GA4GC: Greener Agent for Greener Code via Multi-Objective Configuration Optimization

Jingzhi Gong[1], Yixin Bian[2], Luis de la Cal[3], Giovanni Pinna[4], Anisha Uteem[5], David Williams[6], Mar Zamorano[6], Karine Even-Mendoza[5], W.B. Langdon[6], Hector D. Menendez[5], and Federica Sarro[6]

[1] University of Leeds j.gong@leeds.ac.uk
[2] Harbin Normal University bianyixin@hrbnu.edu.cn
[3] Universidad Politécnica de Madrid l.delacal@upm.es
[4] University of Trieste giovanni.pinna@phd.units.it
[5] King's College London {anisha.uteem, karine.even_mendoza, hector.menendez}@kcl.ac.uk
[6] University College London {david.williams.22, maria.lopez.20, w.langdon, f.sarro}@ucl.ac.uk

Abstract. Coding agents powered by Large Language Models (LLMs) face critical sustainability and scalability challenges in industrial deployment, often incurring costs that may exceed optimization benefits. We introduce GA4GC, the first framework to optimize coding agent runtime (greener agent) and code performance (greener code) trade-offs by discovering Pareto-optimal agent hyperparameters and prompt templates. Evaluation on the SWE-Perf benchmark demonstrates up to 135-fold hypervolume improvement, reducing agent runtime by 37.7% while improving correctness. Baseline comparisons and influence analysis confirm the effectiveness of GA4GC, identify temperature as the most influential hyperparameter, and provide actionable strategies to balance agent and code sustainability in industrial deployment.

Keywords: SBSE · GenAI · Coding Agents · Green SE · AI4SE

1 Introduction

Code performance optimization is fundamental to software development, directly impacting system scalability, resource consumption, and user experience [15]. While Large Language Models (LLMs) show promise in automating this process [10], current approaches focus on simple benchmarks like HumanEval [6] that do not capture real-world software engineering complexity [12].

To address this limitation, researchers and practitioners have increasingly turned to agentic workflows, where LLMs operate as autonomous agents capable of iterative reasoning, tool use, and complex decision-making [3]. These approaches have shown promising results in realistic software engineering (SE) benchmarks such as SWE-Perf [13], which provides code optimization tasks reflecting the complexity that agents face in industry deployments.

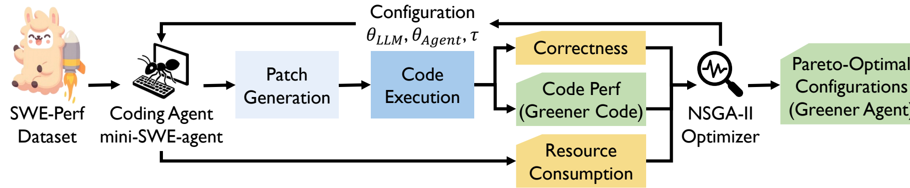

Fig. 1. `GA4GC` workflow of multi-objective configuration optimization.

Unlike single-shot code LLMs, coding agents operate through iterative reasoning processes that require multiple LLM calls, each consuming significant computational resources [4]. Although these agents can successfully solve complex real-world coding tasks, a single agent running on real-world SE problems can consume over 100,000 tokens [1]. Moreover, without careful tuning, the energy consumed by an optimization agent can require hundreds of thousands of code executions to reach an energetic "break-even" point, making some optimizations a net energy loss [7]. As organizations scale deployments, this creates prohibitive costs and threatens environmental sustainability, directly conflicting with Green Software Engineering principles and Net Zero targets [8]. [1]

We address these challenges by proposing `GA4GC` (Greener Agent for Greener Code), which optimizes the trade-off between resource consumption of the coding agent and performance of the generated code. Our key insight is that the vast configuration space of coding agents, including prompt templates, LLM- and agent-specific hyperparameters, is too complex for manual exploration. Therefore `GA4GC` uses NSGA-II Multi-Objective Genetic Algorithm (MOGA) optimization to discover Pareto-optimal agent configurations. **Our contributions are**:

- `GA4GC`, a MOGA framework that discovers Pareto-optimal coding agent configurations that are up to 37.7% faster (943 vs 1513 seconds) while improving correctness, with up to 135 times improved hypervolume over the default.
- Configuration influence analysis via Random Forest reveals that temperature has the highest overall influence. LLM hyperparameters primarily impact task effectiveness, while agent constraints affect resource consumption.
- Actionable suggestions for coding agent practitioners across three scenarios: runtime-critical (low temperature with restrictive top_p), performance-critical (moderate temperature with balanced top_p), and context-specific optimization via `GA4GC`.

