

Accelerating FPGA Physical Design Closure through Generative AI

Mrunal Sagare
Lattice Semiconductor
Pune, Maharashtra, India
mrunal.sagare@latticesemi.com

Abhishek Gade
Lattice Semiconductor
Pune, Maharashtra, India
abhishek.gade@latticesemi.com

Rajashri Lakhotia
Lattice Semiconductor
Pune, Maharashtra, India
rajashri.lakhotia@latticesemi.com

Abstract— Channel-less full-abutment FPGA physical design involves several mechanically repetitive tasks that collectively consume hundreds of engineering hours per iteration. This paper presents four targeted Generative AI applications integrated into a production Cadence Innovus and Calibre flow, implemented at the 28 nm process node. The first and most mature application, GenParser, is a purpose-built log-triage tool using the GPT-4 API with retrieval-augmented generation over engineer-verified Innovus log examples, reducing triage time from several hours to 2–5 minutes per run. The remaining three applications are structured prompt-engineering workflows validated as proof-of-concept implementations using Claude (Anthropic): PnR constraint authoring, feedthrough pin mapping, and combined post-route ECO and DRC closure. Each is designed with a clear pathway to full pipeline automation. Constraint authoring time fell from 40–50 hours to under 30 minutes per block, with zero script errors observed. Post-route closure tasks were accelerated by approximately 300× combined across ECO and DRC sign-off. All four applications integrate with existing EDA flows without tool modification and keep the engineer in the review loop for all generated outputs.

Keywords— *FPGA Physical Design, Cadence Innovus, Calibre DRC, Generative AI, Retrieval-Augmented Generation, PnR Automation, Prompt Engineering.*

I. INTRODUCTION

Channel-less, full-abutment FPGA physical design places unusual demands on implementation. Logic tiles share physical boundaries with no routing channels between them, so every inter-block connection must be planned as a feedthrough occupying a specific routing track through each intermediate tile. This makes the design highly sensitive to how routing constraints are specified, how feedthrough pins are placed, and how post-route violations are corrected. Each of these tasks is mechanically repetitive and requires exact knowledge of the EDA tool interface, but produces output that an engineer can inspect and validate. That combination is where current Generative AI models are most effective.

This paper has focused on following physical design tools from among many popular options available: Cadence Innovus for place-and-route and Calibre v2025.1_16.10 for DRC sign-off on a 28 nm FPGA fabric. Four tasks consistently account for the largest share of manual engineering time: translating IP

specifications from Excel into Tcl constraint scripts, matching feedthrough port names across adjacent block boundaries, triaging Innovus log files, and resolving post-route ECO and DRC violations.

This paper describes four AI applications targeting these tasks. GenParser, is a purpose-built tool using the GPT-4 API with a retrieval layer over verified Innovus log examples; it runs as part of the daily design flow. The other three are proof-of-concept implementations using structured prompt engineering with Claude (Anthropic): the engineer formats the relevant EDA data into a prompt, LLM (Large Language Model) loop generates the output, and the engineer reviews it before applying it. Each proof-of-concept workflow is designed so that the manual extraction step could be replaced by a straightforward Python script calling the Claude API, turning it into a fully automated LLM pipeline. Section II reviews related work. Section III describes the flow and framework. Section IV describes GenParser. Section V describes the three structured workflows. Section VI reports results. Section VII discusses limitations.

II. LITERATURE REVIEW

Transformer-based language models [2] have demonstrated strong capability as code generators, opening practical opportunities to automate structured engineering tasks. Lewis et al. [1] introduced retrieval-augmented generation (RAG), which allows a model to draw on verified external knowledge at inference time without retraining. This is the mechanism behind GenParser, where GPT-4 is grounded in engineer-verified Innovus log examples rather than its general training data.

Liu et al. [5] showed with ChipNeMo that adapting an LLM on EDA documentation improves its accuracy on tool-specific queries. This motivated a key design decision in our structured workflows: every prompt explicitly provides the correct Innovus command syntax rather than relying on the model's general knowledge of Tcl. He et al. [10] survey LLM applications across synthesis, verification, and physical implementation, and observe that tasks with well-defined, structured output formats are consistently the most tractable for current models, consistent with our experience.

