Are Skills Enough? An Agentic Frequency Optimization View for Large Chips

Jiahan Xie, Sakshi Garg, Jose Renau
University of California, Santa Cruz

***Abstract*—A chip designer with a working processor wants higher frequency—not new code from scratch. Yet most LLM-based RTL tools today target code generation or bug fixing on small, single-file modules. We present the first head-to-head comparison of two orchestration styles for large-scale frequency optimization: *Skills*, where domain-specific prompts, scripts, and tool integrations let a coding agent orchestrate the synthesis/STA/LEC workflow on its own; and *Custom agents*, where a bespoke pipeline enforces the same workflow with rigid guardrails. We evaluate both on processor-scale RISC-V CPU and GPGPU designs (CVA6, SoomRV, Vortex, Dino) across three frontier LLMs (Opus 4.6, GPT-5.3-codex, Gemini 3.1) with mandatory SAT-based equivalence checking. Skills and Custom agents achieve similar average Fmax gains (22.8% vs 21.6%), but differ sharply in controllability and localization behavior: Skills agents self-localize critical paths through exploratory tool use—sometimes exhibiting emergent behaviors such as recursive netlist tracing that no one programmed—while Custom agents require explicit localization to function at all.**

***Index Terms*—Custom Agents, Coding Skills, Large Language Models, Frequency Improvement, Chip Design**

I. Introduction

Optimizing for higher frequency is difficult: the designer must trace critical paths across thousands of lines of RTL, propose changes that shorten combinational depth without breaking equivalence, and re-synthesize after every attempt. The goal of this work is to leverage LLMs and coding agents to improve the frequency of large, working designs.

We evaluate on processor-scale hand-coded designs—CVA6, SoomRV, Vortex, and Dino—with tens of thousands to hundreds of thousands of synthesized cells, substantially larger than prior optimization work (Section II). At this scale, the agent must navigate multi-file repositories, identify which of many modules contains the critical path, and make targeted edits without breaking the surrounding design. Every timing-driven change must preserve functional equivalence, verified via sequential SAT-based logic equivalence checking (LEC), and each trial requires running synthesis and static timing analysis, so reducing unproductive iterations matters.

We compare two orchestration styles within the same synthesis/STA/LEC loop. *Skills* delegate orchestration to a general-purpose coding agent equipped with domain prompts, scripts, and tool integrations [1]; *Custom agents* encode the loop as an explicit state machine with prescribed LLM calls. The styles are points in a shared design space: Skills favor deployment speed and exploration, whereas Custom favors controllability and auditability. To our knowledge, no prior work compares these styles head-to-head. They achieve similar Fmax gains (22.8% vs 21.6%) across LLMs and designs. Skills agents consume over 20× more tokens due to self-localization, but wall-clock time remains comparable because synthesis and LEC dominate. Skills offer faster deployment and sometimes surprising creativity (e.g., emergent grep-based signal tracing); Custom agents provide an auditable, inspectable pipeline. Across configurations, locator choice, ware/tech prompts, and model selection each contribute measurable gains.

At this scale, localization—tracing a critical path from the post-synthesis netlist back to RTL source constructs—becomes a prerequisite for cost-effective optimization. Presenting the entire codebase to the LLM at each iteration is prohibitively expensive; prior single-module work avoids this problem entirely. We compare hierarchy resolution via `slang`, endpoint-based netlist flooding, and SynAlign [2] line-level linking. Localization is essential for Custom agents but mainly cost-saving for Skills agents, which can self-localize by grepping netlists and elaboration files and sometimes cross-check handwritten locators. Technique prompts consistently help both paradigms.

This paper makes the following contributions:

- First Skills-vs.-Custom orchestration style study for Fmax optimization, revealing control/exploration/cost trade-offs.
- Benchmarking on processor-scale CPU/GPGPU designs (CVA6, SoomRV, Vortex, Dino) across three frontier LLMs, with benchmark sizes substantially larger than prior optimization works [3], [4], [5], [6], [7], [8], [9].
- An evaluation of localization strategies showing their asymmetric impact: essential for Custom agents, cost-saving for Skills agents.

II. Related Work

A. LLM-Based RTL Frequency/PPA Optimization

RTLRewriter [3] is the first framework to apply LLMs to RTL code optimization, using circuit partitioning and multi-modal program analysis. SymRTLO [4] extends this with symbolic reasoning and AST-based templates, improving over RTLRewriter on the same benchmark but targeting modules averaging only 9 cells. Neither mandates sequential SAT-based LEC for correctness. Timing Logic Metamorphosis [5] shows that without explicit timing-targeted instructions, LLMs miss redundancies in control flow and Mux structures, which directly motivates our encoding of optimization strategies as reusable agent skills.

979-8-3195-1246-8-0/26/$31.00 ©2026 IEEE

VeriPPA [6] optimizes *generated* Verilog through iterative resynthesis; EvolVE [7] applies evolutionary search and MCTS to contest-level designs; VeriOpt [8] uses specialized LLM roles for PPA-aware generation; and Principle-Guided Optimization [9] extracts reusable principles from paired examples. These systems target generated code, small modules, or example-derived optimization rather than pre-existing processor-scale repositories.

Prior agent work [10] shows that LLM agents can adapt to HDLs with LEC validation. HDLCoRe [11] mitigates hallucinations in generated HDL, and VeriMOA [12] uses mixture-of-agents orchestration for spec-to-HDL generation. In contrast, we compare Skills and Custom agents for frequency optimization on large production Verilog repositories.

