The Reach and Ceiling of Rebuilding the Chip Design Toolchain with LLM Agents

Zhengqi Gao[†]
Massachusetts Institute of Technology

Abstract

Within four years, large language models (LLMs) have advanced from conversational assistants in late 2022 (e.g., ChatGPT) to autonomous digital agents in 2026 (e.g., Claude Code and Codex). Agentic LLMs now can operate in complex computer environments and solve long-horizon software engineering tasks. Among such software, integrated circuit (IC) design tools, referred to as electronic design automation (EDA) software, constitute a unique and important class: they encode decades of accumulated algorithmic and engineering expertise, and no chip design can be carried out without them. How far LLM agents can push the optimization of existing EDA tools, and whether they can rebuild more efficient EDA software from scratch, remain open questions. In this paper, we present a systematic investigation of LLM agents as EDA tool developers across three regimes of different scope and ambition: (i) developing nine CPU-based EDA tools from scratch in Rust, where the majority of agent-built tools attain 2–13× runtime speedups over mature open-source references (e.g., ABC, Ngspice, OpenSTA) at matched or better quality of results, with routing proving the most challenging to rebuild; (ii) optimizing three GPU-accelerated tools (i.e., DREAMPlace, InstantGR, and GEM) given their source code, with the best chains achieving 1.3–2× speedups; and (iii) rebuilding the complete digital design flow from scratch in Rust and deploying it for end-to-end digital IC design, where the agent-native flow is competitive with the OpenROAD reference on the Tier-1 suite of small blocks containing <2K standard cells under an open-source PDK, while scaling to substantially larger designs remains out of reach. The full campaign is carried out primarily with a single human intermittently in the loop over roughly two months. Our results suggest that (1) rebuilding open-source-grade individual EDA tools and the overall chip design flow is mostly within the capability of present-day LLM agents, with back-end tools harder than front-end tools; and (2) agent-developed native EDA tools cannot yet target large-scale, state-of-the-art ICs. Taken together, these findings indicate that producing proprietary-grade EDA tools for the latest semiconductor technology nodes **remains beyond current agent capabilities**.

1 Introduction

The Transformer (Vaswani et al., 2017) provides the architectural bedrock of modern large language models (LLMs), and the discovery of neural scaling laws (Kaplan et al., 2020; Hoffmann et al., 2022) shows that model capability grows predictably as models, data, and compute are scaled up. The milestone arrived with the release of ChatGPT in late 2022 (OpenAI, 2022), which brought conversational LLMs to hundreds of millions of users. Since then, the field has progressed through three broad stages: a conversational stage, in which LLMs answer questions in a single turn; a reasoning stage, in which LLMs spend test-time compute to think before answering (OpenAI, 2024; Guo et al., 2025a); and the current agentic stage (Yao et al., 2023; Wang et al., 2024), in which LLMs equipped with an agentic harness such as Claude Code (Anthropic, 2025a) or Codex (OpenAI, 2025) operate over long horizons in complex computer environments automatically. Such agents now operate at the frontier of, or even surpass, human expertise in solving real-world problems, e.g., repository-level software engineering issues (Jimenez et al., 2024), long-open Erdős problems (Tsoukalas et al., 2026), Fermat's Last Theorem (Anthropic, 2026), and the Navier-Stokes problem (OpenAI, 2026). Their growing competence has prompted an active discussion about the future of human society alongside such agents: the future of professions and of the organizations built around them (Anthropic, 2025b), and the alignment of artificial general intelligence (AGI) with human values (Bostrom, 2014; Bai et al., 2022; Ji et al., 2023; Pachocki, 2026).

[†] Corresponding to: zhengqi@mit.edu.

As LLM agents revolutionize one domain after another, it is natural to ask where and how they can contribute to integrated circuit (IC) design. In a typical IC design flow, a design company designs a chip using electronic design automation (EDA) software and a process design kit (PDK), then delivers the tape-out file to the foundry for fabrication. A growing body of work explores how LLM agents can assist this flow, from several directions: improving the capability of LLMs to write hardware description languages (HDL) such as Verilog (Liu et al., 2023b; 2025; Thakur et al., 2023); building agents that orchestrate existing EDA tools to carry out design tasks (He et al., 2023; Liu et al., 2023a); constructing benchmarks that measure progress on realistic hardware design problems, from module-level Verilog generation (Liu et al., 2023b; Lu et al., 2024) to verification and debugging (Pinckney et al., 2025), end-to-end digital implementation in FluxBench (Deng et al., 2026) and Build-A-Chip-Bench (Gunukula et al., 2026), and long-horizon transistor-level analog design in Analog Design Bench (Analog Design Bench Team, 2026); and many more.

However, we argue that an equally important yet largely unexplored question is whether LLM-based agents can optimize the existing EDA tools, or even rebuild them from scratch to be better (e.g., faster than existing tools, and designed for agents to use). This problem is crucial and timely, and, given the rapid progress of coding agents, well within reach, for the following reasons.

- **Better RL environments for LLM post-training.** First, improving the ability of LLMs to design chips is an important goal in its own right, and the standard paradigm is reinforcement learning (RL) post-training (Ouyang et al., 2022; Guo et al., 2025a) in a chip design sandbox environment. Such an environment requires EDA tools, which the agent invokes repeatedly through tool calls (Schick et al., 2023). Better EDA tools shorten rollout time and thereby accelerate the training loop itself.
- **Empower LLM agents for chip design.** Second, most existing EDA tools were built decades ago for human operators, with usage syntax dating back more than fifty years (e.g., the SPICE netlist format (Nagel and Pederson, 1973)). Alternatively, agent-native EDA tools can potentially consume fewer tokens and less context, require fewer interaction turns, run faster, and yield better designs. Together, these gains shorten the design-to-deployment cycle for the hardware that advanced AI systems depend on.
- **A harder testbench for coding agents.** Third, existing agent benchmarks such as DeepSWE v1.1 (Huang and Jiang, 2026), SWE-Bench Pro V2 (Dan et al., 2026), and Terminal-Bench 4.0 (Terminal-Bench Team, 2026) focus largely on general-purpose software engineering; EDA tools pose a qualitatively different challenge, being algorithmically deep, performance-critical, heuristics-heavy, and scarce in public training data. Rebuilding the toolchain is thus a demanding and fully verifiable test of how far frontier coding agents can go.
- **A prerequisite for recursive self-improvement (RSI).** Fourth, recursive self-improvement (Good, 1966; Bostrom, 2014) ultimately requires the ability to design the hardware on which the improving system runs; chip design capability is therefore a prerequisite rather than an afterthought. If LLM agents can build the very EDA tools with which chips are designed, the entire loop, from tool development through chip design to model training, becomes amenable to autonomous improvement.

In this paper, we present a systematic investigation of LLM agents as EDA tool developers. Our campaign is organized into three regimes of different scope and ambition. In the first regime, the agent develops nine CPU-based EDA tools from scratch in Rust under the ASAP7 (Clark et al., 2016), Nangate 45nm (Nangate Inc., 2008), SkyWater 130nm (SkyWater Technology Foundry and Google, 2020), and GF180nm (GlobalFoundries and Google, 2022) PDKs; most of the resulting implementations run 2–13$\times$ faster than mature open-source references such as ABC, Ngspice, and OpenSTA while matching (sometimes exceeding) their quality of results, although the routing tool proves the hardest to reproduce. In the second regime, the agent is given the source code of three GPU-accelerated tools, DREAMPlace (Lin et al., 2019), InstantGR (Lin et al., 2024), and GEM (Guo et al., 2025b), with the best chains achieving 1.3–2$\times$ speedups on A100/H100 GPUs. In the final regime, the agent rebuilds a complete digital design flow from scratch in Rust and applies it to end-to-end digital IC design: on the Tier-1 suite of small blocks containing $<$2K standard cells under the

SkyWater 130nm PDK, the agent-native flow matches the OpenROAD reference, whereas substantially larger designs fail, owing to the limited capability of the agent-developed physical design tools.

Our contributions. We do not intend to claim any human-proposed novelty or improvement on a particular EDA algorithm in this paper; rather, our paper offers a thorough empirical investigation of what LLM agents can accomplish in EDA tool development when tokenmaxxing. Our contributions are threefold. First, we provide practical frameworks for setting up the experimental harnesses, including useful safeguards against reward hacking. Second, we present the key insights and knowledge learnt from the campaign, allowing others to benefit from our findings without re-incurring the token and compute costs of repeating these experiments. Third, we chart the frontier of problems that remain beyond current agents, mapping out where progress must be made next.

2 Background

A brief history of EDA. EDA grew up alongside the integrated circuit itself, and its foundational algorithms were largely laid down within two decades. Circuit simulation was settled first: SPICE (Nagel and Pederson, 1973) established both the numerical formulation and the netlist syntax in the early 1970s. Logic synthesis and technology mapping matured through the 1980s and 1990s around two-level and multi-level Boolean optimization (Brayton et al., 1984; 1987) and binary decision diagrams (Bryant, 1986), and later consolidated on and-inverter graphs with rewriting-based optimization, the formulation that ABC (Brayton and Mishchenko, 2010) implements today. Physical design converged on analytical placement and maze routing, which remain the backbone of modern implementations and of open-source flows such as OpenROAD (Ajayi et al., 2019). Sangiovanni-Vincentelli (2003) described the progression of EDA as a series of tides: successive waves of abstraction, each receding into a stable layer of algorithms and interfaces. That stability is what makes the toolchain both durable and old. The command syntaxes and file formats in use in 2026 were largely fixed decades ago. The implementations are less uniform: many widely used tools still carry the architectural assumptions of the machines they were originally written for, while some have been modernized, as discussed next.

Modern ML-based EDA. Machine learning (ML) has entered the EDA tools along two distinct paths. The first treats the GPU as a general-purpose accelerator and rewrites classical EDA problem formulations in differentiable, data-parallel form. DREAMPlace (Lin et al., 2019) is the canonical example, expressing analytical placement so that it can be solved with GPU-based gradient optimization and obtaining order-of-magnitude speedups over multi-threaded CPU placers at comparable quality; the same strategy has since been applied to global routing, RTL simulation, and logic synthesis, yielding GPU-native engines such as InstantGR (Lin et al., 2024), GEM (Guo et al., 2025b), and CULS (Lin et al., 2022). The second path replaces a circuit heuristic with a learned ML model. A prominent example is AlphaChip (Mirhoseini et al., 2021), which formulates macro placement as a reinforcement learning problem and reports deployment on production TPU designs (Mirhoseini et al., 2021), although its claimed advantages have aroused controversies (Markov, 2024; Cheng et al., 2023). ML models have also been used to predict downstream quality metrics from early-stage designs, such as routability and design-rule violations before routing is run (Xie et al., 2018; Kirby et al., 2019), and to serve as fast surrogate evaluators in place of expensive tool invocations, supported by public datasets curated for this purpose (Chai et al., 2023).

EDA in the LLM era. Many IC design stages are inherently text-based, from HDL sources and constraint files to tool scripts and log reports, so it is natural that LLMs have opened a third path beyond ML-based EDA, one that operates directly on these text artifacts. The earliest and most active line targets HDLs, opened by DAVE (Pearce et al., 2020), which fine-tunes an LLM to generate Verilog from English, and followed by methods for generating and repairing Verilog (Liu et al., 2025; Thakur et al., 2023) and benchmarks that measure this ability at increasing realism (Liu et al., 2023b; Lu et al., 2024). A parallel line equips models with domain context and tool access so that they can drive existing EDA software (He et al., 2023; Liu et al., 2023a), and related efforts extend LLM assistance to verification collateral (Yan et al., 2025) and to analog

circuit design (Lai et al., 2025). What all of these share is that the tools themselves remain fixed: the model writes designs, or issues commands, but the underlying EDA tool implementations are untouched.

Evolving (EDA) software with LLM-based agents. A separate and much more recent line of work places the agent in a long-horizon loop with the software itself: the agent repeatedly proposes modifications, runs an executable evaluator, and learns from the resulting feedback, so that performance improves with sustained interaction rather than in a single shot. FunSearch (Romera-Paredes et al., 2024) demonstrated the pattern by pairing an LLM with an automated evaluator to discover novel mathematical constructions, AlphaEvolve (Novikov et al., 2025) generalized it from evolving single functions to entire codebases, and ShinkaEvolve (Lange et al., 2025) makes the evolutionary loop markedly more sample-efficient. A growing family of benchmarks now measures this iterative-optimization setting, including KernelBench for GPU kernel generation (Ouyang et al., 2025), GSO for repository-scale software optimization (Shetty et al., 2025), ALE-Bench for long-horizon algorithm engineering (Imajuku et al., 2025), and EdgeBench for learning from real-world environments (Zhu et al., 2026). The idea of agentic software evolution has since reached the EDA community: SATLUTION (Yu et al., 2025) scales repository-level evolution to entire SAT solvers, Yu and Ren (2026) evolve the ABC source code under a correctness and quality-of-results loop, improving the tool rather than merely using it, and HORIZON (Yu et al., 2026a) applies repository-level evolution to hardware design artifacts. Our work takes this premise considerably further, asking not only whether agents can improve an existing tool, but whether they can rebuild the tools, and ultimately the entire flow, from scratch.

3 Regime I: Rebuild CPU-Based EDA Tools from Scratch

The first regime asks the most basic question: can an LLM-based agent rebuild an EDA tool from scratch? We select nine open-source CPU-based EDA tools that together span the digital and analog flow (Table 1), and attempt to rebuild tools implementing their core functions from scratch. We use Claude Code with Fable 5 as the agent throughout this section.

Our tool	Design stage	Dataset (dev / held)	Problem type (Ground truth)	Pass gate	Reference tool
rspice	Analog simulation	44 / 47	Approximation (Costly)	Waveform tolerance	Ngspice
rvsim	RTL simulation	152 / 123	Approximation (Costly)	Byte-exact output	Icarus Verilog
rabc	Logic optimization	41 / 41	Optimization (Unknown)	CEC + QoR band	ABC
rsta	Static timing	50 / 50	Approximation (Costly)	Slack within 1 ps	OpenSTA
rplace	Placement	34 / 36	Optimization (Unknown)	Legality + HPWL ratio	OpenROAD
rcts	Clock tree synthesis	46 / 46	Optimization (Unknown)	Skew/latency ratios	TritonCTS
rrouter	Detailed routing	36 / 36	Optimization (Unknown)	Opens/shorts/DRC	OpenROAD
rmagic	DRC and extraction	54 / 58	Approximation (Costly)	Byte parity of outputs	Magic
rlvs	LVS	49 / 45	Exact-math (Known)	Verdict match	Netgen

Table 1: The nine tool-rebuilding campaigns, ordered from the front end to the back end of the design flow. Each row lists our rebuilt tool, its design stage, the dataset size as development / held-out case counts at the start of the campaign, before any burn-rule promotions, the problem type together with the availability of its ground truth in parentheses, the pass gate that checks every output case, and the pinned reference tool the gate compares against. Please refer to the main text for more details.

