MappingEvolve: LLM-Driven Code Evolution for Technology Mapping

Rongliang Fu
Chinese University of Hong Kong

Yi Liu
Chinese University of Hong Kong

Qiang Xu
Chinese University of Hong Kong

Tsung-Yi Ho
Chinese University of Hong Kong

Abstract

Technology mapping is a critical yet challenging stage in logic synthesis. While Large Language Models (LLMs) have been applied to generate optimization scripts, their potential for core algorithm enhancement remains untapped. We introduce MappingEvolve, an open-source framework that pioneers the use of LLMs to directly evolve technology mapping code. Our method abstracts the mapping process into distinct optimization operators and employs a hierarchical agent-based architecture, comprising a Planner, Evolver, and Evaluator, to guide the evolutionary search. This structured approach enables strategic and effective code modifications. Experiments show our method significantly outperforms direct evolution and strong baselines, achieving 10.04% area reduction versus ABC and 7.93% versus mockturtle, with 46.6%–96.0% $S_{overall}$ improvement on EPFL benchmarks, while explicitly navigating the area–delay trade-off. We have open-sourced our framework to foster reproducibility and further research.

1 Introduction

Logic synthesis plays a pivotal role in the circuit design process, primarily comprising two sub-processes: logic optimization and technology mapping. Among them, technology mapping is particularly crucial as it bridges the gap between logical design and physical design.

Existing optimization methods for technology mapping primarily focus on three approaches: i) cut selection to filter for logic-level or physical-level superior cuts, thereby obtaining better gate-level netlists, such as Priority Cuts [1], SLAP [2], LEAP [3], and PigMAP [4]; ii) multi-output or application-specific cell exploitation, such as emap [5] and dual-output LUT [6]; iii) standard cell library filtering and extension to shrink the search range or improve cell quality, such as MapTune [7] and TeMACLE [8]. While these advances improve mapping quality, they predominantly refine heuristic parameters and search strategies rather than the core algorithmic logic within the mapper itself.

Recent advances in Large Language Models (LLMs), including GPT-5 [9], DeepSeek-V3 [10], and Qwen [11], have demonstrated remarkable capabilities in code comprehension and algorithmic reasoning. These capabilities have led to initial research applying LLMs to logic synthesis, such as ChatLS [12] and LLSM [13]. However, these studies mainly focus on generating optimization scripts rather than evolving the logic synthesis algorithms themselves. Unlike conventional automated parameter tuning over predefined search spaces, LLMs can synthesize semantically meaningful algorithmic modifications within bounded code regions [14]. Recent work like OpenEvolve [15] demonstrates population-based code evolution through iterative LLM-driven mutation and selection, but lacks strategic guidance for complex algorithmic optimization. These observations motivate us to explore LLM-driven code evolution tailored for technology mapping algorithms.

To this end, we propose MappingEvolve, an open-source framework that leverages LLMs to directly evolve the core algorithms of technology mapping. Through systematic analysis of existing mapping implementations, we identify three fundamental operators: MatchPhase (delay and area-flow optimization), MatchPhaseExact (exact-area optimization), and MatchDropPhase (phase unification). They encapsulate critical algorithmic trade-offs while providing well-defined boundaries amenable to controlled evolution. Our framework employs a hierarchical Planner → Evolver → Evaluator architecture that decouples strategic operator selection from concrete heuristic mutation. To ensure functional correctness, we enforce syntactic and semantic constraints through bounded edit regions and multi-stage validation comprising compilation, logical equivalence checking, and quality-of-result evaluation. A unified performance metric $S_{overall}$ quantifies the area-delay trade-off to guide the evolution process.

Overall, the main contributions of this work are as follows:

- To our knowledge, we present the first LLM-driven framework that directly evolves the core algorithmic operators of technology mapping, in contrast to prior approaches limited to external script generation.
- We design a hierarchical Planner-Evolver-Evaluator architecture that enforces safety through syntactic boundaries and multi-stage validation (compilation, logical equivalence checking, and quality-of-result assessment), enabling safe exploration of the algorithmic design space.
- Experimental results demonstrate 11.5× performance improvement over direct operator evolution on ISCAS85 benchmarks [16], achieving 10.04% area reduction versus ABC [17] and 7.93% versus mockturtle [18], with 46.6%–96.0% $S_{overall}$ improvement on EPFL benchmarks [19].
- We release our complete implementation, including source code, evolution prompts, and per-iteration artifacts, to facilitate reproducibility and future research in this field.

2 Technology Mapping Analysis

2.1 Technology Mapping Algorithm

Technology mapping translates a technology-independent Boolean network G with node set V, typically an And-Inverter Graph (AIG) [17, 20], into a gate-level netlist built from a given standard cell library L. Given G and L, the goal is to find a mapping $M : V \rightarrow L$ that minimizes optimization objectives such as circuit area $A(M)$ =

DAC '26, Long Beach, CA, USA

ACM ISBN 979-8-4007-2254-7/2026/07
https://doi.org/10.1145/3770743.3803988

$\sum_{v \in V} \text{area}(M(v))$, worst-case delay $D(M) = \max_{p \in \text{Paths}(G)} \sum_{v \in p} \text{delay}(M(v))$, or a weighted combination thereof.

