HELIX: Placement Heuristics Evolution via LLM-Instructed Code Exploration

Andrew B. Kahng
University of California San Diego
La Jolla, CA, USA
abk@ucsd.edu

Sayak Kundu
University of California San Diego
La Jolla, CA, USA
sakundu@ucsd.edu

Bodhisatta Pramanik
University of California San Diego
La Jolla, CA, USA
bopramanik@ucsd.edu

Abstract

Improving placement algorithms demands months of expert effort navigating industrial C++ codebases and NP-hard optimization landscapes. We present Helix, the first framework for LLM-driven *algorithmic evolution* of placement algorithms within a production C++ (RTL-to-GDS) flow. Helix evolves both global and detailed placement in OpenROAD by combining a structured domain corpus with a (μ+λ) evolutionary strategy in which LLM coding agents edit and compile C++ in isolated worktrees. Three mechanisms drive performance: history-aware *focus targeting*, *cross-generation learning* via successful diff injection, and a *refinement retry loop* with placement diagnostics. The evolved placers discover nontrivial algorithmic changes—gradient blending, density-priority segment reordering, adaptive momentum dampening, among others—that reduce half-perimeter wirelength (HPWL) by up to 9% (4.5% geometric mean) and routed wirelength (RWL) by up to 23% (5.6% geometric mean) over default OpenROAD across twelve design–node pairs on NanGate45 and ASAP7, generalizing to unseen designs. Against a comparable-budget LLM single-shot baseline, Helix achieves 7× larger *fitness* gains. Integration with DREAMPlace demonstrates generality of Helix. Helix is open-sourced at [51].

CCS Concepts

• **Hardware** → **Physical design (EDA)**; • **Computing methodologies** → *Machine learning*.

Keywords

physical design, placement, large language models, evolutionary algorithms, code evolution, OpenROAD

ACM Reference Format:
Andrew B. Kahng, Sayak Kundu, and Bodhisatta Pramanik. 2026. HELIX: Placement Heuristics Evolution via LLM-Instructed Code Exploration. In *IEEE/ACM International Conference on Computer-Aided Design (ICCAD '26), November 08–12, 2026, San Jose, CA, USA*. ACM, New York, NY, USA, 9 pages. https://doi.org/10.1145/3831252.3834232

1 Introduction

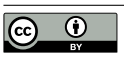
Despite decades of research, the core algorithms inside production placers still improve one hypothesis at a time—each requiring an engineer to navigate many lines of C++, survive a full build system, and evaluate quality across designs and technology nodes before learning whether an idea has merit. Many hypotheses yield no improvement at all, and the algorithmic structure evolves far more slowly than the design and placement needs. Large language models (LLMs) offer a new path: automating this algorithm-development loop itself [21]. Recent paradigms—FunSearch [26], AlphaEvolve [24], AlphaDev [23], and related efforts [18, 20, 29, 31, 33]—combine LLMs with evolutionary search, program execution, and fitness feedback to discover algorithms for optimization problems. Yet these paradigms share a common limitation: the search operates on small, self-contained programs where interfaces are minimal, compilation is trivial, and evaluation is inexpensive. Whether LLM-driven evolution can scale to a large, compiled, industrial codebase remains an open challenge. Classical algorithm configuration [14] and hyper-heuristic methods [8, 15, 17, 25] have operated in this space but without modifying algorithmic semantics.

EDA is a demanding test of this challenge [12]. Recent work has begun applying LLMs to EDA tasks [1, 3, 32]. A very recent effort, EvoPlace [32], evolves Python optimizer functions within DREAMPlace [19], an academic GPU-accelerated global placer, and reports HPWL reductions on standard benchmarks. However, evolving interpreted Python functions in a self-contained research tool poses qualitatively different challenges than evolving compiled C++ algorithms inside an RTL-to-GDS flow: the latter must survive a full industrial build system, comprehend hundreds of thousands (or millions) of lines of code, respect complex cross-module interfaces, and be evaluated through multi-stage physical-design metrics coupled to downstream routing, timing, and power outcomes. More broadly, existing LLM-for-EDA studies either operate on simplified surrogates or restrict search to parameter tuning. A central challenge has remained unaddressed: Can an LLM-driven system propose, compile, evaluate, and iteratively refine **nontrivial algorithmic changes** inside a production EDA codebase—and produce improvements that generalize across designs?

This paper presents Helix, an autonomous framework that meets the challenge above. Helix operates directly on the C++ source of both the global and detailed placement modules in OpenROAD [46], and discovers interpretable algorithmic improvements that a human engineer can read, understand, and adopt independently. Helix evolves the global and detailed placement modules only. Other modules such as CTS, routing, or sizing are not considered in this work although the framework can be extended to them. Helix is enabled, in part, by recent advances in LLM capabilities [38, 44]: long-context windows, tool-using coding agents [30] that edit and compile C++ in isolated environments, and prompt-caching economics [16] that make evaluating multiple *candidates*[1] practical. The resulting code changes are compact, interpretable, and directly adoptable by a human engineer. Our contributions are as follows.

- **Evolutionary code search in a production EDA codebase.** We cast placement algorithm improvement as a (μ+λ) evolutionary search over C++ code edits to OpenROAD's global (GPL) and detailed (DPL) placement modules [47, 48]. Three mechanisms make this tractable: isolated git-worktree sandboxing for each candidate, source-overlay incremental builds, and an automated *Build Repair Agent* that achieves 100% compilation success across all experiments (Section 3).

[1] A *candidate* is a modified version of a baseline implementation (see Section 2).



ICCAD '26, San Jose, CA, USA

ACM ISBN 979-8-4007-2873-0/2026/11
https://doi.org/10.1145/3831252.3834232

- **Domain-grounded search with literature-driven mutation.** We construct an *OpenROAD Corpus* (ORC).[2] This comprises: (i) literature technique cards distilled from placement research; (ii) pseudocode abstractions for every performance-critical function in OpenROAD's GPL and DPL modules; (iii) a curated parameter list with empirically validated *mutation* ranges; and (iv) per-algorithm CPU profiles. ORC grounds every LLM mutation in structured domain knowledge rather than parametric memory. Mutations are decomposed into five semantically distinct operators spanning a "risk gradient" from safe parameter perturbation to unconstrained algorithmic invention, scheduled adaptively as the search matures (Sections 3.1–3.2).
- **Placement-aware feedback and cross-generation learning.** We introduce a feedback loop that parses each candidate's Design Exchange Format (DEF) output, computes cell *displacement* (relative cell movement) statistics against the baseline, and injects placement heatmaps and outlier analysis directly into the LLM's refinement prompts, enabling the model to reason about *where* on the die its code changes had physical impact. *Cross-generation learning* complements this: successful code diffs are injected into future prompts, a *Reflection Agent* extracts actionable insights from each *generation*'s outcomes, and a *tabu* mechanism suppresses unproductive code regions. Together, these mechanisms recover 17 of 42 *near-miss offspring*[3] that would otherwise have been discarded (Section 3.3).
- **Domain grounding and physical feedback are jointly necessary.** Ablation reveals that neither structured domain knowledge nor closed-loop placement feedback suffices on its own: removing ORC eliminates 57% of the total fitness improvement (the LLM defaults to surface-level parameter scaling); removing cross-generation learning loses 36%, as offspring redundantly reattempt failed edits; and removing placement feedback degrades refinement recovery from 40% to near zero. Domain grounding and physical feedback are thus complementary, and not substitutable (Section 4.3).
- **Results on an RTL-to-GDS flow.** Across twelve design-node pairs spanning NanGate45 (NG45) and ASAP7 enablements (including three unseen designs), the evolved placers reduce post-detailed placement HPWL by up to 9% (4.5% geometric mean) and routed wirelength by up to 23% (5.6% geometric mean) over default OpenROAD, while mostly preserving timing and power. Against a comparable-budget LLM single-shot baseline, HELIX achieves 7× larger fitness gains, demonstrating that evolutionary feedback, not just LLM capability, drives the improvement. Integration with DREAMPlace shows the framework generalizes across codebases and languages (Section 4.2). To our knowledge, this is the first LLM-driven evolutionary framework to optimize algorithms within an RTL-to-GDS flow (Section 4).

In the following, Section 2 formulates the problem, Section 3 details HELIX, Section 4 presents results, and Section 5 concludes.

