Research Square

Preprints are preliminary reports that have not undergone peer review.
They should not be considered conclusive, used to inform clinical practice,
or referenced by the media as validated information.

MicroEvolve: Improving Existing Solvers for the Mentorship and Teamwork Problem using an LLM in the Loop

Uran Lajçi
University of Prishtina https://orcid.org/0009-0002-6214-5865
Avni Rexhepi
avni.rexhepi@uni-pr.edu

University of Prishtina
Kadri Sylejmani
University of Prishtina https://orcid.org/0000-0002-4428-1161
Artan Mazrekaj
University of Prishtina https://orcid.org/0009-0006-2605-8771
Jetë Lajçi
University of Prishtina https://orcid.org/0009-0007-2832-3791
Elvir Misini
University of Prishtina https://orcid.org/0009-0006-3262-7440

Research Article

Keywords: Large language models, Combinatorial optimization, LLM-in-the-loop, Google Hash Code, Mentorship and Teamwork

Posted Date: November 19th, 2026

DOI: https://doi.org/10.21203/rs.3.rs-11776181/v1

License: This work is licensed under a Creative Commons Attribution 4.0 International License.
Read Full License

Additional Declarations: The authors declare no competing interests.

MicroEvolve: Improving Existing Solvers for the Mentorship and Teamwork Problem using an LLM in the Loop

Uran Lajçi[1,2], Avni Rexhepi[1*], Kadri Sylejmani[1], Artan Mazrekaj[1], Jetë Lajçi[1], Elvir Misini[1]

[1*]Department of Computer and Software Engineering, Faculty of Electrical and Computer Engineering, University of Prishtina, Prishtinë, 10000, Kosovë.
[2]Care Software, Prishtinë, 10000, Kosovë.

*Corresponding author(s). E-mail(s): avni.rexhepi@uni-pr.edu;
Contributing authors: uran.lajqi@uni-pr.edu; kadri.sylejmani@uni-pr.edu; artan.mazrekaj@uni-pr.edu; jete.lajci@student.uni-pr.edu; elvir.misini@uni-pr.edu;

Abstract

The Mentorship and Teamwork problem, introduced in the Google Hash Code 2022 competition, is a large-scale combinatorial optimization problem in which contributors with different skill levels are assigned to project roles with the aim of maximizing the score of completed projects. This work investigates whether a large language model (LLM) can improve solvers that are already strong, rather than generate a solver from scratch. We introduce MicroEvolve, an automated LLM-in-the-loop code-tuning process in which the model proposes localized modifications to an existing solver, including heuristic, structural, and parameter changes. Each candidate is compiled, executed on a fixed problem instance, and evaluated by an external validator; a modification is retained only when it preserves or improves the current best score. The loop requires no model training or fine-tuning and proceeds automatically after the initial solver is selected. We evaluate MicroEvolve on the five non-trivial official competition instances, starting from an adapted version of the strongest publicly available open-source solver we identified and driven by a single model, Claude Opus 4.8. Five independent 100-iteration runs are performed for each instance, for 25 runs and 2,500 proposed modifications in total. Every run improves its corresponding baseline. Considering the best run for each instance, gains range from 0.46% to 53.69%,

and MicroEvolve establishes new best-known scores on four of the five instances, exceeding previous best-known results by up to approximately 31%. These results provide evidence that LLM-guided code tuning can serve as an effective automated optimization mechanism, analogous to parameter tuning, for improving mature combinatorial optimization solvers.

Keywords: Large language models, Combinatorial optimization, LLM-in-the-loop, Google Hash Code, Mentorship and Teamwork

1 Introduction

Large language models (LLMs)—the systems behind tools such as ChatGPT, Claude, and Google Gemini—have been widely adopted since the release of ChatGPT in 2022 [1]. Part of their appeal lies in their versatility: a single model can edit, summarize, and translate text, extract information from documents, generate code, and solve mathematical problems.

This versatility has prompted growing interest in the use of LLMs for NP-hard combinatorial optimization problems. In this setting, LLMs can be used to construct solutions directly, design new solvers, or improve existing ones. We study the third setting and ask a simple question: **given a strong existing solver, can an LLM improve it further?**

Recent work suggests that LLM-guided evolutionary search can improve algorithms and, in some cases, establish new best-known results. Liu et al. [2] introduced AEL (Algorithm Evolution using Large Language Models), which uses an LLM to evolve algorithms and produced a constructive heuristic for the traveling salesperson problem that outperformed both hand-crafted heuristics and heuristics obtained through direct LLM prompting. Romera-Paredes et al. [3] introduced FunSearch, which combines an LLM with an automated evaluator in an evolutionary loop and discovered new constructions for the long-studied cap set problem that improved on the best previously known results. These studies demonstrate the potential of LLM-guided program evolution, and later systems have applied the same idea to mature code. An important question nevertheless remains for problems of this kind: can a relatively simple LLM-in-the-loop process improve a strong, hand-tuned solver that was written independently of the method, and reach scores beyond the best previously reported?

We investigate this question on Mentorship and Teamwork, a large-scale combinatorial optimization problem introduced in the Qualification Round of Google Hash Code 2022 competition.

We introduce **MicroEvolve**, an LLM-in-the-loop method for improving an existing high-performance C++ solver. At each iteration, the model proposes a localized modification to the current solver. Such modifications may change heuristics, numerical parameters, selection criteria, or other parts of the implementation. The resulting candidate is compiled, executed on a fixed target instance under a ten-minute runtime budget, and scored by an external validator. A candidate is retained if its score matches or improves the best score obtained so far; otherwise, it is discarded.

Accepting score-preserving changes allows later modifications to build on edits that do not immediately improve the objective and enables the search to move across score plateaus. In this sense, MicroEvolve performs a form of automated **code tuning**, in which the implementation itself becomes the object of iterative optimization rather than only a predefined set of numerical parameters.

We start from an adapted version of Coudert-Osmont's open-source solver suite [4], the strongest publicly available open-source approach we identified for the problem. Each MicroEvolve run targets one solver–instance pairing, reflecting the instance-specific nature of the original competition solvers. We select and adapt the initial solver, but after this setup all subsequent code proposals, evaluations, and acceptance decisions proceed automatically, without human intervention. The method requires no model training or fine-tuning and maintains only a single accepted solver rather than a population or archive of candidate programs.

This paper makes three main contributions. First, it introduces MicroEvolve, a minimal and training-free LLM-in-the-loop method for automatically improving an existing high-performance solver through iterative code modifications. Second, it presents an empirical study of 25 independent runs—five on each of the five non-trivial competition instances—comprising 2,500 proposed modifications. Every run improves its corresponding baseline, while the collected run traces provide insight into the types of edits that produce improvements, when those improvements occur, and why the reliability of the method differs across instances. Third, considering the best of five runs for each instance, MicroEvolve establishes new best-known scores on four of the five instances, with improvements over the already strong baseline ranging from 0.46% to 53.69%.

We use the term *best-known score* to mean a score that exceeds every external result available to us for the corresponding instance. Our comparison includes the score of the competition-winning team, the highest score obtained on each instance by any team during the live competition, the FunSearch-evolved results reported by Veličković et al. [5], and the scores reported for the open-source solvers of Coudert-Osmont [4]. This comparison is particularly important because the baseline used by MicroEvolve already exceeds several of these previous results, meaning that the improvements reported in this work are obtained on top of an already competitive starting point.

The remainder of the paper is organised as follows. Section 2 reviews related work on LLM-guided program evolution and situates our contribution within it. Section 3 states the Mentorship and Teamwork problem and illustrates its rules on a small example. Section 4 describes the existing solvers for the problem and the five baselines we derive from the strongest of them. Section 5 presents MicroEvolve itself: the improvement loop, the form its edits take, and a worked trace of four consecutive iterations. Section 6 reports the experimental setup, covering the hardware, the model configuration, the evaluation pipeline, and the run protocol. Section 7 gives the results—the gains over the baselines, the comparison with prior work, the per-iteration outcomes, and the cost—and Section 8 analyses the run traces to account for how the improvements were obtained, closing with the limitations of the study. Section 9 concludes and Section 10 outlines directions for future work. Appendix A reproduces the system prompt and every feedback message verbatim.

2 Related Work

A recent systematic review of large language models (LLMs) for combinatorial optimization (CO) surveys 103 studies and organises them by the role the model plays in the optimization pipeline [6]. Of the 64 studies that address the solution method, the review separates two dominant uses: *solution generation*, in which the model is asked to produce candidate solutions directly (28 studies), and *code generation*, in which it writes or rewrites the algorithm that produces them (36 studies). Far less activity is recorded in parameter tuning (4 studies) and algorithm selection (1 study); these latter counts fall within the 64 rather than adding to it, as they identify studies that address an auxiliary task alongside their primary focus. Application is concentrated in routing—particularly the travelling salesman and vehicle routing problems—followed by scheduling and planning, networks and graphs, and packing, with a thinner tail in combinatorics, engineering, finance, and bioinformatics. This division mirrors the progression set out in Section 1: asking the model for solutions, asking it to design a solver, and asking it to improve one. Our work belongs to the third, and within code generation to a line that uses the LLM as a mutation operator inside an evolutionary search over programs: the model proposes edits to a piece of code, an external evaluator scores the resulting program on the target problem, and higher-scoring programs are retained and fed back into subsequent prompts. We review that line in three parts—the foundational frameworks, their extension to full-scale solvers, and their application to competitive combinatorial programming—before situating our own contribution.

The paradigm was established by FunSearch [3], which pairs a frozen pre-trained LLM with a systematic evaluator and evolves a single designated function inside a fixed program skeleton, maintaining a database of high-scoring programs organised into islands to preserve diversity. The object of search is the procedure rather than the artefact: the program that constructs a solution, not the solution itself. The skeleton supplies boilerplate and whatever prior structure the designer chooses to impose, and it is through the skeleton, rather than through the initial program, that domain knowledge enters; the reported applications begin from minimal implementations, and in this setting FunSearch produced new constructions for the cap set problem and improved heuristics for online bin packing. AlphaEvolve [7] generalises the recipe from a single function to whole code files of hundreds of lines in arbitrary languages, orchestrating an ensemble of LLMs under one or more evaluators and emitting edits as search-and-replace diffs against regions the user marks as evolvable. Among other results it improved data-centre scheduling and circuit design at Google—in both cases starting from mature, already-optimised code—and found a procedure to multiply two 4×4 complex matrices using 48 scalar multiplications, the first improvement over Strassen's algorithm in that setting in 56 years. ShinkaEvolve [8] addresses the sample inefficiency of these methods through balanced parent sampling, novelty-based rejection sampling, and a bandit-based selection over an LLM ensemble, reaching a state-of-the-art circle-packing configuration in fewer than 150 evaluations and releasing its framework as open source. Notably, ShinkaEvolve does not always begin from a blank slate: on the ALE-Bench competitive-programming benchmark it takes the C++ solutions of an existing agent as its starting point and improves all ten of them, signalling a shift in these methods from originating code to refining strong code.

