Invited: Agentic AI for Physical Design R&D: Status and Prospects

Amur Ghose
University of California, San Diego
Department of CSE
aghose@ucsd.edu

Andrew B. Kahng
University of California, San Diego
Departments of CSE and ECE
abk@ucsd.edu

Sayak Kundu
University of California, San Diego
Department of ECE
sakundu@ucsd.edu

Bodhisatta Pramanik
University of California, San Diego
Department of ECE
bopramanik@ucsd.edu

Abstract

Recent advances in large language models (LLMs) and tool-using autonomous agents present new opportunities for accelerating research and development in physical design. Unlike earlier uses of machine learning that focused narrowly on prediction or optimization subroutines, agentic AI systems can comprehend user specifications, modify code, run EDA tools, analyze results, perform multi-step reasoning, and iteratively refine design heuristics. This paper surveys the emerging landscape of agentic AI for physical design R&D, with emphasis on (i) tool-integrated agents for algorithm evolution, debugging, and workflow automation, (ii) autonomous exploration of heuristic spaces in placement, routing, and partitioning, and (iii) interfaces between agents and traditional EDA frameworks. We analyze recent experience with multi-agent workflows and benchmark evaluation, highlighting current capabilities, limitations, and research frontiers. We conclude by articulating the long-term prospects of agentic AI as a catalyst for accelerated innovation in physical design, including autonomous algorithm discovery, continuous tool improvement, and closed-loop learning from large design corpora.

CCS Concepts

• **Hardware → Physical design (EDA)**; **Software tools for EDA**; • **Computing methodologies → Multi-agent systems**; **Intelligent agents**.

Keywords

VLSI physical design, agentic EDA, algorithm evolution, EDA tools

ACM Reference Format:
Amur Ghose, Andrew B. Kahng, Sayak Kundu, and Bodhisatta Pramanik. 2026. Invited: Agentic AI for Physical Design R&D: Status and Prospects. In *Proceedings of the 2026 International Symposium on Physical Design (ISPD '26), March 15–18, 2026, Bonn, Germany.* ACM, New York, NY, USA, 9 pages. https://doi.org/10.1145/3764386.3779612

1 Introduction

Integrated circuit (IC) designers face an ever-increasing challenge to deliver consistent capable design automation tools. At its core, IC design is a large-scale optimization problem in which multiple competing objectives — timing, power, area, and routability — must be simultaneously satisfied. This challenge is further compounded by the growing need for node-specific design-technology co-optimization (DTCO), which leads to both cost and risk increases. Commercial EDA tools, while highly optimized and robust, have to hedge their bets across a wide variety of possible tapeouts. At advanced nodes below 7 nm, DTCO contributes more than 25% of area improvement, with projections exceeding 50% at 3 nm and beyond [47]. Consolidation and oligopoly stymies the necessary innovation required to disrupt these challenges. Closed intellectual property models limit the access to data, impede reproducibility, and constrain the development of OSS benchmarks.

Large Language models (LLMs) and tool-integrated autonomous agents present new opportunities for accelerating research and development in physical design. Unlike earlier applications of machine learning in EDA — typically focused on isolated prediction tasks or parameter tuning — *agentic AI* systems can now interpret user specifications, modify source code, execute EDA tools, analyze outcomes, perform multi-step reasoning, and iteratively refine design heuristics. This capability represents a qualitative shift from ML as a supporting subroutine to ML as an active engineering counterpart. We articulate a vision of *agentic autonomous evolution of EDA*, in which AI agents function as R&D software engineers for physical design tools. This agentic wave is likely to inundate first the open-source EDA ecosystems, such as OpenROAD [84], as opposed to siloed, proprietary, closed-loop incumbents. This paper assesses the emerging landscape of agentic AI for physical design R&D. We make the following contributions:

- We provide a perspective on the evolution of data-driven techniques in EDA from task-specific machine learning models to large language models and tool-integrated agentic systems. This clarifies the limits of prior AI-for-EDA approaches and motivates the transition to agentic AI for physical design R&D (Section 2).
- We introduce a taxonomy of agentic AI for physical design, distinguishing *flow-level agents* that optimize tool orchestration and parameters from *code-level agents* that directly modify EDA algorithms and implementation logic (Section 3).
- We present concrete case studies demonstrating agentic AI in action, spanning flow tuning, detailed placement evolution, functional simulation acceleration, and hypergraph partitioning, all within the OpenROAD ecosystem (Section 4).
- We present a research vision for agentic physical design, identifying challenges and paths forward in verification, safety, benchmarking, and multi-objective optimization, and outlining the

long-term opportunity for a modular, multi-agent OpenROAD framework in which specialized agents collaboratively evolve physical design tools and workflows (Section 5).

The remainder of this paper is organized as follows. Section 2 reviews background on ML in EDA, LLM-based agents, and prior work in automated heuristic discovery. Section 3 defines agentic AI for physical design and outlines core capabilities of flow-level and code-level agents. Section 4 presents some case studies on OpenROAD, Section 5 discusses our vision and near-term prospects, and Section 6 concludes our paper.

2 Background and Related Trends

In recent years, artificial intelligence for chip design — often referred to as AI/ML for EDA or AI-assisted EDA [25, 50] — has emerged as a promising paradigm due to its ability to leverage knowledge from prior circuit design data. A growing number of AI-driven techniques have also been adopted in commercial EDA tools [10, 57]. These approaches typically train machine learning models to provide early-stage predictions or optimizations, thereby bypassing computationally expensive downstream design and simulation steps. By learning from historical design solutions, ML models can evaluate circuit quality at early design stages and guide subsequent optimization decisions. Most existing AI-for-EDA techniques are deployed as specialized prediction or optimization modules within fixed design flows [16]. They thus remain inherently narrow in scope and incremental in accelerating design time, not revolutionary.

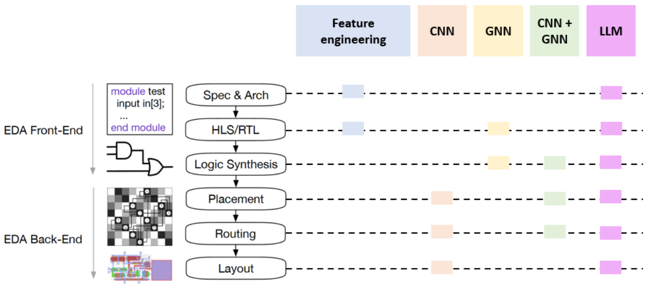

Fig. 1: ML methodologies in EDA stages, adapted from [76].

Recent advances in large language models (LLMs) [1] introduce a distinct capability. LLMs are trained to reason over heterogeneous inputs — including source code, tool logs, configuration files, and natural language specifications — and to interact with external tools in a closed-loop manner [72]. This shift enables a new class of AI systems that can directly augment, and then eventually complement, the EDA engineering workforce. Figure 1, adapted from [76] captures typical ML methodologies per EDA stage.

2.1 Evolution of ML in EDA

Existing AI-for-EDA methods are largely tailored to specific tasks, most commonly the early prediction of design quality metrics such as timing [17, 68], area [18, 19, 55], power [78, 79], IR drop [15, 69], and routability [26, 70]. In addition to prediction, a range of optimization-oriented tasks-including design flow tuning [32, 71], design space exploration [4, 53], and direct PPA optimization [39, 40] — have also been explored using machine learning. Across these applications, ML models are typically used to estimate circuit quality metrics and provide feedback to downstream optimization procedures, rather than to modify the underlying EDA algorithms or tool flows themselves. This paradigm, in which ML serves as a task-specific subroutine within a fixed design flow, is illustrated on the left side of Figure 2. All of these classical supervised and reinforcement learning approaches typically require carefully curated training data, hand-designed features, and problem formulations that are tightly coupled to a specific design stage, technology node, or design style. Today, changing any of these exposes weaknesses in generalization; this motivates us to find more flexible and transferable learning paradigms. Existing approaches are also created to work with largely static and closed EDA toolchains; this environment may shift with the advent of new system integrations and device/fabrication technologies, e.g., 3D heterogeneous integration, integrated photonics, etc.