Related Work. Recent green GenAI research has applied reinforcement learning for energy-efficient code generation [14], compared energy efficiency of LLM versus human-written code [2], and optimized GenAI hyperparameters for domain modeling [5] and text-to-image generation [11,9]. These approaches, however, focus on single-shot generative tasks. By contrast, we address the challenges of complex, multi-turn agentic workflows, mitigating the substantial computational costs of deploying coding agents in real-world software engineering.

[1] `https://www.un.org/en/climatechange/net-zero-coalition`

Table 1. Configuration search space. Decimal range = any value within the range; integer range = only integer values; {set} = only specified values.

Category	Hyperparameter	Abbr.	Range/Values	Description
LLM	Temperature	Temp	[0.0, 1.0]	Controls randomness in token selection
	Top_p	TopP	[0.1, 1.0]	Limits sampled token vocabulary size
	Max_tokens	Token	[512, 4096]	Constrains maximum response length
Agent	Step_limit	Step	[10, 40]	Limits number of LLM calls
	Cost_limit ($)	Cost	[3.0, 10.0]	Constrains total cost of LLM usage
	Env_timeout (s)	ETi	[40, 60]	Timeout for environment operations
	LLM_timeout (s)	LTi	[40, 60]	Timeout for individual LLM calls
Prompt	Template Variant	Pr	{1,2,3}	Different template configurations

2 Methodology and Experimental Setup

MOGA Optimization. Figure 1 illustrates `GA4GC`'s workflow, where we employ NSGA-II to explore the agent configuration space defined by $\mathcal{C} = (\theta_{LLM}, \theta_{agent}, \tau)$, where θ_{LLM} represents LLM-specific hyperparameters, θ_{agent} represents agent-specific operational constraints, and τ represents the prompt template variant[2]. Table 1 details the configuration search space.

We define three fitness functions: $f_1(\mathcal{C})$ = **correctness (passes all test cases),** $f_2(\mathcal{C})$ = **performance gain (code speedup), and** $f_3(\mathcal{C})$ = **agent runtime** (minimize). For each candidate configuration, the agent receives a code optimization task and generates patches through iterative reasoning, during which we measure f_3. Generated patches are executed in isolated Docker environments to measure f_1 and f_2, and the output is a Pareto front of non-dominated configurations.

Research Questions. We address three research questions (RQs):

➤ **RQ1.** To what extent can `GA4GC` improve the resource consumption and performance trade-offs of coding agents compared to default configurations?

➤ **RQ2.** How do different hyperparameters influence agent resource consumption and task performance in the optimization process?

➤ **RQ3.** What actionable strategies can be derived from the Pareto-optimal configurations for sustainable coding agent deployment?

Experimental Setup. We implement mini-SWE-agent[3] with Gemini 2.5 Pro as the base LLM. Our evaluation uses SWE-Perf [13], a benchmark for code optimization tasks in real-world repositories where the goal is to improve code runtime while maintaining functionality. Given the extensive evaluation time required for each candidate configuration, we focus on the astropy project[4], using 9 instances for training and 3 instances for validation.

We use `pymoo`'s default NSGA-II setup[5] with a population of 5 and 5 generations, giving 25 configurations. Each configuration is evaluated by running the

[2] Details on the prompt templates we used can be found on GA4GC's GitHub page.

[3] `https://github.com/pppyb/mini-swe-agent`

[4] `https://github.com/astropy/astropy`

[5] Detailed NSGA-II setup and complete methodology can be found on GitHub.

Table 2. Default vs. GA4GC-optimized configurations. Corr=Correctness, Perf= performance gain (%), Runt=runtime (s), HV=hypervolume, VHV=validation hypervolume. See Table 1 for other definitions. Green cells indicate improvements over default.

