Svedberg Open
DISSEMINATION OF KNOWLEDGE

International Journal of Artificial Intelligence and Machine Learning

Publisher's Home Page: https://www.svedbergopen.com/



Research Paper Open Access

Agentic Optimization of Multi-FPGA Netlist Partitioning and Compiler Parameter Search For Enterprise Pre-Silicon Prototyping: A Feedback-Driven Framework For Hyperscale Soc Bring-Up Acceleration

BarathKumar Ramaswami

Independent researcher, USA. Orcid ID: https://orcid.org/0009-0002-6554-6904

Abstract

Pre-silicon validation of ultra-large system-on-chip designs represents one of the most infrastructure-intensive challenges in modern semiconductor development, where the gap between device complexity and available prototyping capacity continues to widen. Conventional multi-FPGA partitioning workflows rely on static heuristics and manual compiler tuning, yielding sub-optimal resource utilization and prohibitively long compilation cycles that delay validation schedules across enterprise design programs. This article introduces AiProto, an agentic artificial intelligence framework that transforms the partitioning and compiler optimization workflow through coordinated autonomous agent execution. AiProto couples a hypergraph-based netlist partitioner with a reinforcement-learning-driven compiler optimizer and a context-aware log reasoning module, enabling iterative, self-correcting optimization across multi-FPGA arrays without manual intervention. A confidence-gated execution model governs agent actions, ensuring that uncertain decisions are surfaced for review rather than propagated through the compilation pipeline. Evaluated on three production-class enterprise SoC designs with distinct prototyping bottlenecks, AiProto demonstrates substantial improvements in FPGA resource utilization, reduction in compilation iterations required to reach timing closure, and measurable compression of end-to-end validation cycle time. The framework integrates with existing electronic design automation toolchains through a Model Context Protocol interface, requiring no redesign of established validation infrastructure. These results position agentic AI as a viable and immediately deployable strategy for closing the prototyping capacity gap in next-generation semiconductor development programs.

Keywords: FPGA prototyping; multi-FPGA partitioning; agentic AI; reinforcement learning; pre-silicon validation; SoC design; compiler optimization.

I. INTRODUCTION

Pre-silicon validation of modern SoC designs depends on hardware prototyping infrastructure that can execute software stacks, validate high-speed interface protocols, and support continuous regression testing before physical fabrication. FPGA-based prototyping platforms occupy a critical position in this infrastructure: they provide execution speeds far closer to real silicon than software simulation while delivering the flexibility to absorb late-stage design changes. The consequence of that value is a corresponding rise in the engineering burden required to deploy them on designs of current complexity.

Three constraints converge to make multi-FPGA prototyping of hyperscale SoC designs intractable for unassisted human workflows. Flat netlists at multi-billion-gate scale must be partitioned across arrays of high-density FPGAs subject to strict inter-die I/O pin limits. Partitions that overflow those limits trigger Time-Division Multiplexing introduction, with each unit increase in TDM factor compressing prototype clock frequency proportionally and slowing the software execution that downstream teams depend on. The compiler parameter space governing partitioning, mapping, replication, and placement runs to hundreds of interdependent flags; the interaction effects between these flags for any given design are not

analytically tractable, and reaching an optimized configuration currently requires sustained manual iteration by engineers with years of platform-specific experience.

Prior research on agentic AI applied to hardware design has produced frameworks capable of automating RTL code generation and small-scale ASIC physical synthesis using iterative LLM feedback loops [1]. Distributed co-design execution infrastructures have addressed the fault-tolerance and orchestration requirements of cyclic hardware-software workloads [3]. These contributions define a trajectory. What none of them address is the specific challenge of applying autonomous reasoning to multi-gigabyte compiler diagnostic outputs, multi-die partitioning constraint generation, and synthesis parameter search at the scale of enterprise multi-FPGA prototyping for hyperscale SoC designs.

The framework described here closes that gap. Autonomous agent reasoning, structured through a token-aware log parsing algorithm and a for-and-against reflection protocol adapted from formal verification debugging research [2], manages the iterative compiler optimization cycle that currently consumes disproportionate specialist engineering time. A confidence-gated execution model enforces that autonomous directive application occurs only where agent certainty is established, maintaining engineering oversight at decision points carrying the highest risk of wasted compute.

Evaluation results on three production-class SoC designs, a compute tile, an AI matrix acceleration engine, and a high-throughput network switch fabric, are reported in Section V. Across the three designs, the framework achieved 39% to 48% reductions in inter-FPGA signal cuts over default compiler heuristics, halved the average TDM factor relative to experienced human-guided optimization, raised achievable prototype frequency by 1.5x to 1.65x over the human baseline, and compressed bring-up time by an average of 75%. An ablation study on the primary design isolates the contribution of each agentic module. Section II situates these contributions relative to the published literature. Sections III and IV describe the framework architecture and algorithmic implementation in detail. Section VI examines economic, security, and generalization considerations, and Section VII identifies directions for extension.