LLMs address these limitations by learning general-purpose representations over code, text, and execution traces [1]. Rather than learning a single predictive mapping, LLMs can interpret design specifications, inspect and modify EDA source code, invoke tools, and analyze results in a closed-loop fashion [54, 72]. As depicted on the right side of Figure 2, this capability enables a transition from learning *within* fixed design flows to learning *about and across* EDA tools themselves, enabling agentic digital IC engineering.

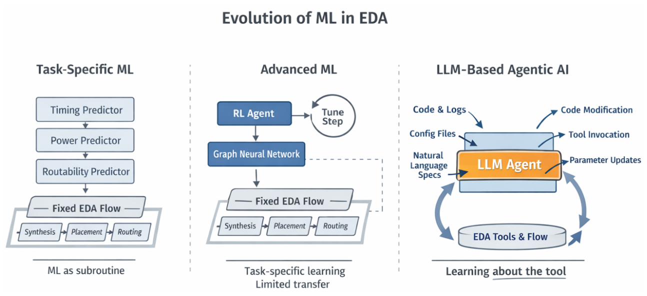

Fig. 2: Evolution of ML in EDA.

2.2 LLMs and Agents

Large language models (LLMs) are foundation models with general-purpose representations of language [1, 6]. LLMs develop the ability to generate coherent, context-aware text and to perform a wide range of tasks without explicit task-specific training [7]. Modern LLMs comprise billions or trillions of parameters, with scale playing a central role in determining model capability [30]. The key architectural innovation enabling this progress is the Transformer model [61]. Empirical studies have shown that language model performance, measured by log-likelihood loss, follows consistent power-law scaling behavior with respect to model size, training data, and computational budget [30]. As models are scaled along these dimensions, they exhibit emergent abilities — capabilities that are not observed in smaller models but arise unexpectedly at larger scales [63]. In Figure 4, this progression corresponds to the leftmost block, where LLMs reason over text and code.

Agents. An *agent* refers to an LLM-based system that couples reasoning with action through interaction with external tools and environments. Unlike a standalone language model that passively

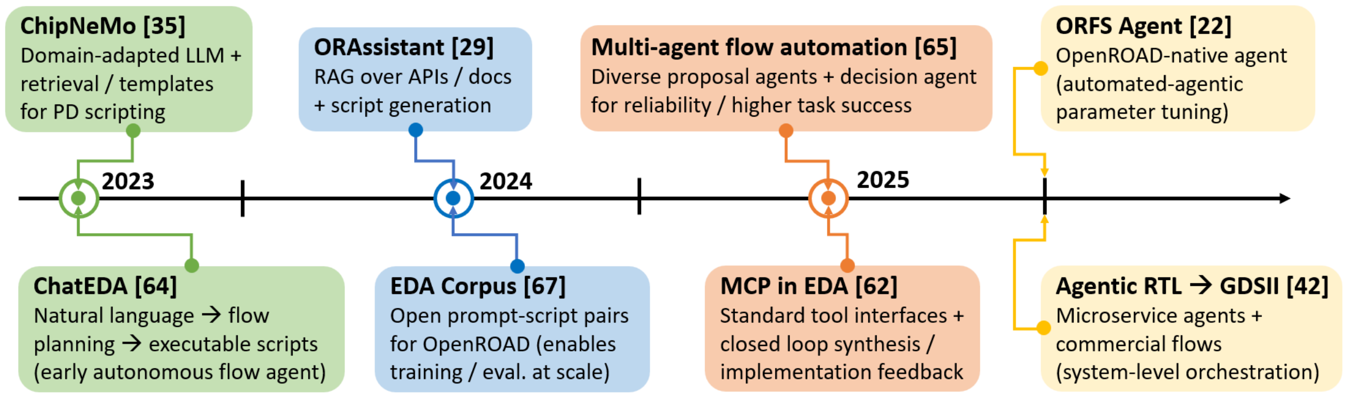
ChipNeMo [35]
Domain-adapted LLM + retrieval / templates for PD scripting
ORAssistant [29]
RAG over APIs / docs + script generation
Multi-agent flow automation [65]
Diverse proposal agents + decision agent for reliability / higher task success
ORFS Agent [22]
OpenROAD-native agent (automated-agentic parameter tuning)
2023
2024
2025
ChatEDA [64]
Natural language → flow planning → executable scripts (early autonomous flow agent)
EDA Corpus [67]
Open prompt-script pairs for OpenROAD (enables training / eval. at scale)
MCP in EDA [62]
Standard tool interfaces + closed loop synthesis / implementation feedback
Agentic RTL → GDSII [42]
Microservice agents + commercial flows (system-level orchestration)

Fig. 3: Timeline of representative LLM-based work for physical design, illustrating the evolution from natural-language-driven flow automation, to open-source grounding via datasets and retrieval, and toward tool-integrated, multi-agent EDA ecosystems.

generates text, an agent maintains state, plans sequences of actions, invokes tools, observes outcomes, and adapts its behavior in a closed-loop manner [54, 73]. Effective EDA problem solving requires not only reasoning about design intent but also executing tools, modifying configurations or source code, and evaluating downstream physical design metrics; fertile ground to critique pure reason. This is illustrated in Figure 4.

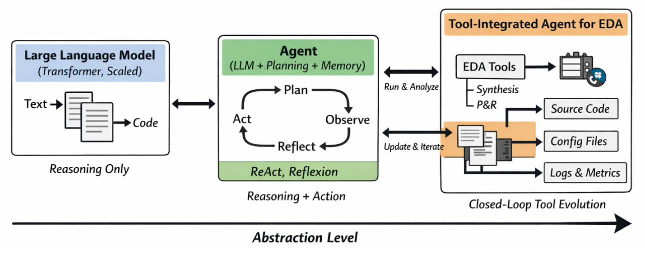

Fig. 4: LLMs, agents and EDA.

Modern agentic infrastructure supports tight control loops, type checking, tool calling [54] dynamic timeouts, and so forth, wrapped around a single base LLM that may or may not be fine-tuned for the purpose. Experiments with Kimi-class models demonstrate that such fine-tuning helps greatly; however, most base LLMs are also usable as core drivers for agentic systems. Prompt optimization, chain of thought, etc. largely form the proto-layer of how the agent pipelines, manages, and compacts context, but with much more abstraction and optimization — e.g. frameworks like DSPy obviate the need for much of hand-tuned prompting. When combined with frontier LLMs from OpenAI [45] and Google [20], these architectures enable tool-using agents that can read documentation, write and execute code, interpret results, and iteratively refine solutions — capabilities that align naturally with the workflows of EDA research and development [42]. Within EDA, agentic guardrails generally take the form of RTL-level checks [37, 58] atop domain-tuned models [35] or LLM-interfacing [58] alongside a "bonsai" ecosystem of benchmarks [36, 41]. A clear-eyed view of the EDA ecosystem would find it severely lacking in the adaptation required. Efforts such as MCP4EDA [62] to port general context pipelining abstractions such as MCP [83] arguably face obsolescence from generic LLM advances, and do not sufficiently emphasize the unique types of data (VCD, HGR, etc.) that make EDA special. And, while coding agents or other agents for generic data might use operations such as "sed" to manipulate their environments, what such primitives might look like for EDA and IC design environments is still unclear.