2 Problem Formulation

We formalize placement algorithm improvement as a search over code modifications to an existing C++ implementation (notation in Table 1). Unlike conventional placement optimization, which searches over continuous or discrete parameter spaces, our formulation defines the search space Θ as the set of all possible compilable C++ implementations reachable from a baseline through source-level edits—a discrete, high-dimensional space where most points do not compile, let alone improve placement quality. A *candidate* $\theta \in \Theta$ is a modified version of a baseline implementation θ_{base} (the unmodified OpenROAD placer), obtained by editing C++ source: tuning numerical parameters, restructuring algorithmic logic, or introducing new optimization passes. The search maintains a *population*[4] of μ candidates and generates λ offspring per generation through mutation and *crossover* operators (see Section 3 for details).

Table 1: Terminology and notation. Our empirical parameter settings are available in [51].

Symbol / term	Description
Γ	Input design suite
$\mathcal{N}$	Design in the design suite Γ
θ	A candidate in an evolution
θ_{base}	Fixed baseline implementation (default OpenROAD global/detailed placer)
$\text{HPWL}_{\mathcal{N}}(\theta)$	HPWL of candidate θ on design $\mathcal{N}$
$t_{\mathcal{N}}(\theta)$	Runtime of the evaluated candidate for θ on design $\mathcal{N}$
w_1, w_2	Positive weights in Equation 1 (default w_1, w_2 = 1.0, 0.5)
γ	Allowed runtime overhead factor relative to baseline (default = 1.5)
γ_g	Generation-dependent runtime tolerance
$\zeta(\theta)$	Penalty for segfaults, legalization failures, etc. (default = 100)
$f_{\mathcal{N}}(\theta)$	Scalar fitness score of design $\mathcal{N}$
Θ	Search space of candidate implementations

Given a benchmark suite $\Gamma = \{\mathcal{N}_1, \ldots, \mathcal{N}_m\}$ of m designs, let $\text{HPWL}_{\mathcal{N}}(\theta)$ and $t_{\mathcal{N}}(\theta)$ denote the half-perimeter wirelength and runtime of candidate θ on design $\mathcal{N} \in \Gamma$. We define per-design fitness as

$$f_{\mathcal{N}}(\theta) \;=\; w_1 \cdot \frac{\text{HPWL}_{\mathcal{N}}(\theta_{\text{base}})}{\text{HPWL}_{\mathcal{N}}(\theta)} \;-\; w_2 \cdot \max\!\left(0,\; \frac{t_{\mathcal{N}}(\theta)}{t_{\mathcal{N}}(\theta_{\text{base}})} - \gamma_g\right) - \zeta(\theta), \tag{1}$$

where w_1 weights HPWL improvement (a 2% reduction yields $f_{\mathcal{N}} \approx 1.02$), w_2 penalizes runtime degradation beyond a generation-dependent *tolerance* γ_g, and $\zeta(\theta) = \zeta$ if θ fails to compile, segfaults, or produces an illegal placement (0 otherwise). A candidate that matches exactly the baseline performance scores $f_{\mathcal{N}} = w_1$. Note that the additive penalty ζ intentionally avoids a multiplicative "set-fitness-to-zero" rule. Valid candidate scores are not limited to $[0, 1]$: runtime-heavy but valid candidates can approach or fall below zero, so zero is not a natural rejection threshold. Moreover, setting the score to zero only for a failed design could still yield a positive cross-design average, whereas the additive penalty ensures that failure on any design removes the candidate from selection. Aggregate fitness is the arithmetic mean over the evaluation suite:

$$\bar{f}(\theta) \;=\; \frac{1}{|\Gamma|} \sum_{\mathcal{N} \in \Gamma} f_{\mathcal{N}}(\theta). \tag{2}$$

Averaging across designs of varying size and complexity ($|\Gamma| = 3$ in our experiments: aes, ibex, jpeg on ASAP7)[5] prevents overfitting to any single benchmark. A key design choice is that γ_g decays linearly across generations: $\gamma_g \;=\; \gamma_{\text{s}} - (\gamma_{\text{s}} - \gamma_{\text{e}}) \cdot \frac{g}{G_{\max}}$, where γ_{s} and γ_{e} are the initial and final tolerance bounds [51]. The decay schedule is critical: without it, early generations consistently converge on conservative parameter tweaks that yield small, safe improvements (see ablation in Section 4.3). The decaying tolerance allows the population to first discover algorithmically promising mutations—which often carry initial runtime overhead—and then refine them toward runtime neutrality under increasing selection pressure (Figure 3). The goal is to find $\theta^\star \in \arg\max_{\theta \in \Theta} \bar{f}(\theta)$ subject to θ compiling correctly and producing valid placements on all designs in Γ. While we optimize post-detailed placement HPWL in this work, Equation 1 extends naturally to additional PPA objectives by appending corresponding

[2] ORC extends the documentation methodology in [1] (see Section 3.1).

[3] An *offspring* is a new candidate implementation produced by applying a mutation or crossover operator to its parent. A near-miss offspring is an offspring that has better fitness than the baseline but falls short of its parent (see Section 3.2).

[4] A *population* is the set of candidates maintained at any given generation.

[5] The evolved placers are evaluated on six designs (aes, jpeg, ibex, swerv, ariane37, and cva6) × two nodes (NG45, ASAP7) = twelve design-node pairs in Section 4.

ratio terms—e.g., one-sided routed wirelength, WNS/TNS, or power terms that guide selection toward candidates *improve* wirelength while preserving downstream PPA.[6]

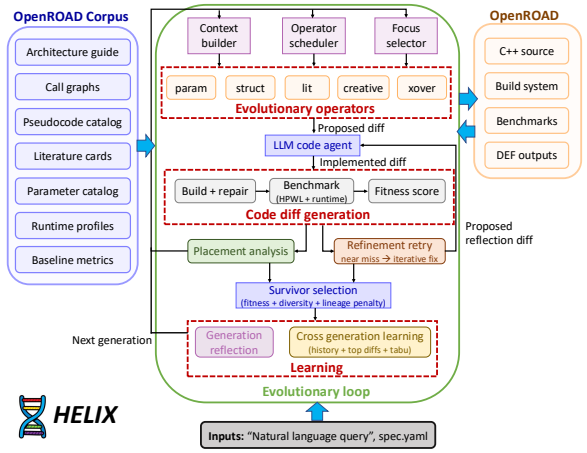


Figure 1: HELIX architecture. Inputs (user request R, codebase C) enter at the orchestrator, which drives a $(\mu+\lambda)$ evolutionary loop (center) drawing domain knowledge from the ORC (left) and dispatching LLM coding agents to edit C++ in isolated git worktrees (right). Each candidate is compiled, benchmarked, and scored, and the output is the best evolved implementation I_{best}.

3 Our Approach

Figure 1 illustrates the HELIX architecture, and Algorithms 1–2 provide formal pseudocode. The framework has three layers: (i) an *OpenROAD Corpus* (ORC) (Section 3.1) that provides structured domain knowledge—distilled literature, pseudocode abstractions, a parameter catalog, and profiling data—grounding every LLM mutation; (ii) a $(\mu+\lambda)$ *evolutionary loop* (Section 3.2) that schedules five mutation operators with adaptive probabilities and a focus selector steering each offspring toward an unexplored code region; and (iii) a *per-offspring pipeline* (Section 3.3) that handles LLM-driven code editing in isolated git worktrees, automated build repair, multi-design benchmarking, placement analysis, and iterative refinement of near-miss candidates.

We now trace the execution flow through Algorithm 1. Given a natural-language optimization request and the OpenROAD codebase, HELIX first assembles the ORC and runs baseline benchmarks (Lines 2–5). Here $\mathcal{B}$ (Line 4) records the per-design baseline HPWL and runtime of the unmodified codebase, providing the reference values for all fitness computations (Equation 1). Offspring evaluation—including build, benchmarking, and fitness scoring—is performed within EVOLVEINDIVIDUAL (Algorithm 2), which is invoked once for each offspring. The population is seeded with $\mu-1$ parameter-only mutations of the unmodified baseline, guaranteeing every seed compiles while spanning diverse search-space regions (Lines 7–13). The main loop (Lines 15–37) executes one generation per iteration—producing λ offspring, evaluating them, and selecting survivors. Each offspring is assigned a specific *focus target* (an algorithm or parameter cluster) to prevent diffuse, overlapping edits—a failure mode we observed consistently in early experiments without this mechanism (Lines 15–25). After benchmarking, survivors are selected via a composite score[7] that balances fitness (60%), focus-target diversity (25%), and a lineage crowding penalty that prevents population collapse to a single ancestral line (Line 27). The selected survivors become the *parents* for the next generation. If fitness stagnates for w generations, a fresh *individual*[8] is evolved from the *unmodified baseline* using a literature-guided operator on a target absent from the current population, ensuring genetic independence from the stagnated lineage (Lines 29–33). Each generation ends with an LLM-driven reflection [27] (*generation-level reflection*) pass that generates placement heatmaps, extracts operator weight adjustments, tabu regions, and the top-three successful diffs, feeding cross-generation learning into the next iteration (Lines 35–37). Details of each component are explained in the following subsections.