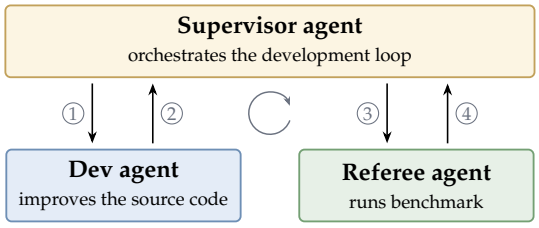


Figure 1: The harness of Regime I. Each iteration, the supervisor ① briefs a fresh dev agent and ② receives its improved source; ③ the rebuilt tool goes to a sealed referee, and ④ its verdict closes the loop.

Experimental setup overview. All nine references are originally written in C or C++. We enforce the agent to rebuild every tool in Rust; the switch of language serves as a low-effort protection against reward hacking, since no line of the reference implementation can be carried over verbatim. Figure 1 shows our experimental setup, or harness. A supervisor agent

orchestrates the whole process and reacts to our prompts. At each iteration it briefs a *fresh* dev agent with no memory of earlier iterations, handing it the current source tree and a dossier of accumulated lessons distilled from previous reports. The dev agent implements whatever improvements it chooses and returns the modified code. The supervisor rebuilds the tool and hands it to the referee agent. For each case in the development and held-out sets, the referee agent executes the rebuilt tool and the pinned reference, interleaved on the same machine to ensure a fair runtime comparison. It then issues the verdict by checking every case against banked reference outputs. The referee agent, the banked references, and the held-out cases live in a sealed location the dev agent can neither read nor write; the dev agent sees only the verdicts on the development set, while verdicts on the held-out set are recorded solely for our own reference. In the following, we describe a few important details in our experimental setup.

- **Scope contract.** An EDA tool has many functionalities: some are core, while others are peripheral. In Ngspice, for instance, the `.op`, `.dc`, `.ac`, and `.tran` analyses over standard device models are core, while interactive plotting commands are peripheral. In our experiments, we concentrate the rebuilding effort on the core, and the *scope contract* states which features of the reference must be reproduced. For instance, in ABC, we focus on and-inverter graph (AIG) optimization and technology mapping.
- **Pass gate and QoR.** The pass gate defines what counts as correct versus failure on each test case in the development and held-out datasets. As shown in Table 1, the EDA tools we consider here can be classified into three types. When an exact answer exists (e.g., LVS), the gate requires exact reproduction of the reference verdict. When the optimum is unknown or the ground truth is costly to obtain, the gate couples a hard correctness check with quality-of-results (QoR) bands relative to the reference. For example, a placement passes only if it is fully legal (the correctness check) and its wirelength is within a fixed ratio of the reference (the QoR band), and a simulated waveform passes if it stays within a tolerance of the reference output.
- **Protocols.** We have two protocols: the *burn rule* and the *source access regime*, and when they apply, they will be clearly logged. The burn rule handles held-out cases that fail persistently: when a held-out case has failed for several consecutive iterations while the development set is fully passing, the supervisor may move it into the development set. The source access regime states whether the agent may read the reference EDA tool implementation starting after a certain iteration. The default is closed-book: the dev agent may run the reference binary and read its documentation, but the reference source code is never present in its workspace. This is enforced by construction, although knowledge of the reference absorbed during pretraining cannot be excluded.
- **Meta plan.** To make our experiments easy to extend to rebuilding new tools, we draft a meta plan document at the beginning of our campaign. Provided a reference tool name (e.g., Ngspice), denoted `xxx`, the meta plan automatically generates a `plan_xxx.md` file, which is reviewed, lightly modified by a human, and then used to launch the experiment that rebuilds from scratch a tool implementing functionality similar to `xxx` in Rust.

Case study: Ngspice and the agent-developed rspice. Ngspice (Vogt et al., 2026) is the standard open-source analog simulator. Its ground truth is the physical system's behavior in the real world, and it is approximated by the solution of the circuit's modeling equations (e.g., the differential-algebraic equations in transient analysis). Our scope contract covers resistors, capacitors, inductors, independent and controlled sources, diodes, Level-1 MOSFETs and BSIM4 (Liu et al., 2000) through the OSDI interface (Kuthe, 2023), and the four analyses `.op`, `.dc`, `.ac`, and `.tran`; noise analysis and several other minor features are excluded. The benchmark comprises 95 circuits, including synthetic circuits as well as cases drawn from AnalogGym (Li et al., 2024), the Open Circuit Benchmark (OCB) (Dong et al., 2023), and the IBM power-grid benchmarks (Nassif, 2008). They range from 14 to 400k elements (the stress cases reach 3.2 million elements) and are split into 44 development, 47 held-out, and 4 stress test cases. The stress test cases are not run in the development iterations, but only once after the whole campaign finishes (i.e., 50 iterations in this case). The pass gate is a tolerance with Ngspice's output as the reference: absolute error at most $10^{-6} + 10^{-4}\,|\text{ref}|$ for operating-point

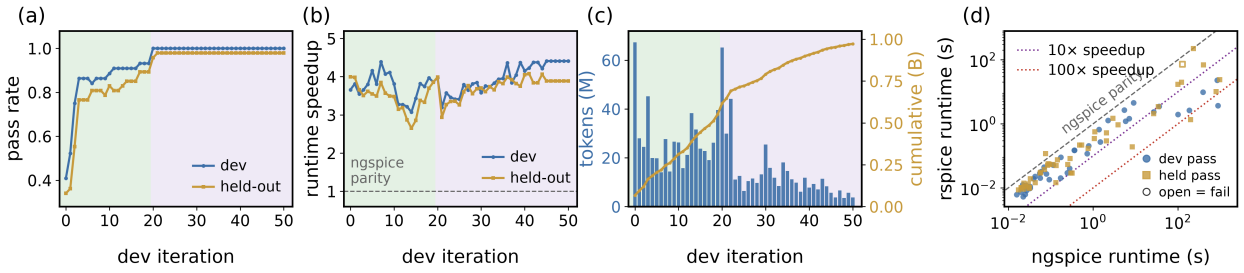


Figure 2: Rebuilding Ngspice from scratch with Claude Code and Fable 5 in Rust. (a) Pass rate on the development and held-out sets over the 50 iterations. (b) Geometric-mean runtime speedup over Ngspice on passing cases of the two sets; the dashed line marks parity. (c) Tokens spent per iteration (bars, left axis) and cumulative tokens (curve, right axis, in billions). (d) Per-case wall time of the rebuilt simulator against Ngspice at the final measurement, both single threaded; the dashed line marks parity, the dotted lines 10× and 100× speedup, and the open marker is a transient simulation that failed outside the tolerance. From (a)-(c), the campaign exhibits a two-stage trend: the first 20 iterations, shaded green, are spent mostly on raising the pass rate, while the remaining 30 iterations, shaded purple, are spent mostly on improving the runtime speedup.

and DC sweeps, per-signal normalized root-mean-square (RMS) error at most 10^{-3} for transients, and 0.05 dB in magnitude and 0.5 degrees in phase for AC. The pinned reference throughout the campaign is `ngspice-46`, released in March 2026 and built from source with KLU and OSDI support; all golden outputs and paired timings come from this single build.

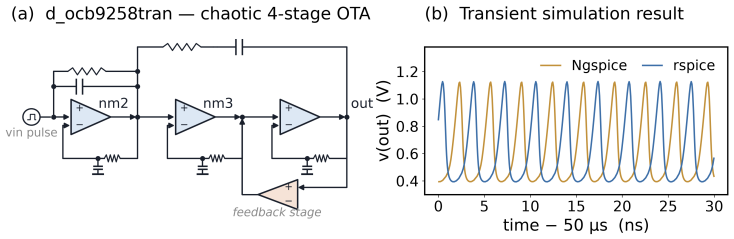


Figure 3: The one persistent held-out failure. (a) The deck: a chaotic four-stage OTA loop of 44 elements; each stage is a five-transistor BSIM4 differential OTA with an RC servo loop stabilizing its DC bias, cross-coupled through high-impedance feedback. (b) The response of Ngspice and our rspice 50 μs into the transient: the same attractor, decorrelated trajectories.

Figure 2 shows the trajectory. The held-out pass rate rises from a third to above 95 percent within 20 iterations and then holds at 46 of 47 (Figure 2a); from iteration 20 onward the protocol is tightened to byte-identical output, and every subsequent speedup (a further roughly 10 percent on the development set) changes no output bit (Figure 2b). The final tool passes 44 of 44 development and 46 of 47 held-out cases and is 4.4× and 3.9× faster than Ngspice on the two sets (Figure 2d). Over the whole campaign the dev agents consumed 0.97 billion input tokens (reported in Figure 2c) and produced 10.7 million output tokens. On the four stress-test IBM power grids of up to 3.2 million elements, evaluated at the end of the development iterations, it passes 4 of 4 at a geometric-mean 7.6× speedup. The one persistent failure is a chaotic oscillator in time-domain transient simulation, shown in Figure 3. Shortly after the input pulse the two trajectories are indistinguishable; tens of microseconds in, Ngspice and our rspice still produce the same oscillation, with matching amplitude, waveform shape, and frequency, but fully decorrelated in phase. This gap is the expected signature of a chaotic system: any difference in floating-point evaluation order, or even in the math library the reference is compiled against, grows exponentially along the trajectory until the pointwise difference saturates at the size of the oscillation itself. In this sense the case is arguably not a real failure of the rebuilt simulator: the two tools agree on the physics and settle on the same attractor, and the same sensitivity means even the reference compared against a differently compiled copy of itself could fail this gate, which cannot certify a chaotic trajectory pointwise.

The speedup does not come from doing less numerical work or from a different algorithm: rspice takes the identical Newton iteration counts (at the operating point, per DC-sweep point, and per accepted timestep),

the identical adaptive timestep sequences in transient analysis, and the identical user-defined sweep and frequency grids in DC and AC analyses; it performs exactly one device-model evaluation per Newton iteration per device; and it runs single-threaded under the interleaved paired measurement protocol. The gains instead arise from executing the same computation more efficiently, through four mechanisms. First, and dominant, is the linear-solver architecture: Ngspice's Sparse 1.3 stores matrix elements in linked lists and re-derives its elimination decisions during each factorization, incurring an indirect memory access per element, whereas rspice records the symbolic elimination schedule once and replays it as a branch-free sweep over contiguous, cache-ordered index streams, so each factorization touches memory in a fixed and predictable order. Second, rspice avoids per-analysis reconstruction overheads, for example updating only the imaginary matrix entries between AC frequency points where Ngspice reloads the full matrix. Third, its statically dispatched, monomorphized implementation compiles under whole-program (link-time) optimization, allowing the compiler to inline device evaluation into the surrounding loops and auto-vectorize independent kernels such as the truncation-error check, none of which is available across Ngspice's function-pointer plugin boundaries and separately compiled translation units. Fourth, rspice allocates its solver and device-state workspaces once, in contiguous arenas sized exactly from the recorded factorization plan, and reuses them across all Newton iterations and timesteps; Ngspice, by contrast, builds its matrix from per-element heap allocations and grows working vectors incrementally, incurring steady-state allocator traffic and the fragmented memory layout that pointer-chasing traversal then pays for. Arena allocation thus contributes to both sides of the efficiency picture: it removes per-iteration heap overhead and underlies the compact, cache-ordered layout exploited by the replay factorization.

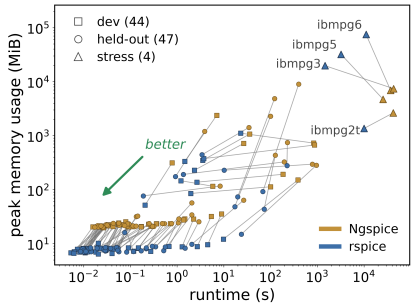


Figure 4: Per-case runtime against peak memory usage for Ngspice and rspice on every measured deck; gray lines connect the two measurements of the same deck. rspice uses less memory on nearly every deck; the only exceptions are the one-shot grid factorizations (`ibmpg1/2` in the suite and the labeled `ibmpg3/5/6`), where its stored elimination schedule dominates.

Consistent with this attribution, the speedup scales with the share of runtime spent in the solver: geometric-mean 34.7× on the nine solver-dominated grid circuits (seven of which exceed 36×; the remaining two, `ibmpg1` and the transient `ibmpg1t`, fall at 4.8× and near parity), 5.0× on the remaining 40 native-device circuits, but only 2.1× on the 41 BSIM4 circuits, where both simulators execute the identical compiled OSDI model binary and only the surrounding solver and assembly costs differ. Notably, the compact data layout that drives the speedup also reduces peak memory: as shown in Figure 4, rspice uses less memory than Ngspice on 89 of the 91 development and held-out circuits (geometric mean 0.30×; the two exceptions are the one-shot operating-point grids `ibmpg1` and `ibmpg2`), while on the four stress-test grids it uses much more memory on the three one-shot factorizations, where the stored elimination schedule grows to as much as 11× Ngspice's footprint on `ibmpg6`, and less on the transient `ibmpg2t`.

Case study: ABC and the agent-developed rabc. ABC (Brayton and Mishchenko, 2010) is the standard open-source logic optimizer. Logic optimization is a problem with no known optimum: correctness is exactly checkable (the output must compute the same Boolean function as the input), but there is no "true" answer for how small or how fast the result should be, so quality can only be judged relative to the reference. Our scope contract covers AIG optimization equivalent to the `resyn2` script, priority-cut 6-input LUT mapping for FPGAs, and standard-cell mapping against the SkyWater 130nm (SkyWater Technology Foundry and Google, 2020) and ASAP7 (Clark et al., 2016) libraries. The suite has 82 circuits from five sources, including the EPFL (Amarù et al., 2015) and ISCAS/ITC99 (Brglez and Fujiwara, 1985; Corno et al., 2000) collections, ranging from 174 to 804k AIG nodes, initially split into 41 development and 41 held-out cases. The pass gate checks two things on every case: (i) the output netlist must be proven formally equivalent to the input by the pinned ABC's combinational equivalence checker within a time budget, and (ii) every QoR metric must be within 1.03× of the pinned ABC's reference result, including AIG node count and logic levels after optimization, LUT count and levels after FPGA mapping, and area and delay after standard-cell mapping.