Config	Temp	TopP	Token	Step	Cost	ETi	LTi	Pr	Corr	Perf	Runt	HV (%)	VHV (%)
Default	0.000	1.000	4096	240	3.00	60	60	-	2/9	0.00	1513.3	0.52	1.1
#4	0.085	0.135	1120	36	9.26	41	57	2	4/9	0.00	943.1	5.82	4.1
#5	0.692	0.384	2972	38	6.73	40	56	3	8/9	6.43	984.8	70.28	14.9
#9	0.725	0.412	2972	22	6.73	43	41	3	7/9	0.00	958.1	9.25	21.6
#15	0.657	0.384	2972	38	6.73	40	56	2	7/9	10.67	1400.1	33.42	2.7
#16	0.085	0.131	1120	36	6.91	41	57	2	0/9	0.00	853.3	1.10	21.6

agent on all 9 training instances and measuring the three objectives (f_1, f_2, f_3). After optimization, we extract the Pareto-optimal configurations and, to assess generalization, validate them on 3 held-out instances. In total, each run took between 25 and 35 hours and cost \$50 to \$150 for LLM API calls.

All experiments are conducted on an isolated Google Cloud Platform server with 4 CPUs, 16 GB RAM, running Ubuntu 25.04. Performance gains for each SWE-Perf instance are measured 20 times, and statistical significance is evaluated using the Mann-Whitney U test with $p < 0.1$ [13].

3 Results and Analysis

RQ1 Results. Table 2 shows the results of RQ1, where NSGA-II identifies five Pareto-optimal configurations: Config#4 achieves 37.7% runtime reduction (943.1s vs 1513.3s) while doubling correctness, Config#15 achieves 10.67% code performance gain with similar runtime overhead, and Config#5 delivers 4 times better correctness (8/9 vs 2/9) while simultaneously improving performance by 6.43%. Notably, **4 out of 5 configurations dominate default in multiple objectives**, addressing both greener agent and greener code requirements.

We computed the hypervolume indicator using `pymoo` with objectives normalized to [0,1] and reference point [-0.1, -0.1, -0.1] (runtime inverted). **Each optimized configuration substantially outperforms the default**: Config#5 achieves 135× higher hypervolume (70.28% vs 0.52%), Config#15 achieves 64× improvement (33.42% vs 0.52%), and even the lowest-performing Config#16 achieves 2× improvement (1.10% vs 0.52%). Validation on three held-out instances confirms generalization, with all optimized configurations maintaining superior hypervolume. Moreover, NSGA-II outperforms the random search baseline (83.0% vs 53.1% cumulative hypervolume, 5 vs 3 Pareto solutions), confirming directed optimization rather than random exploration[6].

RQ1: GA4GC achieves 135 times higher hypervolume, 37.7% faster runtime while improving correctness, and 4/5 Pareto front configurations dominating the default while all maintaining superior hypervolume on unseen tasks.

RQ2 Results. Table 3 shows the hyperparameter influence analysis. We train a Random Forest for each objective using all 25 evaluated configurations to measure influence magnitudes [11]. Among others, **temperature emerges as**

[6] Complete baseline comparison and Pareto front visualizations are available here.

Table 3. Random Forest feature importance for hyperparameters on optimization objectives. Colors indicate importance: Low (0.0-0.1), Medium (0.1-0.2), High (>0.2).

Category	Hyperparameter	Correctness Impact	Performance Impact	Runtime Impact
LLM	Temperature	0.152	0.392	0.199
	Top_p	0.199	0.051	0.097
	Max_tokens	0.057	0.090	0.089
Agent	Step_limit	0.140	0.119	0.049
	Cost_limit	0.199	0.076	0.128
	Env_timeout	0.060	0.034	0.298
	LLM_timeout	0.120	0.109	0.102
Prompt	Template Variant	0.072	0.130	0.038

the most critical hyperparameter, with high-performing Config#5 and #15 using moderate temperatures (0.66–0.69) while low-temperature Config#4 and #16 achieve faster runtime but no performance gain, indicating its role in balancing exploration versus exploitation during token generation.

Top_p shows correctness influence (0.199) with successful configurations using mid-range values (0.38–0.41), indicating that balanced vocabulary sampling avoids both overly restrictive and chaotic token selection. Cost_limit exhibits influence across correctness (0.199) and runtime (0.128), with Pareto-optimal configurations using higher budgets ($6.73–$9.26 vs $3.0 default) to enable more thorough exploration without timeout constraints. Prompt template variants show moderate performance influence (0.130), with templates 2 and 3 dominating the Pareto front, suggesting that task-specific prompt engineering significantly impacts optimization effectiveness.

RQ2: Temperature shows highest overall influence, LLM hyperparameters primarily impact task effectiveness while agent constraints affect resource consumption, confirming the need for MOGA in green coding agent deployment.