Two complementary approaches optimize PPA without rewriting RTL source code. ASPEN [13] uses LLMs to generate e-graph rewriting rules for datapath optimization on an internal representation. ORFS-agent [14] automates OpenROAD flow-script parameter tuning via an LLM-based iterative agent. Both are orthogonal to our source-level approach and could be composed with it.

B. Skills and Tool-Integration Frameworks

MCP4EDA [1] is the closest precedent to our *Skills* paradigm, exposing RTL-to-GDSII tools as Model Context Protocol endpoints. However, it tunes physical-design scripts on small single-file designs, does not rewrite RTL, and does not compare against a Custom pipeline.

C. Critical-Path Localization

Localizing timing-critical constructs in RTL is essential for targeted optimization. RTL-Timer [15] predicts per-signal slack at the RTL stage using graph neural networks. VeriLoC [16] extends this to line-of-code level timing and congestion prediction using LLM embeddings. ViTAD [17] builds a Signal Timing Dependency Graph from AST analysis and post-synthesis reports to diagnose timing violations. These approaches *predict* timing at the RTL stage.

Our localization solves the *inverse* problem: given a critical path in the post-synthesis netlist, identify the corresponding RTL source constructs to rewrite. Our locator combines hierarchy resolution (via `slang-hier`) with a netlist flooding pass that traverses outward from path endpoints to recover matching RTL modules, intentionally extending beyond the reported path so that the agent may optimize an adjacent pipeline stage and rely on synthesis retiming. We also evaluate SynAlign [2], which provides probabilistic line-level linking but considers only gates between the path endpoints, yielding finer-grained yet narrower localization.

D. Benchmarks for LLM-Driven Hardware Tasks

Most LLM hardware benchmarks target generation correctness on small designs [3], [18], [19], [20], [21]. RTL-OPT [22] is closest to our task, pairing suboptimal RTL with expert-optimized code and measuring PPA improvement, but its 36 designs are individual modules, far smaller than our processor-scale benchmarks (1K–30K LoC, tens to hundreds of thousands of synthesized cells).

Require: $code_{ref}$: reference RTL source
Ensure: $code_{ref}$ with improved Fmax
$elab \leftarrow \text{COMPILE}(code_{ref})$
$netlist \leftarrow \text{SYNTH}(elab)$
$report_{ref} \leftarrow \text{STA}(netlist)$
for $i = 1$ **to** N **do**
  $hints \leftarrow \text{SLANG\_HIER}(report_{ref},\ code_{ref})$ {hierarchy localization}
  if SynAlign enabled **then**
    $hints \mathrel{+}= \text{SynAlign}(netlist,\ code_{ref})$ {probabilistic line-level}
  else
    $hints \mathrel{+}= \text{FLOOD}(netlist,\ report_{ref})$ {topological module-level}
  end if
  $\delta \leftarrow \text{LLM}(ware{+}tech\_prompt,\ hints,\ code_{ref})$
  for $j = 1$ **to** M **do**
    $code_{new} \leftarrow \text{APPLY\_PATCH}(\delta)$
    $result \leftarrow \text{SLANG}(code_{new})$
    if $result.fail$ **then**
      $\delta \leftarrow \text{LLM}(syntax\_fixer,\ result,\ \delta)$
      continue
    end if
    $result \leftarrow \text{LEC}(code_{new},\ code_{ref})$
    if $result.fail$ **then**
      $\delta \leftarrow \text{LLM}(lec\_fixer,\ result,\ \delta)$
      continue
    end if
    break {patch compiles and is equivalent}
  end for
  if $\neg\, result.fail$ **then**
    $report_{new} \leftarrow \text{STA}(\text{SYNTH}(\text{COMPILE}(code_{new})))$
    if $report_{new}.Fmax > report_{ref}.Fmax$ **then**
      $code_{ref} \leftarrow code_{new}$;
      $elab \leftarrow \text{COMPILE}(code_{ref})$;
      $netlist \leftarrow \text{SYNTH}(elab)$;
      $report_{ref} \leftarrow report_{new}$
    end if
  end if
end for

Algorithm 1. FREQUENCY OPTIMIZATION LOOP

III. METHODOLOGY

Both paradigms target the same optimization loop (Figure 1, Algorithm 1), iteratively proposing RTL modifications that improve frequency while preserving equivalence. Custom encodes the loop as a deterministic state machine; Skills gives the coding agent guidance and tool access, allowing reordering or extra exploration. After launch, runs are fully automated: agents propose RTL edits, invoke syntax/LEC/synthesis feedback, and accept/reject patches without human RTL edits or per-benchmark intervention. Our goal is to study autonomous agentic optimization, where the AI agent manages the full loop.

a) Localization: Each iteration begins by identifying RTL modules likely to contain or neighbor the critical path, narrowing the optimization target to a handful of modules. This step is essential at processor scale: presenting the entire repository to the LLM on every iteration would be prohibitively