A parallel line of work moves from short, self-contained programs to the internal components of full-scale solvers. AutoSAT [9] confines the LLM to nine hand-designed heuristic slots—restart, reduce, rephase, and variable-bumping policies together with their trigger conditions—inside a modularised conflict-driven clause-learning solver, and searches over them with a greedy hill climber and a $(1 + 1)$ evolutionary strategy, each retaining a candidate whose score equals or improves the incumbent. The starting point is EasySAT, a lightweight implementation chosen because MiniSat and Kissat are an order and two orders of magnitude larger than the available context; the resulting configurations improve on MiniSat on nine of twelve datasets and on Kissat on four. Subsequent work by the same group [10] targets a larger and more intricate codebase, constructing ModSAT—a modularised solver written to be amenable to LLM revision—exposing seven critical heuristics, and reporting solvers that improve on strong SAT baselines. A recent framework, LaGO [11], unifies part of this activity by decomposing automated heuristic design into a forward pass that evaluates a candidate injected into an algorithm skeleton, a backward pass that converts execution traces into structured feedback, and an update step that synthesises the next generation, showing that several popular methods—EoH, ReEvo, and LLM-LNS—are restricted instantiations of this structure. Across these works the recurring pattern is that an existing codebase is iteratively rewritten under an evaluator—frequently within a predefined set of modular slots, and typically tuned to perform well over a distribution or dataset of instances rather than a single input. This constraint is not peculiar to LLM-driven methods: Boualamia et al. [12] replace the roulette-wheel operator selection of an adaptive large neighbourhood search with a Q-learning agent choosing among the same fixed operators, reporting competitive or improved results on capacitated vehicle routing instances while the operators themselves, and the surrounding code, remain untouched.

Closest to our setting, Veličković et al. [5] apply this approach to combinatorial competitive programming, and specifically to past iterations of Google Hash Code. Their pipeline is offline and human-in-the-loop: competitors write an input parser, a greedy backbone, and an evaluator—typically around 200 lines of Python—FunSearch evolves the scoring function embedded within that backbone, and the humans may revise the backbone in response to what the search produces. As with the open-source competition solvers, and with our own runs, the setup is per-input: each provided instance is treated as a separate problem, and different inputs to the same round may receive different backbones. The resulting solvers break into the top percentile on every prior online qualification round and surpass the top human team on five of the eight, including the 2022 Mentorship and Teamwork round that we study; the approach is further validated on a held-out AtCoder Heuristic Contest problem. Their result demonstrates that LLM-guided evolution can amplify human-authored solutions on exactly the class of problems we study, but the amplified artefact is a human-written backbone, and the evolution is restricted to its scoring function.

Our work occupies a gap left open by these three lines. Improving mature code with an LLM in the loop is not itself new—AlphaEvolve does so on production systems at Google—but in the published work the artefact that is improved is, with that exception, written or selected for the method: FunSearch's skeleton, Veličković

et al.'s backbone, AutoSAT's EasySAT, Sun et al.'s ModSAT. Rather than evolving a designated function within a purpose-written backbone, we begin from the strongest publicly available open-source solver for the problem—the C++ submission of the Quiche LORIAinne team [4], which ranks first among the published open-source entries we identified for the 2022 Mentorship and Teamwork qualification problem, exceeding the competition winner on all five of its non-trivial instances and the best score reached during the live round on one of them (Section 4)—and improve it directly, as it stands, with no human in the loop. Because solvers for Hash Code are characteristically tuned per instance rather than written as general-purpose algorithms, we pair a single solver with a single instance and pursue deep, localised tuning of that pairing, in contrast to the distribution-level objectives of the solver-improvement literature. The improvement loop itself, MicroEvolve, is deliberately minimal: it maintains a single accepted version of the solver—not a population, archive, or island set—and applies one localised, exactly-matched textual edit per iteration, accepting any candidate that does not lower the score so that score-preserving edits can carry the search across plateaus. It introduces no training or fine-tuning and no auxiliary search machinery, and it permits edits anywhere in the solver rather than within a fixed modular search space. To our knowledge, the automated improvement of an existing state-of-the-art open-source solver, tuned to a single competition instance through such a minimal loop, has not previously been reported; it is the focus of this paper.

3 Problem Definition

The Mentorship and Teamwork problem was posed in the Qualification Round of the Google Hash Code 2022 competition. This section states the problem definition; because Google no longer maintains an official page for the round, the complete rules are taken from a public archive of past Hash Code competitions [13]. Section 3.1 describes what an instance consists of, Section 3.2 defines what a solution is, Section 3.3 states the feasibility rules, and Section 3.4 gives the scheduling and scoring rules. Section 3.5 illustrates all of them on a small example.

3.1 Instance

An instance consists of a set of contributors C, a set of projects P, and a set of skills.

Each contributor $c \in C$ holds every skill at an integer level; a skill the contributor has not acquired counts as level 0. We write $\ell_c(s)$ for the level at which contributor c holds skill s. Levels are not static: they increase as projects are completed, so $\ell_c(s)$ is a function of time and we write $\ell_c^t(s)$ when the distinction matters.

Each project $p \in P$ is described by three numbers and an ordered list of roles:

- a duration D_p, the number of days of work the project takes once it has started;
- a reward S_p, the score awarded for completing it;
- a best-before day B_p, after which the reward decays;
- an ordered list of roles $R_p = (r_{p,1}, \ldots, r_{p,|R_p|})$, where each role $r_{p,i}$ requires a single skill $s_{p,i}$ at a level $\lambda_{p,i}$.

3.2 Solutions

A solution specifies

1. which projects are executed — a subset $E \subseteq P$; and
2. for every executed project $p \in E$ and every role $r_{p,i}$ of that project, the contributor assigned to it.

Projects need not all be executed, since the constraints of Section 3.3 may make some of them unschedulable and the scoring rules of Section 3.4 may leave others worth nothing. The order in which the executed projects are listed is part of the solution: it fixes, for each contributor, the sequence in which that contributor performs the projects they are assigned to, from which the start day of every project follows deterministically (Section 3.4).

3.3 Feasibility

A solution is feasible if all of the following hold.

(C1) One contributor per role. Every role of every executed project is filled by exactly one contributor.
(C2) One role per project. A contributor fills at most one role on any given project; the contributors assigned to the roles of a project are therefore pairwise distinct.
(C3) Qualification. A contributor c assigned to role $r_{p,i}$ must satisfy $\ell_c^{t_p}(s_{p,i}) \geq \lambda_{p,i}$, where t_p is the start day of project p — *or* qualify through mentorship under (C4).
(C4) Mentorship. A contributor c with $\ell_c^{t_p}(s_{p,i}) = \lambda_{p,i} - 1$, that is exactly one level below the requirement, may still fill role $r_{p,i}$ provided some other contributor c' assigned to a different role of the same project satisfies $\ell_{c'}^{t_p}(s_{p,i}) \geq \lambda_{p,i}$. A mentor qualifies through their own actual level, may mentor several colleagues at once — including on the same skill — and may simultaneously be mentored by someone else on a different skill.
(C5) No overlap. A contributor works on at most one project at a time and never interrupts a project once started.
(C6) Validity is global. A single assignment that violates (C1)–(C5) invalidates the entire solution, which then scores nothing; an infeasible assignment is not merely skipped.

Two further rules govern how the state evolves rather than what is admissible at a given moment.

(G1) Skill growth. When project p completes, every assignee whose role required a level at least equal to their own level gains one level in that role's skill: $\ell_c(s_{p,i}) \leftarrow \ell_c(s_{p,i}) + 1$ whenever $\lambda_{p,i} \geq \ell_c(s_{p,i})$. A contributor who already exceeded the requirement gains nothing, and mentoring grants the mentor no increase.
(G2) Coupling of assignment and scheduling. Because levels rise as projects complete, whether a contributor qualifies for a role under (C3) or (C4) depends on which projects they finished earlier. The order of execution is therefore not a post-processing detail but part of the decision: the problem couples assignment with scheduling.

Table 1 Example instance: the level at which each contributor holds each skill. A dash denotes a skill the contributor has not acquired, which counts as level 0

Contributor	Python	C++	HTML
Lovelace	4	2	–
Dijkstra	–	5	–
Gödel	3	–	1
Murati	–	–	3
Pajaziti	–	–	1

3.4 Scheduling and Scoring

Every contributor is free from day 0. A project starts on the first day on which all of its assigned contributors are simultaneously free, occupies them for D_p consecutive days, and releases them afterwards. Start days are thus derived deterministically from the submission: each project starts as early as its contributors allow, given the order in which the projects appear. A project that starts on day t_p occupies days $t_p, \ldots, t_p + D_p - 1$ and is complete at the beginning of day $t_p + D_p$, which we denote f_p.

A project earns its full reward if its last working day falls strictly before its best-before day, that is if $f_p \leq B_p$. Otherwise it loses one point per day of lateness, and never falls below zero. The score of an executed project is therefore

$$\text{score}(p) \;=\; \max\bigl(0,\; S_p - \max(0,\; f_p - B_p)\bigr), \tag{1}$$

and the objective is to maximise the total

$$\max \sum_{p \in E} \text{score}(p) \tag{2}$$

over all feasible solutions. A project whose score has decayed to zero may still be executed: it earns nothing, but its assignees still gain levels under (G1), which may qualify them for later projects. Conversely, executing it consumes days that other projects could have used, so whether a zero-scoring project is worth executing is itself part of the optimisation.

3.5 Example

Tables 1 and 2 give a small instance with five contributors, five projects and three skills. Table 3 gives an optimal solution for it, worth 68 points, and Figure 1 shows the schedule that solution induces. Each of the main mechanisms of Sections 3.3 and 3.4 is illustrated by a different project, so the example can be read one project at a time.

Reading the example.

AuthService and AdminPanel both start on day 0, since their five contributors are distinct and all free. AuthService is the only project either Lovelace or Dijkstra can

Table 2 Example instance: the five projects, with duration D_p, reward S_p, best-before day B_p, and the skill and level required by each role

Project	D_p	S_p	B_p	Role 1	Role 2	Role 3
AuthService	3	20	6	Python 4	C++ 5	–
AdminPanel	2	10	4	HTML 3	HTML 2	HTML 2
DataPipeline	4	15	7	Python 5	Python 3	–
SearchIndex	5	25	10	C++ 6	C++ 3	–
BuildCache	5	3	2	C++ 6	Python 5	–

Table 3 An optimal solution for the instance of Tables 1 and 2, worth 68 points. A dagger (†) marks a contributor who qualifies through mentorship (C4) rather than by their own level; the skill column shows the growth granted by (G1) on completion

Project	Role	Contributor	Skill	Days	Score
AuthService	Python 4	Lovelace	4 → 5	0–2	20
	C++ 5	Dijkstra	5 → 6		
AdminPanel	HTML 3	Murati	3 → 4	0–1	10
	HTML 2	Pajaziti†	1 → 2		
	HTML 2	Gödel†	1 → 2		
DataPipeline	Python 5	Lovelace	5 → 6	3–6	15
	Python 3	Gödel	3 → 4		
SearchIndex	C++ 6	Dijkstra	6 → 7	7–11	23
	C++ 3	Lovelace†	2 → 3		
BuildCache	not executed			—	0
				Total	68

start immediately: no other contributor reaches Python 4 or C++ 5, and neither role is within one level of anybody else, so mentorship cannot help. Both fill their role at exactly the required level, so (G1) applies to both and they finish the project at Python 5 and C++ 6.