2.3 Automated Heuristic Discovery

The challenge of automated heuristic discovery predates the LLM era by several decades. [51] formalized the task of mapping problem instances to suitable algorithms based on instance features. Evolutionary approaches such as genetic programming [23] and genetic improvement [31] applied search directly to program representations, while automated algorithm configuration methods tuned solver hyperparameters using techniques such as iterative racing and Bayesian optimization [27, 38]. These classical approaches operate over restricted, human-defined search spaces and lack semantic understanding of algorithmic intent or source code, and cannot reason about or modify complex optimization software, particularly in domains such as physical design where algorithms are tightly coupled to large codebases and evolving objectives.

Recent work revisits automated heuristic discovery using large language models. Surveys [34] identify LLM-based approaches that operate directly at the code level, enabling semantic mutation, synthesis, and fitness-guided refinement. Extensions to multi-objective optimization [74] and applications to global placement [75] exist, but are largely still human-engineered and unscalable.

2.4 Autonomous Code Evolution

Code itself can serve as an optimization substrate, enabling autonomous improvement of existing algorithms. Early evidence of this capability emerged from AlphaDev [43], which used reinforcement learning to discover faster low-level sorting algorithms, yielding improvements that had eluded manual optimization for decades. FunSearch [52] further generalized this idea by combining LLM-guided proposal generation with evolutionary search, leading to the discovery of novel solutions to long-standing mathematical problems such as the cap set problem. More general frameworks have emerged that treat algorithm and heuristic design as an explicit search process. The Evolution of Heuristics (EoH) framework [33] formulates heuristic discovery as population-based evolution, where LLMs propose mutations and crossovers over candidate solutions guided by task-specific fitness signals. This approach has achieved state-of-the-art performance across a range of classical combinatorial optimization problems, including bin packing and the traveling salesman problem. AlphaEvolve [44] extends this paradigm to scientific and algorithmic discovery. Open-source efforts such as OpenEvolve [3] make these ideas accessible and extensible by combining quality-diversity evolution (MAP-Elites) with ensembles of LLMs.

In the context of hardware design, the advent of thinking models and autonomous code evolution is particularly significant because most hardware design tasks implicitly solve a variety of optimization problems that are too hard to solve directly. Today's tools and methodologies may still engrain formulations, heuristics and "bags of tricks" whose antecedents are now lost or outdated. As one example: Have the EDA field's early (more formal) emphases on near-planarity, k-coplanarity, k-tree topologies, etc. faded from today's hypergraph-level optimizations? Or, have advances in optimization theory from the early 2000s onward — e.g., hashing and RIP matrices — been slow to permeate the field when so much energy goes toward advancing heuristic frameworks that were first pioneered in the 1980s and 1990s?

3 Agentic AI for Physical Design R&D

This section formalizes the concept of agentic AI in the context of physical design and delineates the capabilities required for such systems to operate effectively in EDA research and development. We first introduce a taxonomy of agentic systems, distinguishing between flow-level and code-level agents, and then identify the core capabilities that enable autonomous reasoning, execution, and refinement. Finally, we outline a concrete workflow that illustrates how agentic systems can be integrated into practical physical design R&D, from code comprehension to verified tool improvements.

Taxonomies. We define an *agentic AI system* for physical design as an autonomous software agent that can: (i) interpret natural-language specifications of design objectives, (ii) invoke EDA tools and analyze their outputs, (iii) modify tool configurations or source code, and (iv) iteratively refine solutions based on measured QoR. This definition distinguishes agentic systems from traditional automation scripts, which execute fixed procedures without adaptation, as well as from ML-based prediction models, which provide estimates without direct control over tools or algorithms. Within this definition, we identify two levels of agentic capability (Figure 5) that naturally arise in physical design R&D, reflecting the layered structure of EDA tools themselves.

3.1 Flow-Level Agents

Flow script generation is a critical enabler of automation in physical design, as scripts determine how EDA tools are orchestrated across synthesis, placement, routing, and verification stages. These scripts encode procedural knowledge about tool invocation order, parameter dependencies, and recovery from intermediate failures. Traditionally, flow scripts are authored and maintained manually by domain experts, requiring deep familiarity with both the toolchain and the target design. As flows grow more complex and design-specific, this manual process becomes increasingly brittle and time-consuming. Flow-level agents operate at the level of tool invocation and configuration. They treat EDA tools as black boxes and optimize the sequence, scheduling, and parameterization of tool runs by adjusting parameters such as target clock period, core utilization, placement density, routing effort, etc. Recent advances in LLMs enable designers to express high-level objectives and constraints in natural language, allowing LLM-based systems to synthesize executable scripts that orchestrate EDA tools accordingly [42].

ChatEDA. Early systems such as ChatEDA [64] demonstrate this concept by treating flow automation as an agentic task: user requirements are decomposed into sub-tasks, translated into executable scripts, and executed through EDA tool APIs. Gains come through richer training corpora, instruction tuning, chain-of-thought prompting, and multi-agent collaboration.

ChipNeMo. Domain-adapted language models further improve script generation by embedding tool-specific knowledge directly into the model. ChipNeMo [35], for example, adapts LLMs to chip design through continued pretraining and instruction alignment on expert-authored scripts. By combining in-context learning with retrieval of relevant templates and documentation, such systems can generate more accurate and maintainable Tcl or Python scripts for timing analysis and physical design tasks.

ORAssistant. Complementary efforts emphasize open-source accessibility and tool-centric reasoning. ORAssistant [29] focuses on script generation and question answering for OpenROAD [84] by coupling LLMs with retrieval over API definitions, code templates, and documentation. Retrieval-augmented training and inference reduce hallucination and improve reliability.

ORFS-Agent. ORFS-Agent [22] advances agentic automation beyond script generation by introducing an LLM-driven iterative optimization agent for parameter tuning in the OpenROAD-flow-script (ORFS) [85]. The agent explores parameter configurations and improves over standard Bayesian optimization in resource efficiency and design quality. ORFS-Agent relies on an LLM capable of (i) reading and modifying files and logs, (ii) invoking external tools via function calling (e.g., Python functions), and (iii) proposing parameter values to optimize objectives under constraints.

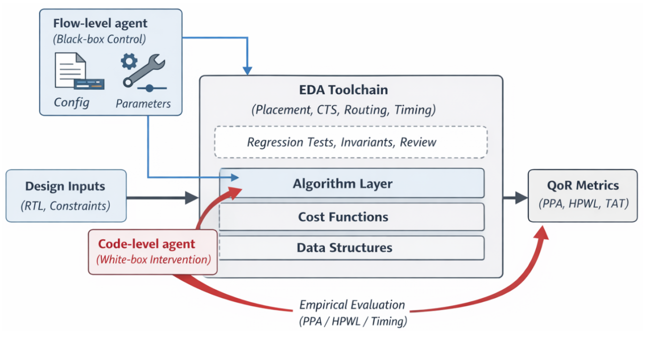

Fig. 5: Flow-level and code-level agents orchestration.

3.2 Code-Level Agents

Code-level agents operate directly on the source code of EDA tools and design artifacts, enabling autonomous modification of algorithms, heuristics, and implementation logic — in sum, everything present in the codebase. Naturally, this approach finds more purchase in the field of OSS EDA than in black-box, siloed settings.

Intervening inside the toolchain. From a functional perspective, code-level agents subsume a broad class of code generation and transformation tasks that are central to modern EDA workflows. These include HDL and HLS code generation [19, 48], flow and verification script synthesis [24, 64, 65], testbench construction [5, 59, 60], and direct modification of EDA tool source code [75, 77]. Recent advances in large language models [11, 46, 81] demonstrate that such agents can generate syntactically and semantically correct code across multiple programming languages.