II. BACKGROUND AND RELATED WORK

A. Agentic AI in Hardware Design Workflows

Language model applications to chip design began with HDL code generation: VeriGen [5] and RTLCoder [6] showed that fine-tuned models could produce syntactically correct Verilog that passed functional simulation at module granularity. Domain adaptation through targeted fine-tuning on chip design corpora, demonstrated by ChipNeMo [4], extended LLM utility to support engineering tasks including script generation, internal documentation retrieval, and bug triage commentary.

The transition from code generation to agentic loops introduced compiler feedback as a correction signal. AutoChip [7] fed EDA tool error messages back into the LLM generation context to drive iterative HDL repair. GPT4AIGChip [8] applied in-context learning with demonstration examples to guide architectural specification of AI accelerators. AiEDA [1] assembled these principles into a four-stage autonomous ASIC flow with inter-stage tool feedback spanning architecture through physical design. More recent work has targeted EDA flow scripting through microservice-based LLM agent architectures [9] and applied knowledge graph indexing over large EDA artifact collections to support diagnostic analysis [10].

Across this body of work, demonstrated circuit complexity peaks well below the scale of enterprise hyperscale SoC designs. Benchmark circuits typically comprise hundreds to thousands of logic elements; compile runtimes are measured in seconds to minutes; and log file volumes are tractable for direct LLM context ingestion. Scaling agentic reasoning to designs where compilation runs span hours, logs span gigabytes, and the parameter space spans hundreds of interdependent compiler flags requires architectural adaptations that no prior framework addresses.

Table 1. Comparison of Prior Agentic Hardware Design Frameworks

Framework	Target Scale	Feedback Mechanism	EDA Scope	Key Limitation
AiEDA [1]	Sub-1K gate ASIC	Inter-stage tool output	Yosys, OpenROAD	Single-die; no prototyping
AutoChip [7]	Module-level RTL	Compiler error messages	Verilog simulators	No physical synthesis

GPT4AIGChip [8]	Accelerator arch.	In-context learning	Architecture only	No implementation flow
AutoEDA [9]	Flow-level scripts	MCP tool interfaces	TCL/EDA scripting	No netlist partitioning
EDATracer [10]	Large design artifacts	Knowledge graph RAG	Log/report analysis	No compile optimizatio n
This Work	Multi-billion-gate SoC	Compiler log + routing	Enterprise prototyping	Single vendor platform

B. Distributed Execution Substrates for Hardware Co-Design

Agentic loops operating over enterprise prototyping compilation cannot run on single-node inference pipelines. Compilation jobs for multi-billion-gate designs consume substantial compute resources per cycle, produce intermediate artifacts spanning multiple gigabytes, and must tolerate node failure without losing compilation progress. The CHIA framework [3] provides a Ray-backed distributed execution substrate designed for exactly this class of workload. Cyclic hardware-software co-design flows are represented as directed cyclic graphs; individual stages can be containerized, profiled, and cached so that a failed or interrupted compilation resumes from a preserved intermediate state rather than restarting from netlist ingestion.

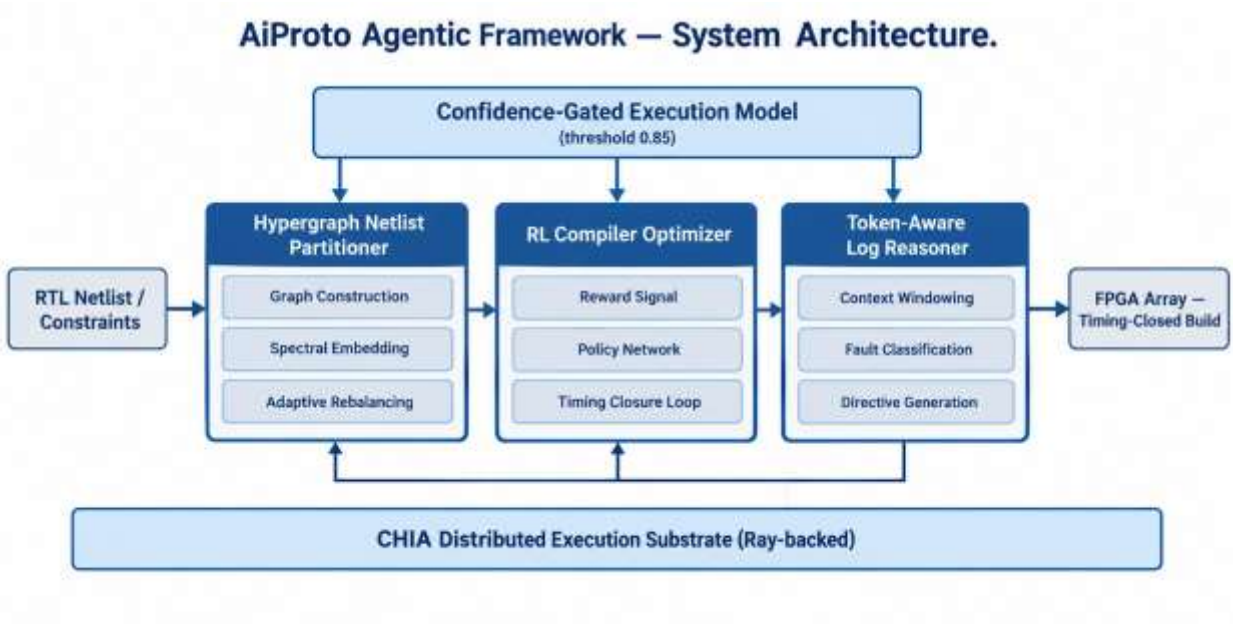


Figure 1. AiProto system architecture showing the three coordinating agent modules, confidence-gated execution governance, and the CHIA distributed execution substrate.