AdminPanel illustrates mentorship. Murati holds HTML 3 and fills the first role on her own; Gödel and Pajaziti hold HTML 1, one level below the two remaining roles, and qualify only because Murati is on the same project at level 3. Murati mentors both at once, which (C4) permits, and gains nothing for it — her own increase, from 3 to 4, comes from her own role under (G1).

DataPipeline and SearchIndex show why order matters. Neither can start on day 0: their first roles require Python 5 and C++ 6, levels that exist in this instance only after AuthService has been completed. The dependency is therefore not declared anywhere in the input but emerges from (G1). SearchIndex additionally reuses mentorship in the opposite direction, with Lovelace at C++ 2 filling the C++ 3 role under Dijkstra. Since Lovelace is assigned to three of the four executed projects, she is the bottleneck of the

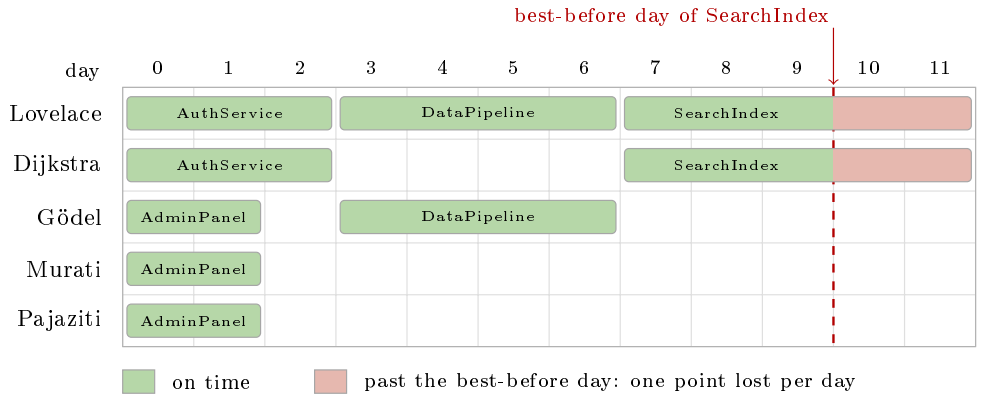


Fig. 1 Schedule induced by the solution of Table 3. Each row is a contributor and each column a day. SearchIndex is still within its best-before day for days 7–9 and beyond it for days 10–11, losing one point for each of those two days

instance, and the two projects that follow AuthService must be sequenced. Running DataPipeline first, as in Table 3, leaves it fully rewarded and pushes SearchIndex two days past its best-before day, for $15 + 23 = 38$ points; the reverse order rewards SearchIndex in full but pushes DataPipeline five days late, for $25 + 10 = 35$. The higher-reward project is not the one to schedule first here, because the penalty falls on whichever project is displaced and DataPipeline's deadline is the tighter of the two.

BuildCache is never executed. Its duration of five days against a best-before day of 2 means that its lateness is at least $5 - 2 = 3$ days however early it starts, so by (1) its score is $\max(0, 3 - 3) = 0$ for a start on day 0 and zero for every later start. It is feasible — Dijkstra and Lovelace qualify for its roles once AuthService is done — but it can never contribute, and executing it before the remaining projects would only occupy the two contributors those projects depend on. Exhaustive enumeration over all subsets, orders and role assignments confirms that 68 points is optimal for this instance and that the schedule of Figure 1 is the only one attaining it, up to exchanging Gödel and Pajaziti between AdminPanel's two identical roles and up to appending zero-scoring projects after day 11. The latter freedom is not vacuous: BuildCache may be run over days 12–16 by Dijkstra and Lovelace, who are idle by then, for a further 0 points, which is a second optimal solution with a different schedule.

4 Existing Solvers

The Mentorship and Teamwork problem was introduced in the Qualification Round of Google Hash Code 2022, which attracted 9,032 teams of up to four members. Following the competition, several teams released implementations and technical descriptions. Among the publicly available implementations identified in our review, the highest-scoring open-source solver we found is the C++ submission by Coudert-Osmont [4]. Its reported scores exceed those of the overall competition-winning team on all five non-trivial instances. The repository provides five instance-specific solver variants, one

for each of B–F, reflecting the extensive per-instance tuning used in Google Hash Code solutions.

Two academic studies also address this problem. Lajçi et al. [14] apply an iterated local search with random restarts, but do not achieve scores comparable with the strongest competition and post-competition solvers. Veličković et al. [5] use FunSearch to evolve the scoring function within a human-designed heuristic backbone. Their approach exceeds the competition winner on four of the five non-trivial instances. The per-instance solution scores were provided to us by the authors. As with the open-source implementations, a separate solver was produced for each instance.

Because the available methods were developed and tuned on an instance-by-instance basis, we evaluate B–F as five distinct solver-instance settings. The comparison includes (i) the score of the overall competition winner, (ii) the highest score obtained on each instance during the live round, (iii) the results of Veličković et al. [5], and (iv) the scores reported by Coudert-Osmont [4]. To keep the quantitative evidence together with the results it contextualises, all reference scores are reported alongside the MicroEvolve scores in the final comparison table in Section 7.2.

We derive the five baseline solvers from the Coudert-Osmont solution, in a fork of the original repository [15]. The changes are deliberately limited to selecting the strongest available solver variant for each instance and making minor parameter and execution adjustments. For instance B, we use solver D rather than the dedicated B solver and add a ten-minute wall-clock limit that aborts the construction and writes the best solution found so far. For instance C, we use an earlier variant of solver F. For instance D, we use solver C and increase its internal capacity parameter from 200 to 800. For instances E and F, we retain the original solvers unchanged.

These choices define the starting points for MicroEvolve rather than a new algorithmic contribution. All five baselines remain instance-specific variants of the original code. Four of them are deterministic functions of their input; the instance-B baseline, because of the wall-clock limit just described, is deterministic only under a fixed execution environment (Section 6.6). In the remainder of the paper, the label "solver B", for example, denotes the selected baseline used on instance B, irrespective of the filename or instance label of the variant from which it was derived. The corresponding baseline scores are reported with the final comparative results in Section 7.2.

4.1 Algorithmic Structure of the Baseline Solvers

All five baselines are greedy constructive heuristics: they commit feasible projects one at a time, update contributor availabilities and skill levels after each commitment, never revisit a committed project, and improve only through a pairwise role swap applied before a team is accepted (absent in the instance-F baseline). Solvers B, C, and F repeatedly select a single best candidate project while D and E process projects in ordered passes; Algorithms 1 and 2 make these two patterns explicit, and Table 4 specifies the solver-specific ranking, assignment, and stopping rules.

This representation makes explicit that the baseline solvers differ primarily in four design choices: project ordering or selection, contributor assignment, the treatment of skill-growth opportunities, and the stopping condition. The released source code provides the exact scoring expressions and implementation details used in each case.

Algorithm 1 Single-best-project construction (solvers B, C, and F)

Initialise contributor availability and skill levels
$U \leftarrow$ set of all unscheduled projects; $\Sigma \leftarrow \emptyset$ ▷ Σ: schedule under construction
while the solver-specific stopping condition is not satisfied **do**
    $K \leftarrow \emptyset$
    for all $p \in U$ **do**
        $a_p \leftarrow$ BestFeasibleAssignment$(p, \text{current state})$
        if a_p exists **then**
            $v_p \leftarrow$ ProjectValue$(p, a_p, \text{current state})$
            add (p, a_p, v_p) to K
        end if
    end for
    if $K = \emptyset$ **then**
        break
    end if
    $(p^*, a^*, v^*) \leftarrow \arg\max_{(p,a,v)\in K} v$
    commit (p^*, a^*); append p^* to Σ
    update contributor availability and skills; remove p^* from U
end while
return Σ

Algorithm 2 Repeated-pass construction (solvers D and E)

Initialise contributor availability and skill levels
$U \leftarrow$ set of all unscheduled projects; $\Sigma \leftarrow \emptyset$ ▷ Σ: schedule under construction
for all solver-specific phases q **do**
    repeat
        added $\leftarrow$ **false**
        order U using the phase-specific priority rule for q
        for all p in the resulting order **do**
            $a_p \leftarrow$ BestFeasibleAssignment$(p, \text{current state}, q)$
            if a_p exists and satisfies the acceptance condition for q **then**
                commit (p, a_p); append p to Σ
                update contributor availability and skills; remove p from U
                added $\leftarrow$ **true**
            end if
        end for
    until *added* = **false**
end for
return Σ

Table 4 Solver-specific instantiations of the two baseline construction templates

Solver	Project control and value	Contributor assignment	Acceptance and termination
B	Uses Algorithm 1. For every project, evaluates 50 randomized restarts, each a fresh role ordering, and retains the ordering with the most exact-level matches. Selects the project with the greatest skill-growth potential (`lvlup`).	Greedy, mentorship-aware assignment. Role orderings are generated by a fixed-seed pseudo-random generator, so the run is reproducible under a fixed execution environment.	Commits one highest-valued feasible project per iteration; stops when no further project can be selected, or when the added ten-minute wall-clock cap (Section 4) aborts the best-project scan, at which point the solution built so far is written.
C	Uses Algorithm 1. Ranks positive-score projects by benefit-to-time ratio; if the immediate score is zero, ranks them by level-ups per unit time.	For homogeneous skill projects, scans contributors sorted by availability to find the contiguous block that maximises score per unit time. Otherwise assigns roles greedily by lower availability, then higher skill, then lower total skill count.	Commits one highest-ranked project per iteration; stops when no project can be added or 3,200 projects have been scheduled.
D	Uses Algorithm 2 with one repeated phase. Orders projects by descending score per duration per role.	Tries 50 randomized restarts from a fixed seed, each a distinct role ordering. Within each restart, assigns contributors greedily by earlier availability, then lower skill, then lower total skill count.	Commits the best assignment only when it yields a positive net score; repeats full passes until a pass adds no project.
E	Uses Algorithm 2 with two phases. Phase 1 orders projects by increasing deadline minus duration. Phase 2 orders the remainder by descending score plus deadline minus duration.	Processes roles in input order using a mentorship-aware key: earlier availability, then lower skill, then lower total skill count. A mentor-swap step subsequently exchanges mutually qualifying contributors where beneficial.	Phase 1 accepts only projects that finish on time with positive score. Phase 2 permits late projects when their assignments still increase contributor skills. Each phase ends when a full pass adds no project.
F	Uses Algorithm 1 and normally ranks projects by benefit-to-time ratio. For zero-score projects, the potential number of skill level-ups replaces immediate score in the ranking.	For homogeneous skill projects above level 1, scans contributors sorted by availability for the contiguous block that maximises score per unit time. Otherwise assigns roles greedily by earlier availability, closer skill match, and lower total skill count. Zero-score assignments are re-optimised with the Hungarian algorithm to maximise level-ups.	Commits one highest-ranked project per iteration; stops when no project can be added or 3,200 projects have been scheduled.