Why PD enables algorithmic evolution. EDA provides a uniquely suitable environment for code-level agentic evolution. Core EDA algorithms-placement cost functions, partitioning gain models, congestion estimators, and timing-driven refinement — are heuristic by nature, tightly coupled to tool architecture, and evaluated primarily through empirical QoR (PPA, HPWL, TAT, etc.) metrics. This fits with the autonomous code evolution paradigm introduced in Section 2.4, where algorithmic logic is iteratively proposed, evaluated, and refined empirically rather than via theory.

Within the above setting, code-level agents specialize general-purpose algorithm evolution into domain-aware transformations. Agents reason over existing implementations, developer intent, and historical design tradeoffs to propose targeted modifications — such as introducing new objective terms, restructuring control logic, or adapting search strategies to emerging design regimes. These changes operate directly on the semantics of the core algorithms and can propose new algorithms altogether, totaling changes in the order of thousands of lines of code. Guardrails are essential: such agents must operate under stricter verification constraints, including regression testing, invariant checking, and human-in-the-loop review — all of which can be integrated within existing CI/CD workflows with a little finesse.

4 Agentic OpenROAD: Case Studies

We present case studies demonstrating agentic AI for physical design R&D, spanning flow-level tuning and code-level evolution across multiple EDA tools. All experiments use OpenROAD [84] [2], a permissively open-sourced RTL-to-GDS tool. Our goal is not to present isolated "AI-for-EDA" demonstrations, but to establish a single methodological claim: *agentic AI can participate in the physical-design R&D loop end-to-end*, i.e., (i) identify bottlenecks, (ii) propose interventions at the appropriate abstraction level (flow, algorithm, or implementation), and (iii) validate improvements through tool execution and measurable QoR/runtime outcomes. Accordingly, we intentionally cover multiple layers of the stack: *flow configuration* (high-dimensional tuning), *algorithmic structure* (operator/heuristic design), and *classical optimization components* (solver stability and solution quality). This breadth is essential for generality: agentic methods are not limited to parameter tuning or scripting, but can drive sustained tool evolution when anchored by reproducible evaluation.

4.1 Flow-Level Agent: End-to-End RTL-to-GDS Optimization

This case study demonstrates that agentic AI can replace ad hoc, expert-driven flow tuning with a systematic, semantics-aware optimization process that scales to the high-dimensional configuration spaces of industrial RTL-to-GDS flows while remaining grounded in physical design metrics. This subsection draws from [22] — ORFS-agent is an LLM-based autotuner integrated with OpenROAD-flow-scripts (ORFS) [85], the standard flow infrastructure for OpenROAD. The agent optimizes design configurations through parallel trials guided by metric-aware refinement, enabling continuous and data-driven improvement of end-to-end physical design flows.

A key property of ORFS-agent is its *model-agnostic* design. The agent can operate with any base LLM that supports tool use or function-calling semantics, without reliance on fine-tuning. This decouples algorithmic progress from a specific foundation model and allows performance to naturally improve as stronger models become available, without retraining or redesign [22]. As a result, ORFS-agent represents a reusable optimization methodology rather than a model-specific solution.

Agent design. RTL-to-GDS flows expose hundreds to thousands of tunable parameters, creating a search space that is both high-dimensional and highly structured. Traditional Bayesian optimization (BO) methods struggle in this regime due to weak domain priors, poor scalability, and limited ability to exploit semantic relationships among parameters. ORFS-agent addresses these challenges through a semantics-aware search strategy that explicitly reasons about the physical meaning and interactions of flow parameters. The agent operates through an iterative tool-using loop with three core operations:

- *INSPECT*: Performs PCA to identify influential parameter loadings, detects outliers, summarizes correlations, and extracts health indicators from logs (e.g., timeouts, DRC violations, CTS warnings).
- *MODEL*: Fits lightweight surrogate models (linear, ridge, lasso, isotonic regression, Gaussian processes) to rank candidate configurations and bracket promising operating regimes.
- *AGGLOMERATE*: Generates a pool of feasible candidates, then applies diversity-aware down-selection (DPP-like or k-medoids) to select a batch of 25 configurations for parallel evaluation.

Table 1: ORFS-agent vs. OR-AutoTuner (normalized to OR-AutoTuner with 4 params and 375 iterations; lower is better).

Setting	Method	Params	Iters	WL↓	ECP↓
4-param	OR-AutoTuner	4	375	1.00	1.00
	ORFS-agent (w/ tools)	4	375	0.97	0.99
	ORFS-agent (no tools)	4	375	1.03	0.99
High-dim	OR-AutoTuner	12	1000	0.92	0.94
	ORFS-agent	12	600	0.94	0.96

The agent supports single-objective optimization (e.g., minimizing wirelength or effective clock period), multi-objective optimization via weighted combinations of normalized metrics, and natural-language constraints such as "Minimize ECP with area, count, power, and PDP degradation ≤ 2%". Experiments are conducted using SKY130HD and ASAP7 enablements on the IBEX, AES, and JPEG designs. The ORFS environment is containerized with pinned commits to ensure reproducibility, and each iteration evaluates 25 parallel trials (12 for JPEG due to longer runtime).

Outcomes and learnings. Compared to OR-AutoTuner [28], a Bayesian optimization baseline, ORFS-agent requires approximately

40% fewer iterations to reach iso-QoR, or achieves approximately 13% improvement in wirelength or effective clock period for single-objective optimization. With 12 tunable parameters and 600 iterations, ORFS-agent matches or exceeds OR-AutoTuner operating on only 4 parameters over 1000 iterations (Table 1). Beyond raw performance, this study highlights three broader insights: (i) semantics-aware reasoning is essential for efficient exploration in high-dimensional PD flows, (ii) tight integration with tool logs and health checks is critical for reliable agent behavior, and (iii) model-agnostic agent designs future-proof optimization workflows by allowing gains to track improvements in foundation models rather than depend on retraining.

4.2 Code-Level Agent: Detailed Placement Algorithm Evolution

OpenROAD's detailed placement (DPL) engine applies a fixed, hand-designed sequence of optimization operators, including maximum independent set matching, global swap, vertical swap, reordering, and zero-temperature simulated annealing. While effective, this sequence reflects accumulated historical design choices rather than systematic optimization. This case study investigates whether agentic AI can participate directly in algorithmic design by discovering improved operator schedules or entirely new placement heuristics within a production-quality detailed placer.

Agent design. We treat two components of DPL as evolvable programs: (i) the sequence and parameterization of move operators, and (ii) the internal reordering algorithm. An LLM-based coding agent operates within an evolutionary loop, proposing mutations and recombinations of existing operator sequences as well as novel reordering strategies. Candidate implementations are instantiated directly in the DPL C++ source code and evaluated through full OpenROAD runs. Selection is driven by a task-specific fitness function that balances placement quality (HPWL) and runtime, grounding evolution in measured physical design outcomes over proxies.

For sequence evolution, the agent explores alternative operator orderings and repetitions, yielding a large combinatorial search space that is difficult to navigate manually. For algorithm evolution, the agent is permitted to synthesize new reordering logic while preserving DPL's external interfaces and correctness constraints. All candidates are evaluated under identical flow conditions to ensure fair comparison. Experiments are conducted on ASAP7 designs (AES, JPEG, IBEX, AES (multi-height) and JPEG (multi-height)).

Table 2: Evolved detailed placement optimization results on ASAP7 designs. HPWL in μm.