Definition 1 (Logic Phase). *Each node v can be realized in two output* phases*: positive ($\phi = 0$, non-inverted) or negative ($\phi = 1$, inverted). When matching a cut c to a gate g, the gate's input phases determine the required phase $\phi_l \in \{0, 1\}$ for each leaf $l \in c$. If a gate requires a phase not currently implemented by its fanin, an inverter must be inserted.*

A key challenge in technology mapping is balancing multiple conflicting optimization objectives. Optimizing solely for delay often results in excessive area overhead (due to gate duplication and increased logic depth buffering), while aggressive area minimization can degrade timing performance (by sharing gates across multiple paths, increasing fanout and delay). To address this trade-off, modern technology mappers adopt an iterative refinement strategy that progressively optimizes different objectives across multiple rounds.

To enable LLM-driven evolution, we analyze the mockturtle [18] library's implementation and formalize its multi-round mapping process. We identify that the algorithm relies on three optimization metrics applied across different rounds:

- **DelayFlow** records the arrival time at node v for cut c matched to gate g:
$$T(v, c, g) = \text{delay}(g) + \max_{l \in c} T(l, \phi_l), \tag{1}$$
where l denotes each leaf in cut c, $\phi_l \in \{0, 1\}$ is the required phase at leaf l (determined by gate g's input phases as defined in Definition 1), and $\text{delay}(g)$ denotes the maximum pin-to-output delay across all inputs of gate g.
- **AreaFlow** estimates the amortized area contribution:
$$F(v, c, g) = \text{area}(g) + \sum_{l \in c} \frac{F(l, \phi_l)}{\text{fo}(l)}, \tag{2}$$
where $\text{fo}(l)$ is the estimated fanout count of leaf node l. This metric recursively accounts for area flow from fanin nodes, assuming each node's cost is shared among its fanouts proportionally.
- **ExactArea** records the true incremental area through reference counting:
$$E(v, c, g) = \text{area}(g) + \sum_{\substack{l \in c \\ \rho(l, \phi_l) = 0}} E(l, \phi_l), \tag{3}$$
where $\rho(l, \phi_l)$ is the current reference count of leaf l at phase ϕ_l. Only leaves with zero reference count (i.e., $\rho(l, \phi_l) = 0$) contribute their local area $E(l, \phi_l)$ to the total.

This staged optimization, beginning with delay-oriented optimization using DelayFlow(T), transitioning to area flow optimization using AreaFlow(F), and culminating in exact area optimization using ExactArea(E), allows the algorithm to efficiently explore quality-runtime trade-offs.

2.2 Operator Abstraction

To enable LLM-driven evolution of technology mapping algorithms, we systematically analyze the implementation and abstract the iterative optimization process into a unified algorithmic framework shown in Algorithm 1. The algorithm executes R optimization rounds, where $R = R_{delay} + R_{flow} + R_{exact}$ combines delay rounds (R_{delay}, typically 1), area flow rounds (R_{flow}, typically 1-2), and exact area rounds (R_{exact}, typically 2-3). Each round processes all nodes in topological order. Critically, we identify that the mapping quality is determined by three core operators repeatedly invoked across all rounds:

Algorithm 1: Technology mapping with three key operators.

Input: Logic network G, standard cell library L
Output: Mapped netlist N
Data: R_{delay}: delay rounds, R_{flow}: area flow rounds, R_{exact}: exact area rounds, $R = R_{delay} + R_{flow} + R_{exact}$

```
// Iterative optimization with varying objectives
1  for round r = 1 to R do
2      for each node v ∈ G in topological order do
3          if r ≤ R_delay + R_flow then
               // Delay and area flow optimization rounds
4              MatchPhase(v, phase = 0, L);
5              MatchPhase(v, phase = 1, L);
6          else
               // Exact area optimization rounds
7              MatchPhaseExact(v, phase = 0, L);
8              MatchPhaseExact(v, phase = 1, L);
9          MatchDropPhase(v);
10     Update mapping coverage and compute metrics;
11     Backward propagate required times;
12 Finalize netlist N from selected cuts;
13 return N
```

- **MatchPhase** (Lines 4-5): For a given node v and output phase $\phi \in \{0, 1\}$, this operator evaluates all feasible cuts $C(v)$ and their compatible library gates. For each cut $c \in C(v)$ and gate $g \in L$, it computes the arrival time $T(v, c, g)$ (via Equation (1)) and area flow $F(v, c, g)$ (via Equation (2)), then selects the best match by minimizing a cost function $C(c, g) = \alpha_r \cdot T(v, c, g) + (1 - \alpha_r) \cdot F(v, c, g)$, where $\alpha_r \in [0, 1]$ determines the delay-area trade-off priority in round r. In practice, α_r starts close to 1 in delay-oriented rounds (emphasizing T) and decreases toward 0 in area-flow rounds (increasing weight on F). This operator is invoked in delay and area flow optimization rounds (when $r \leq R_{delay} + R_{flow}$).
- **MatchPhaseExact** (Lines 7-8): This operator serves the same purpose as MatchPhase but performs exact area computation by recursively dereferencing the previous best cut from the current mapping cover and then evaluating each candidate cut through recursive reference counting (via Equation (3)). Unlike the flow-based heuristics in MatchPhase, this yields the true incremental area $E(v, c, g)$ but at a higher computational cost. In exact-area rounds, candidate selection minimizes $E(v, c, g)$ directly under timing feasibility constraints. It replaces MatchPhase in the exact area optimization rounds (when $r > R_{delay} + R_{flow}$).
- **MatchDropPhase** (Line 9): After both output phases have been matched, this operator attempts to unify them by checking whether a single gate match combined with an output inverter can cover both phases at a lower total cost. If unification reduces cost without violating timing constraints,