Algorithm 1: HELIX Evolutionary Framework.

Input: R: user request; C: OpenROAD codebase; μ: population size; λ: offspring per generation; $G_{\max}$: max generations; γ_s, γ_e: runtime tolerance bounds; Corpus (ORC)
Output: I_{best}: best evolved implementation

1 // Phase 1: ORC and analysis (offline preprocessing)
2 $\mathcal{T} \leftarrow$ identify target modules, files, and functions from R and C
3 $\mathcal{KB} \leftarrow$ assemble knowledge base: literature cards, call graphs, pseudocode, parameter catalog, profiling data, and architecture guide
4 $\mathcal{B} \leftarrow$ run baseline benchmarks on C; collect HPWL and runtime per design
5 $\mathcal{F} \leftarrow$ initialize *Focus Selector* from parameter catalog and profiling data
6 // Phase 2: Initialize population
7 $I_{\text{base}} \leftarrow$ package unmodified codebase as individual with $\bar{f} = w_1$
8 $\mathcal{P} \leftarrow \{I_{\text{base}}\}$
9 **for** $i \leftarrow 1$ **to** $\mu - 1$ **do**
10 $I \leftarrow$ EVOLVEINDIVIDUAL(I_{base}, mutate_param, 0, $\mathcal{H}_\emptyset$, $\mathcal{KB}$, $\mathcal{B}$, γ_s)
11 $\mathcal{P} \leftarrow \mathcal{P} \cup \{I\}$
12 $\mathcal{H} \leftarrow$ initialize evolution history from $\mathcal{P}$; $I_{\text{best}} \leftarrow \arg\max_{I \in \mathcal{P}} I$.fitness
13 $\bar{f}_{\text{best}} \leftarrow I_{\text{best}}$.fitness
14 // Phase 3: Evolutionary loop
15 **for** $g \leftarrow 1$ **to** $G_{\max}$ **do**
16 $\gamma_g \leftarrow \gamma_s - (\gamma_s - \gamma_e) \cdot g / G_{\max}$; // adaptive runtime tolerance
17 $\mathcal{P}_{\text{par}} \leftarrow k$-tournament selection of μ parents from $\mathcal{P}$
18 $\mathcal{O} \leftarrow \emptyset$
19 **for** $j \leftarrow 1$ **to** λ **do** // generate λ offspring
20 $p \leftarrow \mathcal{P}_{\text{par}}[j \bmod \mu]$
21 $op \leftarrow$ sample operator from adaptive distribution (Equation 3)
22 $\tau \leftarrow \mathcal{F}$.select($op$, $\mathcal{H}$) ; // focus target
23 $I_{\text{new}} \leftarrow$ EVOLVEINDIVIDUAL(p, op, g, $\mathcal{H}$, $\mathcal{KB}$, $\mathcal{B}$, γ_g, τ)
24 Record (op, τ, I_{new}.fitness) in $\mathcal{H}$; save code diff
25 $\mathcal{O} \leftarrow \mathcal{O} \cup \{I_{\text{new}}\}$
26 // Survivor selection: fitness + diversity − crowding
27 $\mathcal{P} \leftarrow$ select μ survivors from $\mathcal{P}_{\text{par}} \cup \mathcal{O}$ by composite score[4]
28 // Anti-stagnation injection
29 **if** $\bar{f}_{\text{best}}$ *unchanged for last* w *generations* **then**
30 $\tau' \leftarrow \mathcal{F}$.select_excluding(mutate_lit, $\mathcal{H}$, $\{\tau \mid I \in \mathcal{P}\}$)
31 $I_{\text{fresh}} \leftarrow$ EVOLVEINDIVIDUAL(I_{base}, mutate_lit, g, $\mathcal{H}$, $\mathcal{KB}$, $\mathcal{B}$, γ_g, τ')
32 **if** I_{fresh}.fitness $> w_1$ **then**
33 replace worst-fitness individual in $\mathcal{P}$ with I_{fresh}
34 // Generation-level reflection and cross-generation learning
35 $\mathbf{s} \leftarrow$ LLM analysis of ($\mathcal{P}_{\text{par}}$, $\mathcal{O}$, $\mathcal{P}$) with placement heatmaps (Section 3.3)
36 Update $\mathcal{H}$: operator weight adjustments, tabu regions, top-3 diffs
37 $I_{\text{best}} \leftarrow \arg\max_{I \in \{I_{\text{best}}\} \cup \mathcal{P}} I$.fitness; $\bar{f}_{\text{best}} \leftarrow I_{\text{best}}$.fitness
38 **return** I_{best}

3.1 OpenROAD Corpus (ORC)

The effectiveness of LLM-driven code mutations depends critically on the context provided to the model. Indeed, among all HELIX components, the ORC has the largest measured effect on the final improvement: removing it eliminates 57% of the total fitness gain (Section 4.3). HELIX embodies this principle through a multi-layered

[6] Sensitivity of HELIX to w_2 and γ_g is shown in [51].

[7] Composite survivor score: $S(I) = 0.6\,\hat{f}(I) + 0.25\,\delta(I) - 0.1\,\ell(I)$, where $\hat{f}$ is rank-normalized fitness, $\delta \in \{0, 0.5, 1\}$ is a focus-target novelty bonus, and ℓ counts same-lineage elites already selected [51].

[8] An *individual* is a complete, compilable C++ implementation of a target module (GPL/DPL); details in Section 3.2.

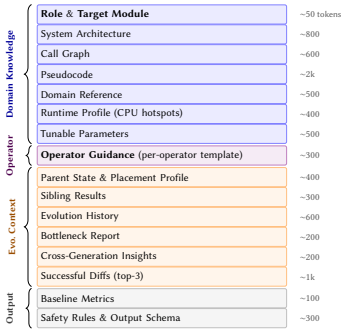


Figure 2: Anatomy of an LLM prompt assembled by the *Context Builder*. Static domain knowledge is reused across generations, while evolutionary context grows.

knowledge base, the ORC, that captures algorithmic intent, tunable parameters, performance characteristics, and the state of the art in placement research. The ORC is assembled once per target module and reused across all generations; its construction is fully automated except for the initial collection of research papers. A deterministic Context Builder assembles each LLM prompt from the ORC: given the current operator type and focus target, it selects the relevant pseudocode files (via the call graph), retrieves matching literature cards (via textual similarity [34]), and concatenates them with evolutionary context—parent state, sibling results, successful diffs—into the 16-section prompt structure shown in Figure 2, totaling 8–12k tokens per mutation call.

Literature technique cards. We collect 97 research papers (list given in [51]) spanning global placement (e.g., [7, 9, 22]), detailed placement (e.g., [4–6, 11]), and related topics (legalization, timing-driven placement, routability optimization). A *Distillation Agent* (a single LLM call per paper, run once offline) processes each paper through a multi-stage pipeline—extraction, chunking, analysis, and merge—producing a structured JSON [41, 42] knowledge card that records the algorithmic technique, pseudocode sketch, quantitative claims with experimental conditions, and identified risks for code evolution. Cards are indexed by placement stage, optimization objective, and technique family, enabling targeted retrieval: the Context Builder ranks cards by textual similarity [34] to the focus-target description and returns the top-k matches.[9] This grounds the `mutate_lit` operator (Section 3.2) in specific published techniques rather than the LLM's parametric memory.

Pseudocode abstractions and call graphs. A *Documentation Agent* (one-time LLM pass per function, run offline) generates algorithm-level pseudocode for every performance-critical function by analyzing the C++ source, stripping implementation details (memory management, logging, GUI hooks), and preserving algorithmic structure, data flow, and loop invariants. This captures algorithmic intent at a level the LLM can reason about, while fitting within token budgets that raw C++ (~20,000 lines per module) would vastly exceed. The ORC contains pseudocode files across both GPL and DPL modules, each including a function signature, purpose statement, step-by-step logic, parameter descriptions, and complexity annotations. Static call graphs extracted via Clang-based analysis complement the pseudocode: the Context Builder uses them to select which pseudocode files to inject based on the focus target's position in the call hierarchy, and the LLM uses them to identify functions on the "critical path" to the hottest inner loops.