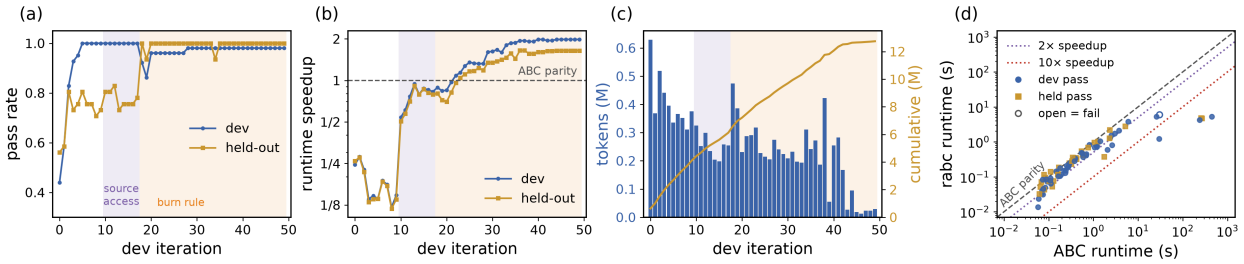


Figure 5: Rebuilding ABC from scratch with Claude Code and Fable 5 in Rust. (a) Pass rate on the development and held-out sets over the 50 iterations. (b) Geometric-mean runtime speedup over ABC on all measured cases of the two sets (failing cases enter at a capped ratio in early iterations); the dashed line marks parity. (c) Output tokens produced per iteration (bars, left axis) and cumulative output tokens (curve, right axis, in millions). (d) Per-case wall time of the rebuilt tool against ABC at the final measurement; the dashed line marks parity, the dotted lines 2× and 10× speedup, and the open marker is the one failing case. In (a)-(c), the lavender band begins at iteration 10, when the agent is granted read access to the ABC source, and the orange region begins at iteration 18, when ten persistently failing held-out cases are burned.

Metric	#rabc better	#Tied (±0.1%)	#rabc worse	#Cases
AIG node count	43	18	21	82
AIG depth	33	44	5	82
LUT count (FPGA)	73	3	5	81
LUT depth (FPGA)	14	65	2	81
Area (ASIC)	24	0	6	30
Delay (ASIC)	28	0	2	30

Table 2: Quality of results of rabc against ABC at the final measurement, counting per-case wins with a strict 0.1 percent tie band (the pass gate allows 3 percent).

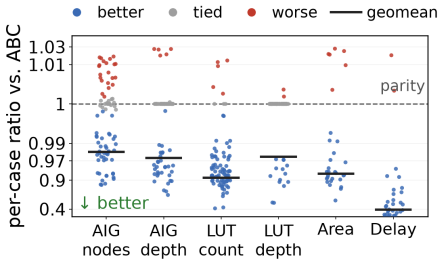


Figure 6: Per-case QoR ratios of rabc against ABC (log scale); horizontal bars mark the geometric mean of each metric.

The pinned reference throughout is ABC at commit `bcfdf59`; all golden outputs, equivalence verdicts, and paired timings come from this single build.

Figure 5 shows the trajectory. The campaign begins closed-book, and after nine iterations the held-out pass rate stalls around 75 percent while the rebuilt tool runs several times slower than ABC. At iteration 10, we instruct the dev agent through the supervisor that it has read access to the ABC source code, together with a directive to prioritize true runtime, and the runtime deficit collapses from 7.3× slower to 2.0× slower within a single iteration, as shown in Figure 5(b). What the dev agent learns from the given source code is unwritten engineering craft: ABC predicts whether a rewrite helps with cheap arithmetic instead of building and discarding it, keeps only a short list of the most promising rewrite candidates instead of examining all of them, and runs a fixed ten-pass optimization recipe instead of iterating to convergence. The trial-and-error against merely the output of black-box ABC never discovers this craft, while reading the source code exposes it immediately. Ten persistently failing held-out cases are burned at iteration 18 following our burn rule; with these cases now visible to the dev agent, the stuck failures are resolved over the following iterations. The single failing case, `leon3mp` (a ∼590k-node system-on-chip design), fails not because the rebuilt tool rabc produces a wrong answer but because the grader cannot check the answer. Namely, our pass gate requires the pinned reference ABC's own equivalence checker `cec` to formally prove the mapped netlist correct within a time budget, and on this circuit that checker never returns a verdict: it exhausts its fixed effort schedule or crashes. At the final measurement the tool passes 81 of 82 cases, is 2.0× faster than ABC on the development set and 1.6× faster on held-out, as shown in Figure 5(a), (b), and (d), and its quality of results is on average *better* than the reference: geometric-mean node count 0.98× and LUT count 0.91× of ABC (Table 2 and Figure 6).

The final tool rabc thus ends the campaign ahead of the reference on both fronts: runtime and quality of results. The runtime advantage traces to the implementation craft described above: predicting instead of building, pruning instead of enumerating, and a fixed recipe instead of iterating to convergence. The quality advantage comes not from a new algorithm but from where effort is spent: the adopted reconvergence-cut heuristics improve node quality as well as speed, and the mapping stage, which the runtime gate does not time, keeps a fuller cut-evaluation effort than ABC's default and therefore delivers smaller mapped netlists. The margin is widest on the ASIC backend: over the 30 standard-cell mapping cases, the mapped netlists average 0.93× of ABC's area and 0.38× of its delay (Table 2 and Figure 6). Over the whole campaign the dev agents consume 2.3 billion input tokens and produce 12.7 million output tokens.

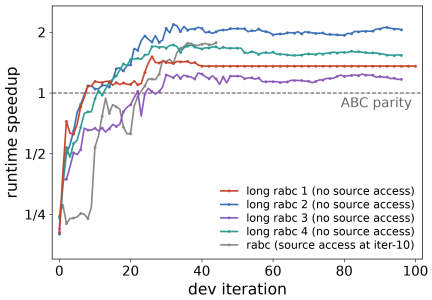


Figure 7: Runtime speedup over ABC per iteration for four independent closed-book runs (*long rabc* 1–4) and the source-access run, all on the 41 dev test cases. Final speedups: 2.05×, 1.53×, 1.36×, and 1.17× for the closed-book runs versus 1.77× with source access.

The source-access jump raises the natural question: is reading the ABC source code necessary at all for the agent? To answer this question, four independent runs of the same task are launched closed-book and never granted source access, each continued for 100 or more development iterations. Figure 7 overlays the runtime-speedup trajectories of these four runs and the source-access run on the 41 development cases. As shown in Figure 7, source access is not necessary for the endpoint: every closed-book run reaches full pass rate on its development set and ends faster than ABC on it, and the best of them (*long rabc* 2, 2.05×) exceeds the source-access run's 1.77× while crossing parity sooner. Whatever the source code provides is, in principle, rediscoverable by simply spending more tokens. Moreover, the outcomes of the closed-book runs are essentially a lottery: the four replicates end at 2.05×, 1.53×, 1.36×, and 1.17×, a factor of 1.8 across otherwise identical campaigns, with no way for us to know in advance which draw one has purchased, whereas the source-access run escapes its closed-book plateau in essentially one step.

Results across remaining seven tools. Figure 8 shows the four remaining campaigns that succeed, and Figure 9 the three that expose the limits of this regime. The four tools of Figure 8 all have exact or closely approximable ground truth, and end at essentially full held-out pass rate with a large speedup over their references:

- **rvsim** produces byte-exact output on 114 of 114 held-out cases[1] at 3.0× the speed of its frozen reference, a pinned `iverilog` v13 build of Icarus Verilog (Williams and Baxter, 2002). The headline is the geometric mean of per-case runtime ratios over all 270 passing cases in Figure 8a, and rvsim is faster than the reference on 259 of them. The speedup is largely architectural: a single native binary that parses, elaborates, and simulates in one process, versus `iverilog`'s compile-then-interpret pipeline, with four-state values kept in packed bit-planes with single-word fast paths so common two-state activity avoids per-bit bookkeeping. At the end of the campaign rvsim still fails 5 of the 161 development cases, all output mismatches of three kinds: two cases hinge on how `iverilog` propagates unknown (X) values, two reproduce only under `iverilog`'s exact internal event-scheduling order, which a completed oracle bisect shows no simple scheduler model matches (the fix would be a full scheduler rewrite, declined with held-out already perfect), and one case stays undiagnosed because the protocol reveals only output digests to the dev agent, not full waveforms.
- **rsta** matches OpenSTA (Cherry, 2019), pinned at version 3.1.0 and commit `08cf90d3fd`, within 1 ps on 45 of 47 held-out cases[2]; the two fails are reference-side crashes rather than mismatches, and the worst held-out error is 0.071 ps. It is 13.3× faster as a geomean over all 100 dev and held cases, with both

[1]Table 1 lists the pre-burn split (152 development / 123 held-out); the burn rule promotes 9 persistently failing held-out cases into the development set during the campaign, and final results are reported on the resulting 161/114 split.

[2]Table 1 lists the pre-burn split (50 development / 50 held-out); the burn rule promotes 3 persistently failing held-out cases into the development set, and final results are reported on the resulting 53/47 split.

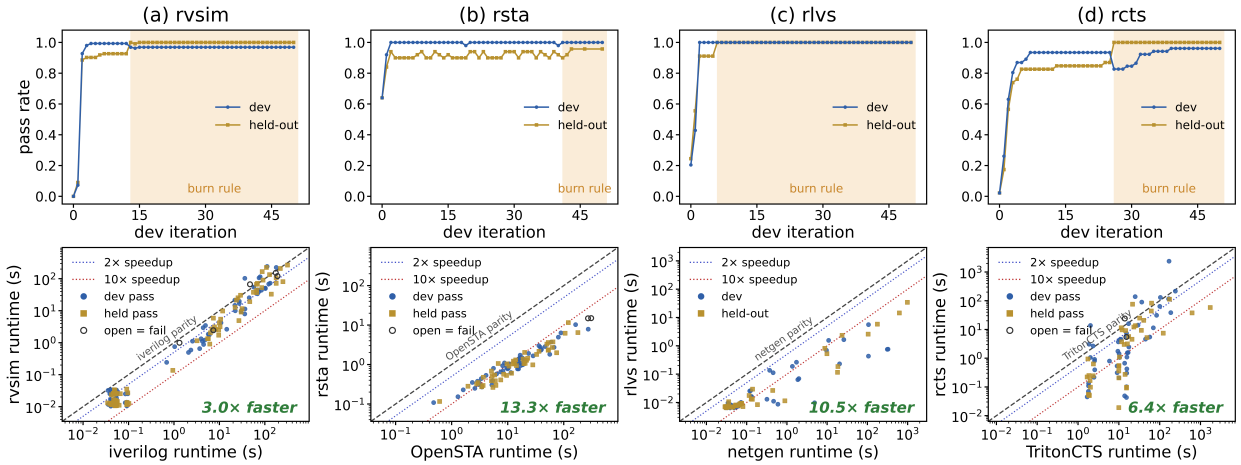


Figure 8: The four remaining campaigns that succeed: rvsim, rsta, rlvs, and rcts. Top: development and held-out pass rate per iteration, with the shaded window marking the period in which the burn rule applied. Bottom: per-case wall time of the rebuilt tool against its reference at the final measurement; the green label gives the geometric-mean speedup on passing cases, and the diagonal marks parity. For rsta the scatter and its geomean include the two officially failed cases: their reference-side crashes are thread-related and do not occur in the 1-thread timing runs.

tools at 1 thread. The gain comes mainly from replacing the Tcl-interpreted front-end with a compiled, interpreter-free path, worth roughly 3.5–4.5×, and from a flat, cache-friendly compressed-sparse-row (CSR) graph layout in place of pointer-linked objects, worth another 2–3×; allocation discipline adds roughly 1.3–1.6× and link-time plus profile-guided optimization a further 1.1×. No work is skipped: timed runs emit full worst-K reports cross-checked against the accuracy run at 1 ps.

- **rlvs** matches Netgen (Sivilotti and Edwards, 2019), pinned at version 1.5.323, on all 94 cases, 53 of 53 development and 41 of 41 held-out after the burn rule promotes four chronic cases at iteration 6 (49/45 at the start), under a checker requiring Netgen's exact verdict, discrepancy counts, and discrepancy sets up to circuit automorphism, with a 10.5× per-case geometric-mean speedup. Whole-suite totals are far better still: the development suite runs in 1.4 percent of Netgen's time, because the speedup grows with design size, from about 7× on sub-second cases, bounded by per-invocation overheads, to 30–170× on the million-device cases that dominate suite time. The comparison algorithm is Netgen's own, reproduced bug for bug; the speedup is implementation-level: a static single binary with no interpreter layer, cache-conscious data structures, and linear hashed scans where quadratic behavior emerges on large netlists.
- **rcts** passes 40 of 40 held-out cases and, over all passing benchmark cases, runs 6.4× faster than TritonCTS (Han et al., 2020), run inside the pinned OpenROAD binary at `26Q3-556-g7fe21ca27c`, under interleaved paired timing. The campaign's own decomposition is candid that this headline mixes two effects. Part is structural: rcts is a standalone binary, while the reference launches TritonCTS inside full OpenROAD and pays 2 to 15 s of startup per case, which dominates the wins on the SkyWater 130nm and ASAP7 decks. The rest is a genuine algorithmic advantage of about 8.5× on contest-dialect decks, later widened by a capacitance-first scoring order and a construction-time abort that skip full delay evaluation of electrically illegal candidates. On million-sink cases, rcts is 2.5× to 4.5× slower than the reference's marginal work. At the end two development cases, `adaptec2` and `adaptec3`, still fail, both on the skew band alone. Their bars derive from reference trees built without the per-driver load cap that rcts must satisfy: meeting them at near-zero skew would take 3.38× and 1.72× of the cases' capacitance budgets, and an exhaustive sweep of the tool's move classes at the 1.0× cap finds no legal tree that crosses the bar, the closest coming within 3.4 ps.