On code generation, Thakur et al. [13] found at DATE 2023 that including correct-syntax examples in the prompt was the

single most important factor in LLM output quality for hardware tasks. Liu et al. [6] introduced VerilogEval and Lu et al. [8] built RTLLM to benchmark RTL generation quality. Blocklove et al. [7] found in Chip-Chat that iterative conversational refinement substantially improves output quality for hardware-specific tasks. GPT-4 [3] and Claude [4] are the models used in this work. Liu et al. [9] demonstrated that AI assistance can reduce cycle time in formal verification workflows, a directly comparable result in an EDA context adjacent to physical design. Murray et al. [11] and the Cadence Innovus system [12] provide the FPGA implementation context throughout.

III. PHYSICAL DESIGN FLOW AND FRAMEWORK

A. Flow Overview

Figure 1 shows the physical design flow for a full-abutment FPGA tile block. Starting from an IP connectivity specification in Excel, the flow runs through floorplanning, power-grid implementation, clock-tree synthesis, place-and-route, timing and signal-integrity sign-off, and Calibre DRC verification. The four AI-assisted stages are highlighted.

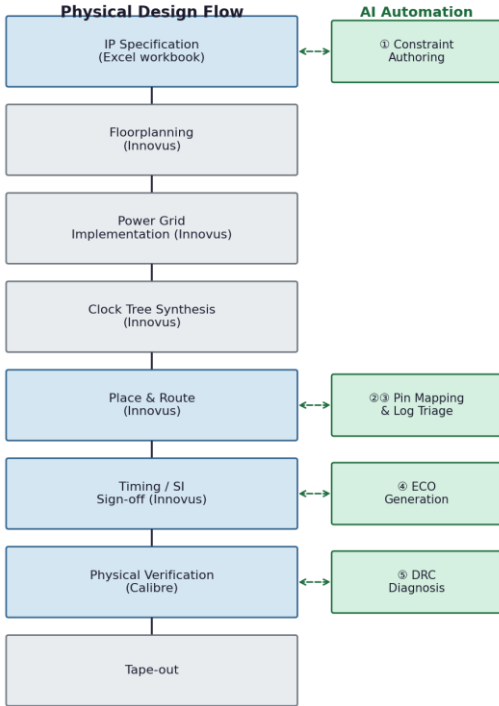


Fig. 1. Physical design flow with four AI-assisted stages (highlighted).

B. IP Blocks Evaluated

The applications were developed and evaluated on IP blocks from the same FPGA fabric family, all at the 28 nm process node. Table I summarises the blocks.

TABLE I. IP BLOCKS EVALUATED

Block	Process	Signals	Applications
Fabric Tile A	28 nm	20k	①②③④
Fabric Tile B	28 nm	10k	①②③④
IO Block C	28 nm	6k	①③④

C. Framework Architecture

The four applications fall into two tiers. GenParser (Application 1) is a tool: it calls the GPT-4 API programmatically, maintains a retrieval database of engineer-verified log examples, and runs automatically after each Innovus run. Applications 2, 3, and 4 are proof-of-concept structured prompt-engineering workflows. For each, a prompt template was developed that provides claude-opus-4-5 with the relevant EDA context and the required tool-specific command syntax. In the current implementation, the engineer extracts the input data manually, queries Claude through its chat interface, and reviews the output before applying it. Each workflow is structured so that the manual extraction step can be replaced by a Python script calling the Claude API, making the transition to a fully automated pipeline a straightforward engineering step rather than a research problem. No modifications to Innovus or Calibre are required in any of the four applications.

IV. APPLICATION 1: GENPARSER — AUTOMATED INNOVUS LOG TRIAGE

A single Innovus run on a complex block produces a log file with tens of thousands of lines spanning timing, routing, clock-tree synthesis, DRC, and tool housekeeping. Triageing this output to identify genuine problems and determine corrective actions has historically taken several hours per run, particularly for engineers who are new to the specific design or tool version.

A. Architecture

GenParser is built on a three-tier architecture: a structured knowledge base of verified error–solution pairs collected from previous design runs, a GPT-4 API reasoning layer, and pattern-matching heuristics for high-frequency error signatures. Each knowledge-base entry contains the original log message, its engineer-verified root cause, and the corrective Tcl command that resolved it. When GenParser processes a new log, it searches the knowledge base for entries similar to the messages in the new file and includes the closest matches in the context passed to GPT-4. This grounds the model's diagnosis in real, verified information from the same tool version and design environment, which is the core benefit of the RAG architecture: GPT-4 is not reasoning from general training data about Innovus but from engineer-confirmed examples of what specific log messages mean in this flow.