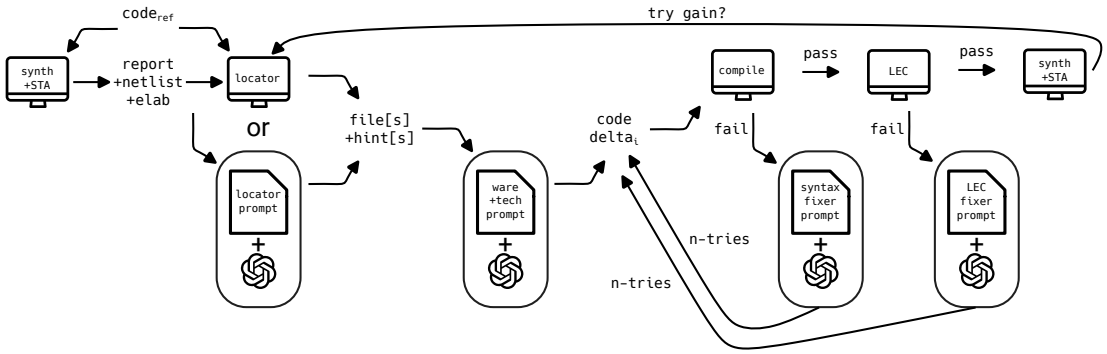

Fig. 1. OPTIMIZATION LOOP SHARED BY SKILLS AND CUSTOM AGENTS. CUSTOM USES A HARD-CODED STATE MACHINE; SKILLS LETS THE CODING AGENT ORCHESTRATE THE SAME STEPS VIA SKILL PROMPTS.

expensive and would dilute attention across many irrelevant files. Our locator has two components.

The first component, `slang-hier`, extracts the design hierarchy from the elaborated source and resolves register or wire names from the STA report to their containing modules. When the start and end points reside in different modules, it walks up the hierarchy to a common ancestor; the modules along that path become the optimization target.

The second component *floods* outward through the post-synthesis netlist from the critical path's begin and end registers (or primary inputs/outputs) until it encounters nodes whose names match identifiers in the original RTL source. These matches identify candidate RTL module files adjacent to the critical registers. This strategy is deliberately broader than tracing only gates on the reported path: it may identify the pipeline stage *ahead* of the critical path, enabling edits that allow synthesis retiming to resolve the violation indirectly.

When SynAlign [2] is enabled (lines 6–7), it replaces the flooding pass with probabilistic netlist-to-RTL linking that points to specific lines of code rather than modules, considering only gates between the path endpoints. Section V-C compares both strategies. Localization also has asymmetric value across paradigms: it is essential for the Custom pipeline, but mainly a cost optimization for Skills agents, which can spend extra tokens to self-localize through codebase exploration.

b) Optimization prompts: The core LLM call (line 12) receives hardware- and technology-aware context, including timing slack information from STA, standard-cell library characteristics, and coding-style guidance such as resource replacement (e.g., carry-select adders), array banking, and pipeline restructuring to shorten critical combinational paths. Resource-replacement patterns are retrieved on demand via RAG from a curated library. All prompts are designed to elicit chain-of-thought reasoning [23] for better optimization decisions in both settings. In the Custom pipeline, an *architectural agent* first generates a per-module optimization plan. Dedicated *optimization subagents* are then spawned for each module to implement the prescribed strategies in parallel.

c) Syntax checking with Slang: Before running the expensive synthesis and LEC steps, each proposed patch is compiled with Slang [24] (line 14). LEC also catches syntax errors, but Slang is significantly faster and produces richer diagnostic messages, enabling the syntax-fixer prompt (line 16) to converge in fewer retries. In practice this pre-filter eliminates most wasted synthesis cycles.

d) LEC gating: Every accepted delta must pass sequential SAT-based logic equivalence checking (LEC) against the reference design (line 19). Failures feed back into the LLM with a fixer prompt (line 21), and the inner loop retries up to M times. LEC is run at each benchmark's designated top boundary, so internal retiming is allowed but externally visible cycle behavior must match the reference.

e) Skills vs. Custom: what differs: For the Custom agent, Algorithm 1 is implemented as a deterministic script: a state machine that executes each step in fixed order, calls the LLM only at prescribed points, and never deviates. For the Skills agent, the same logic is encoded as a skill prompt together with tool integrations (MCP endpoints or shell commands) that the coding agent orchestrates. The key consequence is that Skills agents *may* skip, reorder, or augment steps (for example, running an extra grep to cross-check the locator's output before proposing a patch), while Custom agents cannot. This flexibility is the source of both the emergent capabilities and the occasional unpredictability observed in Section V-D.

IV. EXPERIMENTAL SETUP

A. Benchmark Designs

Table I summarizes the benchmark designs and their optimization targets. Each target selects a subset of the design hierarchy around the critical path. All metrics are measured after synthesis with Yosys/ABC on the SkyWater sky130A standard-cell library, a setup that enables open, reproducible experiments. Absolute Fmax may shift with other libraries, but our comparisons depend mainly on RTL structure, localization quality, and agent-control strategy.

B. Tool Versions

Table II lists the EDA and analysis tools used in the synthesis/STA/LEC flow. All experiments use the same tool versions to ensure reproducibility.

Design	Target	LoC	Files	Gates	Fmax (MHz)
Dino	SingleIssue	1073	18	11570	110.6
	DualIssue	1955	18	23773	80.9
CVA6	Core	39,157	92	212756	37.7
	LSU	8,624	22	34605	116.6
SoomRV	Frontend	4487	19	670888	10.4
Vortex	Core	30588	228	483912	16.3
	FPU	9977	41	70255	71.5

Table I. BENCHMARK DESIGNS AND OPTIMIZATION TARGETS. BENCHMARKS SPAN 1K–30K LOC WITH HUNDREDS OF THOUSANDS OF GATES.