The three campaigns of Figure 9 are a different story, worthy of detailed discussion:

- **rplace has huge run-to-run variance.** Five independent campaigns on the identical frozen suite, referee, and seed end anywhere from 49 of 70 cases after 28 iterations (21 of 34 development, 28 of 36 held-out) to 65 of 70 within 3 (33 of 34 and 32 of 36, with wirelength 6 percent shorter than OpenROAD's, pinned at `26Q3-556-g7fe21ca27c`, on both splits; the best run in Figure 9a). The spread is set at the first submission: iteration-0 scores range from 0 of 70 to 58 of 70, already above where the 28-iteration run ends. In the 0-of-70 run the placements are nearly legal, but the agent tests them with a legality checker it wrote itself, whose definition of when two cells overlap differs slightly from the referee's; the agent therefore submits placements it believes legal, and the referee, which accepts nothing less than zero violations, rejects every case. Moreover, paired runs given identical prompts differ by more than runs given different ones. The variance is architectural rather than incremental: four of the five runs independently commit to an electrostatic-field global placer within their first hours, and the one that opens with a purely quadratic formulation trails until it rebuilds onto the same engine class.
- **rrouter is genuinely hard.** The reference is the router of the OpenROAD build pinned at `26Q3-556-g7fe21ca27c`, and the gate is unforgiving: a case passes only with zero opens, zero shorts, and zero design-rule violations at signoff DRC, with wirelength within $1.05\times$ and via count within $1.10\times$ of the reference. Over a 50-iteration campaign we escalate through every aid we can offer, each marked in Figure 9b: an algorithm architecture hint when the pass rate first goes flat, read access to the reference source, and finally the referee's per-violation marker coordinates on development cases. The agent builds a two-stage global-plus-detailed router with negotiated congestion, in-loop design-rule costs, referee-calibrated estimators for the five dominant violation classes, and a marker-driven repair engine; these cut development-set opens from a peak of 4,560 to 1,869 and design-rule violations from an initial 1.78 million to about 20K, with one late repair upgrade removing more violations on held-out cases than on dev. Yet the pass rate barely moves: it plateaus at 17 of 36 development and 7 of 36 held-out cases, because passing requires driving every count on a case to exactly zero, and the residual defects concentrate on the dense designs where the last few hundred conflicts are hardest. At the end every development-set failure is opens-only, with zero shorts and zero timeouts on both splits.
- **rmagic does "reverse engineering".** The geometry and DRC tiers reach 50 of 58 and 35 of 44 held-out cases, against Magic pinned at version 8.3.675, and the campaign fully decodes Magic's extraction model itself: rather than a field solver, Magic computes parasitics with closed-form, grid-quantized area, perimeter, and sidewall-coupling rules, each of which is recovered, verified to sub-attofarad agreement on purpose-built probe layouts, and cited to a technology-file entry or source line under a strict no-fitting charter. The remaining barrier lies downstream of extraction: the reference flow's resistance pass splits each wire into an RC network and re-apportions its capacitance among the resulting segments, and this apportionment, recorded nowhere in the extraction output the reimplementation targets, accounts for every remaining extraction failure. As a result, 13 of 14 held-out extraction cases fail, capping the all-tiers rate at 37 of 58 despite a $12.8\times$ speedup on passing cases. Notably, the held-out pass rate ends above the development rate (0.638 vs. 0.519): because fixes must explain mechanisms rather than fit outputs, they generalize; seven held-out cases, including the one passing extraction case, flip without ever being examined.

Compare different LLM agents. To assess sensitivity to model choice, we repeat a small subset of the experiments in this section using Codex powered by GPT-6 Astra. These runs lead to conclusions consistent with those reported above, while suggesting that GPT-6 Astra requires fewer tokens. We caution, however, that frontier models are evolving rapidly, with new versions often released every two to three months. Consequently, the quantitative results reported here may quickly become outdated, and future models are likely to achieve substantially stronger performance.

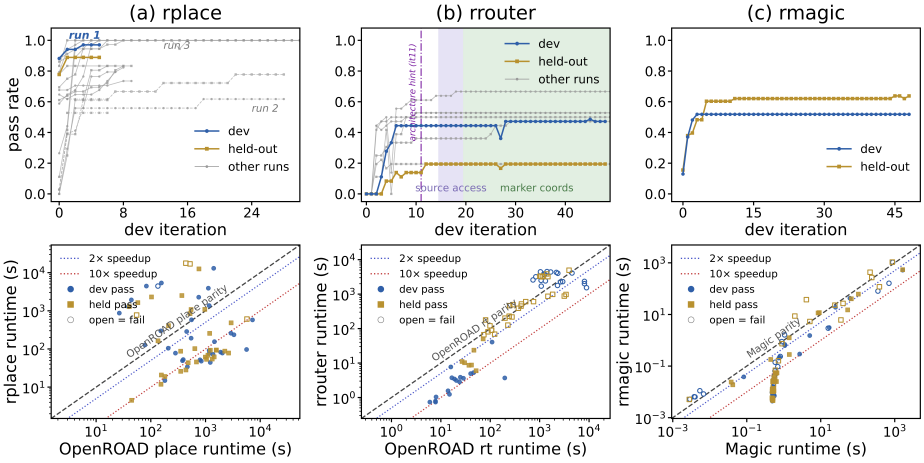


Figure 9: The three campaigns that remain hard: rplace, rrouter, and rmagic. Top: pass rates as in Figure 8; for rplace and rrouter the gray curves are independent replicate runs, and for rrouter the marked events are an architecture hint at iteration 11, source access, and marker-coordinate hints. Bottom: per-case wall times at the final measurement, with failing cases drawn as open markers.

4 Regime II: Optimize GPU-Based EDA Tools

The first regime starts each tool from nothing and asks the agent to reach the reference. In the second regime, the agent instead starts from a mature, already GPU-accelerated tool, with full source code, and is asked to make it faster without changing what it computes.

Experimental setup overview. We select three open-source GPU-accelerated tools from three stages of the digital flow: DREAMPlace (Lin et al., 2019) for global placement, InstantGR (Lin et al., 2024) for global routing, and GEM (Guo et al., 2025b) for RTL simulation. Each tool is frozen at a pinned commit, which serves as the baseline for all measurements: DREAMPlace at release 4.3.0 (commit `2aea7658`), InstantGR at commit `cfce09f`, and GEM at commit `9e913f9`. The benchmark for each tool is the design set used in its own publication: for DREAMPlace, 47 designs drawn from the ISPD 2005 and 2006 placement-contest suites (Nam et al., 2005; Nam, 2006), the ISPD 2015 fence-region suite (Bustany et al., 2015), and the MMS mixed-size suite (Yan et al., 2009); for InstantGR, the 11 designs of the ISPD 2024 global-routing contest (Liang et al., 2024); and for GEM, the 6 processor designs of its own release (Guo et al., 2025b). The pass gate requires that the optimized tool produce the same result as the frozen one: for placement, a legal result with half-perimeter wirelength within 1.005× of the baseline on every design; for routing, a legal result within 1.002×; for simulation, an identical simulation outcome. The score is the geometric mean over designs of the paired runtime ratio of the frozen tool to the optimized tool, where each design is run in the same container in both orders so that warm-up effects cancel. To measure how much of the outcome depends on the agent rather than the task, every tool is optimized independently by three models of the same family at different capability tiers (Fable 5, Opus 4.8, and Sonnet 5) on two GPUs (A100 80GB and H100 80GB HBM3), for 18 isolated chains; no chain may read another's notes, source, or measurements.

Results. Figure 10 shows the banked speedup of every chain over its iterations. All 18 chains improve their tool with every correctness gate intact. The best chain reaches 2.0× on DREAMPlace, 1.5× on InstantGR, and 1.3× on GEM; the median chain reaches 1.7×, 1.4×, and 1.2× respectively, so the range of roughly 1.5 to 2× quoted in the abstract describes placement and the best chains rather than the typical outcome on every tool. The gains are largest on placement, whose runtime is dominated by a handful of CUDA kernels that the agent can profile and rewrite, and smallest on RTL simulation, where the frozen tool is already the product of a recent GPU-specific redesign. None of the gains involves a new algorithm. Across chains the winning levers fall into a few recurring categories: removing host–device synchronization points and redundant metric transfers from the optimization loop, caching temporary buffers and CUDA streams across iterations

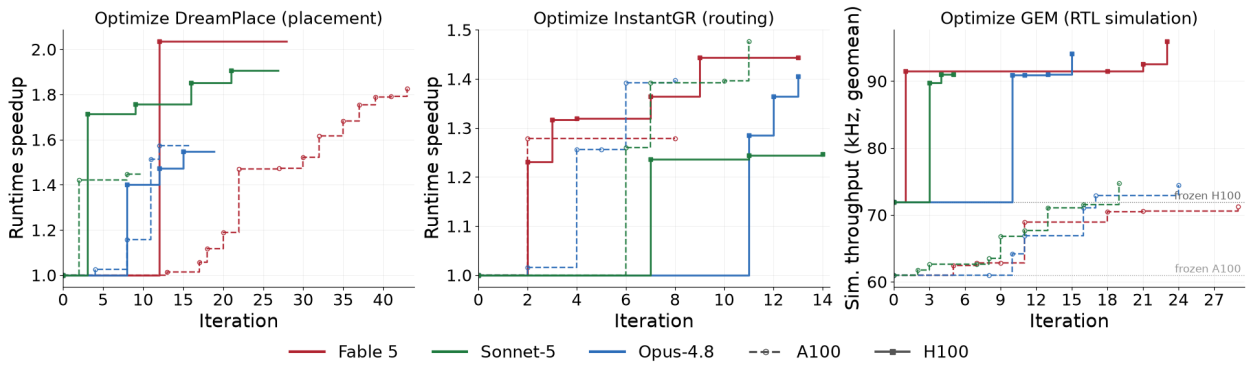


Figure 10: Banked progress over the frozen tool as a function of iteration for the 18 chains at completion: three tools, three models (colors), two GPUs (solid with filled squares: H100; dashed with open circles: A100). The placement and routing panels show runtime speedup; the simulation panel shows simulated throughput in kHz (geometric mean), with the frozen baselines as dotted lines. Each marker is a banked, gate-validated improvement; each curve extends flat to the chain's final scored iteration, so iterations after the last marker produce no further banked gain. Final per-chain values appear in Figure 11 and Tables 3–4.

instead of reallocating them, correcting kernel launch bounds and thread counts to the actual occupancy of the target GPU, and eliminating allocator churn and serial parsing on the host side. The ordering across models is not strict. Fable 5 holds the best result on two of the three tools, but Sonnet 5 holds it on routing (1.5× on the A100) and comes within 0.13× of Fable 5 on placement, and on two tools the same model ranks differently on the two GPUs. The curves also show two failure modes of the setting. Chains frequently stall for ten or more iterations with no banked improvement while the agent pursues a lever that does not survive the paired measurement, and development-time speedups measured in a single container routinely overstate the validated result: one placement chain records 3.2× in its own log before health-gated re-measurement in fresh containers settles at 2.0×. The health gate and the minimum-of-two rule exist precisely to remove this container luck from the score, and we regard them as essential for any speedup claim on shared GPU infrastructure.

Per-design runtimes. Figure 11 and Tables 3–4 report, for every benchmark design, the measured runtime of the frozen tool and the runtime of the agent-optimized tool on both GPUs. GPU runtimes vary from container to container, so we measure the two tools back to back in the same container and report the optimized runtime through the per-design ratio of the pair, which container variation largely cancels: the optimized seconds shown are the frozen runtime divided by this measured ratio. The same container variation means that the per-iteration numbers of Figure 10, banked during development, and the final measurement taken after development finishes can differ slightly (placement H100: 2.04× banked versus 1.97× final); we report both rather than letting either stand in for the other.

What the per-design data show. Three patterns in the per-design data sharpen the aggregate numbers. First, the placement gains are strongly structured by benchmark family: on the H100 the fence-region ISPD2015 designs average 2.2× and reach 3.1–4.2× on the five largest (`mgc_des_perf_b`: 46.7 s → 11.1 s), because the frozen placer re-derives static fence-region membership every iteration and the agent's cached-membership lever removes that entire term; on the A100 the same chain's largest wins are instead on the MMS giants (geometric mean 2.1×; `mms_newblue5`: 85.3 s → 31.6 s), where its allocation-pooling and batched-transfer levers dominate. A speedup harvested from a structural inefficiency is therefore not one number but a distribution over designs, and quoting only the geometric mean understates how uneven the harvest is. Second, the routing gains concentrate where routing is actually expensive: the two `mempool_group` giants improve by 1.4–1.5× on the H100 (taking a full 11-design sweep from 151 s to 106 s), while the sub-5-second `ariane` and `mempool_tile` designs sit near 1.3×, limited by fixed host-side costs that the levers cannot remove. Third, the simulation table shows the opposite regime: GEM's kernel time on the three smallest designs is already below 1.2 s, so a large fraction of each run is launch and transfer floor, and the uniform

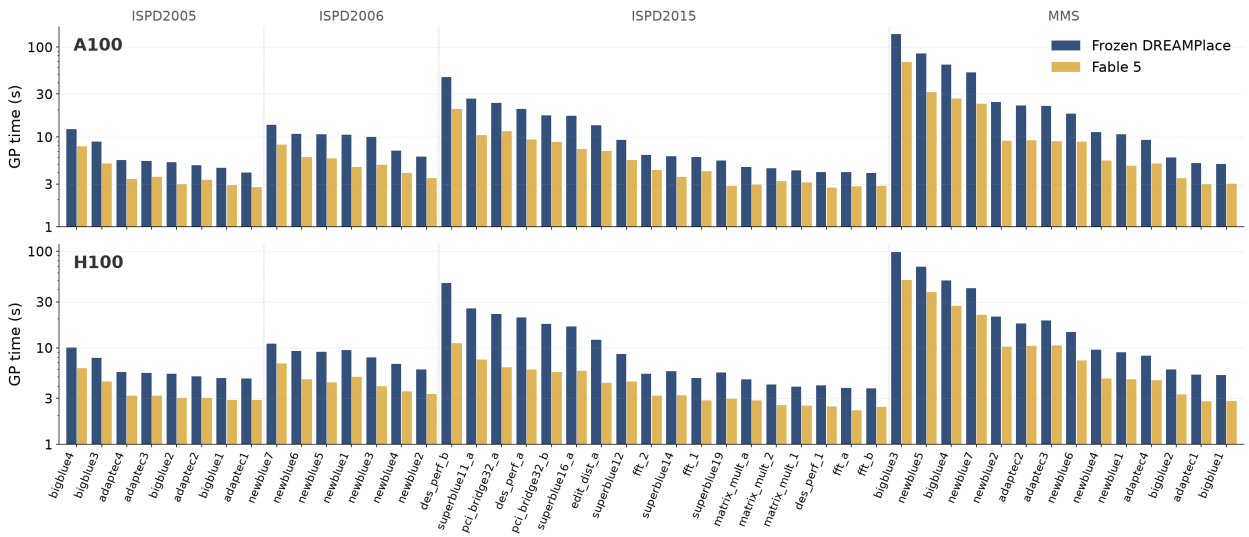


Figure 11: DREAMPlace per-design global-placement time on the 47-design suite: the frozen baseline against the best agent-optimized chain per GPU (Fable 5 on both), on A100 (PCIe, top) and H100 (bottom), log scale, grouped by benchmark suite. Geometric-mean speedup: 1.8× (A100), 2.0× (H100).