Parameter catalog, profiling data, and architecture guide. For each module, a list of tunable constants—struct defaults, `constexpr` values, "magic" numbers, and loop bounds—is extracted from C++ source. Each entry records the parameter name, source location, default value, type, and a suggested mutation range derived from the literature and empirical testing [51]. The `mutate_param` operator draws directly from this catalog, sampling parameters and perturbation magnitudes within the annotated ranges. Per-design CPU profiling via `perf` identifies computational *hotspots*[10] biasing mutations toward performance-critical regions. Finally, a narrative architecture guide per module describes subsystem interactions, key data structures, and which invariants must be preserved. This gives the LLM sufficient structural context to propose valid edits without reading the full source.

[9] HELIX uses $k = 5$ by default. Using $k > 5$ did not improve our results.

Table 2: Evolutionary operators ordered by increasing risk and potential impact. Line budgets guide the LLM but are not enforced; fitness is the sole selection criterion.

Operator	Action	Budget	Focus target
`mutate_param`	Perturb 1–3 constants in annotated ranges	25 lines	Param. cluster
`mutate_struct`	Restructure control flow or data access	100 lines	Algorithm
`mutate_lit`	Implement technique from literature card	200 lines	Algorithm
`mutate_creative`	Novel heuristic guided by profiling data	200 lines	Algorithm
`crossover`	Merge complementary diffs from two parents	500 lines	None

3.2 Evolutionary Loop and Operator Scheduling

HELIX drives optimization through a $(\mu+\lambda)$ evolutionary strategy [2, 13] in which μ parent individuals and λ offspring compete for survival each generation. Here each individual is represented by an isolated git commit, its compound fitness score, per-design benchmark metrics, and the code diff relative to baseline.

Mutation operators. Table 2 summarizes the five operators, ordered by increasing *risk* (compilation failures, seg-faults, degradation, etc.) and potential impact. (i) `mutate_param` perturbs tunable constants within pre-defined ranges—edits are small and compilation almost always succeeds. (ii) `mutate_creative` allows the LLM to propose novel algorithmic modifications unconstrained by the literature, guided by architecture documentation, profiling hotspots, placement diagnostics, and evolution history. (iii) `mutate_struct` restructures control flow without introducing new ideas (e.g., reordering loop phases, adding early exits). (iv) `mutate_lit` implements a specific technique from a retrieved literature card, with the paper's pseudocode and quantitative claims injected directly into the prompt.[11] (v) `crossover` receives both parents' diffs and produces a unified implementation that integrates their complementary strengths (see prompts and implementation in [51]).

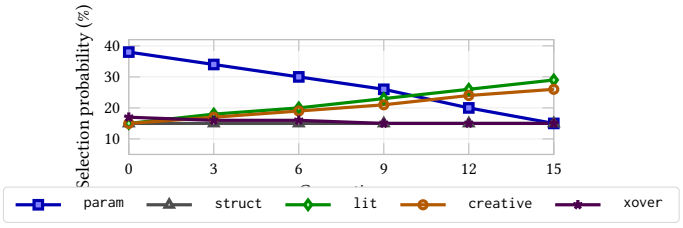


Figure 3: Operator selection probabilities across generations (Equation 3, progress factor α only, β=$\hat{s}$=1). Safe parameter tuning dominates early and decays; literature-guided and creative mutations grow as diminishing returns demand deeper algorithmic changes.

[10] Hotspots are functions consuming the largest fraction of total CPU cycles during placement execution. E.g., for GPL, wirelength and FFT-based density gradient computation together account for ~55% of placement runtime.

[11] `mutate_lit` provides the literature technique as guidance, not as specification. The LLM adapts the idea to fit the existing codebase's data structures and interfaces, and may implement a subset or variant of the published technique.

Adaptive operator scheduling. The probability of selecting operator op at generation g depends on three multiplicative factors:

$$P(op \mid g) \propto \underbrace{\alpha_{op}(g)}_{\text{progress}} \cdot \underbrace{\beta_{op}(g)}_{\text{stagnation}} \cdot \underbrace{\hat{s}_{op}}_{\text{success rate}} \tag{3}$$

The progress factor α gradually shifts probability from low-risk operators (`mutate_param`) toward higher-risk ones (`mutate_lit`, `mutate_creative`) as the population matures and diminishing returns demand deeper algorithmic changes (Figure 3). The stagnation factor β boosts exploratory operators when fitness has not improved for w consecutive generations. The success rate $\hat{s}$ tracks each operator's empirical improvement rate via Laplace smoothing. A probability floor of 8% prevents any operator from being starved. Full scheduling details are in [51].

Focus targeting. Within each operator invocation, the *Focus Selector* assigns the offspring a **narrow** optimization target τ to prevent diffuse, overlapping edits. For `mutate_param`, targets are clusters of related parameters; for structural, literature, and creative operators, targets are specific algorithms or pipeline stages identified from profiling data. Target selection uses history-aware weighted sampling that prioritizes untried targets, up-weights previously successful ones, and down-weights repeatedly failed targets without eliminating them—promoting systematic coverage while allowing the search to deepen on productive regions.

Adaptive scheduling and focus targeting together prevent stagnation on parameter tweaks and redundant structural edits (Section 4.3).

LLM Mutation Plan (`mutate_creative`, DPL, Gen 10, Candidate 5)

"*Ideal-order pruning + improvement-triggered cascade in the cell reorderer. (1) When the net-centroid ideal positions are already monotonically ordered, only the identity permutation (position adjustment) is evaluated—skipping the remaining* $(N!-1)$ *permutations that would place cells against their ideal ordering. (2) When a window reorder succeeds, the overlapping adjacent window is immediately re-evaluated, propagating local improvements to neighboring cells. Changed* `reorder()` *return type from* `void` *to* `bool` *to signal success upstream.*"

Resulting C++ diff: `detailed_reorder.cxx` (42 lines changed)

```
 // Change 1: skip N!-1 permutations when ideal
 // positions already match current cell ordering
+bool hasInversion = false;
+for (int i = 1; i < size; i++)
+  if (idealX[i-1] > idealX[i])
+    { hasInversion = true; break; }
   ...
-} while (next_permutation(order).found);
+} while (hasInversion
+       && next_permutation(order).found);
 // Change 2: cascade re-evaluation on success
-evalWindow(istrt, istop);
+if (evalWindow(istrt, istop) && i > jstrt) {
+  int cs = min(jstop, i + windowSize_ - 2);
+  evalWindow(max(jstrt, cs-windowSize_+1), cs);
+}
```

Figure 4: A complete mutation cycle: the LLM's plan (top, verbatim excerpt) and resulting C++ diff (bottom) for DPL.

3.3 Per-Offspring Lifecycle

Producing an offspring from an LLM-proposed edit is a multi-stage process (Algorithm 2). The raw code change must compile, survive benchmarking, undergo analysis, and—if it *narrowly* misses improving on its parent—receive targeted refinement. Figure 4 shows a complete mutation cycle from LLM plan to C++ diff.

Code editing and build repair. The Context Builder assembles a prompt from the ORC, evolution history, parent state, operator specification, and focus target (Line 2). An isolated git worktree is created from the parent's commit, ensuring edits are sandboxed

Algorithm 2: EvolveIndividual: Per-Offspring Lifecycle.