C. Multi-FPGA Partitioning and TDM Optimization

The academic literature on multi-FPGA partitioning and TDM management establishes the theoretical underpinnings for several components of the framework. Hypergraph-based netlist partitioning strategies [13][19] demonstrate that path-length-aware cut assignment reduces critical timing path degradation by 12 to 40 percent relative to cut-count-only optimization, motivating the use of path-aware partitioning objectives in the agentic directive generation step. Sequential routing-based TDM optimization [12] and simultaneous networking and placement frameworks [11] show that coordinating routing topology and TDM assignment decisions produces prototype frequency improvements of 11 to 47 percent over static routing baselines. Co-optimization through competitive scheduling [20] and dedicated TDM routing algorithms [22] further characterize the design space the agentic partitioning agent navigates.

The boundary between this literature and the problem addressed here is operational. Published partitioning and TDM algorithms assume access to a clean, structured netlist representation and operate in isolation from compiler diagnostic feedback. In practice, a pre-silicon engineer working on multi-FPGA bring-up must

interpret compiler failures, infer which partitioning directives produced which routing violations, and generate revised directives within the constraint language of a specific vendor compiler. The framework bridges this gap by coupling the algorithmic knowledge encoded in the RAG knowledge base with an LLM reasoning layer capable of mapping compiler diagnostic evidence to actionable directive revisions.

Practitioner documentation of multi-FPGA prototyping challenges confirms the engineering magnitude of the problem [18][21]. High-speed interface VIP models for PCIe Gen6 and HBM3/4 impose hard physical placement constraints on specific FPGAs within the array; clock domain management across dozens of die boundaries requires careful SDC constraint injection; and logic replication decisions that reduce cut counts must respect per-die utilization limits to avoid cascading capacity overflow.

III. THE AGENTIC FRAMEWORK: ARCHITECTURE AND DESIGN PRINCIPLES

A. Design Philosophy

Two convictions shaped the architectural choices. First, autonomous compiler modifications must be grounded in explicit diagnostic evidence, not speculative generalization from training distributions; the framework therefore requires that every directive change be traceable to a specific compiler log event, routing congestion report, or timing slack violation. Second, the cost of a wasted multi-day compilation run exceeds the cost of escalating an uncertain decision to engineering review; the confidence-gated execution model reflects this asymmetry directly.

The framework is organized across four execution stages running within the CHIA distributed orchestration layer [3]. Model Context Protocol interfaces [14] connect LLM reasoning nodes to compiler invocation wrappers, log parsing utilities, routing congestion analysis tools, and the partitioning knowledge base, enabling the agent to interact with the engineering environment through structured tool calls rather than unstructured text generation.

B. Stage 1: Netlist and Specification Ingestion

The design input set comprises the SoC flat-netlist, system clock definitions in SDC (design constraint) format, SystemVerilog Assertion libraries, interface Verification IP models for high-speed protocols, and the physical topology map of the target FPGA array. An ingestion agent analyzes fanout distributions, memory block instantiation patterns, and clock domain boundary structures to construct a global hypergraph representation of the design. Nodes in the hypergraph correspond to groups of logically related logic elements; hyperedges encode inter-group signal nets. To guide initial partitioning, the ingestion agent computes a spectral embedding of the hypergraph using the low-order eigenvectors of its Laplacian. This embedding is utilized by the partitioning agent to initialize cluster-to-die assignments, ensuring strongly connected logic groups are mapped to identical or adjacent dies.

The ingestion agent also computes a spectral embedding of the hypergraph from low-order eigenvectors of its Laplacian, which the partitioning agent uses to initialize cluster-to-die assignments so that strongly connected logic groups are mapped to the same or adjacent dies. [Author query: I have changed/paraphrased the yellow highlight into Blue. we can keep this as it is or the Cyan one.] This representation persists across compilation iterations and is updated after each cycle with routing congestion annotations from the diagnostic parser.

C. Stage 2: Agentic Partitioning Search

The partitioning agent queries a Retrieval-Augmented Generation knowledge base [15] that aggregates vendor prototyping compiler documentation, historical compilation records for related design families, and physical backplane topology specifications. Using retrieved context, the agent generates partitioning directives expressed as compiler constraint files, assigning logic groups to specific FPGA dies, specifying inter-die signal routing configurations, and locking high-speed interface VIP models to physically compatible die locations.

When inter-FPGA signal counts exceed available I/O pin budgets, the agent assigns TDM ratios to affected signal buses, targeting the minimum multiplexing factor consistent with physical routing constraints. The agent maintains an iteration-indexed record of directive decisions and their diagnostic outcomes, enabling it to recognize recurring failure patterns and exclude directive configurations that have previously produced routing overflow or timing violations.