Design	Default		Move Seq. Evo		Reordering Evo	
	HPWL	Time (s)	HPWL	Time (s)	HPWL	Time (s)
AES	32412.8	4.3	**31124.6**	42.2	32235.5	**4.0**
JPEG	57018.2	12.5	**55795.4**	171.5	56831.8	**10.4**
IBEX	43229.3	4.3	**42311.2**	22.0	43209.6	**3.6**
AES (MH)	43628.6	**4.8**	**43347.3**	17.2	43524.1	6.5
JPEG (MH)	93642.8	42.0	**92445.2**	100.7	93514.6	**35.2**

Outcomes and learnings. Move operators may repeat with different hyperparameters, yielding a large combinatorial search space that is difficult to explore manually. Using an EoH-style framework [33], the agent proposes alternative operator orderings and evaluates them based on HPWL. Early results demonstrate up to 3.8% HPWL reduction (2.7% geometric mean) relative to the default OpenROAD sequence, at the cost of increased runtime due to longer operator chains. See Table 2 for move sequence evolution results on some ASAP7 designs. Beyond sequence optimization, the agent can synthesize new algorithms. In one experiment, the agent autonomously implemented a particle swarm optimization (PSO)-based reordering heuristic, generating over 600 lines of new C++ code. The algorithm models cells as particles and updates their positions using PSO dynamics, followed by sorting based on refined positions to derive new cell orders. This PSO-based reordering improves HPWL by up to 0.5% while reducing runtime by up to 17% compared to the baseline. Table 2 reports reordering evolution results. Together, these results highlight two key insights: (i) operator sequencing in detailed placement remains significantly under-optimized and amenable to systematic discovery, and (ii) agentic code evolution can produce non-trivial, human-competitive heuristics when tightly constrained by correctness, interfaces, and full-flow evaluation.

4.3 Agentic Optimization of Functional Simulation

Fast functional simulation is critical for physical design R&D, as it directly impacts design-space exploration, regression testing, and iteration latency. Verilator converts RTL to C++ and compiles it into fast executables, but its performance depends heavily on the choice of RTL-level and compiler-level optimization flags. While GCC-level optimizations have been extensively studied, RTL-level flag selection exposes a much larger and less systematically explored design space, with the potential to simultaneously improve compilation time and simulation throughput.

Agent design. We frame flag selection as a performance optimization problem over tuples of (flag set, runtime). An LLM-based agent proposes improved flag combinations using guided evolutionary prompting, without full fine-tuning. This allows the agent to generalize across designs while remaining agnostic to a specific RTL structure. The agent operates directly on Verilator's configuration space, enabling it to reason about interactions between RTL-level transformations and downstream compilation behavior.

Experiments are conducted across a diverse set of RTL benchmarks, including cryptographic cores, filters, arithmetic units, and RISC-V processors. Performance is measured in terms of end-to-end simulation speed (cycles per second), capturing the combined effect of compilation and execution. All configurations are evaluated under identical toolchain settings to ensure fair comparison.

Outcomes and learnings. LLM-evolved flag configurations achieve simulation speedups ranging from 5.7% (AES) to 48.5% (FIR filter) relative to default Verilator settings (Table 3). Designs with regular structure, such as filters and cryptographic cores, exhibit the largest gains, suggesting that the agent learns to exploit architectural regularities that interact favorably with RTL-level optimizations. These results indicate that agentic optimization can uncover non-obvious performance improvements in mature toolchains, even in domains traditionally optimized through manual expertise.

Beyond Verilator. We extend this approach to Arcilator [80], an LLVM-oriented Verilator fork that achieves 2–4× higher cycle-level simulation speed for common RISC-V cores (e.g., `Rocket`, `BOOM`)

Table 3: Verilator simulation speedups; LLM-evolved flags.

Benchmark	Speedup (%)
FIR filter	48.5
CRC32	41.7
MAC	40.0
FIFO	33.3
PicoRV32	33.3
SHA256	29.8
Matrix mul	20.5
SERV	11.3
ALU	7.4
WBUART32	5.9
AES	5.7

by operating directly at the CIRCT/MLIR IR level. Despite its performance advantages, Arcilator has seen limited adoption due to gaps in parsing, elaboration, and lowering support for common verification environments such as UVM. Via agent-assisted development, we achieve full parsing and elaboration for `circt-verilog` on UVM testbenches (improving baseline success rates from ≈ 2% to 100%), enable VCD-level outputs with and without DUT semantics, and retain Arcilator's inherent speedups relative to Verilator [86].

4.4 Agentic Boosting of Classical Optimizers

Hypergraph partitioning remains a foundational optimization problem in physical design, with decades of algorithmic research [87] culminating in highly optimized tools such as TritonPart [9]. Despite these advances, modern partitioners increasingly rely on complex solver components (e.g., ILP-based refinement) whose behavior can exhibit high variance across instances and solver backends. This case study examines whether agentic AI can *strengthen and stabilize* classical optimizers by improving solver robustness and solution consistency rather than replacing established algorithms.

Agent design. TritonPart is a state-of-the-art hypergraph partitioner that outperforms hMETIS and KaHyPar, and includes an ILP-based boundary optimization step via commercial (CPLEX) or open-source (OR-Tools) solvers. We deploy an agent that targets variance reduction in this ILP step by proposing localized code-level modifications. The agent operates within tight constraints: it preserves TritonPart's core algorithmic structure and interfaces, while modifying solver formulations, heuristics, and integration logic to improve consistency across runs and solvers. We evaluate TritonPart and an agent-enhanced **FastPart** [82] across a diverse set of hypergraphs (collected from real netlists) spanning small to extremely large problem sizes, ranging from tens of thousands to over ten million vertices. Experiments are conducted for $K \in \{2, 3, 4\}$ partitions under the same imbalance factor (2%). For each configuration, we report cutsize as the primary quality metric along with wall-clock runtime to reflect practical solver efficiency. FastPart is evaluated under a fixed time budget per run, while TritonPart is run using its standard FM-based refinement pipeline.

Outcomes and learnings. Across the evaluated benchmarks [12] [13] [14], FastPart consistently achieves cutsizes that match or improve upon TritonPart while offering substantial runtime reductions (Table 4 presents detailed results). On large-scale instances (e.g., MPG and MPC), FastPart attains comparable or lower cutsizes with one to two orders of magnitude lower runtime, demonstrating

Table 4: Comparing TritonPart/FastPart cuts and runtimes.

Design	#V	#E	K	TritonPart		FastPart	
				Cut	Time (s)	Cut	Time (s)
			2	**1088**	67.8	1091	**15**
JPEG	51352	54784	3	**1214**	**125.8**	1216	300
			4	1308	314.2	**1283**	**300**
			2	912	**100.8**	**748**	240
ARIANE133	89088	95321	3	1200	**152.9**	**1089**	180
			4	1603	310.2	**1331**	**240**
			2	660	80.9	**630**	**15**
ARIANE136	91877	95575	3	1032	115.0	**979**	**15**
			4	1249	219.4	**1111**	**120**
			2	1953	277.2	**1924**	**15**
BSG	586364	700967	3	2879	**198.1**	**2684**	300
			4	3363	**279.7**	**3081**	300
			2	**323**	83.7	332	**30**
NVDLA	152764	164999	3	383	111.6	**378**	**30**
			4	648	**161.2**	**639**	180
			2	0	117.5	0	**10**
ARIANE_X4	331816	337036	3	**896**	162.8	902	**15**
			4	0	166.4	0	**10**
			2	2229	871.5	**2157**	**30**
MPG	2460278	2488257	3	**3338**	923.5	3369	**240**
			4	3784	996.3	**3517**	**180**
			2	707	3876.3	**665**	**120**
MPC	10486897	10726018	3	**5193**	4193.1	5390	**300**
			4	1060	4074.3	**943**	**30**

significantly improved scalability. On medium-sized designs (e.g., ARIANE133, ARIANE136, BSG), FastPart often produces strictly better cutsizes, particularly for higher partition counts, while maintaining runtimes capped at a few minutes. These results indicate that agent-driven modifications can stabilize and accelerate ILP-based refinement, yielding competitive or best-known solutions.