Input: p: parent; op: operator; g: generation; $\mathcal{H}$: history; $\mathcal{KB}$: knowledge base; $\mathcal{B}$: baselines; γ_g: runtime tolerance; τ: focus target; ψ: max repair trials; $R_{\max}$: max refinement rounds
Output: I: individual

1 // Step 1: LLM-driven code editing
2 $ctx \leftarrow$ assemble prompt from $\mathcal{KB}$, $\mathcal{H}$, p, op, τ
3 $wt \leftarrow$ create isolated git worktree from p's commit
4 Dispatch coding agent to edit C++ source in wt using ctx
5 // Step 2: Build with automated repair
6 $ok \leftarrow$ false
7 $k \leftarrow 0$
8 **while** ok = false ***and*** $k < \psi$ **do**
9 $ok \leftarrow$ source-overlay build of wt
10 **if** $\neg\, ok$ **then** dispatch *Build Repair Agent* with compiler errors;;
11 $k \leftarrow k + 1$;
12 **if** $\neg\, ok$ **then return** individual with $\bar{f} = -\zeta$;
13 // Step 3: Benchmark and fitness
14 $\mathbf{m} \leftarrow$ run benchmarks on wt: HPWL, runtime, DEF output per design
15 $\bar{f} \leftarrow$ compute aggregate fitness (Equation 2) with $\gamma = \gamma_g$
16 // Step 4: Placement analysis
17 $\Delta \leftarrow$ parse DEF outputs; compute region-level displacement statistics vs. baseline
18 Generate placement heatmaps for each design
19 // Step 5: Refinement retry (near-miss offspring)
20 **if** $w_1 < \bar{f} < p$.fitness **then**
21 **for** $r \leftarrow 1$ **to** $R_{\max}$ **do**
22 $ctx_r \leftarrow$ build refinement prompt with $\mathbf{m}$, Δ, fitness gap (p.fitness $- \bar{f}$)
23 Dispatch agent to refine code in wt using ctx_r
24 Rebuild codebase
25 $\mathbf{m}' \leftarrow$ rebenchmark
26 $\bar{f}' \leftarrow$ recompute fitness
27 **if** $\bar{f}' \geq p$.fitness **then**
28 $\bar{f} \leftarrow \bar{f}'$
29 **break**
30 **else if** $\bar{f}' > \bar{f}$ **then**
31 $\bar{f} \leftarrow \bar{f}'$
32 **else**
33 Revert to previous commit
34 **break**
35 // Step 6: Population reflection (runtime regression)
36 **if** *HPWL improved* ***but*** *runtime* $> 2\times$ *baseline* **then**
37 Dispatch *Reflection Agent* with diagnostics and Δ
38 Rebuild and rebenchmark;
39 Keep if fitness improves
40 $I \leftarrow$ package (wt, plan, $\bar{f}$, $\mathbf{m}$, Δ)
41 **return** I

(Line 3). A coding agent edits C++ source within the worktree (Line 4); for later generations, the top three successful diffs and sibling results from earlier offspring in the same generation are included in the prompt. The edited worktree is then compiled using a source-overlay incremental build. If compilation fails, a *Build Repair Agent* is dispatched with parsed compiler errors and attempts targeted fixes for up to ψ trials (Lines 6–11).

Placement analysis. After benchmarking (Lines 14–15), each design's DEF output is compared cell-by-cell against the baseline. The die is partitioned into a 4×4 grid, and per-region displacement statistics (fraction of cells moved and average displacement magnitude) are computed (Lines 17–18). Placement heatmaps are then generated and fed directly into three downstream mechanisms: refinement of near-miss offspring (described below), generation-level reflection (Section 3), and grounding of `mutate_creative` prompts in actual placement outcomes.

Refinement retry. If an offspring beats the baseline but falls short of its parent ($w_1 < \bar{f} < p$.fitness, Line 20), up to $R_{\max}$ refinement rounds are applied (Lines 21–34). Each round constructs a targeted prompt containing per-design metrics, placement heatmap analysis, and the fitness gap to the parent (Line 22). The LLM makes focused adjustments (e.g., tuning constants) and the code is rebuilt and rebenchmarked (Lines 23–34). This mechanism recovers near-miss

offspring that contain promising algorithmic ideas requiring calibration: in our DPL runs, it recovered 17 of 42 near-miss candidates that would otherwise have been discarded. A separate runtime degradation safeguard (Lines 36–39) catches mutations that improve HPWL but introduce overhead exceeding 2× baseline.[12] A *Reflection Agent* diagnoses the source code of mutations and applies targeted fixes, retaining the result only if overall fitness improves. Two design choices bound the per-candidate evaluation cost: (i) inner-loop fitness uses placement-stage HPWL and runtime rather than full post-route PPA metrics; and (ii) source-overlay incremental builds recompile only the files modified by each candidate atop a prebuilt OpenROAD tree.

Table 3: Design statistics.

Design	#StdCell	#Macro	ASAP7		NanGate45	
			TCP (ps)	Util. (%)	TCP (ps)	Util. (%)
aes	14k–16k	0	340	60	750	55
ibex	16k–20k	0	1040	55	2030	70
jpeg	56k–60k	0	550	70	1020	70
cva6	84k–111k	8	800	70	2200	50
swerv	90k–118k	28	1750	40	2100	60
ariane37	137k–192k	37	1050	45	2600	40

Table 4: Default OpenROAD baseline metrics.

Node	Design	HPWL (μm)	RWL (μm)	T_{pl} (s)	T_{rt} (s)	WNS (ps)	TNS (ns)	Pwr (mW)
NG45	aes	208682	258337	73	14	-64.03	-3.75	543
	ibex	197361	255223	48	15	-146.02	-24.54	90
	jpeg	471729	561271	108	48	-105.01	-36.62	557
	cva6	2040118	2409729	466	75	-114.52	-4.90	197
	swerv	3638329	4169496	788	84	-136.00	-27.53	244
	ariane37	3993809	4621172	1878	135	-183.01	-35.56	296
ASAP7	aes	49059	64157	30	14	-49.72	-4.74	204
	ibex	54830	80076	45	23	-37.98	-5.37	47
	jpeg	125527	182151	117	55	-125.94	-19.93	162
	cva6	530923	639179	303	95	-168.10	-204.41	133
	swerv	928310	1146330	297	161	-214.30	-83.40	98
	ariane37	1322894	1540174	1045	222	-133.25	-2.26	281

4 Experimental Evaluation

Helix [51] is implemented in Python. We use Cursor Agent [40] (with Claude Opus 4.6 [38]) for C++ code edits in isolated worktrees, and Claude CLI [37] (Opus 4.6) for analysis-only tasks. All experiments run on a server with eight 2.4 GHz Intel Xeon Gold 6148 processors and 384 GB RAM. Helix supports multiple coding-agent backends (Cursor Agent, Claude CLI, and Codex [39]): evolving OpenROAD's GPL and DPL with Claude Opus 4.6, Claude Opus 4.8, and Codex-GPT5.5 achieves fitness 1.017, 1.016, and 1.012, respectively. Our evolved code diffs are available in [51].

Experimental setup. We evaluate on six designs from the OpenROAD [45] and MacroPlacement [10] repositories—aes, jpeg, ibex, swerv, ariane37, and cva6—under both NanGate45 [43] and ASAP7 [36] enablements (Table 3). All HPWL and PPA metrics are generated with OpenROAD [46]; the baseline is the unmodified default implementation (Table 4). Evolution is performed on three designs—aes, jpeg, and ibex—on ASAP7, with μ=5 parents, λ=8 offspring per generation, and $G_{\max}$=15 generations. We run three independent evolutions per module; Table 5 reports results from the best-fitness run. Across runs, best DPL fitness ranges from 1.026 to 1.028 and best GPL fitness from 1.016 to 1.017. Across the three independent evolution runs, the geometric-mean HPWL (RWL) improvement is 4.5% (5.6%) with standard deviation 0.3 (0.5) percentage points;

[12]We add a stricter threshold than the generation-dependent penalty γ_g in Equation 1 to catch extreme outliers.

Evolved GPL diff (`nesterovBase.cpp`, `nesterovPlace.cpp`)

```
// LSE gradient blend (nesterovBase.cpp)
constexpr float kLseBlend = 0.12f;
+ if (gPin->hasMinExpSumX() && ...)
+   lseX += gPin->minExpSumX()/gn->waExpMinSumX();
+ tmpPair.x += kLseBlend * (lseX - tmpPair.x);
// Density-penalty calibration (nesterovPlace.cpp)
  npUpdateCurGradient(nb);
+ float wlSum  = nb->getWireLengthGradSum();
+ float denSum = nb->getDensityGradSum();
+ if (wlSum > 0 && denSum > 0)
+   nb->setDensityPenalty(
+       (wlSum/denSum) * npVars_.initDensityPenalty);
```

Evolved DPL diff (`detailed_reorder.cxx`, `detailed.cxx`)

```
// Density-priority segment ordering (detailed_reorder.cxx)
- for (int s = 0; s < numSeg; s++) {
+ std::ranges::sort(segOrder, [&](int a, int b){
+     return cellCount[a] > cellCount[b]; });
+ for (int si = 0; si < numSeg; si++) {
+   const int s = segOrder[si];
// Extra refinement passes (detailed.cxx)
+ for (int round = 0; round < 3; ++round) {
+   doDetailedCommand(gsArgs); // global swap x2
+   doDetailedCommand(roArgs); // reorder x2
+   if ((bestRefine-cur)/bestRefine < 0.0003) break;
+   bestRefine = cur;
+ }
```

Figure 5: Code diffs of the evolved global and detailed placers. Top: WA/LSE gradient blend and design-adaptive density-penalty calibration in GPL. Bottom: density-priority segment reordering and multi-pass refinement with early termination in DPL. Lines prefixed '+' are additions; '−' are deletions.

across the five P&R trials over all designs, the mean standard deviation is 0.3 percentage points for HPWL and 0.7 for RWL. The evolved placers are evaluated *without further tuning* on all six designs, including three unseen designs (cva6, swerv_wrapper, ariane37). To reduce noise in post-route metrics, we run five independent P&R trials per design, perturbing the target clock period by ±1 and ±2 ps; all tables report medians over these five runs.