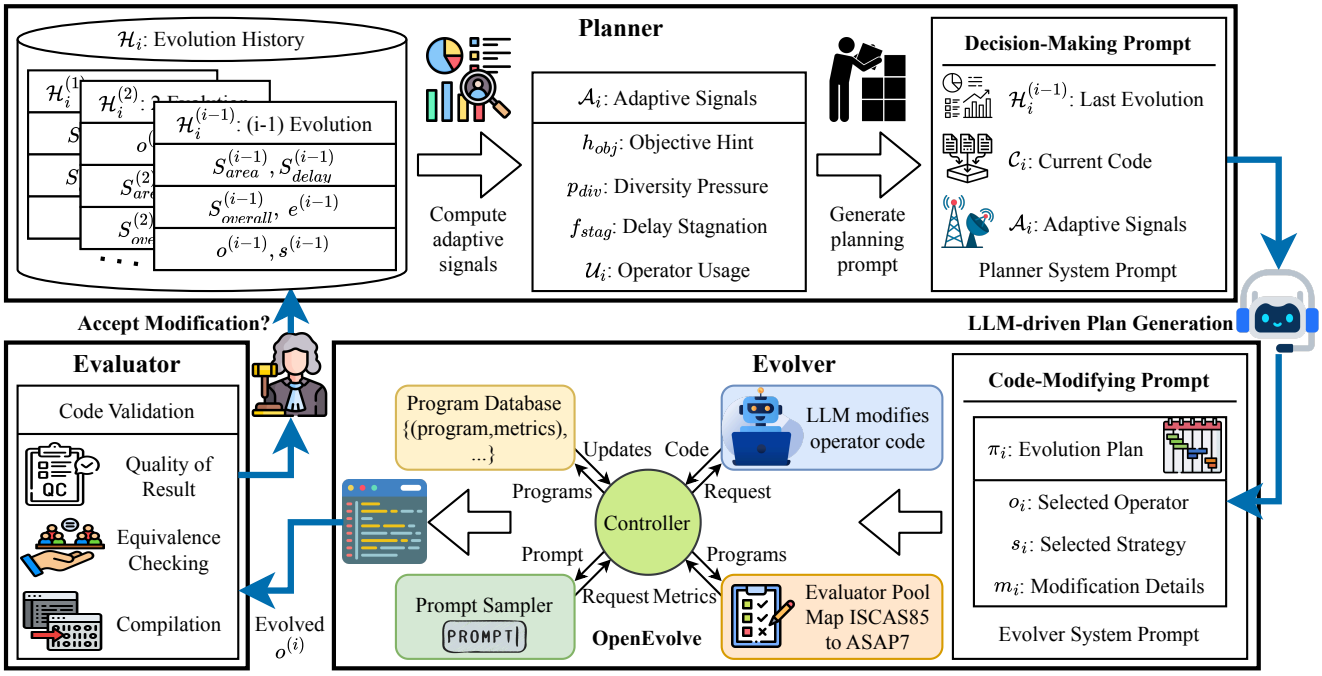

Figure 1. The overall flow of MappingEvolve.

it eliminates the redundant phase match. This operator is applied in every optimization round.

After each round, the algorithm updates the mapping coverage and computes required arrival times to guide subsequent rounds (Lines 10-11).

2.3 LLM-driven Evolution Target Selection

The three operators identified above, O ={MatchPhase, MatchPhaseExact, MatchDropPhase}, encapsulate the core optimization logic of technology mapping algorithms. We select these operators as ideal targets for LLM-driven code evolution because they exhibit three key properties that balance evolvability with safety:

First, each operator implements **self-contained optimization heuristics** with tunable parameters and trade-off decisions (e.g., cost function weights in MatchPhase, timing slack thresholds in MatchPhaseExact, phase unification criteria in MatchDropPhase), precisely the type of algorithmic choices where LLMs explore strategies beyond human-designed rules.

Second, the mockturtle library [18] provides **well-defined modular interfaces**, allowing us to isolate each operator in a separate C++ source file. We designate safe modification regions using `EVOLVE-BLOCK` markers, ensuring LLM edits are confined to optimization logic (e.g., cost function computation) while preserving API integrity (e.g., class member access).

Third, operator modifications directly impact **measurable performance metrics** (area $A(M)$ and delay $D(M)$), enabling quantitative feedback for iterative evolution. Combined with compilation and equivalence checks, this establishes a robust validation process.

By abstracting the mapping algorithm into these three evolvable operators with bounded modification regions, we create a structured search space that balances exploration freedom with safety guarantees, a key enabler for systematic LLM-driven algorithm improvement.

3 MappingEvolve

This section presents MappingEvolve, a framework that leverages LLMs to evolve technology mapping algorithms, as shown in Fig. 1. Based on the operator abstraction established in Section 2.2, MappingEvolve employs a hierarchical Planner→Evolver→Evaluator architecture to iteratively evolve the three core operators. Unlike direct code generation approaches that lack strategic guidance, this architecture decouples strategic planning from implementation, enabling targeted exploration of algorithmic improvements while maintaining code safety through bounded modification regions and multi-stage validation.

3.1 Optimization Objectives

As formalized in Section 2.1, technology mapping seeks to minimize circuit area $A(M)$ and delay $D(M)$ for a given mapping M. These objectives often conflict: aggressive area minimization may increase delay, while delay-optimal mappings frequently require extra gates.