5 Vision and Near-Term Prospects

We envision a future in which physical design tools are no longer static software artifacts, but continuously evolving systems shaped by autonomous agents operating alongside human developers (Figure 6). In this paradigm, agentic AI does not merely accelerate isolated design tasks; instead, it participates directly in EDA research and development, proposing, implementing, and evaluating algorithmic improvements over time. This shift represents a fundamental change in how physical design tools are created, maintained, and advanced.

From single agents to multi-agent tool ecosystems. While early successes in agentic AI often focus on single, monolithic agents, we believe the long-term trajectory lies in *multi-agent ecosystems.* Different aspects of physical design — placement, routing, timing analysis, partitioning, and flow orchestration — naturally decompose into specialized roles. A multi-agent framework allows agents with complementary expertise to collaborate, critique one another, and jointly refine solutions. In such a system, flow-level agents may explore configuration spaces and identify performance bottlenecks, while code-level agents focus on targeted algorithmic modifications within specific tool components. Meta-agents may monitor progress, detect stagnation, and allocate computational effort across agents and objectives. This cooperative structure mirrors how human EDA teams operate today, but with the ability to scale exploration and iteration far beyond human limits.

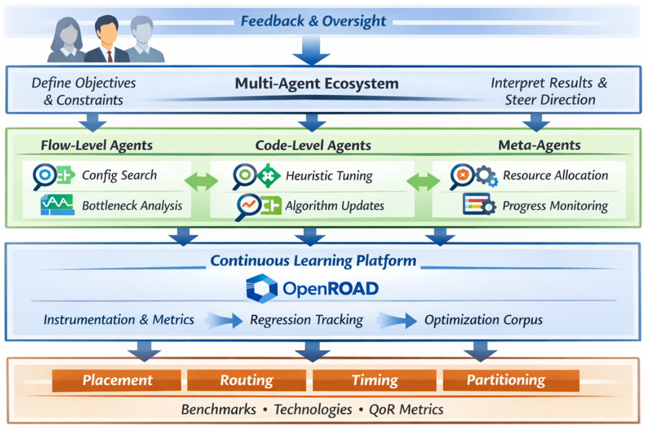

Fig. 6: Vision of agentic AI–driven physical design R&D leveraging OpenROAD.

OpenROAD as a continuous learning platform. We see OpenROAD as a natural foundation for realizing this vision. Its transparent codebase, reproducible flows, and permissive licensing make it uniquely suited to serve as a shared substrate for agentic experimentation and learning. Rather than treating OpenROAD as a fixed reference implementation, we envision it as a *continuous learning platform* in which agents iteratively improve heuristics, cost functions, and workflow structures. Over time, this process can yield a growing corpus of agent-discovered optimizations, design patterns, and algorithmic variants, all grounded in measured quality-of-results across diverse designs and technologies. Such a corpus would not only benefit autonomous agents, but also provide human researchers with new insights into physical design tradeoffs and algorithmic structure.

Near-term impact: agent-assisted algorithm development. In the near term, the most immediate impact of agentic AI is likely to be in accelerating algorithm prototyping and evaluation. Code-level agents can already propose localized modifications to cost functions, heuristics, and control logic within existing placement, routing, and partitioning engines; see, e.g., [21]. When coupled with automated benchmarking pipelines, these agents enable rapid iteration over design alternatives that would be prohibitively time-consuming for human researchers alone. We expect such workflows to become increasingly common for exploring large design spaces, tuning heuristic parameters, and stress-testing algorithms across diverse benchmarks and technology nodes.

Data-driven heuristic discovery and refinement. A second near- to medium-term prospect lies in the emergence of data-driven heuristic design. As agent-driven experimentation produces large volumes of structured performance data, agents can begin to identify patterns linking algorithmic choices to quality-of-results outcomes. Rather than replacing analytical insight, this empirical feedback can help surface non-obvious tradeoffs, regime-dependent behaviors, and corner cases that inform subsequent human-guided refinement. This tight loop between data and interpretation offers a practical path toward more robust and adaptable physical design algorithms.

Incremental evolution of PD infrastructures. In the medium term, we anticipate that open-source infrastructures such as OpenROAD will increasingly support agent-oriented workflows natively. This includes standardized APIs for instrumentation, metric extraction, controlled code modification, and regression tracking. Progress is likely to occur incrementally, with agents contributing small, validated improvements that accumulate into substantial tool evolution. Such incrementalism aligns well with the conservative validation culture required for trustworthy physical design tools.

Human–agent co-evolution and gating challenges. Importantly, this trajectory does not imply fully autonomous tool development that excludes human expertise. Instead, we anticipate a model of *human–agent co-evolution*, in which agents handle large-scale exploration, implementation, and regression testing, while human designers provide high-level guidance, domain knowledge, and judgment. At the same time, several challenges remain. Ensuring correctness, reproducibility, and interpretability of agent-generated code, as well as avoiding overfitting to specific benchmarks or metrics, will be essential for translating early successes into sustained, community-wide impact.

6 Conclusion

This paper has examined the emerging role of agentic AI in physical design research and development, arguing that recent advances in large language models and tool-integrated agents enable a qualitative shift in how EDA tools are improved. We have introduced a taxonomy distinguishing flow-level agents, which optimize tool orchestration and parameterization, from code-level agents, which directly modify algorithms and implementation logic. Through case studies in the OpenROAD ecosystem, we have shown that agentic systems can already deliver meaningful improvements across multiple stages of physical design, including flow tuning, detailed placement, simulation, and hypergraph partitioning. These results demonstrate that autonomous, tool-aware agents can function as effective R&D collaborators rather than merely assistive optimization components. Looking forward, agentic AI points toward a model of continuous human–agent co-evolution of physical design tools. Open-source platforms such as OpenROAD provide a strong foundation for this transition by enabling transparent experimentation, reproducible evaluation, and data-driven algorithm discovery. While challenges remain in verification, benchmarking, and multi-objective reasoning, agentic systems are poised to increasingly shoulder the burden of large-scale exploration and implementation, allowing human experts to focus on objectives, constraints, and architectural insight.

Acknowledgments. Partial support for this research was provided by the Samsung AI Center.