Tool	Version	Role
Yosys	0.60	Synthesis + ABC optimization + LEC
OpenSTA	2.7.0	Static timing analysis
Slang	10.0.13	SystemVerilog elaboration
SkyWater PDK	sky130A	Standard-cell timing library

Table II. EDA TOOLS AND VERSIONS. ALL EXPERIMENTS SHARE IDENTICAL TOOL VERSIONS TO ISOLATE AGENT-PARADIGM EFFECTS.

Custom LLM	Skills Coding Agent	Skills LLM
Opus 4.6	Claude Code 2.1.75	Opus 4.6
GPT-5.3-codex	Codex CLI 0.114.0	GPT-5.3-codex
gemini-3.1-pro-preview	Gemini-cli 0.33.1	gemini-3.1-pro-preview

Table III. CODING AGENTS AND LLM CONFIGURATIONS. EACH ROW IS MODEL-MATCHED ACROSS CUSTOM AND SKILLS.

Label	Paradigm	Locator	Ware/Tech
CA	Coding agent	none	No
C	Custom	slang-hier + flood	Yes
C-L	Custom	slang-hier only	Yes
C-W	Custom	slang-hier + flood	No
CAL	Custom	slang-hier + SynAlign	Yes
S	Skills	slang-hier + flood	Yes
S-L	Skills	slang-hier only	Yes
S-W	Skills	slang-hier + flood	No
SAL	Skills	slang-hier + SynAlign	Yes

Table IV. EVALUATION CONFIGURATIONS. ALL USE SLANG-HIER; FLOOD AND SYNALIGN ARE ALTERNATIVE LOCALIZATION PASSES. WARE/TECH ENABLES HARDWARE- AND TECHNOLOGY-AWARE PROMPTS.

C. Coding Agents and LLMs

Table III lists the LLMs and coding agents used in the evaluation. Opus and GPT are used with the same underlying model in both paradigms, as the backbone of a Skills-augmented coding agent and as the underlying model in the Custom agent pipeline. For Gemini, both paradigms use gemini-3.1-pro-preview. On the Skills side, this was run through Gemini-cli 0.33.1, which in our experiments consistently used gemini-3.1-pro-preview without fallback to Flash. For all three coding agents on the Skills side (Claude Code, Codex CLI, and Gemini-cli), we used the default effort level, which is medium in all three cases.

D. Evaluation Configurations

Table IV summarizes the configurations evaluated. Each configuration combines a paradigm (Custom or Skills) with a localization strategy and optional hardware/technology prompts. The baseline (CA) runs the coding agent without any skills or custom pipeline. All configurations use slang-hier for hierarchy resolution; the locator column indicates what additional localization is enabled on top of slang-hier.

V. EVALUATION

A. Fmax Results

Table V reports configurations with the flooding locator (C/S) and with hierarchy only (C-L/S-L). Failures are categorized as F_{cw} (context window overflow), F_s (syntax error), F_l (LEC failure), F_r (performance regression), and F_f (failed to locate the file). Table VI reports the SynAlign configurations (CAL/SAL), which replace the flooding pass with probabilistic line-level localization (see Table IV for configuration definitions). Each numeric cell is one completed optimization trajectory, not a repeated-seed mean. Custom uses fixed model settings and deterministic EDA/localization tools, whereas Skills CLIs do not expose comparable seed control; we therefore use these tables descriptively, not as statistical significance tests.

The coding-agent baseline (CA), which runs the agent CLI without Skills, rarely improves frequency. Claude CLI never invokes external tools to verify compilation or equivalence, relying instead on re-reading modified files; consequently Opus 4.6 CA exhibits the most syntax (F_s) and LEC (F_l) failures. Codex CLI and Gemini CLI sometimes discover EDA tools (e.g., `iverilog`, `yosys`) by probing the environment, and Codex CLI occasionally compiles the design. However, neither invokes a timing analysis tool even when explicitly instructed to improve frequency. One exception is that Gemini CLI occasionally generates its own SAT-based equivalence checks after discovering tools in the environment.

Overall, Skills and Custom achieve comparable results. Counting the best configuration per benchmark target (bold/underlined in Table V), Opus 4.6 favors Skills on 4 targets and Custom on 3; GPT-5.3 favors Custom on 6 and Skills on 1; Gemini 3.1 favors Custom on 4 and Skills on 3. Neither paradigm consistently dominates.

The largest paradigm gap appears for Opus 4.6: Skills average 26.8% versus 13.9% for Custom, but this comparison is sensitive to outlier cases where exploratory coding agents discover opportunities beyond the locator's scope (Section V-D). For GPT-5.3 (Custom 10.5% vs. Skills 12.7%) and Gemini 3.1 (Custom 18.5% vs. Skills 17.5%), the gap is small, suggesting that the two paradigms achieve similar optimization quality under comparable infrastructure.

Enabling a locator improves Fmax by 13.3% for Custom agents but only 4.5% for Skills agents, which can partially self-localize by exploring the repository. Section V-C analyzes locator designs in detail.