		A100, wall time (s)				H100, wall time (s)			
Design	Reported (4×A800, s)	Frozen	Fable 5	Sonnet-5	Opus-4.8	Frozen	Fable 5	Sonnet-5	Opus-4.8
mempool_group	35.78	76.75	64.12 (1.20×)	62.40 (1.23×)	60.30 (1.27×)	49.20	35.30 (1.39×)	40.05 (1.23×)	39.59 (1.24×)
mempool_group_rank	36.35	68.17	54.26 (1.26×)	52.21 (1.31×)	53.13 (1.28×)	46.18	30.35 (1.52×)	38.76 (1.19×)	34.54 (1.34×)
bsg_chip	14.64	20.29	16.52 (1.23×)	16.76 (1.21×)	14.60 (1.39×)	14.04	10.11 (1.39×)	11.19 (1.25×)	10.00 (1.40×)
bsg_chip_rank	9.45	14.91	10.43 (1.43×)	8.80 (1.69×)	7.56 (1.97×)	10.45	6.25 (1.67×)	7.80 (1.34×)	5.63 (1.85×)
nvdla_rank	3.83	7.16	6.32 (1.13×)	5.42 (1.32×)	5.05 (1.42×)	5.50	4.15 (1.32×)	4.25 (1.29×)	4.14 (1.33×)
nvdla	3.73	6.59	5.59 (1.18×)	3.53 (1.87×)	4.54 (1.45×)	5.97	4.39 (1.36×)	4.70 (1.27×)	4.57 (1.31×)
ariane133_51	—	5.57	5.23 (1.07×)	4.51 (1.23×)	4.36 (1.28×)	4.26	3.24 (1.32×)	3.86 (1.10×)	3.63 (1.17×)
ariane_68_rank	2.98	5.25	4.70 (1.12×)	3.69 (1.42×)	4.13 (1.27×)	4.35	3.35 (1.30×)	3.75 (1.16×)	3.64 (1.20×)
mempool_tile_rank	2.87	5.23	4.51 (1.16×)	3.69 (1.42×)	3.77 (1.39×)	3.88	3.06 (1.27×)	3.38 (1.15×)	2.97 (1.30×)
mempool_tile	2.56	4.81	4.20 (1.14×)	3.50 (1.37×)	3.88 (1.24×)	3.49	2.81 (1.24×)	3.16 (1.11×)	2.56 (1.36×)
ariane133_68	2.55	4.65	4.02 (1.16×)	3.57 (1.30×)	3.46 (1.35×)	3.78	2.96 (1.28×)	3.28 (1.15×)	2.82 (1.34×)

Table 3: InstantGR per-design route wall time (s) on the 11 designs of the ISPD 2024 contest (Liang et al., 2024): paper-reported runtime, our frozen baseline, and each model's optimized tree (speedup vs. frozen in parentheses). Reported column: the runtimes reported in the InstantGR paper (Lin et al., 2024), measured by its authors on 4× NVIDIA A800; "—" where the paper does not report the design. Our values are from each chain's final-validation minimum leg (A100-SXM4 / H100), so chain aggregates sit 1–7% below the banked trajectories of Figure 10.

1.4–1.5× H100 gains on `gemmini`, `rocket`, and `nvdla` come from synchronization-elimination levers whose benefit scales with kernel count rather than design size. Summed over each tool's full scored field on the H100, the optimized tools cut total wall time from 700 s to 338 s (placement), 151 s to 106 s (routing), and 52 s to 36 s (simulation).

Case study: DREAMPlace. The placement chain of Fable 5 on the H100 illustrates the working pattern. The agent begins by profiling the banked build with the framework's own profiler on four representative designs and finds that no single kernel dominates: on a small design the density and gradient kernels share the step, on a high-fanout design the wirelength kernel alone takes a third of the step because it assigns one thread per net and serializes a whole warp on any high-degree net, and on a macro-heavy design the force kernel dominates because macros touch thousands of density bins. The agent then mines its own earlier measurement legs per design and finds that a warp-cooperative force kernel it banked in a previous

Design	Reported (A100, kHz)	A100, throughput (kHz)				H100, throughput (kHz)			
		Frozen	Fable 5	Sonnet-5	Opus-4.8	Frozen	Fable 5	Sonnet-5	Opus-4.8
gemmini	25.6	24.5	29.9 (1.22×)	31.0 (1.26×)	30.7 (1.26×)	31.1	45.7 (1.47×)	46.1 (1.49×)	45.3 (1.46×)
rocket	52.4	57.0	61.3 (1.07×)	64.8 (1.14×)	64.6 (1.13×)	54.8	77.6 (1.42×)	82.8 (1.51×)	80.2 (1.46×)
nvdla	65.4	59.1	67.9 (1.15×)	68.1 (1.15×)	69.2 (1.17×)	62.9	72.6 (1.15×)	82.5 (1.31×)	90.7 (1.44×)
picorv32†	—	90.4	122.2 (1.35×)	121.5 (1.34×)	123.4 (1.36×)	114.8	158.7 (1.38×)	125.3 (1.09×)	169.9 (1.48×)
vexriscv†	—	82.9	93.8 (1.13×)	97.8 (1.18×)	98.2 (1.18×)	106.6	112.0 (1.05×)	117.4 (1.10×)	125.2 (1.17×)
vexriscv_full†	—	83.2	92.3 (1.11×)	97.1 (1.17×)	97.1 (1.17×)	105.7	120.9 (1.14×)	116.0 (1.10×)	124.7 (1.18×)

Table 4: GEM per-design simulation throughput in kHz (simulated cycles per second; higher is better; speedup vs. frozen in parentheses). Reported column: the throughput reported in the GEM paper (Guo et al., 2025b) (Table II), measured by its authors on an A100 (bus variant unstated) with their own test programs (context, not a same-workload baseline); the three expansion designs are not in the paper. Our frozen A100 basis is SXM4 for gemmini/rocket/nvdla and PCIe for the expansion designs (†); our values are from each chain's final-validation minimum leg (for Fable 5 on the H100 this is the degraded archived-binary re-test discussed in the text).

wave wins on macro-heavy designs by up to 1.7× but loses on large standard-cell designs by up to 20 percent, with the number of impacted bins per node as the discriminating variable. The resulting lever dispatches each node to whichever kernel suits it, with a threshold on impacted-bin count, and writes each node exactly once so that the wirelength gate holds by construction. Together with a fused wirelength kernel that handles low- and high-degree nets on separate paths, this lifted the bank from 1.9× to 2.1×. The episode is representative of the regime as a whole: progress comes from measurement, attribution, and a targeted fix, not from rewriting the placer.

5 Regime III: Rebuild the Whole Digital Design Flow from Scratch

The third regime asks for the whole toolchain at once. Rather than one tool at a time against a fixed reference, the agent must build a complete RTL-to-GDSII flow whose tools work with one another, and the flow is then judged by the only standard that matters: whether an agent can use it to design chips. The target technology is the SkyWater 130nm PDK (SkyWater Technology Foundry and Google, 2020), and the reference is the OpenROAD flow (Ajayi et al., 2019) with its companion sign-off tools.

Experimental setup overview. We set up our experiments to make the human as hands-off as possible. Development proceeds in three phases. First, in a survey phase, the agent studies the standard digital chip design flow and itself decides how to decompose it: how many tools to develop, at a granularity it believes will be manageable, together with a per-tool verification benchmark for each. Its decision is a plan of 15 standalone tools, each to be written in Rust as its own binary, communicating through the standard open formats (Verilog, DEF/LEF, Liberty, SDC, SPEF, GDSII). Second, the 15 tools are developed independently, with multiple dev agents working in parallel, each against its tool's verification benchmark. Third, the campaign moves to a *deployed* phase that proceeds in numbered loops over three tiers of benchmark designs (Figure 12): in each loop, agents design chips with the assembled flow on a managed evaluation platform, every failure is audited after the wave, and a gated repair process improves both the flow and the evaluation harness itself.

To judge the flow in this deployed phase, a design agent, Claude Code with Fable 5, is given a natural-language specification (for example, a Reed-Solomon decoder with decoded throughput of at least 1.83 Gb/s in a die of at most 34,300 μm^2), a toolchain, and a time budget, and must deliver a signed-off GDSII layout: a submission comprises the final GDSII together with the matching gate-level netlist, the pair on which every sign-off gate is graded. Six evaluation lanes run every loop (Table 5): the three benchmark tiers, each run twice, once by an agent whose sandbox contains only the agent-developed native flow (lanes B1–B3) and once by an agent whose sandbox contains only the open-source reference stack (lanes B4–B6); at design time the agent thus has exactly one of the two toolchains available. Each task is attempted three times

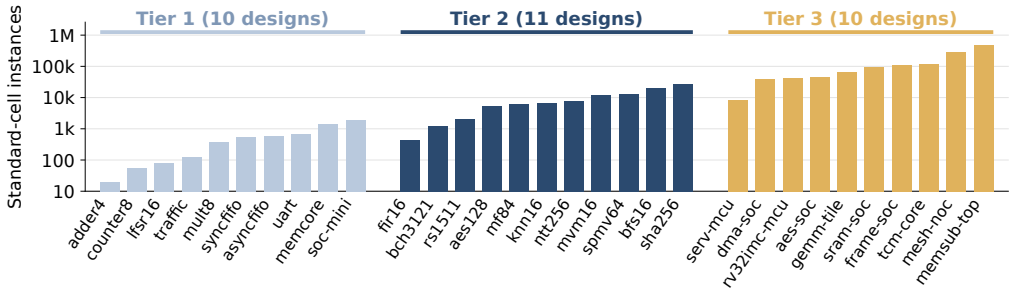


Figure 12: The three tiers of whole-flow benchmark designs, by post-synthesis standard-cell count under the SkyWater 130nm PDK. Tier 1 and Tier 2 are used for the head-to-head evaluation against the reference flow; Tier 3 probes the limit of both flows.

Lane	Tier	Designs	Toolchain in sandbox	Runs per loop
B1	1 (small blocks)	11	native flow (15 Rust tools)	11×3
B2	2 (medium designs)	12	native flow	12×3
B3	3 (large SoCs)	10	native flow	10×3
B4	1 (small blocks)	11	reference (Yosys + OpenROAD)	11×3
B5	2 (medium designs)	12	reference	12×3
B6	3 (large SoCs)	10	reference	10×3

Table 5: The six design-agent evaluation lanes. Twin lanes differ in exactly one variable (which toolchain the agent's sandbox contains) and share tasks, briefs, budgets, grading, and attempt count. Tier membership follows Figure 12; a trivial inverter control task is included in Tiers 1 and 2.

independently; an attempt is a long interactive session, capped at 12 hours, in which the agent may inspect any intermediate result and rerun any stage. At grading time, a submission is scored in $[0, 1]$. Half of the score comes from the five hard sign-off gates (functional simulation, logical equivalence against the RTL, LVS, DRC, and static timing), each checked twice, once by the agent-developed native toolchain and once by the open-source reference toolchain, giving ten verdicts worth 0.05 each; the other half is the mean of the specification's figures of merit (performance, area, power), awarded only when all ten verdicts pass:

$$\text{score} \;=\; \frac{1}{2}\cdot\frac{n_{\text{pass}}}{10} \;+\; \frac{1}{2}\cdot\mathbb{I}\big[n_{\text{pass}}=10\big]\cdot\frac{s_{\text{perf}}+s_{\text{area}}+s_{\text{power}}}{3}, \tag{1}$$

where n_{pass} is the number of passing verdicts, $s_{\text{perf}}, s_{\text{area}}, s_{\text{power}} \in [0, 1]$ score the figures of merit, and $\mathbb{I}[\cdot]$ is the indicator function. A design therefore earns above 0.5 only by passing every sign-off check.

This deployed phase is operated by a small set of specialized agents. A *supervisor agent* owns the campaign: it decides which changes enter each loop, briefs the other agents, and alone launches evaluation waves. *Dev agents* develop and repair the flow's tools between waves. When a development job finishes, a *release agent* verifies that the changed tools build and behave as intended and that the documentation matches, then seals the container image for the next wave under the version pinning and hash manifests discussed later. After a wave completes, a *triage agent* collects every run's verdicts and error messages, flags anomalies such as platform failures, and reports to the supervisor, which folds the feedback into the next loop's briefs. The campaign advances by iterating this loop.

A common issue in agent-based automated experiments is that the setting the agents actually face can drift from what the experimenters believe it to be, and that agents under pressure to reach a goal can exhibit anomalous, reward-hacking behavior. Our experiments are no exception, and their surface is unusually large, spanning many interacting tools, file formats, and checkers. In the following, we discuss some considerations of our experimental design that make the evaluation more verifiable and more robust to hacking:

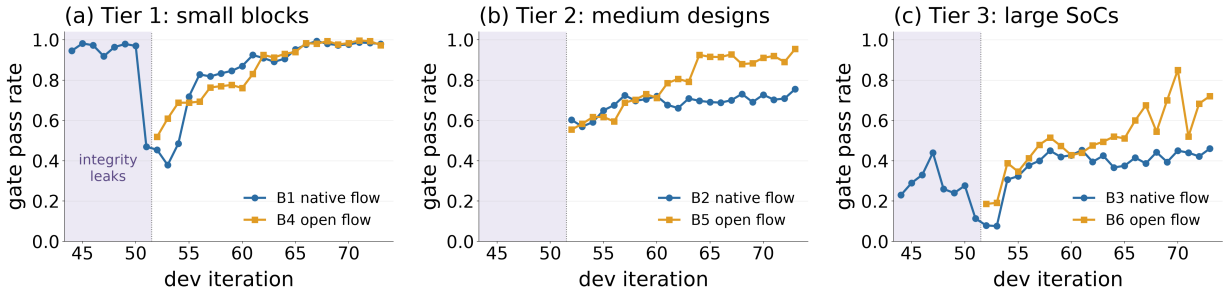


Figure 13: Mean fraction of sign-off gates passed by development loop for the agent-developed native flow (blue, lanes B1–B3) and the open-source reference flow (orange, lanes B4–B6) on the three benchmark tiers. The shaded band marks the loops during which the sandbox leaked the hidden testbenches and grading tools to the agent; the leaks are sealed at loop 51, and the reference lanes exist from loop 52. Tier-3 means are over few scored runs per loop and carry the platform-failure caveat of the text.