Baselines. Beyond the default OpenROAD implementation, we compare against two baselines that consume comparable evaluation budgets: (i) an *LLM single-shot (LLM-SS)* baseline (Section 4.1.2); and (ii) a *Bayesian optimization (BO)* baseline using TPE [28, 53] over flow-level parameters,[13] applied to both the default and evolved codebases (Section 4.1.3). We additionally integrate Helix with DREAMPlace [19] and compare against published EvoPlace [32] configurations to evaluate cross-tool portability (Section 4.2).

Evolved placers. We evolve GPL and DPL in separate Helix runs, then combine the results into a single end-to-end flow evaluated without further tuning. Full diffs are available in [51] with implementation highlights below.

- The best GPL individual emerges at generation 12. Helix modifies OpenROAD's Nesterov engine [9] with four changes in ∼300 lines: (1) a 12% weighted-average/log-sum-exp (WA/LSE) gradient blend that reduces spurious forces on interior pins; (2) design-adaptive density-penalty initialization that calibrates the wirelength–density force balance from first-iteration gradient magnitudes; (3) literature-guided dynamic momentum dampening [35]; and (4) adaptive gradient restart when both HPWL and overflow regress simultaneously.

[13]We use Ray Tune [54] with TPE [53] to search a nine-dimensional flow parameter space (timing repair effort, cell padding in global and detailed placement, CTS cluster size/diameter, routing-driven and timing-driven modes, pin-layer and upper-layer routing adjustments) with 400 trials (20 parallel workers), minimizing RWL.

- The best DPL individual emerges at generation 13. HELIX implements two main changes in ~200 lines: density-priority segment reordering that processes denser segments first when swap partners are abundant, and doubled pass intensity with early termination (improvement < 0.03%). Crucially, all six changes are algorithmic modifications, not parameter tweaks. Notably, density-priority segment reordering was discovered independently in all three DPL runs.

Table 5: HELIX evolved placement (GPL+DPL): %Δ vs. default OpenROAD (negative = better). Bold = improvement. WNS, TNS and Power are absolute values (ps/ns/mW), not %Δ.

Node	Design	HPWL (%)	RWL (%)	T_{pl} (%)	T_{rt} (%)	WNS (ps)	TNS (ns)	Pwr (mW)
NG45	aes	**-8**	**-6**	+3	**-3**	-86.01	-4.37	**536**
	ibex	**-1**	**-2**	+14	+10	**-137.01**	-57.03	**89**
	jpeg	**-5**	**-5**	+2	+4	**-98.00**	**-35.13**	559
	cva6	**-9**	**-9**	+13	+11	**-80.80**	**-4.80**	**192**
	swerv	**-8**	**-8**	+4	+2	**-76.04**	**-15.75**	**243**
	ariane37	**-1**	**-1**	+3	+1	**-125.01**	**-7.81**	297
ASAP7	aes★	**-4**	**-2**	+15	+2	**-47.48**	**-3.82**	**202**
	ibex★	**-2**	**-1**	+12	+4	-80.42	-18.41	47
	jpeg★	**-6**	**-23**	+6	+3	**-49.34**	**-5.76**	**154**
	cva6	**-1**	**-1**	+21	+7	**-136.42**	**-148.71**	**131**
	swerv	**-2**	**-1**	**-25**	**-26**	**-20.03**	**-0.40**	**97**
	ariane37	**-6**	**-6**	+5	+2	**-14.83**	**-0.12**	**268**
	Geo-mean	**-4.5**	**-5.6**	+5.4	+0.9	—		–

★ Training design for HELIX.

4.1 QoR Comparisons

All OpenROAD comparisons below use the combined GPL+DPL evolved flow evaluated end-to-end across twelve design–node pairs.

4.1.1 Comparisons with Default OpenROAD. Table 5 reports the PPA impact of the combined evolved placers (GPL+DPL) on the full six-design suite on NG45 and ASAP7, as %Δ vs. the unmodified OpenROAD baseline (Table 4).

Wirelength. Across all twelve design–node pairs, HELIX reduces post-detailed placement HPWL by 1–9% (4.5% geometric mean) and routed wirelength by 1–23% (5.6% geometric mean).[14] The largest improvement occurs on jpeg-ASAP7 (23% RWL from 6% HPWL), indicating placements that are substantially more routing-friendly beyond pure wirelength minimization.

Generalization to unseen designs. The three unseen designs—all 4–12× larger than the training set and containing macros—exhibit improvements comparable to or exceeding the seen designs, e.g., cva6-NG45 achieves the largest HPWL reduction (9%/9% RWL). Overall, 10 of 12 design–node pairs show HPWL improvement ≥2%, confirming that the evolved changes generalize well rather than overfitting to the training set. Evolution optimizes the *average* fitness across the three training designs (Equation 2), discouraging single-design overfitting. Thus, a mutation that helps one design but hurts the others is unlikely to survive selection. No seen or unseen design regresses in HPWL or RWL.

Timing, runtime, and power. WNS and TNS are not explicitly optimized, yet the evolved placer often preserves or improves timing in the majority of cases. TNS regresses on ibex (both NG45 and ASAP7). This regression is due to timing-agnostic cell displacement: the evolved placement moves cells by ~5 μm on average, including cells in timing-critical cones. Although total critical-path wirelength is nearly unchanged (RWL improves 2% on NG45, 1% on ASAP7), redistributed net segments cause more nets to cross buffer-insertion thresholds; OpenROAD's resizer inserts ~50% more buffers on these paths (nine versus six), degrading setup TNS. Violating endpoints double, while median slack and hold timing remain unchanged. We observe no post-route DRC violations in any evaluated design. Placement runtime increases modestly (+5.4% geometric mean), primarily due to additional global-swap and reorder refinement passes introduced by the evolved DPL. Note that runtime is explicitly included in the fitness function (Equation 1). Routing runtime is largely unaffected (+0.9% geometric mean). Power is stable across all designs.

[14] Measured independently against default OpenROAD: GPL-only evolution achieves ~1.6% geo-mean HPWL improvement; DPL-only evolution achieves ~2.4%.

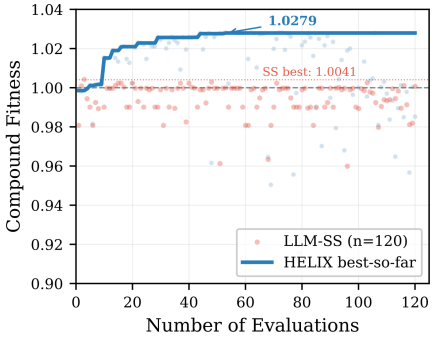


Figure 6: HELIX vs. LLM-SS: best-so-far fitness against total candidate evaluations for DPL. GPL exhibits similar trends. X-axis truncated at 120 to match LLM-SS budget.

4.1.2 Comparisons with LLM-SS Baseline. To isolate the value of the evolutionary loop, we evaluate an LLM single-shot (LLM-SS) baseline that uses the same LLM, ORC, build infrastructure, and *Build Repair Agent* as HELIX but strips all evolutionary components. Each of 120 attempts starts independently from the unmodified baseline, with no parent selection, crossover, reflection, or feedback from prior attempts. Figure 6 overlays the results on DPL evolution. LLM-SS candidates cluster around the baseline, with the best-of-120 reaching only fitness 1.004 (+0.4% HPWL on the training set). HELIX converges to 1.028 (+2.8%)—a 7× improvement. When evaluated end-to-end on the full twelve design–node pairs, the LLM-SS best achieves <1% geo-mean HPWL improvement versus HELIX's 4.5%. Two factors explain the gap: (i) without history, the LLM repeatedly proposes the same few modifications; and (ii) without cumulative edits, each attempt makes a single incremental change, whereas HELIX compounds gains across generations.