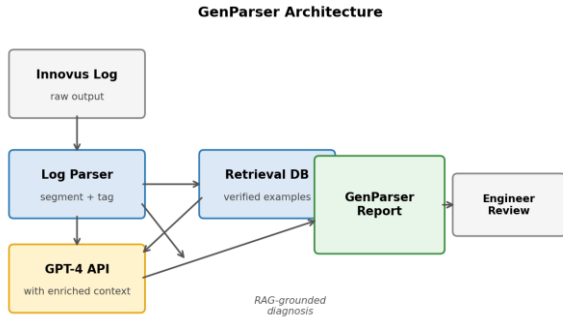


Fig. 2. GenParser architecture: log parser, retrieval from verified-example database, and GPT-4 diagnosis.

B. Operation

GenParser segments the Innovus log into individual message records and tags each by category: timing, routing, clock-tree, DRC, or informational. For each non-informational record, it retrieves the closest matching database entries and passes the enriched context to GPT-4, requesting a root cause, a severity classification (must-fix, investigate, or ignore), and a corrective Tcl command where applicable. The output is a structured triage report grouped by severity, delivered within 2–5 minutes of the Innovus run completing. A representative report entry is shown below.

```

# GenParser TRIAGE REPORT (excerpt)
CATEGORY : Timing          SEVERITY : Must-fix
MSG      : "TNS: -312.4ps WNS: -48.6ps (CLK_CORE
setup)"

ROOT CAUSE
Net DATA_BUS[3] has no NDR assigned. Routed
length 618um exceeds the 500um constraint;
high-congestion path adds parasitic delay.

SUGGESTED FIX
set_route_attributes -nets {DATA_BUS[3]} -
route_rule NDR_2x2
route_opt -effort high -timing_driven \
-nets {DATA_BUS[3]}

SIMILAR CASE (DB): CLK_BUS[1], same block family.
Post-fix WNS improved: -51.3ps to +8.7ps.

```

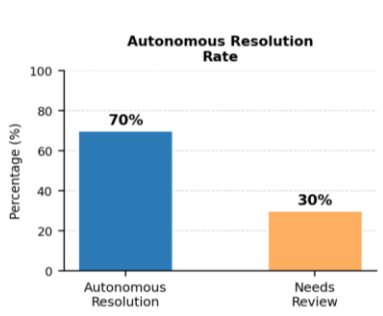


Fig. 3. GenParser triage: autonomous resolution rate.

C. Incremental Improvement

The retrieval database grows with each design iteration: novel resolved errors are annotated and added as new entries, so GenParser improves over time on the same design family. A planned extension is a design-health dashboard that tracks timing, congestion, and DRC metrics across successive runs and surfaces regressions automatically, removing the need to compare run reports manually.

V. APPLICATIONS 2–4: STRUCTURED PROMPT-ENGINEERING WORKFLOWS

Applications 2, 3, and 4 are proof-of-concept implementations of AI-assisted workflows using structured prompting with Claude. In each case, a prompt template was developed that provides Claude with the EDA context it needs along with the exact Innovus or Calibre command syntax for the task. The engineer supplies the input data and reviews the output. The time savings reported in Section VI are measured from this proof-of-concept implementation and include the engineer's extraction and review time.

Each workflow is designed with pipeline automation in mind. The input to every prompt is structured and machine-readable: an Excel workbook, a DEF file, Innovus output, or a Calibre DRC file. A Python script reading each input and calling the Claude API would implement the same workflow without manual extraction, which is the natural next step for any of the three workflows once the proof of concept is validated.