RQ3 Results. Based on the hyperparameter influence analysis, we derive three actionable strategies for green SBSE practitioners across different optimization scenarios: **For runtime-critical scenarios:** Use low temperature (0–0.1) with restrictive top_p (0.13–0.14) to minimize exploration overhead, combined with moderate max_tokens (1120–2000) and step limits (20–36). **Performance-critical scenarios:** Use moderate temperature (0.65–0.70) with balanced top_p (0.38–0.41) to enable creative optimization strategies, combined with higher cost budgets ($6.5–$9.5) and prompt templates optimized for performance tasks. **For most accurate optimization:** For practitioners with specific requirements, we recommend applying `GA4GC` to discover context-specific Pareto-optimal configurations tailored to their deployment priorities.

RQ3: We provide scenario-specific actionable suggestions for greener coding agent deployment. For more accurate optimization, practitioners can apply `GA4GC` to discover context-specific Pareto-optimal configurations.

Threats to Validity. Our evaluation focuses on the astropy project (12 instances) from SWE-Perf and is specific to mini-SWE-agent with Gemini 2.5 Pro

due to computational constraints, which may limit generalizability. The limited search budget may prevent full Pareto front convergence. The stochastic nature of NSGA-II and LLM inference means results may vary across runs. All limitations reveal opportunities for future studies.

4 Conclusion

We introduced `GA4GC`, a framework to optimize coding agent resource-performance trade-offs via multi-objective optimization. On SWE-Perf, it achieves 135-fold hypervolume improvement and 37.7% runtime reduction while improving correctness. Our analysis reveals temperature is the most important LLM parameter and other insights and actionable guidelines to address both green concerns and use by industry.

Availability. Code and results are available at GitHub & DOI 10.5281/zenodo.17177693.

References

1. Anthropic: Raising the bar on SWE-bench Verified with Claude 3.5 Sonnet. https://www.anthropic.com/research/swe-bench-sonnet (Jan 2025)
2. Apsan, R., et al.: Generating energy-efficient code via large-language models–where are we now? arXiv preprint arXiv:2509.10099 (2025)
3. Ashiga, M., et al.: Industrial llm-based code optimization under regulation: A mixture-of-agents approach. arXiv preprint arXiv:2508.03329 (2025)
4. Belcak, P., et al.: Small language models are the future of Agentic AI (2025), https://arxiv.org/abs/2506.02153
5. Bulhakov, V., et al.: Investigating the role of LLMs hyperparameter tuning and prompt engineering to support domain modeling. In: SEAA 2025. pp. 349–366
6. Chen, M., Tworek, J., et al.: Evaluating large language models trained on code (2021), https://arxiv.org/abs/2107.03374
7. Coignion, T., Quinton, C., Rouvoy, R.: When faster isn't greener: The hidden costs of LLM-based code optimization. In: ASE 2025
8. Cruz, L.e.a.: Greening AI-enabled systems with software engineering: A research agenda for environmentally sustainable AI practices. ACM SIGSOFT Software Eng. Notes **50**(3), 14–23 (Jul 2025). https://doi.org/10.1145/3743095.3743099
9. d'Aloisio, G., Fadahunsi, T., Choy, J., Moussa, R., Sarro, F.: SustainDiffusion: Optimising the social and environmental sustainability of Stable Diffusion models. In: 2025 IEEE/ACM ICSE (2025), https://arxiv.org/abs/2507.15663
10. Gong, J., Giavrimis, R., Brookes, P., et al.: Tuning llm-based code optimization via meta-prompting: An industrial perspective. arXiv:2508.01443 (2025)
11. Gong, J., Li, S., d'Aloisio, G., Ding, Z., Ye, Y., Langdon, W.B., Sarro, F.: GreenStableYolo: Optimizing inference time and image quality of text-to-image generation. In: SSBSE. pp. 70–76. Springer (2024)
12. Gong, J., et al.: Language models for code optimization: Survey, challenges and future directions (2025), https://arxiv.org/abs/2501.01277
13. He, X., et al.: SWE-Perf: can language models optimize code performance on real-world repositories? (2025), https://arxiv.org/abs/2507.12415
14. Ilager, S., Briem, L.F., Brandic, I.: Green-Code: Learning to optimize energy efficiency in LLM-based code generation. In: CCGrid 2025. pp. 559–569. IEEE
15. Shypula, A.G., et al.: Learning performance-improving code edits. In: ICLR (2024)