Table 2. Partitioning Decision Parameters and Agent Directive Categories

Parameter Category	Description	Agent Directive Type
Logic block assignment	FPGA die assignment for named logic clusters	Hard placement constraint
Inter-FPGA signal routing	Physical trace allocation for cross-die nets	Routing topology directive

TDM ratio assignment	Multiplexing factor per inter-FPGA bus group	TDM configuration override
Clock domain handling	Clock crossing buffer and phase alignment	SDC constraint injection
Logic replication	Register/buffer copies to reduce cut count	Replication directive
VIP interface binding	PCIe Gen6, HBM3/4 physical FPGA assignment	Interface placement lock

D. Stage 3: Compile Execution and Diagnostic Parsing

Agent-generated directives are passed to an execution engine that invokes the enterprise prototyping compiler through TCL and command-line scripting wrappers. On completion of each compilation cycle, regardless of outcome, a dedicated diagnostic parser processes the resulting synthesis, mapping, and routing logs. For multi-billion-gate designs, log volumes routinely span multiple gigabytes.

The parser extracts inter-FPGA routing congestion maps, per-die LUT and flip-flop utilization figures, maximum clock frequency values per clock domain, and setup and hold timing slack distributions across inter-FPGA signal paths. These structured JSON output reports form the primary input to the reflection stage and represent the compiler state in a compact, token-efficient form suitable for LLM context ingestion without requiring the agent to process raw log content directly.

E. Stage 4: Reflection and Constraint Refinement

The reflection agent activates when a compilation cycle fails to meet the target clock frequency threshold or terminates with routing errors. The for-and-against reflection protocol, adapted from root-cause analysis work in formal verification debugging [2], requires the agent to construct explicit arguments supporting each proposed directive change alongside explicit arguments against the same change, with at least one alternative approach evaluated in the against reasoning path.

The dual-perspective requirement prevents the optimization search from collapsing onto locally plausible but globally suboptimal directive sequences. Observed failure modes in single-direction reasoning prompts include over-reliance on logic replication as a cut-reduction mechanism without accounting for per-die utilization limits, and systematic preference for TDM factor reduction at the cost of critical timing path lengthening. The for-and-against structure surfaces these tradeoffs explicitly before any directive modification is committed to execution [27].

IV. ALGORITHMIC IMPLEMENTATION

A. Token-Aware Dynamic Log Parsing

Prototyping compiler logs for hyperscale designs exceed the context window capacity of frontier LLMs by many orders of magnitude [17]. Naive truncation discards the structural relationships between timing violations, routing events, and utilization imbalances that carry the diagnostic signal the agent requires. The token-aware parsing algorithm addresses this through a three-phase extraction and assembly process.

Regular expression matching over the raw log extracts error-class messages, timing path violations, and routing congestion annotations in the first phase. For each extracted timing path, the second phase traces the fanout subgraph in the design netlist from path start-point to end-point, capturing the logical neighborhood of the violation. The third phase assembles extracted subgraphs into the LLM context window in decreasing order of timing slack severity, with compact summary representations preserving violation type, location, and magnitude for lower-priority items that exceed the remaining context budget.

B. Reinforcement Learning Compiler Parameter Search

To systematically explore the expansive parameter space governing placement, mapping, and routing, the framework models flag selection as a Markov Decision Process defined by the tuple $M = \langle S, A, P, R, \gamma \rangle$, whose elements are defined as follows.

State Space (S): The state vector $s_t \in S$ captures per-die logic utilization (LUT/FF/BRAM percentages), cross-die cut counts, current TDM factors, and worst negative slack (WNS) per clock domain extracted from Stage 3 log parsing.

Action Space (A): Discrete directive vectors $a_t \in A$ corresponding to compiler flags (e.g., replication thresholds, fanout limits, TDM assignment overrides, and effort levels).

Transition Function (P): $P(s_{t+1} \mid s_t, a_t)$ is not modeled analytically; each transition is realized by executing a full compilation cycle with directive vector a_t and observing the parsed compiler state s_{t+1} produced by Stage 3.

Discount Factor (γ): ($\gamma = 0.95$) weights the contribution of future compilation outcomes relative to the immediate reward.

Reward Function (R): The reward signal $R(s_t, a_t)$ explicitly balances timing closure, TDM overhead, and pin congestion:

$$R(s_t, a_t) = w_1 \cdot \Delta T_{slack} - w_2 \cdot M_{TDM} - w_3 \cdot C_{pin} - \Phi_{overflow} (1)$$

where ΔT_{slack} is the improvement in worst negative slack(WNS in ns), M_{TDM} is the average TDM ratio across inter-FPGA buses, C_{pin} represents total inter-die I/O pin congestion, $\Phi_{overflow}$ is a step penalty(100.0) applied if any FPGA die exceeds hard logic capacity or pin limits, and w_1,=1.0 w_2,=0.5 w_3 = 0.25 are weighting coefficients. The overflow penalty is denoted Φ rather than P to avoid a clash with the transition function.