Technique prompts and designwares consistently help: C outperforms C-W by 8.5% and S outperforms S-W by 6.8%. The effect is most pronounced for GPT-5.3, which achieves the best results under C but the worst under C-W, suggesting strong sensitivity to prompt engineering. Gemini 3.1 is more re-

Design	Target	Opus 4.6							GPT-5.3							Gemini 3.1						
		CA	C	C-L	C-W	S	S-L	S-W	CA	C	C-L	C-W	S	S-L	S-W	CA	C	C-L	C-W	S	S-L	S-W
Dino	SingleIssue	F_r	18.5	16.5	16.5	**55.0**	46.0	44.9	**36.9**	22.5	16.5	8.4	22.4	18.1	23.3	F_r	19.2	16.5	19.0	10.8	**43.3**	17.9
	DualIssue	F_r	42.4	3.5	49.6	**50.8**	53.1	19.5	F_r	**56.3**	0.9	6.3	51.4	36.8	34.4	F_r	42.9	3.1	**50.2**	42.4	35.0	39.2
CVA6	Core	F_l	13.2	**29.7**	7.6	15.6	28.0	1.0	F_r	**45.5**	5.2	13.8	10.8	F_r	0.7	F_{cw}	7.9	42.9	21.7	**45.0**	7.3	0.7
	LSU	F_s	**14.7**	1.5	F_r	14.3	10.7	10.3	F_r	**18.3**	F_r	F_r	14.0	2.8	6.6	F_{cw}	**15.7**	14.9	15.5	14.1	11.1	10.6
SoomRV	Frontend	F_s	28.7	F_f	F_r	27.3	**51.4**	51.3	F_r	F_r	F_f	F_r	**26.3**	F_r	F_r	F_l	49.5	F_f	**51.2**	31.1	F_r	28.3
Vortex	Core	F_r	1.2	F_f	3.0	10.4	**17.1**	16.5	F_r	1.2	F_f	**3.1**	1.8	F_r	3.0	F_r	1.2	F_f	**3.0**	0.1	0.6	0.6
	FPU	F_r	19.3	**21.3**	3.5	19.3	21.2	2.8	F_r	**22.6**	F_r	0.7	13.8	F_r	F_r	F_r	12.0	F_r	1.1	1.6	0.7	**26.9**
Average (%)		0.0	19.7	10.4	11.5	27.5	**32.5**	20.5	5.3	**23.8**	3.2	4.6	20.1	8.2	9.7	0.0	21.2	11.1	**23.1**	20.7	14.0	17.7

Table V. FMAX IMPROVEMENT (%) ACROSS CONFIGURATIONS (SEE TABLE IV FOR DEFINITIONS). CA: CODING AGENT BASELINE; C/S: CUSTOM/SKILLS WITH FLOOD LOCATOR; -L: SLANG-HIER ONLY; -W: WITHOUT WARE/TECH PROMPTS. SKILLS AND CUSTOM ACHIEVE SIMILAR FMAX GAINS OVERALL; THE LOCATOR AND WARE/TECH PROMPTS EACH CONTRIBUTE INDEPENDENTLY.

silient and often proposes optimization strategies independently. Because these prompts add negligible tokens and designware retrieval is on-demand, we enable both by default.

B. Time and Cost

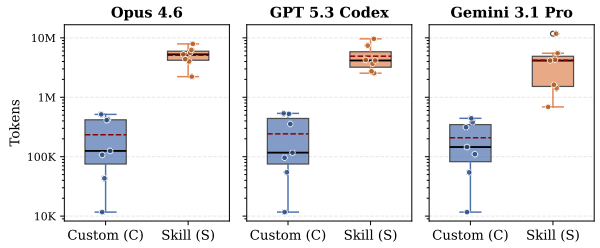

Fig. 2. TOKEN CONSUMPTION FOR SKILLS (S) AND CUSTOM (C). SKILLS USE MORE TOKENS THAN CUSTOM DUE TO SELF-LOCALIZATION EXPLORATION.

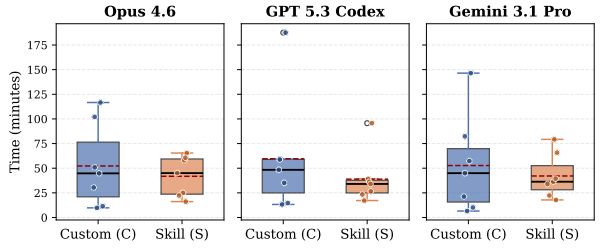

Fig. 3. TOTAL TIME TO COMPLETE FOR SKILLS (S) AND CUSTOM (C). WALL-CLOCK TIME IS DOMINATED BY SYNTHESIS/LEC, NOT THE LLM; BOTH PARADIGMS HAVE SIMILAR TOTAL RUNTIME.

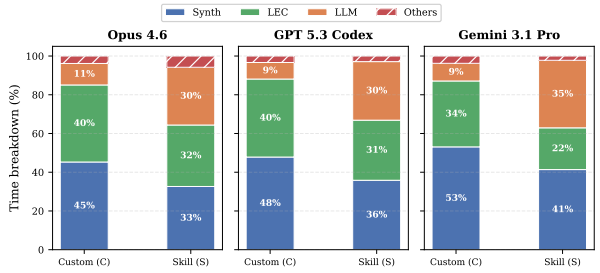

Fig. 4. TIME BREAKDOWN IN SYNTH VS LEC VS LLM VS OTHER FOR SKILLS (S) AND CUSTOM (C). SYNTHESIS AND LEC DOMINATE WALL-CLOCK TIME; REDUCING TOOL LATENCY WOULD BENEFIT BOTH PARADIGMS MORE THAN FASTER LLM INFERENCE.