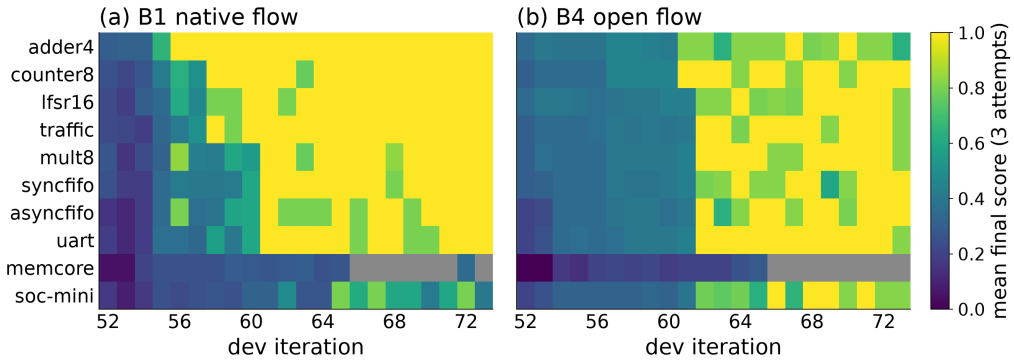


Figure 14: Per-design mean final score (three attempts) by development iteration for the Tier-1 lanes B1 (native flow, left) and B4 (open flow, right). Rows are the ten Tier-1 designs of Figure 12; grey cells mark iterations with no scored runs for the SRAM design memcore.

- **Twin lanes.** Twin lanes refer to B_i and B_{i+3} for $i \in \{1, 2, 3\}$ in our case. The two lanes of a twin pair differ only in which design toolchain the agent's sandbox contains; the tasks, briefs, budgets, grading, and attempt count are identical. Whatever ceiling the design agent itself imposes (specification misreadings, weak floorplanning instincts, budget exhaustion) therefore appears in both lanes: a score gap between twins is attributable to the tools, not the agent, and a movement over loops that appears in one lane but not its twin is attributable to whichever side changed that loop.
- **Every sign-off gate is checked twice.** Every sign-off gate is graded *twice* on the same submitted layout: once by the native flow's own sign-off tools, and once by the open-source checkers. Each run therefore carries a ten-verdict card (5 gates $\times$ 2 checker flanks). Agreement between two independently implemented checkers is strong evidence the verdict reflects the design; disagreement is itself a measurement that localizes a defect and tells us where to look.
- **Version pinning and hash manifests.** Every component of the serving surface, from the tool binaries to the sandbox setup scripts, is version-pinned and content-hashed, and each loop's configuration is recorded as a manifest of those hashes. A change enters only as an explicit, separately hashed delta over the previous loop, with everything else carried byte-verbatim, and the bytes actually served on the evaluation platform are re-verified against the pinned manifest each loop, so exactly which code graded which wave is reconstructable after the fact.

Results and the learned lessons. Figure 13 plots the mean fraction of the ten grading gates passed (five native-instrument and five open-instrument checks, applied identically to every submitted design; i.e., the

Lane	Runs scored	Mean score	Gate pass rate	Turns per episode	Tokens per episode	Mean retries	Episode time	Total tokens
B1 (native)	33 (11×3)	0.960	0.984	103.4	142.8k	1.24	51.6 min	4.7M
B4 (open)	33 (11×3)	0.963	0.993	57.5	74.2k	1.00	35.1 min	2.2M

Table 6: Tier-1 head-to-head between the twin lanes at development loop 72: totals and per-episode means over scored runs. An episode is one attempt at one design task, i.e., a single interactive session of the design agent from brief to submission. The two lanes reach the same quality, while the native-flow agent spends roughly twice the turns, tokens, and wall-clock time per episode.

value of $n_{\text{pass}}/10$ in Equation 1) for the agent-developed native flow (blue) and the open-source reference flow (orange) over development loops 44–73. This deployed phase is itself mainly set up and operated by a supervisor agent over a long horizon; although the supervisor can clear its context and resume, the resulting experimental setup is in many ways complicated, and unintended configurations can arise. The author reads the agents' transcripts roughly once a day, and one such reading uncovers a leakage that is not intended by our design: in lanes B1–B3 the design sandbox is meant to contain only the native flow, yet the open-source grading toolchain is also present on the sandbox PATH. The shaded band marks the loops during which these leaks are present, and the open mirror lanes exist only from loop 52.

In Figure 13(a), Tier 1 (small blocks), sealing the leaks at loop 51 collapses the native lane from an apparent 0.97 to an honest 0.38, after which successive loops of flow and grading-instrument repair rebuild it to around 0.98, converging with the open flow: both flows now saturate this tier, and their agreement across two independently instrumented flanks is the evidence that the recovered scores are real. Figure 14 resolves the same trajectory per design: after the seal, both lanes start from low scores across the design set, and the eight small designs saturate to full scores from around iteration 62 in both lanes, so the tier's recovery is broad-based rather than carried by a few designs. The two macro-bearing designs lag: the SoC design soc-mini reaches consistent full scores in the open lane from about loop 66 while remaining mixed in the native lane, and the SRAM design memcore is the one unsolved cell in both lanes, scoring low when it scores at all, with the grey stretches marking platform-failed runs.

In Figure 13(b), Tier 2 (medium designs), both flows improve steadily but neither reaches 1.0, and a gap opens from around loop 65, with the open flow at 0.90 to 0.95 against the native flow around 0.72. The bottleneck for both flows is the physical-design back end: the failing verdicts sit in the LVS, DRC, and STA sign-off gates downstream of placement and routing, while the front-end gates are rarely the limit. Not all of this failure mass reflects defective designs: a per-cell audit attributes roughly half of the open flow's native-flank failures to cross-flow instrument mismatches in our native graders, progressively cured era over era, with the remainder being genuine design defects. Tier 1's convergence of both flows to nearly 1.0 is the control showing the graders impose no global penalty on either flow, and the residual gap is consistent with the finding of Section 3 that placement and routing are the hardest tools to rebuild.

In Figure 13(c), Tier 3 (large SoCs), both flows sit far from saturation,[3] with noisy means between 0.4 and 0.7, and the dominant limit is not scoring but survival: SoC- and SRAM-macro-bearing runs frequently die in multi-hour platform setup or grading phases before producing verdicts, so per-loop means average over few scored runs. Runtime is a structural constraint at this scale. Each design attempt is capped at 12 hours, and a larger cap could plausibly raise Tier-3 scores; however, since the Tier-3 lanes B3 and B6 run in every development loop, enlarging the cap would make the wall-clock cost of the whole campaign unaffordable.

From these results, we highlight four important notes.

[3]The reference-lane result should not be interpreted as an intrinsic scalability limit of OpenROAD. Here, the 12-hour cap covers the entire interactive agent session; by comparison, a reported expert-configured ORFS run of the larger MegaBoom design takes approximately 15.3 hours on a 48-core machine (The OpenROAD Project, 2023). Moreover, in our setting, we do not provide an expert-configured ORFS flow; instead, the agent must bring up and tune each design from its specification.

- **B1 and B4 reach the same score at different cost.** Table 6 quantifies the converged Tier-1 regime at loop 72: the two lanes are indistinguishable in quality, with mean scores of 0.960 and 0.963 and gate pass rates of 0.984 and 0.993, but B1 spends roughly twice the agent effort (1.7× turns, 1.9× tokens, 1.35× wall time on the shared design set) to get there, and the gap reflects the maturity difference between the two flows rather than any difference in the agents or tasks. Take the asynchronous FIFO as an example: the open-flow agent finishes in 49 turns while the native-flow agent needs 103, spending the surplus on a multi-round DRC repair loop for `met3` spacing violations the native router leaves behind, on the immature flow's learning curve of documentation lookups and tool-refusal errors, and on defensive re-verification; all three are expected to shrink as the native flow matures.
- **Even tiny harness bugs can cost multiple dev iterations, so we gate every fix with a fast smoke test.** A missing tool or a grader-harness defect gets a fix the dev agent believes complete, yet the next evaluation wave fails again for a related reason: one DRC-environment bug takes four sub-iterations to fully cure (the fix lands in the wrong image, then a namespace difference, then a library-version skew, then a missing alias), and one integrity fix passes all checks in the test sandbox but never executes on the real fleet path. The economical answer is a minimal smoke carrier (we use a two-transistor inverter) that exercises the full flow and grading chain in minutes, so a wrong fix costs a smoke run instead of a full evaluation wave.
- **Comparing B2 and B5 separates real defects from instrument mismatches.** Both lanes run Tier 2, B2 with the native flow and B5 with the open-source flow, yet their failure signatures differ in shape. B2's failures are symmetric: when a gate fails, it usually fails on both checker flanks. A genuine metal-spacing violation or connectivity mismatch is a property of the layout itself, so two independent checkers that share no code both measure the same geometry and both find it; since B2's designs come from the still-immature native flow, whose residual bugs are exactly such physical defects (the router's spacing classes, the netlist-emission gaps), dual-flank agreement is independent confirmation that a defect is real, and the campaign treats it as design-earned evidence. B5's failures are asymmetric: its designs, produced by the mature open flow, pass almost all open-source gates and sometimes fail only the native LVS, DRC, or STA. What trips is the native grader consuming the open flow's output conventions: the native checkers are developed in-loop against native-flow output and impose native-specific expectations, such as a constraint-file dialect, port-naming conventions, and LVS pin-matching tables keyed to the native emission style, so an open-flow design that violates no physical rule can still arrive in a dialect the young instrument mishandles and fail on the native flank alone. A full per-design audit at loop 62 sizes this cross-flow class at roughly half of the open lane's failure mass, and a cure campaign (a constraint canonicalizer, a netlist-format converter, port-form pass-through, and a reviewed exception list) drains it era by era, with 13 residual cells at loop 72.
- **Developing physical-design tools is the bottleneck.** Counting the failing verdicts of the B2 lane gate by gate makes the bottleneck precise: of the 64 failing verdicts on the eleven designs' ten-verdict cards at loop 73 (the inverter smoke test excluded), 52 sit in the back-end sign-off gates.

  DRC fails on all eleven designs on both checker flanks, LVS on all eleven on the native flank and ten on the open flank, and STA on three to six, while the front-end functional-simulation and equivalence gates fail only on the three designs where the flow collapses outright. The front end is essentially solved; the back end is not. Figure 15 traces the dominant back-end gate over time, plotting the median number of DRC violations per run for the two native-flow lanes after the seal. On B1, the median starts in the hundreds, and flow repairs drive it to zero at iteration 58; it stays at zero for the rest of the campaign. B2 is harder: the median starts near ten thousand, falls by an order of magnitude at loop 57 when the router and power-grid emitters are cured at source,

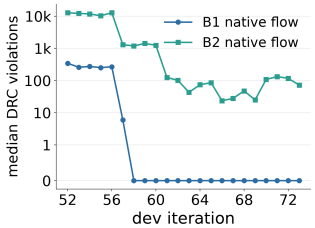


Figure 15: Median DRC violations vs. dev iteration.

  and improves no further than about one hundred violations through the final loops. The residual Tier-2 gap of Figure 13(b) thus sits exactly where Section 3 finds the hardest rebuilds: placement, routing, and

the DRC that judges them. After identifying this bottleneck, we stop treating placer and router bugs as one-shot fix requests and run open-ended improvement campaigns for both tools.

6 Takeaways, Limitations, and Perspectives

The author notes that the AI cycle has greatly expedited and moves extremely fast: models, agents, and evaluations are routinely superseded within months. The observations below are therefore a snapshot taken in October 2026, and parts of them may age quickly. With that caveat, this section collects what we take away from the three campaigns, where our study is limited, and how we see the road ahead.

The limitations of our work. Our measurement has clear boundaries. All three campaigns run on a single harness and primarily one agent stack, so the numbers may shift under other models or scaffolds. The references are open-source tools on open PDKs at mature nodes, and each rebuild covers a scope contract rather than the full tool, so nothing here speaks directly to proprietary flows at advanced nodes. Run-to-run variance is large, and cost limits the number of replicates, so several results carry wide uncertainty bands. Finally, although the grading instruments are audited, repaired, and version-pinned, they are still built and maintained by us, and residual defects in the referee cannot be fully excluded.

The status and the gap to future chip design. We should ultimately aim for autonomous agentic chip design with little or no human intervention: a human is needed, if at all, only at final sign-off, and when sign-off fails, the human provides insights on what to change, which the agent memorizes as skills and applies the next time. Chip design is also essential to realizing recursive self-improvement: model capability is bounded by the compute of the chips it trains and runs on, and today those chips are designed by human engineers on multi-year cycles; once an agent can design the hardware that its successors train and run on, the self-improvement loop closes at the physical layer, and hardware progress begins to compound at the pace of software iteration rather than of human design cycles. The current status of agents for chip design, and the gap to what we desire in the future, can be summarized as follows.

- **Agents can already carry out end-to-end design, but at old technology nodes.** Kimi K3 designs, optimizes, and verifies a chip serving a nano model of itself in a single 48-hour autonomous run with open-source EDA tools on a 45nm library (Moonshot AI, 2026); OpenAI's GPT-6 Astra demonstrates PCB layout in KiCad (OpenAI, 2026a); and the campaign of Section 5 reaches sign-off at 130nm.
- **Agents do better at the front end than at the back end.** The front end is close to solved on standard benchmarks: frontier models plateau near a 91 percent pass rate on VerilogEval, with the residual failures resistant to further sampling (Liu et al., 2026), and recent suites treat the classic RTL benchmarks as saturated, with state-of-the-art models above 95 percent (Yu et al., 2026b). The back end is a different picture: end-to-end evaluations such as FluxBench question whether agents can complete the RTL-to-GDS back end at all (Deng et al., 2026), and our own campaigns agree, concentrating the failure mass in placement, routing, and extraction.
- **Frontier AI labs have not yet carefully crafted chip design data in RL post-training.**[4] They potentially include only RTL and Verilog data, which are no different from ordinary code, and little or no physical design data. That both OpenAI and Anthropic are now advertising full-time positions dedicated to building RL environments and data for chip design (OpenAI, 2026b; Anthropic, 2026) points the same way.

Towards the future fully autonomous agentic chip design, the first step is to include complete chip design data in post-training, in an end-to-end or staged manner; data built on open-source tools and PDKs can already be helpful. Once this is settled, the question becomes how well the model performs on real proprietary PDKs calling proprietary EDA tools: there is certainly a train-with-open versus use-in-closed gap. The gap is not large in the sense that the agent does not care about the EDA tools' internal algorithms; it is not small in the sense that the design complexity is much larger (proprietary PDKs have far more design rules and standard

[4]The author's best personal guess.

cells). If amending post-training with open-source data is insufficient, then proprietary data is a must, but including it in RL post-training has huge legal problems depending on the licenses of the EDA tools and PDKs. What is worse, in the RL environment the EDA tools are called multiple times, and each usage costs money, so the whole RL training now carries an extra tool-call cost, which might be very expensive. Encouragingly, industry leaders are beginning to move in this direction. OpenAI and Synopsys, for example, have announced GPT-Synopsys, a specialized model designed to operate Synopsys EDA tools (Synopsys, Inc. and OpenAI, 2026). This partnership could help address some of the access, tooling, and domain-data barriers described above, and similar initiatives from other EDA vendors and AI labs are likely to follow. Nevertheless, even if EDA vendors permit their commercial tools to be used for model training, the author's best guess is that models (e.g., GPT-Synopsys) trained with proprietary tools, PDKs, or design data will not be publicly releasable.