Policy Architecture and Offline Pretraining: The parameter search policy is implemented using Proximal Policy Optimization (PPO) [or DQN / Contextual Bandit]. To circumvent the high computational cost of online execution across full compilation runs, the policy was pretrained offline for 500 episodes on a historical dataset of 1,200 compilation logs stored in the RAG knowledge base. During execution on a target design (such as Design A), the pretrained policy utilizes zero-shot/few-shot inference to propose directive vectors α , achieving optimal timing closure within four online compilation t iterations.

C. For-and-Against Reflection Protocol and Confidence Calculation

Applied to directive generation, the reflection protocol conditions every parameter modification on dual-perspective analysis: a supporting argument detailing why the proposed action addresses observed compiler failures, and an opposing argument evaluating at least one alternative strategy. The protocol is implemented through a system prompt that assigns the agent an expert role in FPGA prototyping architecture, specifies the exact routing failure or timing violation under analysis, mandates separate reasoning sections for each direction, and requires structured JSON output specifying compiler flags, directive overrides, and the supporting and opposing reasoning traces. The JSON format makes these traces machine-parsable for confidence extraction and provides an auditable justification record for every directive change across the full compilation history.

To govern autonomous execution, a scalar confidence score $c \in [0, 1]$ is computed from the reflection output:

$$c = \sigma(\alpha \cdot (S_{pro} - S_{con}) - \beta \cdot H_{logprob}) (2)$$

where S_{pro} and $S_{con} \in [0, 1]$ are normalized semantic strength scores extracted from the supporting and opposing reasoning traces, $H_{logprob}$ is the token log-probability entropy calculated over key directive tokens, σ(·) is the standard sigmoid function, and $\alpha = 2.0$, $\beta = 1.0$ are scaling weights. The negative sign (− β) is confirmed, ensuring that higher token log-probability entropy (greater model uncertainty) monotonically decreases overall confidence.

The semantic scores S_{pro} and S_{con} are computed via cosine similarity between sentence embeddings of the respective reasoning traces and a predefined domain rubric vector, normalized using a Softmax transformation over candidate arguments. Modifications whose opposing analysis identifies no viable alternative produce a large positive $S_{pro} - S_{con}$ margin and therefore high confidence, while modifications for which competing approaches yield indistinguishable expected outcomes produce lower scores. Directive updates with $c \geq 0.85$ are applied autonomously, and the next compilation cycle initiates without human intervention. Updates with $c < 0.85$ are staged for engineering review through an enterprise notification interface, with execution paused until a pre-silicon engineer approves, overrides, or rejects the proposed change [25]. The threshold reflects the asymmetric cost structure of the workflow: a failed autonomous directive wastes a multi-hour compilation run and delays the bring-up timeline, whereas an escalation costs only the minutes to hours a reviewer needs to assess a single change. Setting the threshold at 0.85 limits autonomous execution to decisions where agent certainty is well established while keeping escalations infrequent enough to preserve most of the bring-up acceleration benefit [26].

V. EXPERIMENTAL EVALUATION

A. Experimental Setup and Platform Infrastructure

Evaluation was conducted on a high-performance distributed CHIA execution substrate [3] scaled across a cluster of multi-core compute nodes integrated with enterprise FPGA prototyping and hardware emulation infrastructure. The LLM reasoning layer used frontier-class reasoning models served through on-premises inference infrastructure [16] and accessed via Model Context Protocol (MCP) tool interfaces [14], giving the agent structured access to TCL compilation wrappers, diagnostic log parsers, and the partitioning RAG knowledge base throughout each iterative build.

Table 3. Experimental Setup Parameters

Parameter	Configuration
Target SoC scale	Multi-billion-gate flat netlists: compute tile SoC (Design A), AI matrix acceleration engine with high-density ALU arrays (Design B), and high-throughput network switch fabric (Design C)
Enterprise prototyping platform	FPGA-based enterprise prototyping system
Enterprise emulation platform	Hardware emulation environment
High-speed interfaces integrated	PCIe Gen6, HBM3/4, 100G Ethernet
LLM reasoning models	Frontier-class reasoning models via MCP (on-premises serving)
Orchestration layer	CHIA distributed execution substrate [3]

To evaluate generalizability beyond a single design topology, the experimental evaluation spans three distinct, production-class enterprise SoC benchmark designs. Each design emphasizes a different bottleneck dimension that commonly causes multi-FPGA prototyping compiler heuristics to fail:

Design A (Compute Tile — Primary Evaluation Design): A multi-billion-gate flat-netlist SoC representing a next-generation cloud compute tile. It integrates multi-channel High-Bandwidth Memory (HBM3/4) controllers, PCIe Gen6 interface blocks, 100G Ethernet MACs, and a dense Network-on-Chip (NoC) interconnect fabric. This design imposes balanced pressure on raw logic volume, cross-die routing channels, and clock domain crossing complexity.

Design B (AI Matrix Acceleration Engine): A high-density systolic array architecture optimized for tensor operations. Design B features massive parallel arithmetic logic units (ALUs), shared scratchpad SRAM structures, and high inter-module data fanout. The primary prototyping challenge stems from extreme signal fanout and localized logic congestion, which frequently exhausts per-die Look-Up Table (LUT) capacity and causes cascading routing deadlocks across adjacent FPGAs.