To guide the evolution process and quantify the area-delay trade-off, we define four metrics for evaluating evolved mapping M' relative to the original M:

- $S_{area}(M') = \frac{A(M)-A(M')}{A(M)}$: Normalized area reduction.
- $S_{delay}(M') = \frac{D(M)-D(M')}{D(M)}$: Normalized delay reduction.
- $S_{overall}(M') = \alpha \cdot S_{area}(M') + (1-\alpha) \cdot S_{delay}(M')$: Weighted combination where $\alpha \in [0,1]$ controls area-delay priority.
- $e(M') \in [0,1]$: Logical equivalence failure rate (must be zero for valid implementations).

To navigate the area-delay trade-off, we define three optimization strategies $\mathcal{S}$:

- **Area-opt** prioritizes area reduction while maintaining timing feasibility.
- **Delay-opt** prioritizes delay reduction, tolerating modest area increases.
- **Balanced** seeks simultaneous improvements via $S_{overall}$.

Strategy selection adapts to performance signals from recent iterations: when area improves but delay consistently degrades, **delay-opt** restores timing quality; when area shows severe degradation, **area-opt** focuses on area optimization; when both objectives show no clear trend, **balanced** explores the Pareto frontier.

3.2 Hierarchical Agent Architecture

The framework comprises three components: a **Planner** selecting optimization targets based on evolution history, an **Evolver** generating plan-conditioned mutations, and an **Evaluator** validating implementations. This decouples strategic planning from code generation and quality assessment.

3.2.1 Planner The Planner prepares structured inputs for an LLM-based decision maker and specifies the required output format for downstream Evolver. It decouples *what to optimize* (strategic planning) from *how to implement* (code generation), enabling systematic exploration while avoiding local optima.

At iteration i, the Planner processes evolution history $\mathcal{H}_i = \{(S_{area}^{(j)}, S_{delay}^{(j)}, S_{overall}^{(j)}, e^{(j)}, o^{(j)}, s^{(j)})\}_{j=\max(1,i-w)}^{i-1}$ of the most recent w iterations, where $o^{(j)} \in \mathcal{O}$ is the modified operator at iteration j and $s^{(j)} \in \mathcal{S}$ is the employed strategy. It extracts current code context $C_i = \{c_o\}_{o\in\mathcal{O}}$ (where c_o is operator o's implementation) and computes adaptive signals $\mathcal{A}_i = \{h_{obj}, p_{div}, f_{stag}, \mathcal{U}_i\}$ from $\mathcal{H}_i$ to guide decision-making:

- *Objective hint* $h_{obj} \in \{\text{delay}, \text{area}, \text{balanced}\}$ guides strategy prioritization, computed as:

$$h_{obj} = \begin{cases} \text{area} & \text{if } (n_{area} > 0 \wedge \bar{\Delta}_{delay} \geq 0) \\ \text{delay} & \text{else if } (n_{delay} > 0 \wedge \bar{\Delta}_{area} \geq 0) \\ \text{balanced} & \text{otherwise} \end{cases}, \quad (4)$$

$$n_{area} = |\{j \in \mathcal{H}_i : S_{area}^{(j)} > S_{area}^{(j-1)}\}|, n_{delay} \text{ (similarly)}, \quad (5)$$

$$\bar{\Delta}_{area} = \frac{1}{|\mathcal{H}_i|} \sum_{j\in\mathcal{H}_i} (S_{area}^{(j)} - S_{area}^{(j-1)}), \bar{\Delta}_{delay} \text{ (similarly)}. \quad (6)$$

  If one metric improves while the other does not worsen, the hint guides continued optimization of the improving metric (e.g., if area reduces but delay remains stable, h_{obj} = area).

- *Diversity pressure* $p_{div} = \begin{cases} \text{HIGH} & \text{if } n_{cs} \geq \lceil w/2 \rceil \\ \text{LOW} & \text{otherwise} \end{cases}$ detects operator over-exploitation, where n_{cs} counts consecutive selections of the same operator. If p_{div} = HIGH, least-used operator $o_i \in \{o \in \mathcal{O} : n_o = \min_{o'\in\mathcal{O}} n_{o'}\}$ is selected to escape local optima by exploring complementary operators rather than over-exploiting a single search space.
- *Delay stagnation* $f_{stag} = \mathbb{I}[n_{area} > 0 \wedge \forall j \in \mathcal{H}_i, S_{delay}^{(j)} < 0]$ flags dangerous trade-off patterns when area gains consistently sacrifice timing. If $f_{stag} = 1$, s_i = delay-opt.
- *Operator usage* $\mathcal{U}_i = \{(o, n_o)\}_{o\in\mathcal{O}}$, where $n_o = |\{j \in \mathcal{H}_i : o^{(j)} = o\}|$, tracks selection frequency to identify under-explored operators.

The Planner constructs a decision-making prompt $(\mathcal{H}_i^{(i-1)}, C_i, \mathcal{A}_i)$ and sends it to an LLM, where $\mathcal{H}_i^{(i-1)}$ contains performance data from iteration $i-1$. The Planner specifies the required output format: evolution plan $\pi_i = (o_i, s_i, m_i)$, where $o_i \in \mathcal{O}$ is the selected operator, $s_i \in \mathcal{S}$ is the optimization strategy, and m_i describes the modification (target code region, implementation approach, expected impact $\Delta_{exp} \in \mathbb{R}^2$, and constraints). The LLM generates π_i by reasoning over the previous iteration's results, code structure, and adaptive signals. The Planner validates the output format and forwards π_i to the Evolver for implementation.