5 MicroEvolve

5.1 Overview

MicroEvolve improves a single existing solver on a single problem instance. A run takes as input one baseline solver (Section 4) together with the corresponding instance file, and returns the best solver it has produced after a fixed budget of N iterations. Figure 2 shows the structure of the loop.

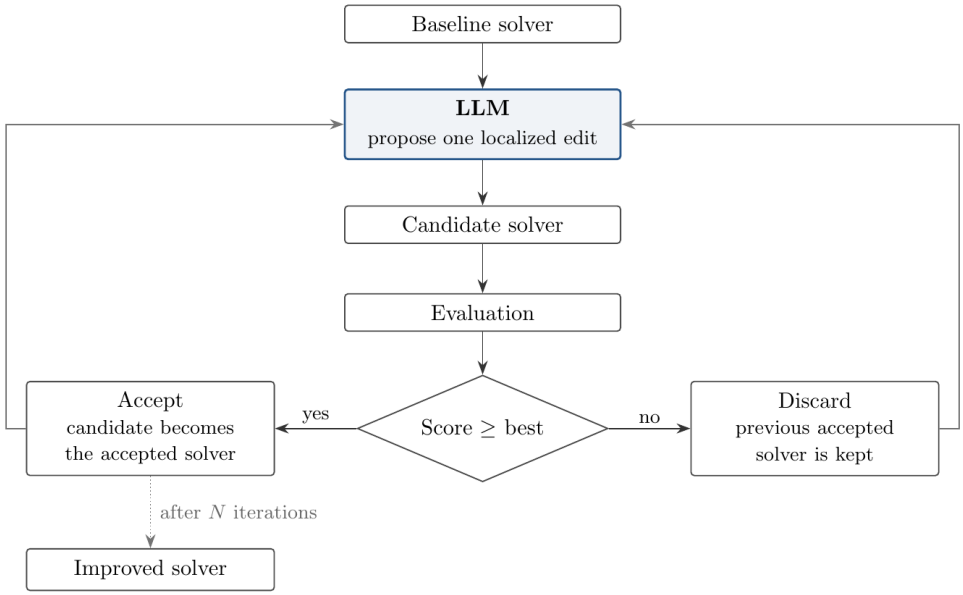


Fig. 2 High-level architecture of MicroEvolve. At each iteration the LLM proposes one localized edit to the accepted solver, and the resulting candidate is evaluated on the target instance. A candidate whose score matches or exceeds the current best is accepted and becomes the solver used in the next iteration; otherwise it is discarded and the previously accepted solver is kept. In each case the outcome is appended to the conversation and returned to the LLM, and the loop repeats. After N iterations the accepted solver is returned as the improved solver

Before the loop begins, the baseline solver is compiled, executed on the instance, and scored by the validator. That score becomes the initial best score, and the baseline source becomes the initial *accepted solver*. The accepted solver is the only version the loop carries forward: it is the best-scoring solver found so far, and every proposed edit is applied to it rather than to the candidate produced in the previous iteration, so that a discarded edit leaves no trace in the code.

Each iteration then performs one turn of the cycle in Figure 2. The language model is asked to propose a single localized modification to the accepted solver, expressed as a search-and-replace edit against its source. Applying the edit yields a *candidate solver*,

which is compiled, executed on the instance under a ten-minute wall-clock budget, and passed to an external validator that recomputes its score and checks feasibility.

The candidate is accepted if its score matches or exceeds the best score obtained so far, and discarded otherwise. On acceptance it becomes the new accepted solver and the best score is updated; on rejection the accepted solver is left unchanged. The acceptance test deliberately admits ties. Keeping an edit that leaves the score unchanged allows the search to drift across plateaus in the score landscape, so that a sequence of individually neutral edits can open the way to a later improvement; Section 8.4 examines how far the run traces bear this out.

Whichever outcome occurs, it is reported back to the model as the next turn of a single continuous conversation, which is the only memory the loop carries from one iteration to the next. After N iterations the accepted solver is returned as the improved solver.

The design is deliberately minimal. MicroEvolve maintains no population, archive, or island set of candidate programs; it performs no training or fine-tuning of the model; and it adds no search machinery beyond the single accepted solver, the edits applied to it, and the conversation that records what has been tried.

5.2 The Improvement Loop in Detail

Algorithm 3 states the loop precisely. The operation Append adds a feedback message to the conversation, and every edit is applied to the accepted solver A rather than to a previous candidate. The notation of Algorithms 1–3 is local to those algorithms and independent of the instance notation of Section 3.

Edits.

A proposed modification is returned as one or more search-and-replace blocks, each pairing a verbatim fragment of the current source with the text that should replace it. The blocks are applied by exact textual substitution, and each search fragment is required to occur exactly once in the accepted solver. This uniqueness requirement is what makes an edit unambiguously localized: a fragment that matches in several places, or in none, is rejected without being evaluated. The exact block syntax and the substitution procedure are given in Section 6.5.

Conversation and memory.

The loop is driven by a single, continuous conversation with the model. The system prompt is fixed for the whole run and contains the problem statement, the complete source of the baseline (version-0) solver, and the rules the model must follow. After each iteration the outcome is appended to the conversation as a new message: on a scored candidate, the score together with the accept-or-reject decision; on a failure, the reason. The updated source itself is never re-sent. The model therefore reconstructs the current state of the solver from the version-0 source in the system prompt together with the accepted edits recorded in the conversation, and this running history is the only memory the loop carries from one iteration to the next. Section 6.4 gives the prompt and the exact wording of each feedback message.

Algorithm 3 MicroEvolve improvement loop

Require: baseline solver source A_0; instance x; validator $\mathcal{V}$; iteration budget N
Ensure: improved solver source A^*; best score b

compile A_0, run on x, $b \leftarrow \mathcal{V}$(output) ▷ baseline score
$A \leftarrow A_0$; $A^* \leftarrow A_0$ ▷ A: current accepted solver
messages $\leftarrow$ [initial request to propose the first change]
for *it* $\leftarrow 1$ **to** N **do**
  $r \leftarrow \textsc{Llm}(system_prompt, messages)$
  Append r to *messages*
  $\Delta \leftarrow \textsc{ParseBlocks}(r)$ ▷ Δ: set of search-and-replace blocks
  if $\Delta = \emptyset$ **then**
    Append "no valid search/replace block"; **continue** ▷ parse failure
  end if
  $A' \leftarrow \textsc{Apply}(A, \Delta)$ ▷ A': candidate; each search fragment must match A exactly once
  if Apply failed **then**
    Append apply-error detail; **continue** ▷ apply failure
  end if
  if not $\textsc{Compile}(A')$ **then**
    Append compiler output; **continue** ▷ compile failure
  end if
  $z \leftarrow \textsc{RunAndScore}(A', x, \mathcal{V})$
  if $z = \perp$ **then**
    Append failure detail; **continue** ▷ runtime / infeasible
  end if
  if $z \geq b$ **then**
    $b \leftarrow z$; $A \leftarrow A'$; $A^* \leftarrow A'$ ▷ accept (improvement or tie)
    Append "score $= z$, accepted"
  else
    Append "score $= z$, rejected" ▷ reject; A unchanged
  end if
end for
return A^*, b

Per-iteration outcomes.

Every iteration therefore ends in one of seven outcomes: an accepted candidate that improves the score, an accepted candidate that ties it, a rejected candidate, or one of four failures — a response from which no valid block could be parsed, a search fragment that did not match uniquely, a compilation error, or a runtime or feasibility failure. All four failures leave the accepted solver untouched, exactly as a rejection does, but they differ from a rejection in that the candidate never receives a score. MicroEvolve records which outcome occurred at each iteration; these counts form the basis of the analysis in Section 7.3.

Scoring of candidates.

Each candidate is scored by a single execution on the target instance, so the acceptance decision rests on one measurement. The conditions under which this is sound, and the one baseline for which they hold only approximately, are discussed in Section 6.6.

5.3 Worked Example

To make the loop concrete, we trace four consecutive iterations from early in a representative run on instance B, which starts from a baseline score of 1,032,469. The four illustrate, in turn, an accepted improvement, a rejection, an accepted tie, and a compile failure.

Iteration 1: accepted improvement.

The model targets the rule by which the solver chooses which project to commit a contributor to next. The original code keeps the candidate project with the greatest level-up gain, and the model adds a tie-break:

```
// before
if(lvlup > bestlvlup) { ... }
// after
if(lvlup > bestlvlup || (lvlup == bestlvlup && (bestEnd == -1 || beste < bestEnd))) { ... }
```

so that, among projects offering the same gain, the one that finishes earlier (a smaller completion day `beste`) is preferred, freeing contributors sooner and reducing exposure to lateness penalties. The candidate compiles, runs, and scores 1,044,071, an improvement of 11,602 over the baseline, and is accepted as the new best.

Iteration 2: rejection.

The model increases the number of randomized restarts from 50 to 120:

```
// before
for(int test = 0; test < 50; ++test)
// after
for(int test = 0; test < 120; ++test)
```

but the resulting solver scores 1,032,987, below the current best of 1,044,071, so the candidate is discarded and the accepted solver is left unchanged. Because the solver uses a fixed seed, raising the number of randomized restarts does not extend the previous search but redirects it along a different sequence of random constructions, and within the ten-minute budget that trajectory settles on a worse result than the one the tie-break had reached.

Iteration 3: accepted tie.

The model makes a change analogous to the first — an earliest-completion tie-break, this time in a different selection step:

```
// before
if(oo > besto)
// after
if(oo > besto || (oo == besto && mav+D[p] < beste))
```

and the candidate scores exactly 1,044,071, equal to the current best. Because the acceptance test admits ties, the edit is kept even though it does not improve the score. Such neutral edits accumulate and can set up a later improvement, a pattern examined in Section 8.4.

Iteration 4: compile failure.

The model attempts to reweight the level-up score and add a deadline-aware bonus, but its response leaves a stray block delimiter (`=======`) in the replacement text, which is written into the source. The compiler rejects it — `error: version control conflict marker in file` — so the candidate is discarded before it can be scored, and the compiler's message is returned to the model as feedback, allowing it to correct course on the following iteration.

6 Experimental Setup

This section reports the hardware, the language-model configuration, the evaluation pipeline, and the run protocol used in all MicroEvolve runs. The full system prompt and the exact feedback messages are given in Appendix A.