A. Application 2: PnR Constraint Authoring

Before starting with PNR process, the engineer requires several input files including netlist, floorplan, routing and timing constraints. Application 2 automates PnR constraint authoring by translating IP connectivity specifications from Excel into Innovus-ready Tcl scripts using claude-opus-4-5. For each IP block, the signal data is extracted in the form of workbook and formatted into a structured prompt that specifies the target tool and provides the correct Innovus command form for each constraint categories like create_route_rule, set_route_attributes, create_pg_stripe, etc. along with one reference example per category. This explicit syntax grounding is the key design decision: without it, the model produces plausible but tool-incorrect commands. Claude returns the complete constraint script through its chat interface; the engineer reviews it for net-name consistency and command correctness before loading it into Innovus. Review takes under 10 minutes per block. The prompt structure is designed to be pipeline-compatible: a Python script reading the workbook and calling the Claude API would implement the same workflow without the manual extraction step. Across all IP blocks evaluated, constraint-authoring time was reduced from 40–50 engineering hours to under 30 minutes, with zero script-generation errors requiring a correction cycle.

INPUTS to claude opus:

pins of IO	net name	Connecting cell	Connecting pin	Instance name
Io_1	Net_1	fcint_cell_io	Pin_1	inst_io_1
Io_2	Net_2	fcint_cell_io	Pin_2	inst_io_1
Io_3	Net_3	fcint_cell_io	Pin_3	inst_io_1
Io_4	Net_4	fcint_cell_io	Pin_4	inst_io_1
Io_5	Net_5	fcint_cell_io	Pin_5	inst_io_1

pins	route rule	routing layer	buffering	yes	no	sharding
cfg_to_adc	create_route_rule -name config_bus_1 -width {M5 0.2 M6 0.2} -spacing {M5 0.2 M6 0.2} -hard_spacing	M5,M6	yes	450 um		single sharding
cfg_to_sspi_1_10	create_route_rule -name sspi -width {M3 0.05 M4 0.05 M5 0.2 M6 0.15} -spacing {M3 0.05 M4 0.05 M5 0.2 M6 0.15} -hard_spacing	M5,M6	yes	450 um		
cfg_to_sspi_1_11	create_route_rule -name sspi -width {M3 0.05 M4 0.05 M5 0.2 M6 0.15} -spacing {M3 0.05 M4 0.05 M5 0.2 M6 0.15} -hard_spacing	M5,M6	yes	450 um		
cfg_to_sspi_1_12	create_route_rule -name sspi -width {M3 0.05 M4 0.05 M5 0.2 M6 0.15} -spacing {M3 0.05 M4 0.05 M5 0.2 M6 0.15} -hard_spacing	M5,M6	yes	450 um		
cfg_to_sspi_1_13	create_route_rule -name sspi -width {M3 0.05 M4 0.05 M5 0.2 M6 0.15} -spacing {M3 0.05 M4 0.05 M5 0.2 M6 0.15} -hard_spacing	M5,M6	yes	450 um		
cfg_to_sspi_1_14	create_route_rule -name sspi -width {M3 0.05 M4 0.05 M5 0.2 M6 0.15} -spacing {M3 0.05 M4 0.05 M5 0.2 M6 0.15} -hard_spacing	M5,M6	yes	450 um		group sharding
cfg_to_sspi_1_15	create_route_rule -name sspi -width {M3 0.05 M4 0.05 M5 0.2 M6 0.15} -spacing {M3 0.05 M4 0.05 M5 0.2 M6 0.15} -hard_spacing	M5,M6	yes	450 um		
cfg_to_sspi_1_16	create_route_rule -name sspi -width {M3 0.05 M4 0.05 M5 0.2 M6 0.15} -spacing {M3 0.05 M4 0.05 M5 0.2 M6 0.15} -hard_spacing	M5,M6	yes	450 um		
cfg_to_sspi_1_17	create_route_rule -name sspi -width {M3 0.05 M4 0.05 M5 0.2 M6 0.15} -spacing {M3 0.05 M4 0.05 M5 0.2 M6 0.15} -hard_spacing	M5,M6	yes	450 um		
cfg_to_sspi_1_18	create_route_rule -name sspi -width {M3 0.05 M4 0.05 M5 0.2 M6 0.15} -spacing {M3 0.05 M4 0.05 M5 0.2 M6 0.15} -hard_spacing	M5,M6	yes	450 um		

```
# PROMPT (excerpt)
Generate Innovus Tcl for the signals below.
Command forms to use:
  NDR      : create_route_rule -name <N> -spacing {...}
             -width {...}
             set_route_attributes -nets {<net>} -
route_rule <N>
  Length   : set_route_attributes -nets {<net>} (add
max_length)
  Shield   : set_route_attributes ... -shield_net
<supply>
             -shield_sides
both
  PG       : create_pg_stripe -nets {<net>} -layer <L>
             -width <W>
```