References

[1] J. Achiam, S. Adler, S. Agarwal, L. Ahmad, I. Akkaya, F. L. Aleman et al., "GPT-4 technical report", *arXiv:2303.08774*, 2023, https://www.arxiv.org/abs/2303.08774.
[2] T. Ajayi, V. A. Chhabria, M. Fogaça, S. Hashemi, A. Hosny, A. B. Kahng et al., "Toward an Open-Source Digital Flow: First Learnings from the OpenROAD Project", *Proc. DAC*, 2019, pp. 1–4.
[3] Algorithmic Superintelligence, "OpenEvolve: Evolutionary code optimization". https://github.com/algorithmicsuperintelligence/openevolve
[4] C. Bai, Q. Sun, J. Zhai, Y. Ma, B. Yu and M. D. F. Wong, "BOOM-Explorer: RISC-V BOOM microarchitecture design space exploration framework", *Proc. ICCAD*, 2021, pp. 1–9.
[5] 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. DAES* 30(6) (2025), pp. 100:1–100:26.
[6] R. Bommasani, D. A. Hudson, E. Adeli, R. Altman, S. Arora, S. von Arx et al., "On the opportunities and risks of foundation models", *arXiv:2108.07258*, 2021, https://www.arxiv.org/abs/2108.07258.
[7] T. B. Brown, B. Mann, N. Ryder, M. Subbiah, J. Kaplan, P. Dhariwal et al., "Language models are few-shot learners", *Proc. NeurIPS*, 2020, pp. 1877–1901.
[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] I. Bustany, G. Gasparyan, A. B. Kahng, Y. Koutis, B. Pramanik and Z. Wang, "An Open-Source Constraints-Driven General Partitioning Multi-Tool for VLSI Physical Design", *Proc. ICCAD*, 2023, pp. 1–8.
[10] Cadence, "Cadence Cerebrus intelligent chip explorer", 2021. https://www.cadence.com/en_US/home/tools/digital-design-and-signoff/soc-implementation-and-floorplanning/cerebrus-intelligent-chip-explorer.html
[11] K. Chang, Y. Wang, H. Ren, M. Wang, S. Liang, Y. Han et al., "ChipGPT: How far are we from natural language hardware design", *arXiv:2305.14019*, 2023, https://arxiv.org/abs/2305.14019.
[12] 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.
[13] C.-K. Cheng, A. B. Kahng, S. Kundu, Y. Wang and Z. Wang, "An Updated Assessment of Reinforcement Learning for Macro Placement", *IEEE Trans. CAD* (2025) (DOI 10.1109/TCAD.2025.3644293).
[14] V. A. Chhabria, V. Gopalakrishnan, A. B. Kahng, S. Kundu, Z. Wang, B.-Y. Wu and D. Yoon, "Strengthening the Foundations of IC Physical Design and ML EDA Research", *Proc. ICCAD*, 2024, pp. 1–9.
[15] Y.-C. Fang, H.-Y. Lin, M.-Y. Sui, C.-M. Li and E. J.-W. Fang, "Machine-learning-based dynamic IR drop prediction for ECO", *Proc. ICCAD*, 2018, pp. 1–7.
[16] W. Fang, J. Wang, Y. Lu, S. Liu, Y. Wu, Y. Ma et al., "A survey of circuit foundation model: Foundation AI models for VLSI circuit design and EDA", *arXiv:2504.03711*, 2025, https://www.arxiv.org/abs/2504.03711.
[17] W. Fang, S. Liu, H. Zhang and Z. Xie, "Annotating slack directly on your Verilog: Fine-grained RTL timing evaluation for early optimization", *Proc. DAC*, 2024, pp. 1–6.
[18] W. Fang, Y. Lu, S. Liu, Q. Zhang, C. Xu, L. W. Wills et al., "MasterRTL: A pre-synthesis PPA estimation framework for any RTL design", *Proc. ICCAD*, 2023, pp. 1–9.
[19] W. Fang, Y. Lu, S. Liu, Q. Zhang, C. Xu, L. W. Wills, H. Zhang and Z. Xie, "Transferable pre-synthesis PPA estimation for RTL designs with data augmentation techniques", *IEEE Trans. CAD* 44(1) (2024), pp. 200–213.
[20] Gemini team, "Gemini 2.5: Our most intelligent AI model", *The Keyword*, 2025.
[21] A. Ghose, J. Jang, A. B. Kahng and J. Lee, "Automated QoR improvement in OpenROAD with coding agents", *arXiv:2601.06268*, 2026, https://arxiv.org/abs/2601.06268.
[22] A. Ghose, A. B. Kahng, S. Kundu and Z. Wang, "ORFS-agent: Tool-using agents for chip design optimization", *Proc. MLCAD*, 2025, pp. 1–13.
[23] D. E. Goldberg, *Genetic algorithms in search, optimization, and machine learning*, Addison-Wesley, 1989.
[24] Z. He, H. Wu, X. Zhang, X. Yao, S. Zheng, H. Zheng et al., "ChatEDA: A large language model powered autonomous agent for EDA", *Proc. MLCAD*, 2023, pp. 1–6.
[25] G. Huang, J. Hu, Y. He, J. Liu, M. Ma, Z. Shen et al., "Machine learning for electronic design automation: A survey", *ACM Trans. DAES* 26(5) (2021), pp. 1–46.
[26] Y.-H. Huang, Z. Xie, G.-Q. Fang, T.-C. Yu, H. Ren et al., "Routability-driven macro placement with embedded CNN-based prediction model", *Proc. DATE*, 2019, pp. 180–185.
[27] F. Hutter, H. H. Hoos, K. Leyton-Brown, and T. Stützle, "ParamILS: An automatic algorithm configuration framework", *Journal of Artificial Intelligence Research* 36 (2009), pp. 267–306.
[28] J. Jung, A. B. Kahng, S. Kim and R. Varadarajan, "METRICS2.1 and flow tuning in the IEEE CEDA robust design flow and OpenROAD", *Proc. ICCAD*, 2021, pp. 1–9.
[29] A. Kaintura, Palaniappan R, S. S. Luar and I. Iyer Almeida, "ORAssistant: A Custom RAG-based conversational assistant for OpenROAD", *arXiv:2410.03845*, 2024, https://arxiv.org/abs/2410.03845.
[30] J. Kaplan, S. McCandlish, T. Henighan, T. B. Brown, B. Chess, R. Child et al., "Scaling laws for neural language models", *arXiv:2001.08361*, 2020, https://www.arxiv.org/abs/2001.08361.
[31] W. B. Langdon and J. Petke, "Genetic improvement of software for multiple objectives", *Proc. ISSBSE*, 2015, pp. 12–28.
[32] R. Liang, J. Jung, H. Xiang, L. Reddy, A. Lvov, J. Hu and G.-J. Nam, "FlowTuner: A multi-stage EDA flow tuner exploiting parameter knowledge transfer", *Proc. ICCAD*, 2021, pp. 1–9.
[33] 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.
[34] 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.
[35] M. Liu, N. Pinckney, B. Khailany and H. Ren, "ChipNemo: Domain-adapted LLMs for chip design", *arXiv:2311.00176*, 2023, https://arxiv.org/abs/2311.00176.
[36] M. Liu, N. Pinckney, B. Khailany and H. Ren, "VerilogEval: Evaluating large language models for verilog code generation", *Proc. ICCAD*, 2023, pp. 1–8.
[37] S. Liu, W. Fang, Y. Lu, J. Wang, Q. Zhang, H. Zhang and Z. Xie, "RTLCoder: fully open-source and efficient LLM-assisted RTL code generation technique", *IEEE Trans. CAD*, 2024.
[38] M. L.-Ibáñez, J. D.-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.
[39] Y.-C. Lu, S. Nath, V. Khandelwal and S. K. Lim, "RL-Sizer: VLSI gate sizing for timing optimization using deep reinforcement learning", *Proc. DAC*, 2021, pp. 733–738.
[40] Y.-C. Lu, W.-T. Chan, D. Guo, S. Kundu, V. Khandelwal and S. K. Lim, "RL-CCD: Concurrent clock and data optimization using attention-based self-supervised reinforcement learning", *Proc. DAC*, 2023, pp. 1–6.
[41] Y. Lu, S. Liu, Q. Zhang and Z. Xie, "RTLLM: An open-source benchmark for design RTL generation with large language model", *arXiv:2308.05345*, 2023, https://arxiv.org/abs/2308.05345.