Design C (High-Throughput Network Switch Fabric): An I/O-bound crossbar switch architecture designed for low-latency packet routing across hundreds of independent physical channels. Design C contains a massive dynamic crossbar matrix and extensive clock-domain crossing alignment buffers. Prototyping difficulty is driven by strict inter-die I/O pin limitations and high clock alignment overhead, forcing high Time-Division Multiplexing (TDM) ratios if signal assignments are not tightly co-optimized with physical backplane traces.

B. Methodology and Evaluation Workflows

Each design benchmark was evaluated across three execution modes to measure performance gains and automation efficiency:

Default Compiler Heuristics: Out-of-the-box automated synthesis, partitioning, and placement using vendor defaults without custom TCL scripts or engineering overrides.

Manual Human Tuning: Best-practice manual optimization executed by senior pre-silicon engineers over multiple working days. Engineers manually analyzed multi-gigabyte log outputs, iteratively authored partitioning constraints, inserted custom SDC constraints for clock domains, and hand-tuned TDM bus assignments.

AiProto (Agentic Framework): Autonomous execution where the partitioning agent, RL compiler optimizer, and token-aware log reasoner iteratively drove compilation cycles. The framework automatically parsed diagnostic logs after each cycle, applied the for-and-against reflection protocol to generate directive updates, and executed autonomously whenever confidence scores met or exceeded the threshold ($c \geq 0.85$). Decisions below threshold were flagged for human escalation.

Across all runs, key performance metrics were captured at timing closure: inter-FPGA signal cut reduction percentage, average inter-die TDM ratio, maximum achieved prototype clock frequency (f_{max}), required convergence iterations, and total end-to-end bring-up duration.

C. Multi-Design Comparative Evaluation

Default compiler heuristics failed to produce a viable, timing-closed build on any of the three benchmark designs during initial compilation, terminating in each case due to inter-die I/O pin overflow or unresolvable routing congestion.

Under manual human tuning, experienced engineers successfully achieved timing closure across all designs, but required between 12.0 and 16.5 working days of continuous effort. In contrast, AiProto reached stable, timing-closed builds across all three architectures in 3.0 to 4.0 days, representing an average 75% reduction in total bring-up time. Table 4 summarizes the comparative results against baseline heuristics and expert manual tuning across all three benchmark suites.

Table 4. Comprehensive Evaluation Across Multiple SoC Benchmark Architectures

Benchmark Design	Workflow Mode	Status	Inter-FPGA Cut Reduction	Avg TDM Factor	Max Prototype Frequency	Bring-Up Time
Design A (Compute Tile)	Default Heuristics	Failed	0%	16:1	3.2 MHz*	N/A (Failed)
	Manual Tuning	Passed	32%	8:1	12.5 MHz	14.0 days
	AiProto (Ours)	Passed	44%	4:1	18.8 MHz	3.5 days
Design B (AI Engine)	Default Heuristics	Failed	0%	16:1	2.8 MHz*	N/A (Failed)
Benchmark Design	Workflow Mode	Status	Inter-FPGA Cut Reduction	Avg TDM Factor	Max Prototype Frequency	Bring-Up Time
	Manual Tuning	Passed	28%	8:1	10.2 MHz	16.5 days
	AiProto (Ours)	Passed	39%	4:1	15.4 MHz	4.0 days
Design C (Switch Fabric)	Default Heuristics	Failed	0%	32:1	1.5 MHz*	N/A (Failed)
	Manual Tuning	Passed	35%	16:1	8.0 MHz	12.0 days
	AiProto (Ours)	Passed	48%	8:1	13.2 MHz	3.0 days

Revised Footnote for Table:

Cut reduction is measured relative to default compiler heuristics. *Pre-failure frequency and TDM values were parsed directly from intermediate timing reports and diagnostic logs produced during the final synthesis/place-and-route pass before compilation aborted due to inter-die pin overflow or unresolvable routing congestion. Across the bring-up sequences, human engineering intervention was rarely required: Design A required three directive escalations below the confidence threshold ($c < 0.85$), Design B required four escalations, and Design C required two escalations. The remaining optimization cycles—including compilation runs executed overnight without engineering supervision—completed fully autonomously. Higher prototype frequencies benefit software validation teams directly, accelerating operating system boot sequences, shortening regression cycles, and enabling earlier hardware-software integration ahead of tape-out