TCL Files and netlist generated by Claude:

```
# CLAUDE OUTPUT
Answer 1 : module top (vss, vcc);
  inout vss;
  inout vcc;
  wire Net_1;
  wire Net_2;
  wire Net_3;
  wire Net_4;
  wire Net_5;
  fcint_piolo piolo2 (
    .Io_1 (Net_1),
    .Io_2 (Net_2),
    .Io_3 (Net_3),
    .Io_4 (Net_4),
    .Io_5 (Net_5)
  );
  fcint_cell_io inst_io_1 (
    .Pin_1 (Net_1),
    .Pin_2 (Net_2),
    .Pin_3 (Net_3),
    .Pin_4 (Net_4),
    .Pin_5 (Net_5)
  );
endmodule

Answer 2:
# — NDR Rule Definitions
—

create_route_rule -name config_bus_1 \
  -width {M5 0.2 M6 0.2} \
  -spacing {M5 0.2 M6 0.2} \
  -hard_spacing
create_route_rule -name sspi \
  -width {M3 0.05 M4 0.05 M5 0.2 M6 0.15} \
  -spacing {M3 0.05 M4 0.05 M5 0.2 M6 0.15} \
  -hard_spacing
set_route_attributes -nets {cfg_to_adc} \
  -route_rule config_bus_1 \
  -preferred_top_routing_layer M6 \
  -preferred_bottom_routing_layer M5 \
```

```
-shield_net VSS -shield_sides single
```

```
set_route_attributes -nets {cfg_to_sspi_1_10
cfg_to_sspi_1_11} \
  -route_rule sop_width \
  -preferred_top_routing_layer M6 \
  -preferred_bottom_routing_layer M3 \
  -max_length 450
```

Similar methodology can be used for generating various constraint files for power grids and high frequency signal constraints. We have tested this method on multiple designs and it drastically reduced the manual overload and made the process much faster.

Across all blocks evaluated, no generated script required a correction cycle before being accepted into Innovus. The pipeline path for this workflow is direct: a Python script reads the Excel workbook row by row, formats each signal's constraint parameters into the prompt template, calls the Claude API, and writes the returned Tcl to a file.

Constraint Authoring Pipeline

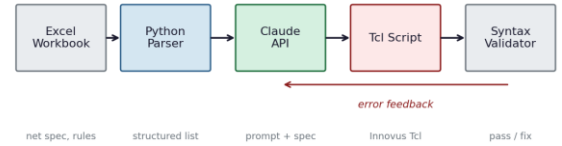


Fig. 4. Constraint authoring: from Excel workbook to reviewed Innovus Tcl.

B. Application 3: Feedthrough Pin Mapping

In full-abutment FPGA designs, feedthrough signals must land on identical routing tracks on both sides of each shared block boundary. Port names on adjacent blocks are frequently inconsistent because different design teams assign them independently. Matching port lists and generating geometrically compatible assign_pin commands is slow and error-prone, particularly for blocks with large feedthrough counts.

For each boundary, the port list from both blocks is extracted from their DEF files, including layer, width, height, signal direction, and placed coordinates for the confirmed left-block pins. This structured data is formatted into a prompt alongside the exact command syntax. Then the port name semantics, bus width, signal direction, and layer compatibility are used to infer the correspondence for each port pair, assigns a confidence level to each match, and generates a Tcl script for every confirmed pair using the left-block geometry as the reference. The engineer reviews high-confidence matches in under 10 minutes and resolves any low-confidence or flagged ports manually before sourcing the output into Innovus.

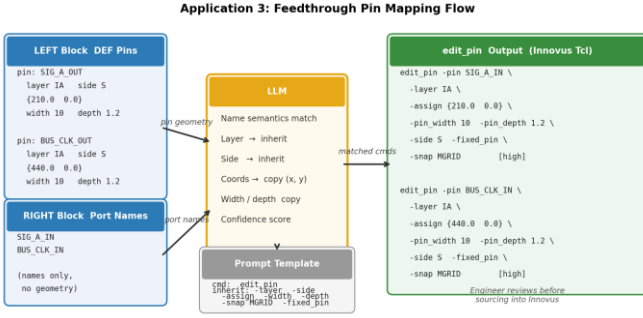


Fig. 4. Constraint authoring: from Excel workbook to reviewed Innovus Tcl.