3.2.2 Evolver The Evolver translates the Planner's evolution plan $\pi_i = (o_i, s_i, m_i)$ into concrete code modifications through plan-conditioned population evolution. Unlike general-purpose code generation that explores the full modification space $\mathcal{X}_{all}$, the Evolver constrains search to a strategically reduced subspace $\mathcal{X}_{o_i,s_i} \subset \mathcal{X}_{all}$ aligned with π_i.

Plan-Guided Search Space Reduction. We implement the Evolver using OpenEvolve [15], a population-based LLM code evolution framework that maintains multiple candidate solutions and employs island-based parallel evolution with iterative mutation and selection. Crucially, we extend OpenEvolve from general-purpose code generation to plan-conditioned evolution by injecting π_i into its system prompt as structured constraints:

- **Target Region**: m_i specifies the code region in current implementation $c_{o_i} \in C_i$ of the selected operator o_i to confine mutations.
- **Implementation Guidance**: m_i provides algorithmic hints aligned with s_i (e.g., for **delay-opt**, "prioritize cuts with minimal depth increase").
- **Expected Impact**: $\Delta_{exp} = (\Delta_{area}, \Delta_{delay})$ filters candidate solutions during evaluation.
- **Semantic Constraints**: m_i encodes operator invariants (e.g., "ensure matching respects library gate fanin limits") to prevent invalid mutations.

This transforms OpenEvolve from undirected search into a strategic exploration engine: the population evolves within $\mathcal{X}_{o_i,s_i}$ defined by both *where* (operator o_i, code region) and *how* (strategy s_i, implementation hints), improving convergence while maintaining diversity through island-based parallelism.

Multi-Layered Safety Enforcement. The Evolver enforces safety through hierarchical constraints:

(1) **Syntactic Boundaries**: Edits confined to `EVOLVE-BLOCK` regions containing only optimization heuristics, excluding function signatures, API interfaces, and data structures.
(2) **Semantic Constraints**: m_i specifies algorithmic invariants (e.g., "k-LUT input limit") validated before compilation, filtering invalid mutations early.
(3) **Incremental Validation**: OpenEvolve's island evolution creates a feedback loop (compilation → equivalence → QoR) through the Evaluator, enabling convergence to valid solutions.

This combination of plan-guided search reduction and multi-layered constraints enables complex modifications while maintaining correctness. It is critical for technology mapping, as subtle logic errors produce functionally incorrect circuits undetected by compilation.

3.2.3 Evaluator The Evaluator receives evolved operator code from the Evolver, merges it with the unchanged operators from C_i to form a complete mapper implementation, then validates the integrated code through a three-stage pipeline and generates a reward signal R that quantifies modification quality. Each candidate undergoes hierarchical validation with corresponding penalties:

$$R = \begin{cases} r_{compile} & \text{if compilation fails} \\ r_{equiv} - \beta \cdot e & \text{if equivalence fails, } e > 0 \\ \max(r_{equiv}, S_{overall}) & \text{if } S_{overall} < 0 \\ \frac{S_{overall}}{1+S_{overall}} & \text{if } S_{overall} \geq 0 \end{cases}, \quad (7)$$

where $r_{equiv} < 0$, $\beta > 0$, and $r_{compile} = r_{equiv} - \beta$. This hierarchical design maps validation stages to differentiated feedback:

- **Compilation** ($r_{compile}$): Merges the evolved operator c_{o_i} with unchanged operators from C_i into a complete mapper, then builds the integrated code and returns diagnostics. Failures receive the harshest penalty, immediately rejecting syntactically invalid code.
- **Equivalence Checking** ($r_{equiv} - \beta \cdot e$): Verifies output netlist functional equivalence using ABC [17]'s combinational equivalence checker `cec`. This is critical for technology mapping, where performance optimizations are meaningless if circuit functionality is corrupted. Violations incur strong penalties proportional to the failure rate e, discouraging semantic errors while allowing partial progress.
- **Quality of Result** ($\max(r_{equiv}, S_{overall})$ or $\frac{S_{overall}}{1+S_{overall}}$): Evaluates the validated mapper M' on benchmark suite $\mathcal{B}$ and computes $S_{area}, S_{delay}, S_{overall}$ as defined in Section 3.1. For regressions ($S_{overall} < 0$), capping at r_{equiv} prevents catastrophic penalties for valid but suboptimal code, enabling exploration of temporary regressions. For improvements ($S_{overall} \geq 0$), sigmoid transformation bounds rewards to $(0, 1)$ to prevent unbounded optimization bias.

The evaluation record $(S^{(i)}_{area}, S^{(i)}_{delay}, S^{(i)}_{overall}, e^{(i)}, o^{(i)}, s^{(i)})$ enters the evolution history $\mathcal{H}_{i+1}$. The Planner analyzes this accumulated history through adaptive signals $\mathcal{A}_{i+1}$, establishing a strategic feedback loop where Evaluator results at iteration i shape the Planner's inputs for iteration $i + 1$.