6.1 Hardware and Software

All experiments were executed on a single machine without parallelism. The machine has an AMD Ryzen 5 5600H processor (6 cores, 12 threads, up to 4.28 GHz) and 16 GB of RAM, and runs Ubuntu 22.04.5 LTS (kernel 6.8.0, x86_64). Candidate solvers were compiled with g++ 11.4.0 at optimization level `-O2` under the C++17 standard.

6.2 Baseline Solvers

The starting point for each instance is the adapted Coudert-Osmont solver described in Section 4. Each is a single C++ file of modest size: 496 lines for instance B, 195 for C, 138 for D, 127 for E, and 281 for F. Each file was frozen before the runs and served both as the initial accepted solver and as the version-0 reference embedded in the system prompt.

6.3 Language Model and API

We use Claude Opus 4.8 (`claude-opus-4-8`, as available through the Anthropic API in June–July 2026). Each call allows at most 16,000 output tokens; temperature, top-p, and top-k are left at the provider defaults. The system prompt, which contains the full baseline source, is marked for ephemeral prompt caching, so that consecutive calls falling within the cache window reuse it rather than re-sending it. Token usage is logged per call and converted to cost at the published per-million-token prices of US $5.00 (input), $25.00 (output), $0.50 (cache read), and $6.25 (cache write); the resulting figures are reported in Section 7.4.

6.4 Prompt and Feedback

Each run uses a fixed system prompt containing the problem statement archived by Avner [13], the complete version-0 source for the target instance, the baseline score on that instance, and the rules the model must follow: propose exactly one change per turn as one or more search-and-replace blocks, match the code verbatim, keep the solver deterministic, respect the ten-minute budget, and never repeat a change already proposed. The conversation opens with a single request for the modification the model expects to have the largest positive effect on the score.

After each iteration the loop appends one standardized feedback message as the next user turn. For a scored candidate the message gives the new score, the previous best, and the accept-or-reject decision; for a failed iteration it gives the failure reason, which is the parse error, the apply error, the compiler's standard error truncated to 1,500 characters, or the validator's rejection detail. The messages carry no iteration number or other metadata, and never include the updated source, so the model must track the state of the solver from the conversation alone. The prompt and all feedback templates are reproduced verbatim in Appendix A.

6.5 Evaluation Pipeline

Each proposed modification must be expressed as at least one block of the form

```
<<<<<<< SEARCH
... verbatim code from the accepted solver ...
=======
... replacement code ...
>>>>>>> REPLACE
```

The pipeline locates each SEARCH string in the current accepted source. A string that does not occur exactly once causes an apply error and the iteration is abandoned. Otherwise the substitution is performed, the modified solver is written to disk and compiled, and the resulting binary is executed with the competition instance on standard input. Execution is limited to 600 seconds of wall-clock time; a solver that exceeds this limit is terminated and the iteration is recorded as a runtime failure. The output is passed to an open-source JavaScript validator [16], which recomputes the score and verifies feasibility against the rules of Section 3.3. A submission that violates any rule is rejected and the validator's message is returned to the model; a valid submission yields the integer score on which the acceptance decision is made.

6.6 Determinism and the Basis of the Acceptance Decision

Each candidate is scored by a single execution, so the acceptance decision of Section 5.1 rests on one measurement. This is sound only to the extent that a candidate's score is reproducible, and the system prompt therefore instructs the model to keep the solver deterministic and to use a hard-coded fixed seed for any pseudo-random generator. Randomization is permitted, but only in a form that yields the same output on every execution.

Two qualifications apply. The first concerns the baseline. Of the five, only the instance-B baseline reads a clock: it includes the ten-minute wall-clock cap that aborts

the construction and writes the best solution found so far (Section 4), so its output is in principle sensitive to machine speed and load rather than being a pure function of the input. The other four are pure functions of their input. Because all runs were executed sequentially on the single machine of Section 6.1, the execution environment was held constant across every iteration of every instance-B run. We nevertheless note that instance-B scores are reproducible under this configuration rather than hardware-independent, and that a candidate whose score sits within the measurement noise of the incumbent could in principle be accepted or rejected on a single noisy comparison.

The second concerns the evolved solvers. All five instance-B lineages inherit the clock read from their baseline, as expected. Beyond these, two runs—one on instance E and one on instance F—introduced wall-clock timing of their own, despite the system prompt's explicit determinism requirement. In both cases the fitness gate accepted the resulting solver, because a single execution cannot detect a violation of a reproducibility contract: the acceptance decision tests the score, not the property that makes the score meaningful. This is a limitation of single-execution scoring rather than of the prompt, and a check that re-runs an accepted candidate and compares scores would catch it. More generally, the method as described is not applicable without modification to solvers whose score depends on wall-clock timing or on non-fixed randomness; we return to this in Section 8.6.

6.7 Runs and Iteration Budget

For each of the five non-trivial instances we performed five independent runs, each starting from the identical frozen baseline in a separate conversation; because model sampling is stochastic, the runs diverge without any explicit diversification mechanism. Every run consisted of exactly 100 iterations with no early stopping, giving 25 runs and 2,500 proposed modifications in total. The pipeline checkpoints after each iteration and retries on API errors with exponential backoff, terminating a run only if eight consecutive attempts fail; in practice all 25 runs completed their full budget without forced termination.

7 Results

We report the outcome of the 25 MicroEvolve runs: five independent runs on each of the five non-trivial instances, every run consisting of exactly 100 iterations starting from the frozen baseline, for 2,500 proposed modifications in total. Four views summarise them. Section 7.1 reports what the loop achieved relative to its own starting point; Section 7.2 places those results against prior work; Section 7.3 reports the per-iteration outcomes that produced them, and Section 7.4 their computational cost.

7.1 Improvement over the Baseline

Following the competition convention, the score attributed to an instance is the best obtained for it, here the best of its five runs. Table 5 reports this alongside the mean and the worst of the five, so that the spread is visible next to the headline figure.

Table 5 Improvement over the baseline. The baseline is the adapted Coudert-Osmont solver of Section 4; the worst, mean, and best rows are taken over the five independent runs on each instance

	B	C	D	E	F
Baseline	1,032,469	255,918	2,608,736	1,648,976	899,143
MicroEvolve, worst of 5	1,044,428	275,615	2,723,747	1,651,070	914,400
MicroEvolve, mean of 5	1,049,603	280,839	2,795,662	1,653,702	1,105,388
MicroEvolve, best of 5	1,057,446	285,714	2,853,365	1,656,494	1,381,929
Gain, best run (abs.)	+24,977	+29,796	+244,629	+7,518	+482,786
Gain, best run (%)	+2.42	+11.64	+9.38	+0.46	+53.69
Gain, mean run (%)	+1.66	+9.74	+7.17	+0.29	+22.94
Gain, worst run (%)	+1.16	+7.70	+4.41	+0.13	+1.70

MicroEvolve improves the baseline on every instance, and does so in every run: all 25 runs finished with a best score strictly above their starting point, so the improvement is not an artefact of the best-of-five convention. What varies is the magnitude. Instance E moves least, by +7,518 points (+0.46%); instance F moves most, from 899,143 to 1,381,929 (+53.69%). Instances C and D gain substantially in relative terms (+11.64% and +9.38%), which on D amounts to +244,629 points in absolute terms, the second-largest absolute gain of the five.

The spread across runs is itself instance-dependent, and the gap between the best-run and mean-run columns is the clearest indicator of it. On B, C, D, and E the two are close, so any single run lands near the reported best; E ranges only from 1,651,070 to 1,656,494 across its five runs. On F they diverge sharply, the best run gaining 53.69% against a mean of 22.94%, because three of the five F runs settle near the baseline (914,400, 920,571, 948,294) while two reach a far better region (1,361,746 and 1,381,929). Section 8.3 accounts for this bimodality.

7.2 Comparison with Prior Results

Table 6 places these scores against the external results introduced in Section 4. The baseline is reported as a separate row, because it does not stand in the same relation to prior work on every instance, and the contribution of the adaptation must therefore be distinguishable from that of the loop.

MicroEvolve sets a new best-known score on four of the five instances: B, D, E, and F. On C it improves the baseline by 11.64% and exceeds the competition winner, Veličković et al., and Coudert-Osmont, but its 285,714 remains 0.97% below the 288,508 reached by *Scheming Kittens* during the live round, so we do not claim a new best-known score there.

The two denominators must be kept apart when reading the margins. On instance D the adaptation of Section 4 already exceeds the best prior result by 19.75%, and MicroEvolve adds a further 9.38% on top of it; the resulting +30.98% over the previous best-known score is the product of both, and should not be attributed to the loop alone. On instance F the relationship is reversed: the baseline sits 24.73% *below* the best prior

Table 6 Comparison with prior results. The baseline is our adaptation of the Coudert-Osmont solver (Section 4). Entries marked † exceed every external result we are aware of. The competition winner was *Make love, not war*; the per-instance best scores were reached by *<(OvO)>* (B), *Scheming Kittens* (C), *Make love, not war* (D), *QR-Coders* (E), and *Proof by Submission* (F), each uniquely, as verified against the full 9,032-team archived scoreboard [13]

	B	**C**	**D**	**E**	**F**
Competition winner	969,087	229,517	674,945	1,640,454	706,200
Competition per-instance best	1,005,020	288,508	674,945	1,650,488	1,194,515
Veličković et al.	969,376	250,129	2,063,388	1,637,258	953,685
Coudert-Osmont	1,003,496	242,898	2,178,519	1,648,976	899,143
Best prior result	1,005,020	288,508	2,178,519	1,650,488	1,194,515
Our baseline	1,032,469	255,918	2,608,736	1,648,976	899,143
vs. best prior (%)	+2.73	−11.30	+19.75	−0.09	−24.73
MicroEvolve, best of 5	1,057,446†	285,714	2,853,365†	1,656,494†	1,381,929†
vs. best prior (%)	+5.22	−0.97	+30.98	+0.36	+15.69

result, and the entire +15.69% margin over that result is produced by MicroEvolve. Instances B and E lie between these cases, with the baseline slightly above and slightly below the best prior result respectively, so that most of the final margin comes from the loop. The margin on E is the narrowest of the four new best-known scores, at +0.36% over a figure it improves by 6,006 points.

Read against the live scoreboard as a whole, no team among the 9,032 that scored on this round exceeded our best run on B, D, E, or F, and exactly one did so on C. This comparison carries the caveat of Section 7.4: competitors worked under a fixed multi-hour deadline, whereas a single MicroEvolve run occupies far longer, so the comparison is one of attainable score rather than of equal effort.

7.3 Iteration-Level Outcomes

Table 7 aggregates, for each instance, the seven per-iteration outcomes of Section 5.2 over its five runs, together with three derived rates. All 25 runs completed their full budget without forced termination.