Possible implications for different stakeholders. The author does not presume to advise any of these stakeholders, and offers the following notes simply as observations from one measurement campaign, in the hope that we all evolve better and prosper together in the future.

- **To EDA companies:** the moat is the optimization engines, not the wrappers around them; this paper shows that agents readily rebuild interfaces, file formats, and mediocre implementations, yet **cannot reproduce heuristics** of placement and routing accumulated over decades. For EDA companies, making tools agent-friendly (fast startup, parseable output, scriptable operation) and rethinking license models for high-frequency agent calls would capture the coming usage rather than resist it. Tool calls issued by RL training environments, for example, are a large use case of their own, unlike the previous era in which human designers invoke the tools manually: RL post-training of an LLM on 1,000 RTL-to-GDSII tasks, at a nominal 100 rollouts per task and 100 tool invocations per rollout, issues on the order of 10^7 EDA tool calls per run. Even at ten cents per call, the tool bill alone is one million dollars per run, or ten million dollars for a lab performing ten such runs per year, and prices anchored to per-seat licenses would be far higher. Embracing high-volume EDA tool calls from RL training environments can only add to an EDA company's revenue and carries almost no harm: the AI training company never sees the black-box heuristics, so the training makes their model better at calling the tools, not at designing better heuristics inside them. This is a low-risk opportunity with a potentially large revenue.
- **To hardware design companies:** agents can already help in many chip design tasks, though in many others they remain weak. Deploying them today lets a company log the design data and agent trajectories, especially on the tasks handled poorly, and distill the human designer knowledge into skills that make the next agent attempt better. Since each ASIC product line carries its own insights and knowledge, the skills distilled from it are product-specific, and this growing in-house library becomes a moat that others cannot reproduce. An emerging business mode is to sell the design-phase data of human designers, together with such assets, to AI training companies, which would likely pay a large price for them. For a frontier hardware company this is probably unreasonable, since it hands over the moat itself; for a mid-scale company not competing at the frontier, however, it is a valid business model with a large potential upside.
- **To frontier AI labs training LLMs:** RTL and Verilog data alone are not chip design data; the missing mass is physical design, and complete, staged or end-to-end design data built on open-source tools and PDKs is achievable today. The author expects that exhausting all open-source data will still bring a tangible improvement in chip design capability, and that this stage takes perhaps 3 months to 1 year. The next question is then where to obtain proprietary data, and one potential opportunity is to buy it from a mid-scale hardware company. This is tricky in two ways. First, as noted above, a model trained on proprietary tools or PDKs potentially cannot be released publicly. Second, several frontier AI labs want to design chips themselves for their own inference (for example, OpenAI's Jalapeño chip (OpenAI, 2026c)), so the same hardware company can be a data seller and a competitor at once. Managing the relationships among these parties can be very tricky, and careful decisions should be made. The recently announced GPT-Synopsys partnership (Synopsys, Inc. and OpenAI, 2026) is an encouraging signal for the community. In any case, improving chip-design capabilities, whether through RL post-training or

other forms of training, is essential for frontier AI labs because it closes a critical part of the recursive self-improvement loop.

- **To AI-native chip design/EDA startups:** the author sees four reasonable business modes at this moment. The first is to build the agent harness specifically for chip design: going beyond a general coding agent such as Codex or Claude Code by incorporating gathered or acquired chip design skills, together with internally built EDA tools that are cheaper and faster, possibly at lower fidelity, for the agent's repeated tool calls during design. The second is to post-train open-source LLMs in collaboration with a hardware company on their proprietary data and deploy them on-site: a hardware company is more willing to collaborate with a startup than with a frontier AI lab, because the frontier labs may build chips of their own, while a startup that does not build its own chip poses no such competitive concern. The third is to operate as a hardware IP company that simply delivers the hardware, while internally designing it faster with agents. The fourth is to sell data to frontier labs for training purposes. In summary, the author believes the key for such startups is to use their small scale advantage to foster trust with hardware companies and foundries within a short-term window, and this trust is the key to their success.

7 Conclusion

We measure whether present-day coding agents can improve the EDA toolchain itself, not merely drive it, and organize the measurement as three campaigns of increasing scope. The agent rebuilds nine CPU-based EDA tools from scratch in Rust, matching the quality of mature open-source references on six of them at 2 to 13× their speed and on a seventh in quality but not yet in speed; it optimizes three GPU-accelerated tools given their source, with the best chains achieving 1.3 to 2× speedups; and it rebuilds a complete RTL-to-GDSII flow that, deployed for end-to-end design by an agent, reaches sign-off parity with the OpenROAD reference on its Tier-1 suite of small blocks containing <2K standard cells, trails it on medium designs where placement and routing quality of results binds, and, like the reference, leaves the largest SoCs short of sign-off. Along the way we develop a harness with safeguards against reward hacking, and an evaluation discipline that audits, repairs, and version-pins the grading instruments with the same rigor as the system under test; we believe both transfer to any long software-optimization campaign.

The results support two conclusions. First, rebuilding open-source-grade EDA tools, and a working flow assembled from them, is mostly within reach of present agents: the failures are concentrated in a small and well-characterized set of physical design problems rather than spread across the flow. Second, the agents' speedups come from implementation rather than algorithmic insight, and the tools where no implementation slack exists, the optimization engines at the heart of placement and routing, are exactly the ones the agent fails to reproduce. Proprietary tools embody decades of such heuristics tuned to the latest technology nodes, and nothing in our results suggests that an agent could reproduce them today.

References

Tutu Ajayi, Vidya A. Chhabria, Mateus Fogaça, Soheil Hashemi, Abdelrahman Hosny, Andrew B. Kahng, Minsoo Kim, Jeongsup Lee, Uday Mallappa, Marina Neseem, Geraldo Pradipta, Sherief Reda, Mehdi Saligane, Sachin S. Sapatnekar, Carl Sechen, Mohamed Shalan, William Swartz, Lutong Wang, Zhehong Wang, Mingyu Woo, and Bangqi Xu. Toward an open-source digital flow: First learnings from the OpenROAD project. In *Proceedings of the 56th Annual Design Automation Conference (DAC)*, pages 1–4, 2019. doi: 10.1145/3316781.3326334.

Luca Amarù, Pierre-Emmanuel Gaillardon, and Giovanni De Micheli. The EPFL combinational benchmark suite. In *International Workshop on Logic & Synthesis (IWLS)*, 2015.

Analog Design Bench Team. Long-horizon analog design bench: Benchmarking agents on hours-long analog and mixed-signal circuit design tasks, 2026.

Anthropic. Claude code. https://www.anthropic.com/claude-code, 2025a.

Anthropic. The anthropic economic index. https://www.anthropic.com/economic-index, 2025b.

Anthropic. Research engineer, chip design RL. Job posting, https://www.anthropic.com/careers/, 2026.

Anthropic. Formalizing Fermat's last theorem. https://www.anthropic.com/research/formalizing-fermats-last-theorem, 2026.

Yuntao Bai, Saurav Kadavath, Sandipan Kundu, Amanda Askell, Jackson Kernion, Andy Jones, Anna Chen, Anna Goldie, Azalia Mirhoseini, Cameron McKinnon, et al. Constitutional AI: Harmlessness from AI feedback. *arXiv preprint arXiv:2212.08073*, 2022.

Nick Bostrom. *Superintelligence: Paths, Dangers, Strategies*. Oxford University Press, 2014.

Robert Brayton and Alan Mishchenko. ABC: An academic industrial-strength verification tool. In *International Conference on Computer Aided Verification (CAV)*, pages 24–40, 2010. doi: 10.1007/978-3-642-14295-6_5.

Robert K. Brayton, Gary D. Hachtel, Curtis T. McMullen, and Alberto L. Sangiovanni-Vincentelli. *Logic Minimization Algorithms for VLSI Synthesis*. Kluwer Academic Publishers, 1984. doi: 10.1007/978-1-4613-2821-6.

Robert K. Brayton, Richard L. Rudell, Alberto L. Sangiovanni-Vincentelli, and Albert R. Wang. MIS: A multiple-level logic optimization system. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 6(6):1062–1081, 1987. doi: 10.1109/TCAD.1987.1270347.

Franc Brglez and Hideo Fujiwara. A neutral netlist of 10 combinational benchmark circuits and a targeted translator in FORTRAN. In *IEEE International Symposium on Circuits and Systems (ISCAS)*, pages 695–698, 1985.

Randal E. Bryant. Graph-based algorithms for Boolean function manipulation. *IEEE Transactions on Computers*, 35(8):677–691, 1986. doi: 10.1109/TC.1986.1676819.

Ismail S. Bustany, David Chinnery, Joseph R. Shinnerl, and Vladimir Yutsis. ISPD 2015 benchmarks with fence regions and routing blockages for detailed-routing-driven placement. In *Proceedings of the International Symposium on Physical Design (ISPD)*, pages 157–164, 2015. doi: 10.1145/2717764.2723572.

Zhuomin Chai, Yuxiang Zhao, Wei Liu, Yibo Lin, Runsheng Wang, and Ru Huang. CircuitNet: An open-source dataset for machine learning in VLSI CAD applications with improved domain-specific evaluation metric and learning strategies. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 2023. doi: 10.1109/TCAD.2023.3287970.

Chung-Kuan Cheng, Andrew B. Kahng, Sayak Kundu, Yucheng Wang, and Zhiang Wang. Assessment of reinforcement learning for macro placement. In *Proceedings of the 2023 International Symposium on Physical Design (ISPD)*, pages 158–166, 2023. doi: 10.1145/3569052.3578926.

James Cherry. OpenSTA: Parallax static timing analyzer. https://github.com/parallaxsw/OpenSTA, 2019. Parallax Software.

Lawrence T. Clark, Vinay Vashishtha, Lucian Shifren, Aditya Gujja, Saurabh Sinha, Brian Cline, Chandarasekaran Ramamurthy, and Greg Yeric. ASAP7: A 7-nm finFET predictive process design kit. *Microelectronics Journal*, 53:105–115, 2016. doi: 10.1016/j.mejo.2016.04.006.

Fulvio Corno, Matteo Sonza Reorda, and Giovanni Squillero. RT-level ITC'99 benchmarks and first ATPG results. *IEEE Design & Test of Computers*, 17(3):44–53, 2000. doi: 10.1109/54.867894.

Soham Dan, Jeff Da, Sanyam Bhutani, Miguel Romero Calvo, Ivan Guevara, Fernando Carabedo, Jose Bernal, Julio Reyna, Luis Leandro Monsalve Mendez, Ying Liu, and Daniel Yue Zhang. SWE-Bench Pro V2: A cleaner, harder-to-game coding benchmark, 2026.

Jinyuan Deng et al. Can AI agents really complete RTL-to-GDS? Lessons from benchmarking tool-interactive EDA workflows. *arXiv preprint arXiv:2607.17528*, 2026.

Zehao Dong, Weidong Cao, Muhan Zhang, Dacheng Tao, Yixin Chen, and Xuan Zhang. CktGNN: Circuit graph neural network for electronic design automation. In *International Conference on Learning Representations (ICLR)*, 2023.

GlobalFoundries and Google. GlobalFoundries 180nm MCU open source PDK. https://github.com/google/gf180mcu-pdk, 2022.

Irving John Good. Speculations concerning the first ultraintelligent machine. In Franz L. Alt and Morris Rubinoff, editors, *Advances in Computers*, volume 6, pages 31–88. Elsevier, 1966. doi: 10.1016/S0065-2458(08)60418-0.

Surya Gunukula, Jerry Yicheng Wang, and Kevin Zhang. Introducing Build-A-Chip-Bench: From workload specification to GDS. https://chipagents.ai/research/build-a-chip-bench, 2026.

Daya Guo, Dejian Yang, Haowei Zhang, Junxiao Song, Ruoyu Zhang, Runxin Xu, Qihao Zhu, Shirong Ma, Peiyi Wang, Xiao Bi, et al. Deepseek-r1: Incentivizing reasoning capability in llms via reinforcement learning. *arXiv preprint arXiv:2501.12948*, 2025a.

Zizheng Guo, Yanqing Zhang, Runsheng Wang, Yibo Lin, and Haoxing Ren. GEM: GPU-accelerated emulator-inspired RTL simulation. In *ACM/IEEE Design Automation Conference (DAC)*, pages 1–7, 2025b. doi: 10.1109/DAC63849.2025.11132713.

Kwangsoo Han, Andrew B. Kahng, and Jiajia Li. Optimal generalized H-tree topology and buffering for high-performance and low-power clock distribution. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 39(2):478–491, 2020. doi: 10.1109/TCAD.2018.2889756.

Zhuolun He, Haoyuan Wu, Xinyun Zhang, Xufeng Yao, Su Zheng, Haisheng Zheng, and Bei Yu. Chateda: A large language model powered autonomous agent for eda. In *ACM/IEEE Workshop on Machine Learning for CAD (MLCAD)*, 2023. doi: 10.1109/MLCAD58807.2023.10299852.

Jordan Hoffmann, Sebastian Borgeaud, Arthur Mensch, Elena Buchatskaya, Trevor Cai, Eliza Rutherford, Diego de Las Casas, Lisa Anne Hendricks, Johannes Welbl, Aidan Clark, et al. Training compute-optimal large language models. *arXiv preprint arXiv:2203.15556*, 2022.

Wenqi Huang and Peter Jiang. DeepSWE v1.1: A cleaner, more reproducible benchmark for frontier coding agents. https://deepswe.datacurve.ai/blog/deepswe-v1-1, 2026.

Yuki Imajuku, Kohki Horie, Yoichi Iwata, Kensho Aoki, Naohiro Takahashi, and Takuya Akiba. ALE-Bench: A benchmark for long-horizon objective-driven algorithm engineering. In *Advances in Neural Information Processing Systems (NeurIPS) Datasets and Benchmarks Track*, volume 38, 2025. doi: 10.52202/085713-5226.

Jiaming Ji, Tianyi Qiu, Boyuan Chen, Borong Zhang, Hantao Lou, Kaile Wang, Yawen Duan, Zhonghao He, Jiayi Zhou, Zhaowei Zhang, et al. AI alignment: A comprehensive survey. *arXiv preprint arXiv:2310.19852*, 2023.

Carlos E Jimenez, John Yang, Alexander Wettig, Shunyu Yao, Kexin Pei, Ofir Press, and Karthik Narasimhan. Swe-bench: Can language models resolve real-world github issues? In *International Conference on Learning Representations*, 2024.