Across all boundaries evaluated, pin mapping time was reduced from 5 engineering hours to under 10 minutes per boundary. The pipeline path for this workflow is direct: a Python script extracts pin records from both DEF files, formats the structured prompt, calls the Claude API, and writes the assign_pin commands to a Tcl file that can be sourced directly into Innovus without further manual formatting.

C. Application 4: Post-Route Closure -ECO and DRC

Post-route closure is typically the most time-intensive phase of the physical design cycle, encompassing iterative signal-integrity ECO loops and Calibre DRC sign-off. Slew and capacitance violations on high-fanout nets require manual inspection of net topology, parasitic estimates, and driver strength before a buffer-insertion plan can be formulated. Antenna violations from the Calibre DRC deck similarly demand cross-referencing polygon-level error geometry against routed net data in the DEF, this lookup is systematic but slow when performed manually across hundreds of violations per sign-off run. The fourth application addresses both tasks through a unified LLM-assisted closure framework, exploiting the fact that both workflows share the same proof-of-concept structure: structured EDA output in, execution-ready Tcl commands out.

For signal-integrity ECO generation, the net topology, driver cell specification, per-sink fanout load figures, slew-violation magnitudes, available buffer cell library, and routing layer and segment length are extracted from Innovus reports and provided to the LLM with the exact eco_add_repeater command syntax. The model determines optimal repeater insertion points based on load distribution and segment length, proposes a buffer cell selection consistent with the target-load specification, and generates execution-ready ECO Tcl directly. This eliminates the manual net-by-net inspection loop, enabling an engineer to close a full ECO iteration which was previously spanning several hours, in under 10 minutes.

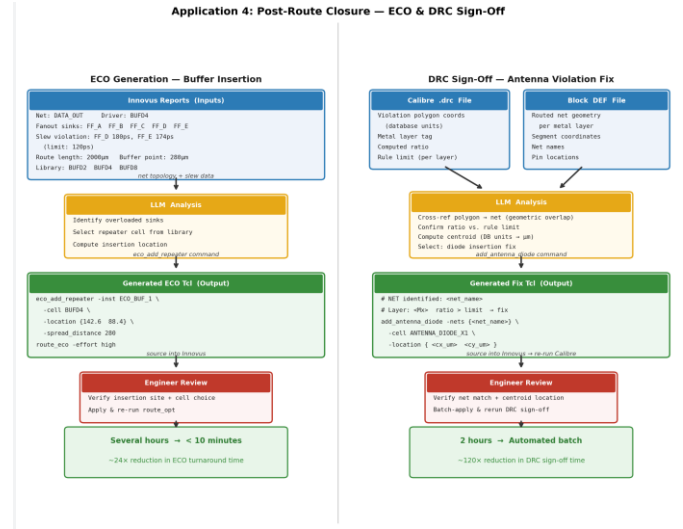


Fig. 4. Constraint authoring: from Excel workbook to reviewed Innovus Tcl.

For Calibre DRC antenna sign-off, the .drc results file and the routed block DEF are passed to the LLM together with the relevant rule specification. Calibre antenna rules check accumulated gate-area ratios across multiple metal layers and process-specific geometries; violations are reported as polygon coordinates in database units referencing the offending layer and net segment. The model parses the violation records, cross-references the error polygon geometry against routed net data in the DEF to identify the offending net on the flagged layer, confirms the computed ratio against the applicable rule limit, and generates the add_antenna_diode command with a placement location computed at the violation centroid, converted from database units to microns. The same prompt structure handles violations across different metal layers and antenna rule variants without modification, since the input format is violation polygon, layer tag, rule limit, DEF routing which is consistent across rule categories. This path requires no manual net lookup and produces a complete, sourceable fix script per Calibre run. The DRC workflow is the closest of the three structured proof-of-concept applications to full pipeline automation: both input files are machine-readable, and a Python script calling the LLM API can replace the manual extraction step entirely, driving toward a zero-touch antenna sign-off loop.