Two patterns stand out. First, accepted edits are dominated by ties rather than improvements: 1,095 of the 1,257 accepts (87%) leave the score unchanged, against 162 that raise it, so only 6.5% of all iterations produce an improvement. Because the acceptance test admits ties, these score-preserving edits accumulate freely; instance E is the extreme case, with a 79.0% acceptance rate of which 348 of 395 accepts are ties. Section 8.4 examines whether this accumulation contributes to the improvements or is merely tolerated by the acceptance rule.

Second, the share of the budget lost to failed edits varies by almost an order of magnitude across instances, from 1.8% on D to 22.6% on F, and the dominant failure mode shifts with it. On C the losses are almost entirely apply failures (94 of 95), edits whose search fragment did not match the accepted source exactly once. On F they are mostly runtime or feasibility failures (68 of 113), edits that compiled but either

Table 7 Per-iteration outcomes, aggregated over the five runs of each instance (500 iterations per instance, 2,500 in total). The acceptance rate counts improving and tie accepts; the failure rate counts parse, apply, compile, and runtime failures. Rates use a nominal denominator of 500 iterations per instance. In one instance-F run a single iteration is not attributable to any recorded category, most plausibly an unlogged parse failure, so the instance-F counts total 499; the reported rates are unaffected except for instance F's acceptance rate, which would read 38.5% on a denominator of 499

Outcome	**B**	**C**	**D**	**E**	**F**	**All**
Accepted — improvement	33	30	21	47	31	162
Accepted — tie	159	228	199	348	161	1,095
Rejected (strictly worse)	269	147	271	69	194	950
Parse failure	0	0	0	0	0	0
Apply failure	30	94	8	18	44	194
Compile failure	7	0	0	4	1	12
Runtime / feasibility failure	2	1	1	14	68	86
Acceptance rate (%)	38.4	51.6	44.0	79.0	38.4	50.3
Improving acceptances (%)	6.6	6.0	4.2	9.4	6.2	6.5
Failure rate (%)	7.8	19.0	1.8	7.2	22.6	11.7
Runs improving baseline	5/5	5/5	5/5	5/5	5/5	25/25

exceeded the ten-minute budget or produced an infeasible schedule. Compile failures are rare throughout, 12 across all 2,500 iterations, and no iteration in any run ended in a recorded parse failure. Section 8.5 interprets these differences.

7.4 Computational Cost

Table 8 reports token usage and cost, computed from the API token logs and the prices of Section 6.3.

Table 8 Computational cost and token usage over 100 iterations. Token figures are in thousands, means per run over the five runs of each instance; the *Overall* column is the mean per run across all 25 runs, except in the final row, which gives the grand total

Per-run mean	**B**	**C**	**D**	**E**	**F**	**Overall**
Input tokens (k)	1215.6	942.4	1006.5	1698.2	1048.5	1182.2
Output tokens (k)	69.9	72.6	97.4	108.3	84.3	86.5
Cache-read tokens (k)	525.1	602.5	473.9	321.0	72.3	399.0
Cache-write tokens (k)	6.4	4.9	3.8	142.9	597.3	151.0
Cost per run (US$)	8.13	6.86	7.73	12.25	11.12	9.22
Total, 5 runs (US$)	40.64	34.30	38.64	61.26	55.60	230.44

A 100-iteration run costs between US$6.86 (instance C) and US$12.25 (instance E) on average, and the full study costs US$230.44. The two most expensive instances become so for different reasons. Instance E is driven by input volume, carrying the largest mean input-token count, which reflects long conversations in which many accepted edits, mostly ties, accumulate in the history the model re-reads each turn. Instance F is driven instead by cache writes, priced at the highest input-side rate, which alone add roughly US$3.70 per run; its low cache-read figure indicates that consecutive calls less often fell within the ephemeral cache window, so the long system prompt was re-cached rather than re-read. The cheaper instances combine moderate input sizes with high cache-read reuse and negligible cache writes.

In absolute terms the method is inexpensive: a 100-iteration run costs about nine dollars in model calls on average, and the best-of-five protocol behind the headline scores of Section 7.1 costs between US$34.30 and US$61.26 per instance, with no training, no fine-tuning, and no human intervention during a run. The dominant cost is not monetary but wall-clock. Per-iteration timing was not instrumented, so we state a bound rather than a measurement. Of the 100 iterations of a run, a mean of 91.7 reached the execution stage—the remainder failed to parse, apply, or compile and consumed no solver time—and each execution was capped at 600 seconds (Section 6.5), which bounds a run at roughly 15 hours and the five runs of an instance at roughly 76 hours on the single machine of Section 6.1. The cap was actually reached in 75 executions across all 25 runs, and these were concentrated in the two instances whose solvers search until they are stopped: a mean of 12.0 per run on instance F and 2.6 on instance E, against at most 0.2 per run on B, C and D. Executions elsewhere terminate well inside the budget, so realised times were correspondingly lower; we report the bound because it is what the released traces support.

8 Discussion

The 25 runs let us look past the headline scores of Section 7 and ask *how* MicroEvolve improved the solvers: what kinds of edits carried the gains, when in a run they arrived, why some instances were reliable and others were not, and what role the tie-accepting rule played. We classified each of the 162 score-improving edits — the accepted edits that strictly raised the best score, pooled across all five runs of an instance — into one of five categories: tie-break additions, selection-criterion changes, scoring-formula reweightings, numeric parameter tuning, and algorithmic or structural changes. The per-edit score gains reported below are computed directly from the run traces as the increase in best score at the iteration where each edit was accepted, and they reconcile exactly with the total baseline-to-final improvement of every run.

8.1 Each Instance Has a Signature Edit Type

Table 9 shows that the distribution of improving edits is far from uniform across instances; each solver is improved through a characteristic kind of change. The characterisations below are stated in terms of the *share of the total gain* a category contributes, which is not the same ordering as the edit counts in Table 9. The two diverge because of the introduce-then-tune rhythm described below: a mechanism is

introduced by one scoring or structural edit and then refined by several parameter edits, so parameter tuning is often the most numerous category on an instance while contributing less of the gain than the rarer edit that opened the mechanism it tunes.

On instance B the gains come mainly from *tie-breaks* — secondary comparisons added to the greedy selection so that, among options the primary key rates equally, the solver prefers the one that (for example) finishes earliest or carries the higher project score. On instance C the gain comes chiefly from *reweightings of the scoring formula* and the subsequent tuning of their coefficients, adjusting the balance between reward, completion-time, idle-time and duration terms; not a single tie-break appears. On instance D the dominant lever is *parameter tuning* — above all the number of randomized restarts and the random seed — supplemented by a few structural changes; here the count and the gain agree. Instance E is the least concentrated of the five: its largest individual contributions are *structural*, chiefly reordering a project's roles by descending required skill level and adding a perturbation-based multi-start, but these are only 6 of its 47 improving edits, and most of the remainder is a long tail of small reweightings and tie-breaks that together account for a comparable share of the gain. Instance F is driven by *scoring reweighting*, specifically the introduction of a penalty proportional to a project's completion day, which alone accounts for most of the gain in the two runs that found it (Section 8.3). The model was never told which kind of edit suited which solver; it inferred an instance-appropriate strategy from the source and the feedback alone.

Table 9 Improving edits by category, per instance (162 edits total, pooled over the five runs of each instance). Each edit is assigned its single dominant effect

Edit category	**B**	**C**	**D**	**E**	**F**	**All**
Tie-break addition	13	0	3	15	5	36
Selection-criterion change	6	3	2	3	2	16
Scoring-formula reweighting	0	11	3	15	10	39
Parameter tuning	11	16	10	8	13	58
Algorithmic / structural	3	0	3	6	1	13
Total	33	30	21	47	31	162

A recurring pattern within these categories is a two-phase rhythm: the model *introduces* a new term or mechanism in one edit, then spends several later iterations *tuning* its coefficient. On instance C, for instance, one run adds a bonus proportional to the raw project score and then walks its coefficient through $1.0\times10^{-6} \rightarrow 1.2\times10^{-6} \rightarrow 1.3\times10^{-6}$; on instance F a completion-day penalty is introduced and then re-tuned across a sequence of nearby values. This rhythm is also why parameter tuning is the single largest category (58 of 162 edits): much of the search budget is spent exploiting a mechanism that an earlier edit discovered. The same motifs, moreover, are rediscovered independently across runs — an earliest-completion tie-break appears in all five B runs, role-ordering-by-required-level in four E runs, and the completion-day penalty in

the two successful F runs — which indicates that the loop is locating genuine structure in each problem rather than exploiting run-specific noise.

8.2 Improvements Arrive Early and Then Saturate

Figure 3 plots, for each instance, the cumulative gain as a fraction of the run's final gain against the iteration number, showing both the five individual runs and their mean. The curves rise steeply and then flatten: averaged over runs, roughly 62–75% of the eventual gain is already secured by iteration 25, 87–98% by iteration 50, and at least 95% by iteration 75. Very few improvements land in the final quarter of a run. This has a direct methodological consequence. A natural way to seek larger gains would be to run the loop for longer — 200, 300, or 500 iterations — but the saturation in Figure 3 suggests that, with the present single-conversation design, additional iterations on the *same* trajectory would yield little. The more promising lever is diversity of trajectories rather than their length: because independent runs diverge (Section 8.3), the budget may be better spent on more runs, or on mechanisms that restart or perturb the search, than on extending any one run.

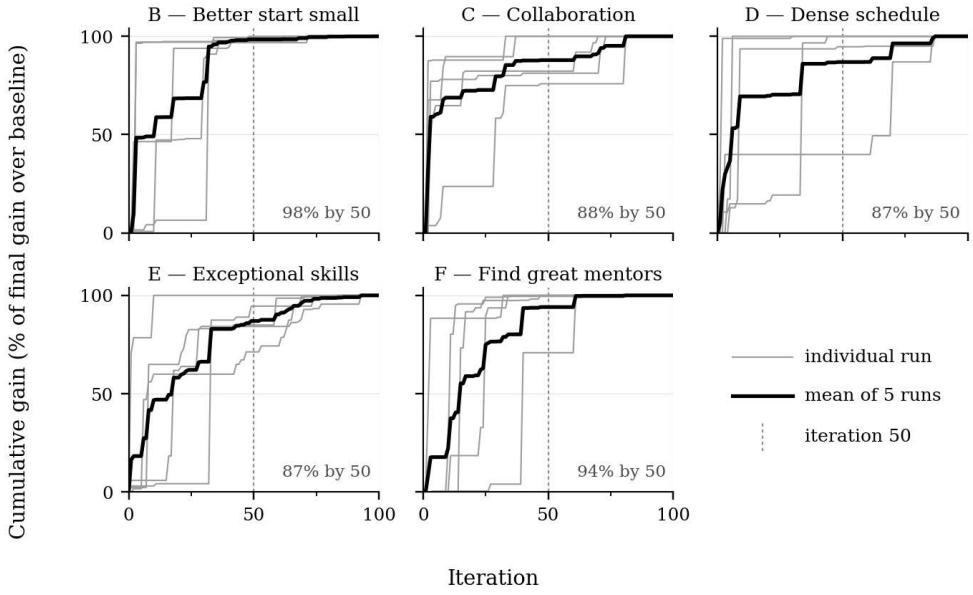


Fig. 3 Convergence of MicroEvolve, one panel per instance. Each grey curve is a single run's cumulative score gain, expressed as a percentage of that run's final gain over the baseline, against iteration number; the heavy black curve is the mean over the five runs. The annotation in each panel gives the mean percentage of the final gain secured by iteration 50 (dashed line). Improvements concentrate in the first half of the run and largely saturate thereafter

8.3 Reliability Is Governed by the Redundancy of Breakthroughs

Instances differ sharply in how concentrated their gains are. Measured per run, the single largest edit accounts on average for about 75% of the run's total gain on B, 70% on D, 65% on C, 61% on E, and 60% on F, so every instance depends substantially on one large edit, with B the most concentrated. Concentration alone, however, does not explain the between-run variance reported in Section 7: instances B, C, D and E produce tightly clustered final scores across their five runs, whereas instance F is strongly bimodal, with two runs near 1.36–1.38 million and three stalled near 0.91–0.95 million.