Skills agents consume substantially more tokens than Custom agents (Figure 2), with median S/C ratios of 32× (Opus 4.6), 23.4× (GPT-5.3), and 14.7× (Gemini 3.1). Custom agents operate on the focused file set from the locator, whereas Skills agents explore the repository autonomously. Opus 4.6 shows the largest gap due to its aggressive grep-based self-localization.

Despite this token gap, total runtime is similar (Figure 3): median S/C ratios are 1.01× (Opus 4.6), 0.75× (GPT-5.3), and 1.05× (Gemini 3.1). Custom runs are slightly slower because synthesis and LEC dominate wall-clock time (Figure 4), and coding agents can overlap tool invocations with exploration, whereas the Custom pipeline executes most steps sequentially. This is an implementation choice rather than a paradigm limit; further parallelizing the Custom pipeline would likely reduce the remaining runtime gap.

LLM inference accounts for only 25.8% of runtime on average; the majority is spent in synthesis and LEC. Faster EDA tools would therefore yield larger end-to-end speedups than faster LLM inference.

C. Locator

We compare three localization strategies: (1) hierarchy-only via `slang-hier`, (2) netlist flooding, and (3) SynAlign probabilistic line-level linking.

For Custom agents, enabling a locator improves Fmax by 13.3% on average. Without it, the pipeline receives fewer candidate files and only observes start/end registers from the timing report, lacking intermediate-signal context about where the bottleneck resides. One exception is CVA6 Core, where the no-locator Custom configuration discovered an ad-hoc writeback-datapath restructuring that the locator-enabled runs missed across multiple temperature settings— illustrating the inherent stochasticity of LLM-driven optimization.

Skills agents are more resilient: locators add only 4.5% on average. Notably, Claude Code's S-L configuration outperforms S by ∼5% because without a locator it explores the codebase more freely, performing broader cross-file analysis that sometimes surfaces opportunities outside the locator's scope. This reinforces the asymmetry of localization utility: for Skills it is often a token-saving hint rather than a hard prerequisite.

The hierarchy-only strategy (`slang-hier`) works only when synthesis preserves register names. In larger designs, retiming often reconstructs names, making RTL mapping

Design	Target	Opus 4.6		GPT-5.3		Gemini 3.1	
		CAL	SAL	CAL	SAL	CAL	SAL
Dino	SingleIssue	11.7	18.8	11.4	13.4	11.8	16.1
	DualIssue	8.9	49.0	8.9	41.6	8.9	46.5
CVA6	Core	6.3	14.0	5.2	7.9	6.5	8.7
	LSU	6.6	11.5	F_r	10.7	14.7	12.4
SoomRV	Frontend	F_f	51.3	F_f	F_r	F_f	27.9
Vortex	Core	1.0	1.2	F_r	1.8	0.6	3.0
	FPU	4.6	16.3	3.6	11.6	2.6	18.8

Table VI. FMAX IMPROVEMENT (%) WITH SYNALIGN LOCALIZATION: CAL (CUSTOM + SYNALIGN) AND SAL (SKILLS + SYNALIGN). COMPARE WITH C/S (FLOOD) AND C-L/S-L (HIERARCHY ONLY) IN TABLE V. SYNALIGN'S LINE-LEVEL PRECISION HELPS WHEN FLOODING IDENTIFIES TOO MANY MODULES, BUT FLOODING'S BROADER SCOPE ENABLES RETIMING-BASED FIXES THAT SYNALIGN MISSES.

impossible—hence the F_f failures for SoomRV Frontend and Vortex Core under C-L.

The *netlist flooding* locator addresses this by traversing outward from critical-path endpoints through the netlist graph, recovering nearby RTL modules even after name reconstruction. Flooding resolves the SoomRV and Vortex failures (average 13.6% Fmax gain) and expands the optimization scope: in Dino DualIssue, C outperforms C-L by 44.7%.

SynAlign provides line-level localization but consistently underperforms flooding, with an average degradation of 17.2% for Custom agents (CAL vs. C in Tables VI and V). Although SynAlign often identifies the correct modules, its probabilistic nature can inject incorrect signal associations that mislead the agent into modifying irrelevant procedural blocks—visible in CVA6 Core and Dino DualIssue. Skills agents are more resilient (SAL degrades only 4.6% vs. S) because they can cross-check localization independently.

These results suggest that *accurate but coarse* localization outperforms *precise but uncertain* localization for LLM-driven RTL optimization. We therefore adopt flooding as the default.

D. Interesting Cases

Several benchmarks reveal distinctive strengths and limitations of each paradigm.

a) Serendipitous optimization (Dino SingleIssue): Claude Code with Skills achieved a much larger Fmax gain than Custom despite incorrectly identifying the ALU as the bottleneck—the ALU was not on the reported critical path. Netlist inspection shows that the modified ALU enabled the synthesis tool to retime the surrounding datapath beneficially. This *serendipitous optimization* illustrates both the upside and unpredictability of exploratory agents: incorrect reasoning about the bottleneck still led to a large improvement because the synthesis tool re-pipelined the design.

b) Emergent codebase exploration (Vortex Core): Synthesis reconstructs register names in Vortex, so all locators identify only a small dual-port RAM wrapper near the critical region. The true bottleneck lies in how that RAM is instantiated within parent modules. Claude Code with Skills went beyond this hint: it grepped the repository to find relevant instantiation sites, inferred that the surrounding memory-banking structure was the real bottleneck, and traced configuration parameters through the highly parameterized codebase before applying a targeted edit.