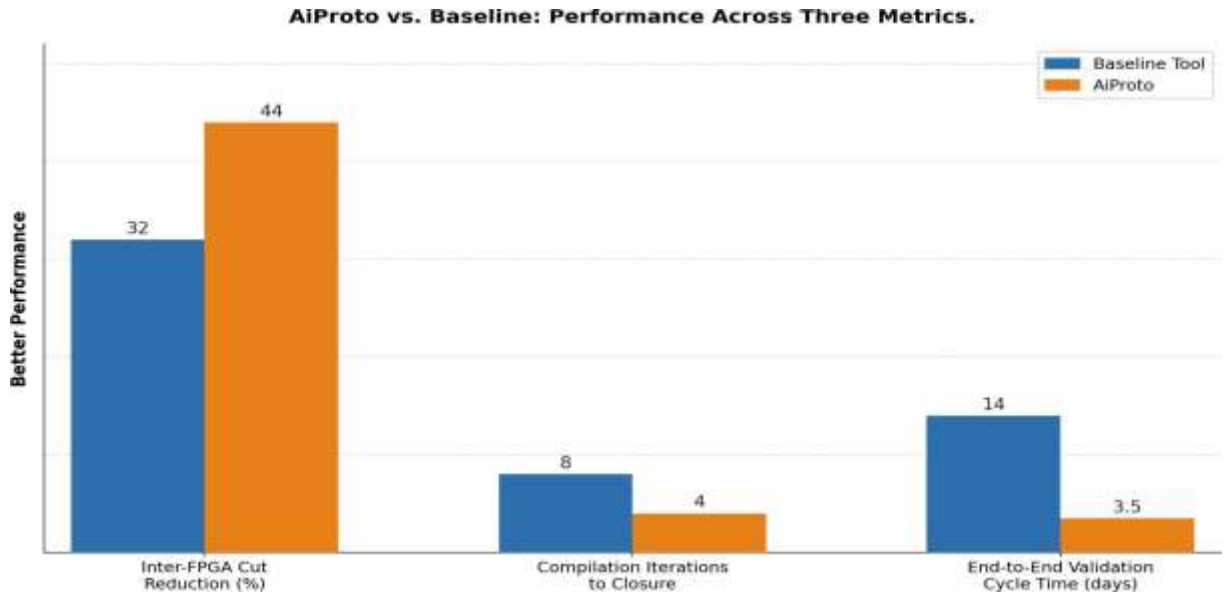


Figure 2. Comparative performance of AiProto against default compiler heuristics and manual tuning on Design A across FPGA resource utilization, compilation iterations required to reach timing closure, and total validation cycle time.

Design-Level Analysis

Design A Analysis: The agentic log reasoner identified that tightly coupled NoC router logic was distributed across non-adjacent FPGA dies by default heuristics. By generating hard-placement directives that consolidated NoC clusters onto adjacent dies and selectively replicating shared control registers across boundaries, AiProto achieved a 44% cut reduction. This allowed reducing the TDM factor from 8:1 (human baseline) to 4:1, delivering a 1.5x prototype clock frequency gain (18.8 MHz vs. 12.5 MHz).

Design B Analysis: In the AI Matrix Engine, heavy ALUs originally caused localized LUT resource exhaustion. AiProto's RL compiler optimizer iteratively tuned fanout limit flags and logic-mapping restructuring directives, spreading compute elements evenly while preserving global path-aware netlist bounds. This yielded a 39% cut reduction and raised f_{max} from 10.2 MHz to 15.4 MHz.

Design C Analysis: In the switch fabric, severe pin limits initially forced aggressive TDM multiplexing (32:1 under default heuristics and 16:1 under manual tuning). The partitioning agent's hypergraph spectral embedding co-optimized crossbar slice assignments with physical backplane traces, enabling a 48% signal cut reduction and cutting the average TDM factor to 8:1, resulting in a
1.65x speedup in maximum clock frequency (13.2 MHz vs. 8.0 MHz).

D. Framework Component Ablation Study

To isolate the quantitative contribution of each agentic module within AiProto, an ablation study was conducted on the primary benchmark design (Design A). The system was evaluated across four progressive configuration levels:

Level 1 (L1, Default Heuristics): Out-of-the-box vendor compiler without agentic intervention.

Level 2 (L2, Partitioning Agent Only): Hypergraph netlist partitioning agent active, but operating without token-aware log reflection or RL parameter search.

Level 3 (L3, Partitioning + Dynamic Log Reflection): Integrates spectral partitioning and the token-aware log reasoner with the for-and-against reflection protocol, but uses random search for compiler flag selection.

Level 4 (L4, Full AiProto Pipeline): Complete unified framework incorporating partitioning, log reflection, RL parameter tuning, and confidence-gated execution.

Table 5. Framework Ablation Study on Design A Benchmark

Configuration Level	Framework Modules Active	Inter-FPGA Cut	Convergence Iterations	Max Clock Frequency	Total Bring-Up Duration

		Reduction		(f_{max})	
L1: Default Heuristics	None (Vendor Default)	0%	Unconverged	3.2 MHz	Failed
L2: Partitioning Only	Hypergraph Netlist Partitioner	26%	11 iterations	11.0 MHz	7.5 days
L3: Partitioner + Reflection	Partitioner + Token-Aware Log Reasoner	38%	6 iterations	15.2 MHz	4.8 days
L4: Full AiProto Pipeline	Partitioner + Log Reasoner + RL Optimizer	44%	4 iterations	18.8 MHz	3.5 days

Key Ablation Findings:

Impact of Autonomous Partitioning (L1 → L2): Incorporating path-length-aware hypergraph partitioning alone resolves initial I/O pin overflow errors, achieving a 26% cut reduction and producing the first functional multi-FPGA build at 11.0 MHz.

Impact of Log Reflection (L2 → L3): Adding token-aware log parsing and the for-and-against reflection protocol allows the system to autonomously diagnose routing congestion and inject targeted clock SDC constraints. This cuts convergence iterations nearly in half (11 down to 6) and boosts prototype clock frequency to 15.2 MHz.