The traces show that the deciding factor is not *whether* a run needs a breakthrough but *how many distinct* breakthroughs are available to it. Instance F hinges on a single idea — subtracting a term proportional to a project's completion day from the greedy selection score, which pushes the solver to finish projects sooner and free contributors earlier. The two successful runs discovered this penalty early (at iterations 11 and 15, worth +268,548 and +353,898 respectively) and immediately crossed one million; the three stalled runs never sampled it, their largest single edits worth only about +10,000 to +28,000, and none ever reached one million. The per-run scores reported in Section 7.1 show the split directly. Instance D also depends on large edits, yet stays reliable, because it offers *several* independent ones: raising the number of randomized restarts, changing the random seed, and adding a multi-seed outer loop each yield between roughly +130,000 and +189,000 on their own, and four of the five D runs found at least one such edit (the fifth still improved by about +115,000 through a set of mid-sized edits). Reliability, then, is a property of the *search landscape the solver exposes to the model*: when high-value edits are redundant, most runs find one and the outcome is stable; when the gain depends on a single narrow discovery, the method becomes a lottery over whether that discovery is sampled within the budget. This also reframes instance F's headline +53.69%: it is real and reproducible — two independent runs reached it — but it is not something a single run can be relied upon to reproduce, and the best-of-five reporting convention is what surfaces it.

8.4 The Role of Score-Preserving Edits

MicroEvolve accepts any edit that does not lower the best score, on the hypothesis that score-preserving (tie) edits let the search drift across plateaus toward a later improvement. The traces give this qualified support. Ties dominate the accepted edits — 1,095 of 1,257 accepts (87%) leave the score unchanged — and a majority of the improving edits are reached across at least one preceding tie: 57% of improvements are immediately preceded by one or more score-preserving accepts, with a mean run of about four ties before an improvement. Improvements do therefore frequently arrive after the search has moved along a plateau rather than from the exact state that produced the previous best. We stop short of claiming that ties are *necessary* — many improvements follow directly from a prior improvement, and the two-phase introduce-then-tune rhythm means some "ties" are simply coefficient nudges that neither help nor hurt — but the data are consistent with the design rationale, and an explicit

ablation against a strict-improvement acceptance rule (noted in Section 10) would settle the question.

8.5 Failure Modes

The share of iterations lost to failed edits varies widely: on average per run, D wastes only 1.8% of its iterations (about 1.8 per 100-iteration run), against 19.0% on C and 22.6% on F. The dominant failure mode also differs. On C the losses are overwhelmingly apply failures — edits whose SEARCH fragment did not match the accepted source exactly once — which is consistent with the model having to reconstruct the current source from the version-0 solver plus the edit history rather than being shown the updated file. On F the losses are mostly runtime or feasibility failures, reflecting edits that compiled but either exceeded the ten-minute budget or produced an infeasible schedule; combined with F's single-breakthrough fragility, this lost budget compounds its unreliability. Compilation failures are rare throughout (12 across all 2,500 iterations), and exactly one instance-F iteration is unaccounted for in the logs, most plausibly a parse failure. That the model sustains a low structural-error rate while editing C++ solvers of up to 496 lines, blind to the current file state, is itself a notable robustness result.

8.6 Limitations

Several limitations qualify these findings. First, the reported per-instance scores are the best of five runs, following the competition convention; the mean-of-five figures in Table 5 are a fairer estimate of a single run's expected outcome, and on the high-variance instance F the two differ substantially. Second, five runs per instance is a small sample, adequate for demonstrating the effect but not for tight estimates of its variance. Third, every result uses a single model, Claude Opus 4.8; we cannot separate the contribution of the loop from that of the particular model, and the sensitivity of the method to the underlying model remains open (Section 10). Fourth, each candidate is scored by a single execution, so the acceptance decision assumes that a candidate's score is reproducible. That assumption holds exactly for four of the five baselines, holds for the instance-B baseline only under a fixed execution environment, and was violated outright by two evolved solvers that introduced wall-clock timing of their own (Section 6.6); the loop has no mechanism to detect such a violation, and applying it to solvers whose score depends on timing or non-fixed randomness would require repeated scoring. Fifth, and most fundamentally, MicroEvolve is tuned to one solver–instance pairing at a time by design; we make no claim that an improved solver generalizes to other instances or that the method produces general-purpose algorithms, and the per-instance signature of the edits (Table 9) suggests the improvements are indeed instance-specific. Finally, the instance-F results are a concrete caution that a strong best-of-n score can rest on a discovery that most individual runs miss.

9 Conclusion

We introduced MicroEvolve, a deliberately minimal LLM-in-the-loop method for improving an existing high-performance solver. It proposes one localized, exactly-matched textual edit per iteration, accepts any candidate that does not lower the score, and carries forward a single accepted version, with no population, archive, island set, training, fine-tuning, or auxiliary search machinery. Applied to the Mentorship and Teamwork problem from Google Hash Code 2022 and driven by a single model, the method improved its baseline on all five non-trivial instances, and did so in every one of the 25 runs rather than only in the best of each five.

The margins depend on which starting point they are measured against, and the two should not be conflated. Measured against the baseline, the best run of each instance gains between 0.46% and 53.69%. Measured against the best prior result we are aware of, MicroEvolve sets new best-known scores on four of the five instances. The largest of those margins, 30.98% on instance D, is the product of two contributions rather than one: our selection and minor adaptation of the open-source solver accounts for 19.75% of it and the loop for a further 9.38% on top. The clearest single-instance case for the method is instead F, where the baseline begins 24.73% *below* the best prior result and the entire 15.69% margin over that result is produced by the loop. On instance C the method improves the baseline substantially but remains just under the best score reached during the live contest, so we claim no new best-known score there.

Beyond the scores, the run traces yield a consistent account of how the method works. Each solver is improved through a characteristic type of edit that the model settles on without being told which kind suits which solver; improvements concentrate in the first half of a run and then saturate, so that additional iterations along one trajectory buy little; and the reliability of a run is governed less by how large its decisive edit must be than by how many independent high-value edits the solver's search landscape exposes to the model. That last observation is what separates instance D, where several distinct breakthroughs are available and every run finds one, from instance F, where a single narrow discovery decides the outcome and only two runs of five sample it. The tie-accepting rule is exercised heavily—87% of accepted edits leave the score unchanged—and the traces are consistent with its rationale without establishing that it is necessary. The method is inexpensive, at roughly nine dollars per 100-iteration run, and requires no human intervention once the initial solver is chosen; its real cost is wall-clock time rather than money.

These results should be read as a demonstration rather than a controlled attribution. We do not separate the contribution of the loop from that of the particular model, we report five runs per instance, and each improved solver is tuned to one instance with no claim of generality (Section 8.6). What the study does establish is narrower and, we think, still worth stating: a stripped-down loop—one accepted version, one edit at a time, accept on tie—is already sufficient for a current general-purpose language model to make measurable and in several cases best-in-class improvements to a mature, hand-tuned combinatorial solver that was written independently of the method, at a cost of about nine dollars per 100-iteration run and, on one machine, a compute budget bounded by fifteen hours per run (Section 7.4).

10 Future Work

Four directions follow most directly from this study.

The first is breadth of problems. The public Hash Code archive [13] contains twenty-one problems and forms a ready-made benchmark: qualification-round problems such as Book Scanning, Traffic Signaling, and Photo Slideshow come with multiple open-source solvers and would test the method across different problem structures from comparably strong baselines, while the final-round problems, having far fewer published solvers, would test whether it can improve less mature starting points. Beyond Hash Code, applying MicroEvolve to a solver refined over decades, such as Lin–Kernighan for the Travelling Salesman Problem, would probe the limits of what code-level edits can still find.

The second concerns the model. Every result here comes from a single model, so the contribution of the loop cannot be separated from that of Claude Opus 4.8. Substituting cheaper variants of the same family, models from other providers, and open-weight models into the identical loop would establish whether the effectiveness is a property of the method or of one particular model.

The third is a set of controls and ablations that the present design leaves open. Chief among them is a no-LLM control: because several of the largest single edits were parameter and seed changes, a matched-compute sweep over the same parameters would quantify how much of the gain requires the model at all. Two further ablations concern the loop itself—replacing the accept-on-tie rule with strict improvement, and discarding the running conversation so that each edit is proposed in a fresh session—and would settle the two design choices that Sections 8.4 and 8.5 can only report evidence consistent with.

The fourth is comparison against established program-evolution systems. AlphaEvolve [7] has not been publicly released, but OpenEvolve [17] and ShinkaEvolve [8] could be run on the same baselines and instances. Doing so requires wrapping our C++ solvers and JavaScript validator behind the Python harness these systems expect, an integration cost rather than a limitation of the tools, and would place MicroEvolve within the broader landscape of LLM-driven code evolution.

Appendix A Prompts and Feedback Messages

This appendix reproduces verbatim the complete system prompt and every feedback message the harness can emit. Both are defined in `evolve.py`: the system prompt as the `RULES` template, rendered once per run by `build_system`, and the feedback strings inline in the iteration loop. The conversation is a single continuous exchange: the system prompt is fixed for the whole run, each model reply is appended as an assistant turn, and exactly one of the feedback messages of Appendix A.3 is appended as the following user turn.

A.1 System prompt

The template below is rendered with four substitutions:

- `{instance}` — the instance name passed to `--instance` (e.g. `c_collaboration`);

- {problem} — the full natural-language problem statement (the file problem_definition.md in the repository, reproduced in Appendix A.4);
- {baseline} — the score of the unmodified seed solver on that instance, thousands-separated;
- {source} — the complete C++ source of the seed solver.