3.3 Iterative Evolution Loop

The evolution process operates in iterations. Each iteration executes the following steps:

(1) The Planner analyzes performance history and proposes an evolution plan.
(2) The Evolver implements the proposed modifications to the selected operator.
(3) The Evaluator merges modified code without any changes, then compiles, tests, and measures the evolved mapper.
(4) MappingEvolve applies a multi-criteria acceptance policy to decide whether to accept the modification, then updates the state and feeds results back to the Planner. A modification is accepted if it achieves sufficient reward ($R_i \geq \tau$) or demonstrates significant delay improvement ($S^{(i)}_{delay} \geq \gamma \cdot S^{best}_{delay}$, where $\gamma \in (0, 1]$). The latter criterion preserves delay optimizations even when the overall reward is suboptimal, recognizing that delay reduction is harder to achieve than area optimization. State updates include:
   - **Code state** C_{i+1}: If accepted, $c^{(i+1)}_{o_i} \leftarrow$ evolved code; otherwise $c^{(i+1)}_{o_i} \leftarrow c^{(i)}_{o_i}$. The code for unselected operators remains unchanged.
   - **Evolution history** $\mathcal{H}_{i+1}$: Appends the current evaluation record $(S^{(i)}_{area}, S^{(i)}_{delay}, S^{(i)}_{overall}, e^{(i)}, o^{(i)}, s^{(i)})$, retaining only the most recent w iterations.
   - **Best performance** S^{best}_{delay}: Updates to $\max(S^{best}_{delay}, S^{(i)}_{delay})$ for acceptance criteria.

The evolution continues for a maximum of N iterations or no improvement over consecutive iterations.

Table 1. Comparison of $S_{overall}$ and e scores across different models and methods on ISCAS85 benchmarks [16].

Model		Base Model	$S_{overall}$	e
OpenEvolve	MatchPhase	DeepSeek-V3	0.00	0.00
	MatchPhaseExact		0.00	0.00
	MatchDropPhase		0.02	0.09
MappingEvolve		DeepSeek-V3	0.23	0.00
		Qwen3-Max	0.18	0.00
		GPT-5	0.30	0.00

4 Experimental Results

4.1 Experimental Setup

4.1.1 Implementation Details MappingEvolve is implemented in Python and evaluated on Ubuntu 22.04 with Intel Xeon Gold 6226R CPU @ 2.90GHz and 256GB memory. We use OpenEvolve [15] as the Evolver component, executing $N = 30$ outer iterations with 3 OpenEvolve iterations per step. During evolution, candidates are validated on ISCAS85 benchmarks [16] (11 circuits), where failure rate e measures the ratio of non-equivalent circuits. For QoR evaluation, circuits are optimized via ABC's `compress2`, then mapped to ASAP7 [21] using the evolved mapper. We evaluate using DeepSeek-V3 [10], Qwen3-Max [11], and GPT-5 as base LLMs.

Hyperparameters. $\alpha = 0.5$ (balanced area-delay weighting), $\tau = -0.1$ (reward threshold allowing slight degradation), $\gamma = 0.8$ (delay preservation factor), $w = 5$ (history window). Reward function: $r_{equiv} = -0.4$, $\beta = 0.1$, $r_{compile} = -0.5$, thus $R \in [-0.5, 0.5]$.

Computational Cost. The entire evolution process of MappingEvolve takes $\sim$ 1 **million tokens (**\$10**)** and $\sim$ 1.5 **hours**.

4.1.2 Benchmarks and Baselines We use ISCAS85 benchmarks [16] for ablation studies (TABLE 1) and EPFL benchmarks [19] for performance comparison (TABLE 2).

Experiment 1: Framework Effectiveness. We compare MappingEvolve against OpenEvolve using DeepSeek-V3 [10]. OpenEvolve evolves each operator (MatchPhase, MatchPhaseExact, MatchDropPhase) separately for 90 iterations (matching $30 \times 3 = 90$ total OpenEvolve calls in MappingEvolve for fair comparison).

Table 2. Technology mapping results on EPFL benchmarks [19] with ASAP7 standard cell library [21].