Jared Kaplan, Sam McCandlish, Tom Henighan, Tom B Brown, Benjamin Chess, Rewon Child, Scott Gray, Alec Radford, Jeffrey Wu, and Dario Amodei. Scaling laws for neural language models. *arXiv preprint arXiv:2001.08361*, 2020.

Robert Kirby, Saad Godil, Rajarshi Roy, and Bryan Catanzaro. CongestionNet: Routing congestion prediction using deep graph neural networks. In *IFIP/IEEE International Conference on Very Large Scale Integration (VLSI-SoC)*, pages 217–222, 2019. doi: 10.1109/VLSI-SoC.2019.8920342.

Pascal Kuthe. OpenVAF: An innovative Verilog-A compiler. https://github.com/pascalkuthe/OpenVAF, 2023.

Yao Lai, Sungyoung Lee, Guojin Chen, Souradip Poddar, Mengkang Hu, David Z Pan, and Ping Luo. Analogcoder: Analog circuit design via training-free code generation. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 39, pages 379–387, 2025. doi: 10.1609/aaai.v39i1.32016.

Robert Tjarko Lange, Edoardo Cetin, et al. ShinkaEvolve: Towards open-ended and sample-efficient program evolution. *arXiv preprint arXiv:2509.19349*, 2025.

Jintao Li, Haochang Zhi, Ruiyu Lyu, Wangzhen Li, Zhaori Bi, Keren Zhu, Yanhan Zeng, Weiwei Shan, Changhao Yan, Fan Yang, Yun Li, and Xuan Zeng. AnalogGym: An open and practical testing suite for analog circuit synthesis. In *IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*, pages 1–9, 2024. doi: 10.1145/3676536.3697117.

Rongjian Liang, Anthony Agnesina, Wen-Hao Liu, and Haoxing Ren. GPU/ML-enhanced large scale global routing contest. In *Proceedings of the International Symposium on Physical Design (ISPD)*, pages 269–274, 2024. doi: 10.1145/3626184.3639693.

Shiju Lin, Jinwei Liu, Tianji Liu, Martin D. F. Wong, and Evangeline F. Y. Young. NovelRewrite: Node-level parallel AIG rewriting. In *Proceedings of the 59th ACM/IEEE Design Automation Conference (DAC)*, pages 427–432, 2022. doi: 10.1145/3489517.3530462.

Shiju Lin, Liang Xiao, Jinwei Liu, and Evangeline F. Y. Young. InstantGR: Scalable GPU parallelization for global routing. In *IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*, pages 1–8, 2024. doi: 10.1145/3676536.3676787.

Yibo Lin, Shounak Dhar, Wuxi Li, Haoxing Ren, Brucek Khailany, and David Z. Pan. DREAMPlace: Deep learning toolkit-enabled GPU acceleration for modern VLSI placement. In *ACM/IEEE Design Automation Conference (DAC)*, pages 1–6, 2019. doi: 10.1145/3316781.3317803.

Guan-Ting Liu, Chao-Han Huck Yang, Chenhui Deng, Zhongzhi Yu, Brucek Khailany, and Yu-Chiang Frank Wang. How LLMs fail and generalize in RTL coding for hardware design?, 2026.

Mingjie Liu, Teodor-Dumitru Ene, Robert Kirby, Chris Cheng, Nathaniel Pinckney, Rongjian Liang, Jonah Alben, Himyanshu Anand, Sanmitra Banerjee, Ismet Bayraktaroglu, et al. Chipnemo: Domain-adapted llms for chip design. *arXiv preprint arXiv:2311.00176*, 2023a.

Mingjie Liu, Nathaniel Pinckney, Brucek Khailany, and Haoxing Ren. Invited paper: VerilogEval: Evaluating large language models for Verilog code generation. In *IEEE/ACM International Conference on Computer Aided Design (ICCAD)*, pages 1–8, 2023b. doi: 10.1109/ICCAD57390.2023.10323812.

Shang Liu, Wenji Fang, Yao Lu, Jing Wang, Qijun Zhang, Hongce Zhang, and Zhiyao Xie. Rtlcoder: Fully open-source and efficient llm-assisted rtl code generation technique. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 44(4):1448–1461, 2025. doi: 10.1109/TCAD.2024.3483089.

Weidong Liu, Xiaodong Jin, Kanyu M. Cao, and Chenming Hu. BSIM 4.0.0 MOSFET model user's manual. Technical Report UCB/ERL M00/38, EECS Department, University of California, Berkeley, 2000.

Yao Lu, Shang Liu, Qijun Zhang, and Zhiyao Xie. Rtllm: An open-source benchmark for design rtl generation with large language model. In *Asia and South Pacific Design Automation Conference (ASP-DAC)*, 2024. doi: 10.1109/ASP-DAC58780.2024.10473904.

Igor L. Markov. The false dawn: Reevaluating google's reinforcement learning for chip macro placement. *arXiv preprint arXiv:2306.09633*, 2024.

Azalia Mirhoseini, Anna Goldie, Mustafa Yazgan, Joe Wenjie Jiang, Ebrahim Songhori, Shen Wang, Young-Joon Lee, Eric Johnson, Omkar Pathak, Azade Nazi, et al. A graph placement methodology for fast chip design. *Nature*, 594(7862):207–212, 2021. doi: 10.1038/s41586-021-03544-w.

Moonshot AI. Kimi k3: Open frontier intelligence. https://www.kimi.ai/blog/kimi-k3, 2026.

Laurence W. Nagel and Donald O. Pederson. SPICE (simulation program with integrated circuit emphasis). Technical Report UCB/ERL M382, EECS Department, University of California, Berkeley, 1973.

Gi-Joon Nam. The ISPD 2006 placement contest: Benchmark suite and results. In *Proceedings of the International Symposium on Physical Design (ISPD)*, page 167, 2006. doi: 10.1145/1123008.1123042.

Gi-Joon Nam, Charles J. Alpert, Paul Villarrubia, Bruce Winter, and Mehmet Yildiz. The ISPD2005 placement contest and benchmark suite. In *Proceedings of the International Symposium on Physical Design (ISPD)*, pages 216–220, 2005. doi: 10.1145/1055137.1055182.

Nangate Inc. Nangate 45nm open cell library. https://si2.org/open-cell-library/, 2008.

Sani R. Nassif. Power grid analysis benchmarks. In *Asia and South Pacific Design Automation Conference (ASP-DAC)*, pages 376–381, 2008. doi: 10.1109/ASPDAC.2008.4483978.

Alexander Novikov et al. AlphaEvolve: A coding agent for scientific and algorithmic discovery. *arXiv preprint arXiv:2506.13131*, 2025.

OpenAI. Introducing ChatGPT. https://openai.com/blog/chatgpt, 2022.

OpenAI. Learning to reason with LLMs. https://openai.com/index/learning-to-reason-with-llms/, 2024.

OpenAI. Introducing Codex. https://openai.com/index/introducing-codex/, 2025.

OpenAI. GPT-6 Astra: A new generation of intelligence. https://openai.com/index/gpt-6-astra/, 2026a.

OpenAI. Research engineer, AI for chip design. Job posting, https://openai.com/careers/, 2026b.

OpenAI. OpenAI and Broadcom unveil LLM-optimized inference chip. https://openai.com/index/openai-broadcom-jalapeno-inference-chip/, 2026c.

OpenAI. On the Navier–Stokes Millennium Prize problem. https://openai.com/index/navier-stokes-solution/, 2026.

Anne Ouyang, Simon Guo, Simran Arora, Alex L. Zhang, William Hu, Christopher Ré, and Azalia Mirhoseini. KernelBench: Can LLMs write efficient GPU kernels? *arXiv preprint arXiv:2502.10517*, 2025.

Long Ouyang, Jeffrey Wu, Xu Jiang, Diogo Almeida, Carroll Wainwright, Pamela Mishkin, Chong Zhang, Sandhini Agarwal, Katarina Slama, Alex Ray, et al. Training language models to follow instructions with human feedback. In *Advances in Neural Information Processing Systems*, volume 35, 2022.

Jakub Pachocki. An alien mind. https://openai.com/index/an-alien-mind/, 2026.

Hammond Pearce, Benjamin Tan, and Ramesh Karri. DAVE: Deriving automatically Verilog from English. In *Proceedings of the 2020 ACM/IEEE Workshop on Machine Learning for CAD (MLCAD)*, 2020. doi: 10.1145/3380446.3430634.

Nathaniel Pinckney, Chenhui Deng, Chia-Tung Ho, Yun-Da Tsai, Mingjie Liu, Wenfei Zhou, Brucek Khailany, and Haoxing Ren. Comprehensive Verilog design problems: A next-generation benchmark dataset for evaluating large language models and agents on RTL design and verification. *arXiv preprint arXiv:2506.14074*, 2025.

Bernardino Romera-Paredes, Mohammadamin Barekatain, Alexander Novikov, Matej Balog, M. Pawan Kumar, Emilien Dupont, Francisco J. R. Ruiz, Jordan S. Ellenberg, Pengming Wang, Omar Fawzi, Pushmeet Kohli, and Alhussein Fawzi. Mathematical discoveries from program search with large language models. *Nature*, 625:468–475, 2024. doi: 10.1038/s41586-023-06924-6.

Alberto Sangiovanni-Vincentelli. The tides of EDA. *IEEE Design & Test of Computers*, 20(6):59–75, 2003. doi: 10.1109/MDT.2003.1246165.

Timo Schick, Jane Dwivedi-Yu, Roberto Dessì, Roberta Raileanu, Maria Lomeli, Eric Hambro, Luke Zettlemoyer, Nicola Cancedda, and Thomas Scialom. Toolformer: Language models can teach themselves to use tools. In *Advances in Neural Information Processing Systems*, volume 36, 2023.

Manish Shetty, Naman Jain, Jinjian Liu, Vijay Kethanaboyina, Koushik Sen, and Ion Stoica. GSO: Challenging software optimization tasks for evaluating SWE-agents. In *Advances in Neural Information Processing Systems (NeurIPS) Datasets and Benchmarks Track*, volume 38, pages 167761–167804, 2025. doi: 10.52202/085713-5054.

Massimo Sivilotti and R. Timothy Edwards. Netgen: LVS and netlist comparison. http://opencircuitdesign.com/netgen/, 2019.

SkyWater Technology Foundry and Google. SkyWater open source PDK. https://github.com/google/skywater-pdk, 2020.

Synopsys, Inc. and OpenAI. OpenAI and Synopsys announce GPT-Synopsys: Frontier intelligence to revolutionize chip design. Press release, https://news.synopsys.com/2026-09-30-OpenAI-and-Synopsys-Announce-GPT-Synopsys-Frontier-Intelligence-to-Revolutionize-Ch September 2026.

Terminal-Bench Team. Terminal-Bench 4.0: Calibrating task resources, fixing tasks, and removing saturated tasks. https://www.tbench.ai/news/terminal-bench-4-0, 2026.

Shailja Thakur, Jason Blocklove, Hammond Pearce, Benjamin Tan, Siddharth Garg, and Ramesh Karri. Autochip: Automating hdl generation using llm feedback. *arXiv preprint arXiv:2311.04887*, 2023.

The OpenROAD Project. Some observations on OpenROAD and ORFS for a larger design: MegaBoom. GitHub discussion, https://github.com/The-OpenROAD-Project/OpenROAD-flow-scripts/discussions/1710, 2023.

George Tsoukalas, Anton Kovsharov, Sergey Shirobokov, Anja Surina, Moritz Firsching, Gergely Bérczi, Francisco J. R. Ruiz, Arun Suggala, Adam Zsolt Wagner, Eric Wieser, et al. Advancing mathematics research with AI-driven formal proof search. *arXiv preprint arXiv:2605.22763*, 2026.

Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. Attention is all you need. In *Advances in Neural Information Processing Systems*, volume 30, 2017.

Holger Vogt, Giles Atkinson, Dietmar Warning, and Paolo Nenzi. Ngspice user's manual, version 46. https://ngspice.sourceforge.io/, 2026.

Lei Wang, Chen Ma, Xueyang Feng, Zeyu Zhang, Hao Yang, Jingsen Zhang, Zhiyuan Chen, Jiakai Tang, Xu Chen, Yankai Lin, et al. A survey on large language model based autonomous agents. *Frontiers of Computer Science*, 18(6):186345, 2024.

Stephen Williams and Michael Baxter. Icarus Verilog: Open-source Verilog more than a year later. *Linux Journal*, 2002(99), 2002.

Zhiyao Xie, Yu-Hung Huang, Guan-Qi Fang, Haoxing Ren, Shao-Yun Fang, Yiran Chen, and Jiang Hu. RouteNet: Routability prediction for mixed-size designs using convolutional neural network. In *IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*, pages 1–8, 2018. doi: 10.1145/3240765.3240843.

Jackey Z. Yan, Natarajan Viswanathan, and Chris Chu. Handling complexities in modern large-scale mixed-size placement. In *Proceedings of the 46th Design Automation Conference (DAC)*, pages 436–441, 2009. doi: 10.1145/1629911.1630028.

Zhiyuan Yan, Wenji Fang, Mengming Li, Min Li, Shang Liu, Zhiyao Xie, and Hongce Zhang. Assertllm: Generating hardware verification assertions from design specifications via multi-llms. In *Asia and South Pacific Design Automation Conference (ASP-DAC)*, pages 614–621, 2025. doi: 10.1145/3658617.3697756.

Shunyu Yao, Jeffrey Zhao, Dian Yu, Nan Du, Izhak Shafran, Karthik Narasimhan, and Yuan Cao. React: Synergizing reasoning and acting in language models. In *International Conference on Learning Representations*, 2023.

Cunxi Yu and Haoxing Ren. Autonomous evolution of EDA tools: Multi-agent self-evolved ABC. In *ACM/IEEE Design Automation Conference (DAC)*, 2026. doi: 10.1145/3770743.3804221.

Cunxi Yu, Rongjian Liang, Chia-Tung Ho, and Haoxing Ren. Autonomous code evolution meets NP-completeness. *arXiv preprint arXiv:2509.07367*, 2025.

Cunxi Yu, Chenhui Deng, Nathaniel Pinckney, and Brucek Khailany. Agentic hardware design as repository-level code evolution. *arXiv preprint arXiv:2606.28279*, 2026a.

Zhongkai Yu, Chenyang Zhou, Yichen Lin, Hejia Zhang, Haotian Ye, Junxia Cui, Zaifeng Pan, Jishen Zhao, and Yufei Ding. ChipBench: A next-step benchmark for evaluating LLM performance in AI-aided chip design, 2026b.

Deyao Zhu, Xin Zhou, Shengling Qin, Xuekai Zhu, Hangliang Ding, Shu Zhong, et al. EdgeBench: Unveiling scaling laws of learning from real-world environments. *arXiv preprint arXiv:2607.05155*, 2026.