A human-in-the-loop experiment confirmed this finding: when the files and signals Claude Code discovered were manually injected into the Custom pipeline, all models achieved comparable improvements. The primary advantage here is the Skills agent's ability to autonomously discover design context that fixed localization heuristics miss, rather than access to a fundamentally different optimization primitive.

c) Parameter selection sensitivity (SoomRV Frontend): The main opportunity in SoomRV Frontend is register-array banking of the branch prediction table. Both paradigms detect and apply this transformation, but Fmax varies across runs because agents select different banking factors. When the register-array structure and banking configuration space are made explicit in the prompt, agents consistently select better schemes—suggesting that structural hints improve optimization quality even when agents can already identify the opportunity. These sensitivities explain many F_r entries: equivalent patches can pick poor banking factors or miss parent context, lowering Fmax.

E. Chaining Models

Under the same Custom pipeline, GPT-5.3 consistently achieves the strongest results, particularly on Dino DualIssue and CVA6 Core. When other models' optimized outputs are fed back into GPT-5.3, it can improve them further but does not reach the Fmax it achieves from the original code. This suggests that early architectural decisions strongly influence the final achievable frequency—once the design evolves along one optimization trajectory, later improvements are constrained.

Conversely, when GPT-5.3 fails to improve a design (e.g., SoomRV Frontend), passing its unchanged output to other models allows them to discover improvements, suggesting that model diversity can be beneficial.

VI. CONCLUSIONS

Skills and Custom agents achieve comparable Fmax gains overall. Skills consume over 20× more tokens due to self-localization but are not slower—synthesis and LEC dominate wall-clock time (∼75%). Each configuration dimension—locator, ware/tech prompts, and model selection—contributes measurable improvements. Localization is critical for Custom agents but optional for Skills, which can self-localize through emergent grepping. Technique prompts consistently help both paradigms.

Skills agents suit rapid deployment or unfamiliar designs where exploratory tool use can surface optimization targets beyond pre-programmed locators. Custom agents suit workflows requiring inspectable, reproducible steps. The two can be combined: a Custom pipeline can leverage a coding agent's self-localization as secondary validation of its locator output and occasionally surface missed opportunities. Reducing EDA tool latency would yield larger end-to-end speedups than faster LLM inference.

ACKNOWLEDGMENTS

We like to thank the reviewers for their feedback on the paper. This work was partially funded, Google Academic Research Award 2024, and Amazon Research Award 2025, and the National Science Foundation under grant 2504580. Any opinions, findings, and conclusions or recommendations expressed herein are those of the authors and do not necessarily reflect the views of the funding agencies.

REFERENCES

[1] Y. Wang, W. Ye, Y. He, Y. Chen, G. Qu, and A. Li, "MCP4EDA: LLM-powered model context protocol RTL-to-GDSII automation with backend aware synthesis optimization," *arXiv preprint arXiv:2507.19570*, 2025. [Online]. Available: https://arxiv.org/abs/2507.19570

[2] S. Garg and J. Renau, "Aligning Netlist to Source Code using SynAlign," 2025. [Online]. Available: https://arxiv.org/abs/2501.00921

[3] X. Yao, Y. Wang, X. Li, Y. Lian, R. Chen, L. Chen, M. Yuan, H. Xu, and B. Yu, "Rtlrewriter: Methodologies for large models aided rtl code optimization," in *Proceedings of the 43rd IEEE/ACM International Conference on Computer-Aided Design*, ser. ICCAD '24. New York, NY, USA: Association for Computing Machinery, 2025. [Online]. Available: https://doi.org/10.1145/3676536.3676775

[4] Y. Wang, W. Ye, P. Guo, Y. He, Z. Wang, B. Tian, S. He, G. Sun, Z. Shen, S. Chen, A. Srivastava, Q. Zhang, G. Qu, and A. Li, "SymRTLO: Enhancing RTL code optimization with LLMs and neuron-inspired symbolic reasoning," *arXiv preprint arXiv:2504.10369*, 2025. [Online]. Available: https://arxiv.org/abs/2504.10369

[5] Z. Xu, B. Li, and L. Wang, "Rethinking llm-based rtl code optimization via timing logic metamorphosis," 2025. [Online]. Available: https://arxiv.org/abs/2507.16808

[6] K. Thorat, J. Zhao, Y. Liu, A. Hasan, H. Peng, X. Xie, B. Lei, and C. Ding, "LLM-VeriPPA: Power, performance, and area optimization aware Verilog code generation with large language models," in *2025 ACM/IEEE 7th Symposium on Machine Learning for CAD (MLCAD)*, 2025, pp. 1–7.

