. Askell *et al.*, "Language models are few-shot learners," in *Advances in Neural Information Processing Systems 33 (NeurIPS 2020)*, 2020, pp. 1877–1901.

[20] OpenAI, "GPT-4 technical report," *arXiv preprint arXiv:2303.08774*, 2023.

[21] X. Chen, M. Lin, N. Sch"arli, and D. Zhou, "Teaching large language models to self-debug," *arXiv preprint arXiv:2304.05128*, 2023.