Circuit	AIG		ABC			mockturtle			MappingEvolve (Ours)					
									DeepSeek-V3			GPT-5		
	size	depth	area	delay	t(s)	area	delay	t(s)	area	delay	t(s)	area	delay	t(s)
adder	1019	255	100.53	2574.36	0.05	92.42	2574.36	0.06	76.94	2583.24	0.02	90.6	2768.87	0.06
bar	3141	12	263.63	169.9	0.08	298.82	171.90	0.07	297.54	171.9	0.08	225.74	180.16	0.07
div	40633	4394	4021.98	43768.73	0.76	3510.25	43295.70	1.11	2930.53	45227.7	1.26	3012.57	43543.04	1.14
hyp	211329	24893	16756.18	198844.69	5.4	15895.39	197081.23	7.05	13996.27	238083.28	7.27	15296.85	198729.45	6.51
log2	29371	387	2019.22	3990.67	1	2047.72	3968.44	2.11	1641.84	4642.54	2.59	1853.82	4126.81	2.4
max	2832	206	256.14	2101.85	0.1	225.89	2101.13	0.12	183.96	2151.64	0.19	202.65	2220.27	0.18
multiplier	24556	262	1816.61	2730.91	0.54	1913.22	2662.96	0.82	1632.12	2956.62	1	1743.2	2673.95	0.88
sin	5041	179	433.57	1843.39	0.18	405.80	1832.67	0.37	324.23	2032.53	0.47	367.25	1910.26	0.4
sqrt	18368	6048	1525.72	49967.46	0.67	1484.04	47964.09	0.56	1206.42	57824.42	0.7	1586.17	55116.87	0.59
square	16623	248	1243	2509.4	0.43	1208.58	2509.22	0.39	1161.13	2856.92	0.45	1149.15	2519.15	0.41
arbiter	11839	87	783.69	898.75	0.16	766.44	898.75	1.65	766.44	898.75	1.78	591.09	969.11	1.67
cavlc	662	16	41.19	188.71	0.05	40.69	185.77	0.03	40.86	185.77	0.03	38.34	191.66	0.03
ctrl	108	8	7.51	104.36	0.06	6.92	103.17	0.00	6.9	103.17	0.01	6.9	106.33	0.01
dec	304	3	28.73	66.15	0.06	30.83	65.72	0.06	29.75	65.72	0.08	27.59	66.15	0.07
i2c	1161	15	70.43	165.99	0.07	70.23	164.10	0.06	70.23	164.1	0.07	69.13	174.18	0.07
int2float	214	15	13.39	181.17	0.05	12.80	181.00	0.01	12.8	181	0.02	12.3	193.8	0.01
mem_ctrl	45547	106	2798.58	1015.48	0.89	2716.20	1006.24	1.84	2623.27	1072.94	2.15	2631.09	1040.39	1.85
priority	683	214	52.63	2155.94	0.07	52.20	2155.87	0.03	57.73	2288.23	0.03	50.81	2264.96	0.04
router	182	18	12.7	187.2	0.06	12.85	187.20	0.02	12.93	187.2	0.03	12.25	192.22	0.03
voter	9654	59	1004.29	729.37	0.22	954.64	723.58	0.37	705.63	843.44	0.37	761.58	773.44	0.38
Ave. ratio	–		1	1	1	0.9771	0.9926	1.5474	0.8996	1.0571	1.7550	0.8996	1.0369	1.6557
$S_{overall}$			0			0.0174			0.0255			**0.0341**		

Experiment 2: Model Generalization. We evaluate MappingEvolve with different base LLMs, including DeepSeek-V3 [10], Qwen3-Max [11], and GPT-5 [9].

Experiment 3: Performance Comparison. We assess the evolved mappers against ABC (`&nf`) [17] and mockturtle (`map`) [18] on EPFL benchmarks [19]. All netlists pass ABC's `cec` equivalence checking, ensuring functional correctness.

4.2 Results and Analysis

Effectiveness of Hierarchical Architecture TABLE 1 compares OpenEvolve and MappingEvolve on ISCAS85 using DeepSeek-V3. OpenEvolve achieves minimal improvements: MatchPhase and MatchPhase-Exact yield $S_{overall} = 0.00$, while MatchDropPhase reaches only $S_{overall} = 0.02$ but introduces equivalence failures ($e = 0.09$). In contrast, MappingEvolve achieves $S_{overall} = 0.23$ with $e = 0.00$, an 11.5× improvement while maintaining perfect correctness. This validates that our architecture effectively coordinates cross-operator evolution through strategic planning and adaptive selection.

Model Generalization Across Different LLMs TABLE 1 evaluates MappingEvolve with three LLMs on ISCAS85. GPT-5 achieves $S_{overall} = 0.30$, followed by DeepSeek-V3 (0.23) and Qwen3-Max (0.18), all with $e = 0.00$. Even Qwen3-Max outperforms the OpenEvolve baseline (0.02) by 9×, confirming framework generalization across model families.

Comparison with State-of-the-Art Tools on EPFL Benchmarks TABLE 2 compares evolved mappers against ABC (`&nf`) [17] and mockturtle (`map`) [18] on EPFL benchmarks [19]. MappingEvolve achieves **10.04% area reduction** versus ABC and **7.93% area reduction** versus mockturtle, with a trade-off of 4.46%-6.50% delay increase. The $S_{overall}$ metric shows 46.6%-96.0% improvement over mockturtle, demonstrating that despite the delay trade-off, the evolved mappers discover beneficial area-delay trade-offs. This reflects our balanced area-delay weighting ($\alpha = 0.5$), enabling MappingEvolve to effectively navigate the area-delay Pareto frontier.

Analysis of Evolved Code Improvements To understand how MappingEvolve discovers effective optimizations, we analyze the GPT-5 evolved mapper at iteration 29 ($S_{overall} = 0.298$), which demonstrates coordinated improvements across three operators:

- **MatchPhase (Delay round):** Introduces area tolerance $t_{tol} = 0.25 \cdot \text{area(inv)}$, accepting delay-improving cuts only if: (1) area increase $\leq t_{tol}$, OR (2) delay gain $\geq 0.5 \cdot \text{delay(inv)}$.
- **MatchPhase (Area round):** Adds area slack $t_{slack} = 0.5 \cdot \text{area(inv)}$ for delay-improving cuts, enabling opportunistic timing recovery.
- **MatchPhaseExact:** Applies area gain threshold $t_{gain} = 0.5 \cdot \text{area(inv)}$, accepting area-improving cuts that worsen delay only if area gain $\geq t_{gain}$.
- **MatchDropPhase (Delay round):** Enforces zero area tolerance for phase consolidation, strictly preserving area during delay optimization.

5 Conclusion

We presented MappingEvolve, a framework that leverages LLMs to evolve technology mapping algorithms through hierarchical planning, controlled mutation, and rigorous validation. By abstracting the mapping process into three evolvable operators, we enable systematic algorithmic exploration while maintaining functional correctness. Experimental results demonstrate the framework's effectiveness: MappingEvolve achieves 11.5× improvement over direct operator evolution on ISCAS85 benchmarks with perfect logical equivalence. On EPFL benchmarks, the evolved mapper achieves 10.04% area reduction versus ABC, and $S_{overall}$ improvements of 46.6%–96.0% over mockturtle, effectively navigating the area-delay trade-off. We release all code, prompts, and evolution artifacts to facilitate future research. Promising directions include delay-oriented optimization and extending to other EDA tools.