[7] X. Zhou, Y. Sheng, X. Chen, S. Zhang, J. Li, and Z. Tang, "EvolVE: Evolutionary search for LLM-based Verilog generation and optimization," *arXiv preprint arXiv:2601.18067*, 2025. [Online]. Available: https://arxiv.org/abs/2601.18067

[8] K. Tasnia, A. Garcia, T. Farheen, and S. Rahman, "VeriOpt: PPA-aware high-quality Verilog generation via multi-role LLMs," *arXiv preprint arXiv:2507.14776*, 2025, also in IEEE Xplore, DOI: 10.1109/ICLAD65657.2025.11240771. [Online]. Available: https://arxiv.org/abs/2507.14776

[9] J. Wang, Z. Li, L. Li, F. He, L. Lin, Y. Lai, Y. Li, X. Zeng, and Y. Guo, "Principle-guided Verilog optimization: IP-safe knowledge transfer via local-cloud collaboration," 2025. [Online]. Available: https://arxiv.org/abs/2508.05675

[10] F. Rabiei, M. Zakharov, and J. Renau, "Beyond verilog: Agents for emerging hdls," in *2025 IEEE International Conference on Omni-layer Intelligent Systems (COINS)*, 2025, pp. 1–7.

[11] H. Ping, S. Li, P. Zhang, A. Cheng, S. Duan, N. Kanakaris, X. Xiao, W. Yang, S. Nazarian, A. Irimia, and P. Bogdan, "HDLCoRe: A training-free framework for mitigating hallucinations in LLM-generated HDL," in *2025 IEEE International Conference on LLM-Aided Design (ICLAD)*, 2025, pp. 108–116.

[12] H. Ping, A. Bhattacharjee, P. Zhang, S. Li, W. Yang, A. Cheng, X. Zhang, J. Thomason, A. Jannesari, N. Ahmed, and P. Bogdan, "VeriMoA: A mixture-of-agents framework for spec-to-HDL generation," 2026. [Online]. Available: https://arxiv.org/abs/2510.27617

[13] N. Zhang, C. Deng, J. M. Kuehn, C.-T. Ho, C. Yu, Z. Zhang, and H. Ren, "ASPEN: LLM-guided e-graph rewriting for RTL datapath optimization," in *International Symposium on Machine Learning for CAD (MLCAD)*, 2025, pp. 1–9. [Online]. Available: https://ieeexplore.ieee.org/document/11189222

[14] A. Ghose, A. B. Kahng, S. Kundu, and Z. Wang, "ORFS-agent: Tool-using agents for chip design optimization," *arXiv preprint arXiv:2506.08332*, 2025. [Online]. Available: https://arxiv.org/abs/2506.08332

[15] W. Fang, S. Liu, H. Zhang, and Z. Xie, "Annotating slack directly on your verilog: Fine-grained rtl timing evaluation for early optimization," 2024. [Online]. Available: https://arxiv.org/abs/2403.18453

[16] R. V. Hemadri, J. Bhandari, A. Nakkab, J. Knechtel, B. P. Gopalan, R. Narayanaswamy, R. Karri, and S. Garg, "Veriloc: Line-of-code level prediction of hardware design quality from verilog code," 2025. [Online]. Available: https://arxiv.org/abs/2506.07239

[17] W. Lv, Y. Xia, X. Chen, and L. Kuang, "Vitad: Timing violation-aware debugging of rtl code using large language models," 2025. [Online]. Available: https://arxiv.org/abs/2508.13257

[18] Liu, Mingjie and Pinckney, Nathaniel and Khailany, Brucek and Ren, Haoxing, "VerilogEval: Evaluating Large Language Models for Verilog Code Generation," *arXiv preprint arXiv:2309.07544*, 2023.

[19] Nathaniel Pinckney and Chenhui Deng and Chia-Tung Ho and Yun-Da Tsai and Mingjie Liu and Wenfei Zhou and 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," 2025. [Online]. Available: https://arxiv.org/abs/2506.14074

[20] Ghorab, Razine Moundir and Parisi, Emanuele and Gutierrez, Cristian and Alberti-Binimelis, Miquel and Moreto, Miquel and Garcia-Gasulla, Dario and Kestor, Gokcen, "NotSoTiny: A Large, Living Benchmark for RTL Code Generation," *arXiv preprint arXiv:2512.20823*, 2025.

[21] P. Jin, D. Huang, C. Li, S. Cheng, Y. Zhao, X. Zheng, J. Zhu, S. Xing, B. Dou, R. Zhang, Z. Du, Q. Guo, and X. Hu, "RealBench: Benchmarking Verilog generation models with real-world IP designs," 2025. [Online]. Available: https://arxiv.org/abs/2507.16200

[22] Y. Wang, W. Ye, P. Guo, Y. He, Z. Wang, B. Tian, S. He, G. Sun, Z. Shen, S. Chen, A. Srivastava, Q. Zhang, G. Qu, and A. Li, "A new benchmark for the appropriate evaluation of RTL code optimization," *arXiv preprint arXiv:2601.01765*, 2025. [Online]. Available: https://arxiv.org/abs/2601.01765

[23] Jason Wei and Xuezhi Wang and Dale Schuurmans and Maarten Bosma and Brian Ichter and Fei Xia and Ed Chi and Quoc Le and Denny Zhou, "Chain-of-Thought Prompting Elicits Reasoning in Large Language Models," 2023.

[24] "slang - SystemVerilog Language Services," https://github.com/MikePopoloski/slang, online; accessed on 30 June 2026. [Online]. Available: https://github.com/MikePopoloski/slang