[42] Y. Lu, H. I. Au, J. Zhang, J. Pan, Y. Wang, A. Li et al., "AutoEDA: Enabling EDA flow automation through microservice-based LLM agents", *arXiv:2508.01012*, 2025, https://arxiv.org/abs/2508.01012.
[43] 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.
[44] 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.
[45] OpenAI team, "Introducing OpenAI o3 and o4-mini", *OpenAI*, 2025, https://openai.com/index/introducing-o3-and-o4-mini/.
[46] OpenAI, "Evaluating large language models trained on code", *arXiv:2107.03374*, 2021, https://www.arxiv.org/abs/2107.03374.
[47] M. Papermaster, "AMD CDNA 2 architecture", *IEEE Int. Solid-State Circuits Conf.*, 2022, pp. 1–3.
[48] H. Pearce, B. Tan and R. Karri, "Dave: Deriving automatically Verilog from English", *Proc. MLCAD*, 2020, pp. 27–32.
[49] Z. Pei, H.-L. Zhen, M. Yuan, Y. Huang and B. Yu, "BetterV: Controlled Verilog generation with discriminative guidance", *Proc. ICML* (PMLR 235), 2024, pp. 40145–40153.
[50] M. Rapp, H. Amrouch, Y. Lin, B. Yu, D. Z. Pan, M. Wolf et al., "MLCAD: A survey of research in machine learning for CAD keynote paper", *IEEE Trans. CAD* 41(10) (2021), pp. 3162–3181.
[51] J. R. Rice, "The algorithm selection problem", *Advances in Computers*, 1976, pp. 65–118.
[52] 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.
[53] B. C. Schafer and Z. Wang, "High-level synthesis design space exploration: Past, present, and future", *IEEE Trans. CAD* 39(10) (2020), pp. 2628–2639 .
[54] T. Schick, J. D.-Yu, R. Dessì, R. Raileanu, M. Lomeli, L. Zettlemoyer et al., "Toolformer: Language models can teach themselves to use tools", *Proc. NeurIPS*, 2023, pp. 68539–68551.
[55] P. Sengupta, A. Tyagi, Y. Chen and J. Hu, "How good is your Verilog RTL code? A quick answer from machine learning", *Proc. ICCAD*, 2022, pp. 1–9.
[56] N. Shinn, F. Cassano, A. Gopinath, K. Narasimhan and S. Yao, "Reflexion: Language agents with verbal reinforcement learning", *Proc. NeurIPS*, 2023, pp. 8634–8655.
[57] Synopsys, "DSO.ai: AI-driven design applications", 2021. https://www.synopsys.com/ai/ai-powered-eda/dso-ai.html
[58] S. Thakur, J. Blocklove, H. Pearce, B. Tan, S. Garg and R. Karri, "AutoChip: Automating HDL generation using LLM feedback", *arXiv:2311.04887*, 2023, https://arxiv.org/abs/2311.04887.
[59] S. Thakur, B. Ahmad, Z. Fan, H. Pearce, B. Tan, R. Karri, B. Dolan-Gavitt and S. Garg, "Benchmarking large language models for automated Verilog RTL code generation", *Proc. DATE*, 2023, pp. 1–6.
[60] S. Thakur, B. Ahmad, H. Pearce, B. Tan, B. Dolan-Gavitt, R. Karri and S. Garg, "VeriGen: A large language model for Verilog code generation", *ACM Trans. DAES* 29(3) (2024), pp. 1–31.
[61] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez et al., "Attention is all you need", *Proc. NeurIPS*, 2017, pp. 5998–6008.
[62] Y. Wang, W. Ye, Y. He, Y. Chen, G. Qu and A. Li, "MCP4EDA: LLM-powered model context protocol RTL-to-GDSII automation with backend aware synthesis optimization", *arXiv:2507.19570*, 2025, https://arxiv.org/abs/2507.19570.
[63] J. Wei, Y. Tay, R. Bommasani, C. Raffel, B. Zoph, S. Borgeaud et al., "Emergent abilities of large language models", *arXiv:2206.07682*, 2022, https://www.arxiv.org/abs/2206.07682.
[64] H. Wu, Z. He, X. Zhang, X. Yao, S. Zheng, H. Zheng and B. Yu, "ChatEDA: A large language model powered autonomous agent for EDA", *IEEE Trans. CAD* 43(9) (2024), pp. 2717–2730.
[65] H. Wu, H. Zheng, Z. He and B. Yu, "Divergent thoughts toward one goal: LLM-based multi-agent collaboration system for electronic design automation", *arXiv:2502.10857*, 2025, https://www.arxiv.org/abs/2502.10857.
[66] 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
[67] B.-Y. Wu, U. Sharma, S. R. D. Kankipati, A. Yadav, B. K. George, S. R. Guntupalli et al., "EDA Corpus: A large language model dataset for enhanced interaction with OpenROAD", *arXiv:2405.06676*, 2024, https://arxiv.org/abs/2405.06676.
[68] Z. Xie, R. Liang, X. Xu, J. Hu, C.-C. Chang, J. Pan et al., "Pre-placement net length and timing estimation by customized graph neural network", *IEEE Trans. CAD* 41(11) (2022), pp. 4667–4680.
[69] Z. Xie, H. Ren, B. Khailany, Y. Sheng, S. Santosh, J. Hu and Y. Chen, "PowerNet: Transferable dynamic IR drop estimation via maximum convolutional neural network", *Proc. ASP-DAC*, 2020, pp. 13–18.
[70] Z. Xie, Y.-H. Huang, G.-Q. Fang, H. Ren, S.-Y. Fang, Y. Chen and J. Hu, "RouteNet: Routability prediction for mixed-size designs using convolutional neural network", *Proc. ICCAD*, 2018, pp. 1–8.
[71] Z. Xie, G.-Q. Fang, Y.-H. Huang, H. Ren, Y. Zhang, B. Khailany, S.-Y. Fang, J. Hu, Y. Chen and E. C. Barboza, "FIST: A feature-importance sampling and tree-based method for automatic design flow parameter tuning", *Proc. ASP-DAC*, 2020, pp. 19–25.
[72] K. Xu, D. Schwachhofer, J. Blocklove, I. Polian, P. Domanski, D. Pflüger et al., "Large language models (LLMs) for electronic design automation (EDA)", *arXiv:2508.20030*, 2025, https://www.arxiv.org/abs/2508.20030.
[73] S. Yao, J. Zhao, D. Yu, N. Du, I. Shafran, K. R. Narasimhan and Y. Cao, "ReAct: synergizing reasoning and acting in language models", *arXiv:2210.03629*, 2022, https://arxiv.org/abs/2210.03629.
[74] 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.
[75] 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.
[76] B. Yu, "Machine learning in EDA: When and how", *ACM/IEEE MLCAD*, 2023, pp. 1–6.
[77] C. Yu, R. Liang, C.-T. Ho and H. Ren, "Autonomous code evolution meets NP-completeness", *arXiv:2509.07367*, 2025, https://arxiv.org/abs/2509.07367.
[78] Q. Zhang, Y. Lu, M. Li and Z. Xie, "AutoPower: Automated few-shot architecture-level power modeling by power group decoupling", *Proc. DAC*, 2025, pp. 1–7.
[79] Y. Zhang, H. Ren and B. Khailany, "GRANNITE: Graph neural network inference for transferable power estimation", *Proc. DAC*, 2020, pp. 1–6.
[80] CIRCT: Circuit IR compilers and tools, https://circt.llvm.org/.
[81] Claude Code, https://github.com/anthropics/claude-code.
[82] FastPart, https://vlsicad.ucsd.edu/hypergraphs/.
[83] Model Context Protocol, "Model context protocol specification", Version 2025-03-26, 2025. https://modelcontextprotocol.io/specification/2025-03-26.
[84] OpenROAD. https://github.com/The-OpenROAD-Project/OpenROAD
[85] OpenROAD-flow-scripts, https://github.com/The-OpenROAD-Project/OpenROAD-flow-scripts
[86] Ergodex-Core. circt-verilog https://github.com/Ergodex-Core/circt-verilog, sv-tests-arcilator https://github.com/Ergodex-Core/sv-tests-arcilator.
[87] Hypergraph partitioning leaderboard, https://github.com/TILOS-AI-Institute/HypergraphPartitioning.