4.1.3 Evaluations with Bayesian Optimization. To test whether HELIX's gains persist after flow-level parameter tuning, we run TPE [53] over OpenROAD's user-exposed flow parameters with 400 trials on both the *default* and *evolved* codebases. BO+*evolved* outperforms BO+*default* on all designs in wirelength (1.3% to 5.9% HPWL, 3.4% geo-mean; 1.1% to 5.6% RWL, 3.3% geo-mean), with power remaining stable. HELIX's algorithmic changes thus provide value that flow-level parameter tuning cannot replicate [51]. To reiterate, our setup provides an equal-tuning-budget comparison. That the evolved placer remains better after tuning shows that HELIX's gains are beyond what can be achieved through flow-level parameter tuning and cannot be obtained by tuning the default codebase alone.

4.2 Generality of HELIX

To test whether HELIX's architecture is tool-agnostic, we integrate it with DREAMPlace [19, 49], a GPU-accelerated analytical placer whose Python/PyTorch codebase is architecturally distinct from

Table 6: DREAMPlace global placement HPWL ($\times 10^6$ μm) comparisons on ISPD 2005. Lower is better; best in bold.

Design	Base	EvoPlace		Helix	
		%Δ	Gen.	%Δ	Gen.
adaptec1★	72.68	**−6.3**	930	−0.8	15
adaptec2	82.19	−0.1	130	**−1.0**	15
adaptec3	194.30	**−1.9**	80	−1.1	15
adaptec4	174.10	−0.6	30	**−0.7**	15
bigblue1★	89.19	div.	430	**−0.3**	15
bigblue2	138.10	−0.5	200	**−0.6**	15
bigblue3	302.78	+4.6	150	**+0.04**	15
bigblue4	741.15	+1.6	90	**−0.1**	15
Geo-mean†		−0.50		**−0.61**	

★ Training design for Helix. † Computed over the same 7 non-diverged designs for both methods. Helix achieves −0.57% across all 8.

OpenROAD's C++ infrastructure. The integration requires only a new configuration file, a build script, and a benchmark parser [51]; Helix's evolutionary loop, operator templates, and fitness model are used without modification. Helix evolves with a single cross-design fitness on two training designs (adaptec1, bigblue1) and evaluates on all eight ISPD 2005 benchmarks [52]. Table 6 compares against the published EvoPlace configurations [50] evaluated on the same benchmarks. Because EvoPlace's source code is not available, we use the authors' released best configurations from their public repository, integrated into the DREAMPlace version mentioned therein. EvoPlace evolves per-design configurations over 30–930 generations; one design diverges and two regress, yielding −0.50% geometric-mean improvement on non-diverged designs. Helix achieves −0.61% geometric-mean improvement on the same seven non-diverged designs—but with 60× fewer generations, no per-design tuning, and no divergence. The above comparison is not strictly head-to-head: EvoPlace performs per-design evolution and also modifies initial macro placement, whereas Helix applies a single evolved codebase across all designs and restricts edits to the global placer. We therefore use Table 6 primarily to demonstrate Helix's portability across different placement engines.

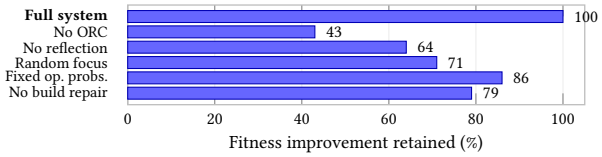


Figure 7: Component ablation on DPL evolution.

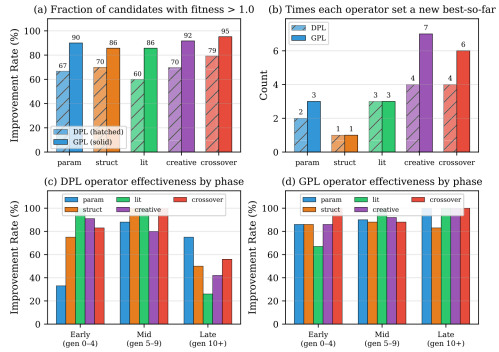


Figure 8: Operator analysis across evolutions. (a) Improvement rate (fraction of compiled candidates with fitness > 1.0). (b) Number of times each operator produced a new population best-so-far. (c, d) Improvement rate by evolution phase for DPL and GPL, respectively.

4.3 Ablation and Operator Analysis

We run ablation experiments on DPL evolution (same setup as Section 4.1.1; 15 generations, μ=5, λ=8), disabling exactly one component per run (Figure 7). Removing the *ORC* eliminates 57% of improvement: the LLM defaults to surface-level parameter edits and rarely attempts structural mutations. *Cross-generation learning/reflection* accounts for 36%; without it, offspring redundantly reattempt failed edits across generations. *Focus targeting* contributes 29%: random assignment causes 3.2× more offspring to redundantly target the same code region. *Build repair* contributes 21%, but disproportionately affects structural mutations, which fail to compile at 2× the rate of parameter mutations. *Adaptive scheduling* has the smallest individual impact (14%), but matters most in later generations when structural operators should dominate.[15]

Operator analysis. Figure 8 complements the component ablation with per-operator analysis across ∼300 compiled candidates. `Crossover` achieves the highest improvement rate (79% DPL, 95% GPL); `mutate_creative` produces the most new best-so-far individuals in GPL (7 of 20), and ties with `crossover` in DPL (4 each). Notably, `mutate_struct` produces the *final* best individual for both targets despite low frequency. Phase dynamics (Figures 8c, d) reveal that in DPL, `mutate_lit` dominates early (100%) but collapses late (26%), while `mutate_param` strengthens from 33% to 88% once structural scaffolding is in place. No single operator dominates across targets and phases (*portfolio effect*).

Resource usage. A 15-generation evolution on OpenROAD's GPL and DPL costs ∼$250–300 in LLM API calls at ∼$1.9–2.5 per candidate; LLM invocations account for ∼53% of wall-clock time, with the remainder C++ compilation (∼3 min per build) and benchmarking (∼5 min per candidate). The DREAMPlace evolution is cheaper per candidate ($1.5) because mutations target Python only; there, LLM calls dominate wall-clock time (∼89%) with total cost ∼$263.

5 Conclusion

We have presented Helix, a framework that uses LLM-driven evolutionary search to discover improved placement algorithms directly in OpenROAD's C++ codebase, reducing HPWL by up to 9% (4.5% geometric mean) and routed wirelength by up to 23% (5.6% geometric mean) while mostly preserving timing and power. Integration with DREAMPlace confirms that the framework generalizes across codebases and languages without modification. Our ablation study identifies a design principle that may extend beyond placement: LLM-driven code evolution in production systems requires both structured domain grounding and closed-loop physical feedback.

Several directions remain open. Multi-objective evolution over timing, power, and routability would enable PPA-aware algorithmic discovery. In particular, incorporating timing-aware fitness terms for WNS and TNS could prevent selection of wirelength-improving mutations that degrade critical paths, while richer analysis feedback could help attribute timing and congestion changes to specific code edits. Extension to other OpenROAD modules—CTS, GRT, RSZ—would test the generality of the ORC-and-evolution paradigm across the RTL-to-GDS flow, and RL- or bandit-based operator selection could improve sample efficiency in longer runs. Helix and all ORC artifacts are available at [51].

Acknowledgments

We thank Samsung AI Center for partial support of ABKGroup research.

[15] While ORC is most impactful, `mutate_creative`—which operates without literature guidance—often produces the best individuals (see operator analysis studies in [51]).

References

[1] A. Ghose, J. Jang, A. B. Kahng and J. Lee, "Automated QoR improvement in OpenROAD with coding agents", *arXiv:2601.06268*, 2025, https://arxiv.org/abs/2601.06268.

[2] H. G. Beyer and H. P. Schwefel, "Evolution strategies: a comprehensive introduction", *Natural computing* 1(1) (2002), pp. 3–52.

[3] J. Blocklove, S. Thakur, B. Tan, H. Pearce, S. Garg and R. Karri, "Automatically improving LLM-based Verilog generation using EDA tool feedback", *ACM Trans. on DAES* 30(6) (2025), pp. 1–26.

[4] U. Brenner, A. Hermann, N. Hoppmann and P. Ochsendorf, "BonnPlace: a self-stabilizing placement framework", *Proc. ISPD*, 2015, pp. 9–16.

[5] S. Cauley, V. Balakrishnan, Y. C. Hu and C.-K. Koh, "A parallel branch-and-cut approach for detailed placement", *ACM Trans. on DAES* 16(2) (2011), pp. 18:1–18:19.