Critically, {source} is *always* version 0 of the solver, never the current accepted version. Accepted edits are carried in the conversation turns instead. This keeps the cached prompt prefix byte-identical across every iteration of a run, and identical between a fresh run and a resumed one.

```
You are an expert competitive programmer specializing in combinatorial optimization heuristics.

You are given the problem statement for the Mentorship and Teamwork problem from Google Hash Code
and the complete source of a state-of-the-art C++ solver. The solver is evaluated exclusively on the
 single input instance `{instance}`. Your goal is to maximize the solver's score on this one
instance; performance on other instances is irrelevant.

Rules, which apply for this entire conversation:

1. Propose exactly ONE change per response: a single, small, incremental modification to the solver.

2. Return the change as one or more search/replace blocks, and nothing else - no explanation, no
rationale, no full source file.
Each block must have exactly this form:

<<<<<<< SEARCH
(lines copied VERBATIM from the current version of the solver, including all whitespace and
indentation)
=======
(the replacement lines)
>>>>>>> REPLACE

The SEARCH text must appear exactly once in the current solver; include enough surrounding lines to
make it unique. The blocks are applied programmatically by exact string replacement, so any
deviation (paraphrased code, altered whitespace, line numbers, markdown fences inside the block)
will cause the change to be rejected.

3. The modified solver must remain fully deterministic: repeated runs must produce the identical
score. Do not introduce randomness; any pseudo-random generator must use a hard-coded fixed seed.

4. The solver must produce its final output within a fixed budget of 10 minutes; changes that exceed
 this budget will be rejected.

5. Do not re-propose a change that has already been attempted in this conversation, nor a trivial
variant of one.

6. After each change I will apply it, compile and run the solver on `{instance}`, and report the
resulting score and my accept/reject decision. A change is accepted only if its score is >= the best
 score so far; otherwise it is reverted. Your next diff must always apply to the current accepted
version of the solver, which I will keep you informed of.

Problem statement:
{problem}

Current solver (version 0, score on `{instance}`: {baseline}):
```cpp
{source}
```
```

A.2 Initial user turn

Iteration 1 is seeded with a single user message; every later user turn is one of the feedback messages below.

```
Propose your first change now: the single modification you expect to have the highest positive
impact on the score.
```

A.3 Feedback messages

Every outcome—including every failure mode—is fed back as a user turn and the loop continues; no outcome aborts the run. The six messages below are mutually exclusive and exhaustive. In each, {`score`} and {`best`} are thousands-separated integers, and the interpolated diagnostics are truncated to the character limits noted.

(1) Unparseable reply.

Emitted when the reply contains no well-formed search/replace block after stripping Markdown fences. Logged as `parse_error`.

```
Your response contained no valid search/replace block. Reply with one or more blocks in the exact
required format, applied to the current accepted version of the solver.
```

(2) Diff does not apply.

Emitted when some block's SEARCH text matches the accepted source zero times, or more than once. {`err`} names the offending block index, its match count, and its first SEARCH line. Logged as `apply_error`.

```
Your change could not be applied: {err}. Remember the next diff must apply verbatim to the current
accepted version. The accepted version is unchanged; propose a different single change.
```

where {`err`} has the form:

```
block {i}: SEARCH text matched {count} times in the current accepted solver (must match exactly once
). First SEARCH line: {first_line!r}
```

(3) Compile error.

The patched source is written to disk, fails `g++ -O2`, and the working copy is reverted. {`stderr`} is the compiler's stderr, stripped and truncated to 1500 characters. Logged as `compile_error`.

```
Your change was applied but the solver failed to compile. Compiler errors:
{stderr}

This change is REJECTED; the accepted version is unchanged. Propose a different single change that
compiles.
```

(4) Runtime or validation failure.

The binary compiled but crashed, exceeded the 600 s wall-clock limit, produced an invalid submission, or the validator emitted no parseable `Score:` line. {`detail`} carries the diagnostic. Logged as `run_error`.

```
Your change compiled but failed at runtime/validation: {detail}. REJECTED; the accepted version is
unchanged. Propose a different single change.
```

(5) Accept.

Emitted when `score >= best_score`; note that ties are accepted, which lets the solver drift laterally across plateaus. {`prev`} is the best score before this iteration. Logged as `accept`.

```
Applied. New score: {score} (previous best: {prev}). ACCEPTED - this is now the current accepted
version. Propose your next single change.
```

(6) Reject on score.

Emitted when the candidate is strictly worse than the incumbent; the working copy on disk is rewritten back to the accepted source. Logged as `reject`.

```
Applied. New score: {score} (previous best: {best}). REJECTED - strictly worse than the best, so
reverted. The accepted version remains the previous best. Propose a different single change.
```

A.4 Problem statement supplied to the model

The {`problem`} field of the system prompt is filled with the file `problem_definition.md`, a self-contained restatement of the Google Hash Code 2022 qualification-round problem covering the contributor and project definitions, the mentorship rule, scheduling and validity conditions, skill levelling, scoring, the input and output formats, and a worked three-contributor example. It is reproduced in the repository rather than here for space; see https://github.com/uran-lajci/MicroEvolve/blob/main/problem_definition.md.

Acknowledgements. The authors thank Yoann Coudert-Osmont (nanored) for licensing the baseline solver used in this work and making it open source. The authors also thank Petar Veličković for kindly providing the results of their FunSearch-based approach for the Mentorship and Teamwork problem. The authors further thank Nysret Musliu for discussing the initial general idea.

Declarations

Funding. This work received no external funding, grants, or financial support.

Competing interests. The authors declare that they have no competing interests.

Ethics approval and consent to participate. Not applicable. This study did not involve human participants, animal subjects, or data requiring ethics board approval.

Consent for publication. Not applicable. This paper does not contain any identifiable personal data, images, or individual case details.

Data availability. All data underlying the results reported in this paper, including problem instances, run results, and evaluation logs, are publicly available at https://github.com/uran-lajci/MicroEvolve.

Materials availability. This work makes use of two third-party materials: the baseline solver originally developed by Coudert-Osmont [4] and licensed for this work, and the solution validator developed by Holecz [16].

Code availability. The code for MicroEvolve, including all evolved solvers, prompts, and run traces underlying the results reported in this paper, is publicly available at https://github.com/uran-lajci/MicroEvolve.

Author contribution. Uran Lajçi conceived the study, developed the MicroEvolve methodology, implemented the pipeline, conducted the experiments, and wrote the original draft. Avni Rexhepi supervised and coordinated the research, advised on the methodology and experimental design, and reviewed and revised the manuscript. Kadri Sylejmani, Artan Mazrekaj, Jetë Lajçi, and Elvir Misini advised on the methodology and experimental design and contributed to reviewing and editing the manuscript. All authors read and approved the final manuscript.

References

[1] OpenAI: Introducing ChatGPT. https://openai.com/index/chatgpt/. Accessed 25 June 2026 (2022)

[2] Liu, F., Tong, X., Yuan, M., Zhang, Q.: Algorithm evolution using large language model. arXiv preprint arXiv:2311.15249 (2023) https://doi.org/10.48550/arXiv.2311.15249

[3] Romera-Paredes, B., Barekatain, M., Novikov, A., Balog, M., Kumar, M.P., Dupont, E., Ruiz, F.J.R., Ellenberg, J.S., Wang, P., Fawzi, O., Kohli, P., Fawzi, A.: Mathematical discoveries from program search with large language models. Nature **625**(7995), 468–475 (2024) https://doi.org/10.1038/s41586-023-06924-6

[4] Coudert-Osmont, Y.: HashCode2022. Source code, https://github.com/Nanored4498/HashCode2022. Accessed 25 June 2026 (2022)

[5] Veličković, P., Vitvitskyi, A., Markeeva, L., Ibarz, B., Buesing, L., Balog, M., Novikov, A.: Amplifying human performance in combinatorial competitive programming. arXiv preprint arXiv:2411.19744 (2024) https://doi.org/10.48550/arXiv.2411.19744

[6] Da Ros, F., Soprano, M., Di Gaspero, L., Roitero, K.: Large language models for combinatorial optimization: A systematic review. ACM Computing Surveys **58**(11), 1–53 (2026) https://doi.org/10.1145/3801961

[7] Novikov, A., Vũ, N., Eisenberger, M., Dupont, E., Huang, P.-S., Wagner, A.Z., Shirobokov, S., Kozlovskii, B., Ruiz, F.J.R., Mehrabian, A., Kumar, M.P., See, A., Chaudhuri, S., Holland, G., Davies, A., Nowozin, S., Kohli, P., Balog, M.: AlphaEvolve: A coding agent for scientific and algorithmic discovery. arXiv preprint arXiv:2506.13131 (2025) https://doi.org/10.48550/arXiv.2506.13131

[8] Lange, R.T., Imajuku, Y., Cetin, E.: ShinkaEvolve: Towards open-ended and sample-efficient program evolution. arXiv preprint arXiv:2509.19349 (2025) https://doi.org/10.48550/arXiv.2509.19349

[9] Sun, Y., Zhang, X., Huang, S., Cai, S., Zhang, B.-Z., Wei, K.: AutoSAT: Automatically optimize SAT solvers via large language models. arXiv preprint arXiv:2402.10705 (2024) https://doi.org/10.48550/arXiv.2402.10705

[10] Sun, Y., Ye, F., Chen, Z., Cai, S., Wei, K.: Automatically discovering heuristics in a complex SAT solver with large language models. arXiv preprint arXiv:2507.22876 (2025) https://doi.org/10.48550/arXiv.2507.22876

[11] Yu, M., Yang, R., Fan, C.: Heuristic search as language-guided program optimization. arXiv preprint arXiv:2602.16038 (2026) https://doi.org/10.48550/arXiv.2602.16038

[12] Boualamia, H., Metrane, A., Hafidi, I., Idrissi, A.: A new adaptation mechanism of the ALNS algorithm using reinforcement learning. Operations Research Forum **6**(3) (2025) https://doi.org/10.1007/s43069-025-00513-1

[13] Avner, P.: Google Hash Code Archive. GitHub repository, https://github.com/pierreavn/google-hashcode-archive. Accessed July 2026 (2024)

[14] Lajçi, U., Misini, E., Sylejmani, K., Rexhepi, A.: Iterated Local Search with Random Restarts for the Mentorship and Teamwork Problem (Google Hash Code 2022). Presented at AlbShkenca, Prishtinë, Kosovë. Conference presentation (2023)

[15] Lajçi, U.: Mentorship and Teamwork — Google Hash Code 2022. Source code, https://github.com/uran-lajci/mentorship-teamwork. Fork of https://github.com/Nanored4498/HashCode2022; accessed 25 June 2026 (2026)

[16] Holecz, K.: Hash Code 2022 Mentorship and Teamwork Insights Reporter. Source code, https://github.com/KristofHolecz/hash-code-2022-mentorship-and-teamwork-insights-reporter (2022)

[17] Sharma, A.: OpenEvolve: An Open-Source Evolutionary Coding Agent. Source code, https://github.com/algorithmicsuperintelligence/openevolve (2025)