Acknowledgment

The research work described in this paper was conducted in the JC STEM Lab of Intelligent Design Automation funded by The Hong Kong Jockey Club Charities Trust and was supported in part by the Research Grants Council of Hong Kong SAR (Grant No. CUHK14207523).

References

[1] A. Mishchenko, Sungmin Cho, Satrajit Chatterjee, and R. Brayton, "Combinational and sequential mapping with priority cuts," in *IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*. IEEE, 2007, pp. 354–361.

[2] W. L. Neto, M. T. Moreira, Y. Li, L. Amaru, C. Yu, and P. E. Gaillardon, "SLAP: A supervised learning approach for priority cuts technology mapping," in *ACM/IEEE Design Automation Conference (DAC)*, vol. 2021-December, 2021, pp. 859–864.

[3] C. R. Chigarapally, H. N. Bhakkad, A. B. Chowdhury, C. Karfa, and S. Bhattacharjee, "LEAP: Learning guided quality cut selection for faster technology mapping," in *IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*, 2025.

[4] H. Pan, C. Lan, Y. Liu, Z. Wang, L. Shang, X. Zeng, F. Yang, and K. Zhu, "Physically aware synthesis revisited: Guiding technology mapping with primitive logic gate placement," in *IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*, 2025.

[5] A. T. Calvino and G. De Micheli, "Technology mapping using multi-output library cells," in *IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*, 2023, pp. 1–9.

[6] L. Shang, S. Lu, S. Jung, Q. Liang, and C. Pan, "Novel fpga technology mapping for dual-output luts: Methodology and application," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems (TCAD)*, pp. 1–1, 2025.

[7] M. Liu, D. Robinson, Y. Li, J. Maximilian Kuehn, R. Liang, H. Ren, and C. Yu, "MapTune: Versatile ASIC technology mapping via reinforcement learning guided library tuning," *ACM Transactions on Design Automation of Electronic Systems (TODAES)*, 2025.

[8] R. Fu, C. Wang, B. Yu, and T.-Y. Ho, "TeMACLE: A technology mapping-aware area-efficient standard cell library extension framework," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems (TCAD)*, vol. 44, no. 8, pp. 3034–3045, 2025.

[9] OpenAI, "Introducing gpt-5," 2025. [Online]. Available: https://openai.com/index/introducing-gpt-5

[10] DeepSeek-AI, "DeepSeek-V3 technical report," 2024. [Online]. Available: https://arxiv.org/abs/2412.19437

[11] Q. Team, "Qwen3 technical report," 2025. [Online]. Available: https://arxiv.org/abs/2505.09388

[12] H. Zheng, H. Wu, and Z. He, "ChatLS: Multimodal retrieval-augmented generation and chain-of-thought for logic synthesis script customization," in *ACM/IEEE Design Automation Conference (DAC)*, 2025, pp. 1–7.

[13] S. Huang, J. Li, Z. Yu, J. Ye, J. Xu, N. Xu, and G. Dai, "LLSM: LLM-enhanced logic synthesis model with EDA-guided CoT prompting, hybrid embedding and AIG-tailored acceleration," in *IEEE/ACM Asia and South Pacific Design Automation Conference (ASPDAC)*, 2025, p. 974–980.

[14] A. Novikov, N. Vũ, M. Eisenberger, E. Dupont, P.-S. Huang, A. Z. Wagner, S. Shirobokov, B. Kozlovskii, F. J. R. Ruiz, A. Mehrabian, M. P. Kumar, A. See, S. Chaudhuri, G. Holland, A. Davies, S. Nowozin, P. Kohli, and M. Balog, "AlphaEvolve: A coding agent for scientific and algorithmic discovery," 2025. [Online]. Available: https://arxiv.org/abs/2506.13131

[15] A. Sharma, "OpenEvolve: an open-source evolutionary coding agent," 2025. [Online]. Available: https://github.com/algorithmicsuperintelligence/openevolve

[16] M. C. Hansen, H. Yalcin, and J. P. Hayes, "Unveiling the ISCAS-85 benchmarks: A case study in reverse engineering," *IEEE Design & Test*, vol. 16, no. 3, pp. 72–80, 1999.

[17] Berkeley Logic Synthesis and Verification Group, "ABC: A system for sequential logic synthesis and verification," http://www.eecs.berkeley.edu/ alanmi/abc/, Version 1.01.

[18] EPFL Integrated Systems Laboratory, "mockturtle: A C++ logic network library," https://github.com/lsils/mockturtle, Accessed on November 2025.

[19] L. Amarù, P.-E. Gaillardon, and G. De Micheli, "The EPFL combinational benchmark suite," in *IEEE/ACM International Workshop on Logic Synthesis*, 2015.

[20] L. Hellerman, "A catalog of three-variable or-invert and and-invert logical circuits," *IEEE Transactions on Electronic Computers*, vol. EC-12, no. 3, pp. 198–223, 1963.

[21] V. Vashishtha, M. Vangala, and L. T. Clark, "ASAP7 predictive design kit development and cell design technology co-optimization: Invited paper," in *IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*, 2017, pp. 992–998.