VI. RESULTS

Table II summarises the measured outcomes. Manual baseline figures are taken from design-activity logs from the previous implementation iteration. AI-assisted times include the engineer's time to prepare the prompt input, review the generated output, and apply it; they do not include model response time, which is under one minute in all cases. For the three structured prompt workflows (Applications 2, 3, 4), the times also include the manual extraction step; a pipeline

implementation would reduce AI-assisted time further by eliminating that step.

TABLE II. MEASURED OUTCOMES

Application	Manual Baseline	AI-Assisted	Reduction
① GenParser — Log triage (per run)	Several hr	2–5 min	$\sim 70\times$
② Constraint authoring (per block)	40–50 hr	<30 min	$>100\times$
③ Pin mapping (per boundary)	5 hr	10 min	$\sim 30\times$
④ Post-route closure (ECO + DRC, per batch)	Several hr	10 min	$\sim 300\times$

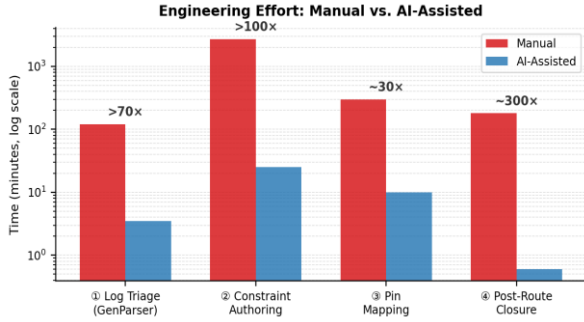


Fig. 6. Engineering effort comparison: manual baseline vs. AI-assisted time across all four applications.

The constraint authoring result is the most significant in absolute terms. Reducing a 40–50 hour task per IP block to under 30 minutes fundamentally changes the iteration model: a constraint update that previously deferred to the next working day can now be completed before the next Innovus run queues. Across all IP blocks evaluated, zero generated scripts required a correction cycle before being accepted into the implementation flow. This zero-error rate is a direct consequence of providing the exact Innovus command syntax in the prompt; without it, the model produces commands that are syntactically plausible but tool-incorrect and rejected at load time. GenParser reduced per-run log triage from several hours to 2–5 minutes ($\sim 70\times$ reduction), and reduced time-to-root-cause by 15–20% compared to manual triage. Of the error categories covered by the knowledge base, 70% were resolved autonomously without engineer intervention. The more significant outcome is qualitative: the engineer receives a structured, severity-ranked, actionable triage report within minutes of an Innovus run completing, rather than beginning a manual scan of a multi-thousand-line log file. The three-tier architecture ensures the diagnosis is grounded in verified error–solution pairs from the same tool version and design family, eliminating the class of false diagnoses that arise from reasoning over general LLM training data alone. Feedthrough pin mapping per block boundary was reduced from 5 hours to approximately 10 minutes ($\sim 30\times$). The LLM’s ability to

resolve port correspondences despite inconsistent naming conventions — using name semantics, bus width, signal direction, and layer compatibility together — is the key capability enabling this reduction. High-confidence matches required no manual intervention beyond a final review pass; uncertain or low-confidence ports were flagged explicitly and resolved manually. For ECO generation, buffer-insertion turnaround was reduced from several hours to under 10 minutes per batch ($\sim 300\times$). The dominant saving is the elimination of the manual net-by-net inspection loop. For Calibre antenna DRC, the per-batch sign-off cycle was reduced from approximately 2 hours to a fully automated script-generation step (). Violations across multiple metal layers and antenna rule categories were handled by the same prompt structure without modification, confirming that the approach generalises across process rule variants without per-rule prompt engineering.

Two observations apply across all four applications. First, prompt specificity is the primary determinant of output quality. Providing the correct Innovus or Calibre command form in the prompt, with one reference example per constraint type, was the difference between tool-correct and tool-incorrect output. Second, the proof-of-concept results already demonstrate substantial time savings even with manual extraction; pipeline automation — replacing manual data extraction with Python scripts calling the LLM API directly from EDA tool output — would reduce the reported AI-assisted times further by eliminating the extraction step entirely. These results collectively demonstrate that targeted, domain-grounded LLM integration can substantially compress physical-design cycle time and reduce manual error without disrupting established EDA flows.

VII. LIMITATIONS

The manual baseline figures for Applications 3 and 4 are taken from activity logs and engineer estimates rather than controlled reruns. The absolute times behind the reported reductions are therefore approximate.