[6] T.-C. Chen, Z.-W. Jiang, T.-C. Hsu, H.-C. Chen and Y.-W. Chang, "NTUplace3: an analytical placer for large-scale mixed-size designs with preplaced blocks and density constraints", *IEEE Trans. on CAD* 27(7) (2008), pp. 1228–1240.

[7] Y. Chen, Z. Wen, Y. Liang and Y. Lin, "Stronger mixed-size placement backbone considering second-order information", *Proc. ICCAD*, 2023, pp. 1–9.

[8] E. K. Burke, M. Gendreau, M. Hyde, G. Kendall, G. Ochoa, E. Ozcan et al., "Hyper-heuristics: a survey of the state of the art", *Journal of the Operational Research Society* 64(12) (2013), pp. 1695–1724.

[9] C.-K. Cheng, A. B. Kahng, I. Kang and L. Wang, "RePlAce: advancing solution quality and routability validation in global placement", *IEEE Trans. on CAD* 38(9) (2019), pp. 1717–1730.

[10] C.-K. Cheng, A. B. Kahng, S. Kundu, Y. Wang and Z. Wang, "Assessment of reinforcement learning for macro placement", *Proc. ISPD*, 2023, pp. 158–166.

[11] W.-K. Chow, J. Kuang, X. He, W. Cai and E. F. Y. Young, "Cell density-driven detailed placement with displacement constraint", *Proc. ISPD*, 2014, pp. 3–10.

[12] M. R. Garey, D. S. Johnson and L. Stockmeyer, "Some simplified NP-complete problems", *Proc. STOC*, 1974, pp. 47–63.

[13] D. E. Goldberg, *Genetic algorithms in search, optimization, and machine learning*, Addison-Wesley, 1989.

[14] H. H. Hoos, "Programming by optimization", *Communications of the ACM* 55(2) (2012), pp. 70–80.

[15] F. Hutter, T. Stützle, K. Leyton-Brown and H. H. Hoos, "ParamILS: an automatic algorithm configuration framework", *arXiv:1401.3492*, 2014, https://arxiv.org/abs/1401.3492.

[16] I. Gim, G. Chen, S.-S Lee, N. Sarda, A. Khandelwal and L. Zhong, "Prompt cache: modular attention reuse for low-latency inference", *Proc. MLSys* , 2024, pp. 325–338.

[17] M. López-Ibáñez, J. Dubois-Lacoste, L. P. Cáceres, M. Birattari and T. Stützle, "The irace package: Iterated racing for automatic algorithm configuration", *Operations Research Perspectives*, 2016, pp. 43–58.

[18] W. B. Langdon and J. Petke, "Genetic improvement of software for multiple objectives", *Proc. ISSBSE*, 2015, pp. 12–28.

[19] P. Liao, S. Liu, Z. Chen, W. Lv, Y. Lin and B. Yu, "DREAMPlace 4.0: timing-driven global placement with momentum-based net weighting", *Proc. DATE*, 2022, pp. 939–944.

[20] F. Liu, X. Tong, M. Yuan, X. Lin, F. Luo, Z. Wang et al., "Evolution of heuristics: towards efficient automatic algorithm design using large language model", *arXiv:2401.02051*, 2024, https://arxiv.org/abs/2401.02051.

[21] F. Liu, Y. Yao, P. Guo, Z. Yang, Z. Zhao, X. Lin et al., "A systematic survey on large language models for algorithm design", *arXiv:2410.14716*, 2024, https://arxiv.org/abs/2410.14716.

[22] J. Lu, P. Chen, C.-C Chang, L. Sha, D. J.-H. Huang, C.-C. Teng et al., "ePlace: electrostatics-based placement using fast Fourier transform and Nesterov's method", *ACM Trans. on DAES* 20(2) (2015), pp. 1–34.

[23] D. J. Mankowitz, A. Michi, A. Zhernov, M. Gelmi, M. Selvi, C. Paduraru et al., "Faster sorting algorithms discovered using deep reinforcement learning", *Nature* 618(7964) (2023), pp. 257–263.

[24] A. Novikov, N. Vũ, M. Eisenberger, E. Dupont, P.-S. Huang, A. Z. Wagner et al., "AlphaEvolve: a coding agent for scientific and algorithmic discovery", *arXiv:2506.13131*, 2025, https://arxiv.org/abs/2506.13131.

[25] J. R. Rice, "The algorithm selection problem", *Advances in Computers*, 1978, pp. 65–118.

[26] B. Romera-Paredes, M. Barekatain, A. Novikov, M. Balog, M. P. Kumar, E. Dupont et al., "Mathematical discoveries from program search with large language models", *Nature* 625(7995) (2024), pp. 468–475.

[27] N. Shinn, F. Cassano, A. Gopinath, K. Narasimhan and S. Yao, "Reflexion: language agents with verbal reinforcement learning", *Proc. NeurIPS*, 2023, pp. 8634–8655.

[28] S. Watanabe, "Tree-structured Parzen estimator: understanding its algorithm components and their roles for better empirical performance", *arXiv:2304.11127* , 2023, https://arxiv.org/abs/2304.11127.

[29] W. Weimer, T. Nguyen, C. L. Goues and S. Forrest, "Automatically finding patches using genetic programming", *Proc. ICSE*, 2009, pp. 364–374.

[30] Q. Wu, G. Bansal, J. Zhang, Y. Wu, B. Li, E. Zhu et al., "AutoGen: enabling next-gen LLM applications via multi-agent conversation", *arXiv:2308.08155*, 2023, https://arxiv.org/abs/2308.08155.

[31] S. Yao, F. Liu, X. Lin, Z. Lu, Z. Wang and Q. Zhang, "Multi-objective evolution of heuristic using large language model", *Proc. AAAI*, 2025, pp. 27144–27152.

[32] X. Yao, J. Jiang, Y. Zhao, P. Liao, Y. Lin and B. Yu, "Evolution of optimization algorithms for global placement via large language models", *arXiv:2504.17801*, 2025, https://arxiv.org/abs/2504.17801.

[33] Y. Yao, F. Liu, J. Cheng and Q. Zhang, "Evolve cost-aware acquisition functions using large language models", *Intl. Conf. on PPSN*, 2024, pp. 374-390.

[34] S. Xu, Z. Wu, H. Zhao, P. Shu, Z. Liu, W. Liao et al., "Reasoning before comparison: LLM-enhanced semantic similarity metrics for domain specialized text analysis", *arXiv:2402.11398*, 2024, https://arxiv.org/abs/2402.11398.

[35] W. Zhu, J. Chen, Z. Peng and G. Fan, "Nonsmooth optimization method for VLSI global placement", *IEEE Trans. on CAD* 34(4) (2015), pp. 642–655.

[36] ASAP7 PDK and cell libraries repo. https://github.com/The-OpenROAD-Project/asap7

[37] Claude code. https://www.claude.com/product/claude-code

[38] Claude Opus 4.6. https://www.anthropic.com/news/claude-opus-4-6

[39] Codex CLI. https://developers.openai.com/codex/cli

[40] Cursor CLI. https://cursor.com/cli

[41] JSON schema draft 7. https://json-schema.org/draft-07

[42] JSON schema. https://json-schema.org/specification

[43] NanGate45 PDK. https://github.com/The-OpenROAD-Project/OpenROAD-flow-scripts/tree/master/flow/platforms/nangate45

[44] OpenAI GPT-5.4. https://openai.com/index/introducing-gpt-5-4/

[45] OpenROAD-Flow-Scripts (commit hash: df27ce6). https://github.com/The-OpenROAD-Project/OpenROAD-flow-scripts

[46] OpenROAD repo (commit hash: 56355d0). https://github.com/The-OpenROAD-Project/OpenROAD.git

[47] Global placement module in OpenROAD (commit hash: 56355d0). https://github.com/The-OpenROAD-Project/OpenROAD/tree/master/src/gpl

[48] Detailed placement module in OpenROAD (commit hash: 56355d0). https://github.com/The-OpenROAD-Project/OpenROAD/tree/master/src/dpl

[49] DREAMPlace repo (commit hash: 105ff79). https://github.com/limbo018/DREAMPlace

[50] EvoPlace evolved DREAMPlace configurations. https://github.com/yaoxufeng/EvoPlace/tree/master/total_best_configs

[51] Helix repository. https://github.com/ABKGroup/HELIX

[52] ISPD2005 benchmarks. https://github.com/limbo018/DREAMPlace/tree/master?tab=readme-ov-file#how-to-get-benchmarks

[53] Optuna. https://github.com/optuna/optuna

[54] Ray Tune. https://docs.ray.io/en/latest/tune/index.html