Applications 2, 3, and 4 are proof-of-concept implementations: the engineer manually extracts the input data and queries Claude through its chat interface. The extraction time is included in the AI-assisted figures in Table II, but it represents an avoidable cost. Each workflow is designed so that this step can be automated, but that automation has not yet been built and validated in production.

GenParser’s retrieval database is specific to the Innovus version and design family. Applying it in a different environment would require rebuilding the database from locally verified log examples.

All four applications keep the engineer in the review loop: no generated output is applied to the design without human inspection. This is a deliberate design decision, not a limitation of the underlying models. It reflects the practical reality that errors in constraint scripts, pin assignments, ECO commands, or waiver justifications have real consequences in a production design flow and that current models are not yet reliable enough to apply their outputs autonomously in that context.

Finally, all four applications required a one-time prompt-engineering investment of one to two days per workflow during initial development. This setup cost is not reflected in the per-iteration figures in Table II.

VIII. CONCLUSION

This paper has described four Generative AI applications integrated into a production FPGA physical design flow. GenParser, the most mature, uses GPT-4 with retrieval-augmented generation over verified Innovus log. The other three are proof-of-concept structured prompt-engineering workflows using Claude, each validated on production IP blocks and each designed with a clear path to full pipeline automation.

The core contribution is a practical framework for identifying where Generative AI assistance is most effective in a physical design flow: tasks with structured, machine-readable inputs, well-defined output formats, and outputs that an engineer can validate. When these conditions are met and the correct tool syntax is provided explicitly in the prompt context, current language models produce tool-correct outputs reliably. Constraint authoring time fell from 40–50 hours to under 30 minutes per block, with zero script errors observed. GenParser reduced log triage from several hours to 2–5 minutes per run. Post-route ECO and DRC closure activities were accelerated by approximately 300× combined.

The broader implication is that the same pattern — structured input, explicit syntax grounding, engineer-in-the-loop review — is transferable to any EDA task that produces machine-readable output and requires tool-correct commands in return. The framework described here is therefore not specific to FPGA physical design: it provides a replicable blueprint for domain-grounded GenAI integration across semiconductor implementation flows. Three extensions are in active development: pipeline automation for Applications 2, 3, and 4 replacing manual extraction with Python scripts calling the Claude API; a GenParser design-health dashboard tracking metrics across successive runs; and incremental growth of the GenParser retrieval database as engineers annotate resolved errors from new design iterations.

REFERENCES

- [1] P. Lewis et al., "Retrieval-augmented generation for knowledge-intensive NLP tasks," *NeurIPS*, vol. 33, pp. 9459–9474, 2020.
- [2] A. Vaswani et al., "Attention is all you need," *NeurIPS*, vol. 30, 2017.
- [3] OpenAI, "GPT-4 technical report," arXiv:2303.08774, 2023.
- [4] Anthropic, Claude API Documentation, <https://docs.anthropic.com>, 2025.
- [5] S. Liu et al., "ChipNeMo: Domain-adapted LLMs for chip design," arXiv:2311.00176, 2023.
- [6] K. Liu, B. Pinckney, S. Godil, and H. Ren, "VerilogEval: Evaluating large language models for Verilog code generation," *ICCAD*, 2023.
- [7] J. Blocklove, S. Garg, R. Karri, and H. Pearce, "Chip-Chat: Challenges and opportunities in conversational hardware design," *MLCAD*, 2023.
- [8] Y. Lu, S. Liu, Q. Zhang, and Z. Shi, "RTLML: An open-source benchmark for design RTL generation with large language models," *ASP-DAC*, 2024.
- [9] L. Liu et al., "Application of AI to accelerate formal verification workflow," *WISH*, 2025.
- [10] H. He et al., "LLM4EDA: Emerging progress in large language models for electronic design automation," arXiv:2401.12224, 2024.
- [11] K. E. Murray et al., "VTR 8: High-performance CAD and customizable FPGA architecture modelling," *ACM TRET*, vol. 13, no. 2, 2020.
- [12] Cadence Design Systems, Innovus Implementation System User Guide, v21.1, 2023.
- [13] S. Thakur et al., "Benchmarking large language models for automated Verilog RTL code generation," *DATE*, 2023.