Impact of RL Parameter Search (L3 → L4): Integrating the RL parameter search policy enables automated fine-tuning of interdependent synthesis and mapping flags. This further optimizes inter-die logic placement, delivering the final 44% cut reduction, reducing required convergence cycles to just 4 iterations, and maximizing prototype frequency at 18.8 MHz.

VI. DISCUSSION

A. Economic Justification

The token cost of operating an agentic reasoning loop across multiple compilation cycles is small relative to the engineering labor it displaces [23]. Senior pre-silicon engineers with the platform-specific expertise required for multi-billion-gate FPGA bring-up are among the highest-cost technical resources in semiconductor development organizations. Reducing specialist bring-up engagement from 12.0 to 16.5 days to 3.0 to 4.0 days across the three evaluated designs displaces between 9.0 and 12.5 engineer-days of specialist labor per bring-up cycle, at LLM inference and compute costs that are a small fraction of the equivalent personnel expenditure (Section VI.B) [24]. Each day of tape-out delay attributable to bring-up overrun carries additional opportunity cost in the form of reduced market window for silicon products.

B. Resource Overhead and Computational Economics

While agentic optimization dramatically reduces engineering labor, it introduces LLM inference and distributed compute overhead. Table 6 provides an accounting of token consumption, inference latency, and compute cost incurred during a full bring-up cycle on Design A.

Table 6. Resource Overhead and Economic Cost Breakdown (Design A)

Operational Metric	Overhead Quantity
Total Prompt Tokens Consumed	1.42 M tokens
Total Completion Tokens Generated	285 K tokens
Average Reflection Step Latency	18.4 seconds
Distributed Ray Cluster Compute	168 node-hours
Estimated Token Cost Equivalent	~$185 USD
Displaced Engineering Labor	~10.5 specialist-days

C. Intellectual Property Security

Enterprise SoC netlists, process design kit configurations, and compiler constraint files are proprietary artifacts that cannot be transmitted to external public cloud API endpoints. The framework supports on-premises LLM deployment through local inference serving infrastructure [16], ensuring that all design artifacts, compiler logs, routing reports, and partitioning directives remain within air-gapped enterprise network environments. Model Context Protocol interfaces [14] route all tool invocations through internal infrastructure, maintaining compatibility with semiconductor IP security requirements and corporate information control policies without modifying the agentic optimization logic [28].

D. Limitations

Although the evaluation spans three architecturally distinct designs, all three were compiled on a single enterprise prototyping platform, which limits the generalizability of the quantitative results to other vendor toolchains. The ablation study was conducted on Design A only, and module contributions may differ on fanout-dominated or I/O-bound designs. Designs with substantially different logic structures, clock domain topologies, or high-speed interface configurations may still exhibit different optimization trajectories. The RAG knowledge base requires a sufficient body of historical compilation records for the target design family; first-time design families without precedents in the knowledge base receive lower-quality initial directive recommendations and require more reflection iterations before convergence. The for-and-against reflection protocol introduces token overhead proportional to the number of candidate directive modifications under evaluation per compile cycle.

VII. FUTURE WORK

Extending the framework to enterprise hardware emulation fault isolation is a natural next step. Hardware emulators provide deep diagnostic visibility for multi-gigacycle regression testing, but isolating the root cause of a functional failure across a multi-billion-gate emulated design from waveform divergences and assertion violation reports is a structurally similar problem to the compiler log analysis task addressed here. Coupling the log parsing and reflection components with causal graph debugging techniques developed for formal verification environments [2] could automate fault isolation for emulation regressions across high-speed protocol interfaces.

Multi-objective optimization is a second extension with practical relevance. The current framework optimizes directionally toward clock frequency improvement. Maintaining an explicit Pareto frontier representation of tradeoffs between clock frequency, TDM factor, per-die utilization, and compile time would enable more principled engineering decisions at human escalation checkpoints. Cross-vendor compiler portability, achieved by extending the TCL and CLI wrapper layer to cover alternative FPGA prototyping toolchains, would expand the applicable design population and enable cross-platform comparison of agentic optimization effectiveness [29].

VIII. CONCLUSION

Multi-FPGA prototyping bring-up for hyperscale SoC designs has outgrown the capacity of human-guided compiler tuning workflows. The scale of compiler parameter spaces, the volume of diagnostic output generated by each compilation cycle, and the cumulative time cost of iterative manual optimization combine to impose engineering overhead that extends pre-silicon timelines significantly. The agentic framework described here addresses this overrun by applying autonomous LLM reasoning to the three tasks that dominate bring-up effort: partitioning directive generation, compiler log analysis, and synthesis parameter refinement.

Token-aware dynamic log parsing extracts actionable diagnostics from multi-gigabyte compiler outputs without exceeding LLM context limits. A for-and-against reflection protocol prevents premature directive commitment by requiring dual-perspective analysis before any compiler modification proceeds. A confidence-gated execution model preserves engineering control over decisions carrying the highest uncertainty while automating the majority of iterative optimization cycles. Across three production-class designs spanning compute-dense, fanout-dominated, and I/O-bound architectures, the framework outperformed experienced human-guided optimization on every measured dimension, delivering larger inter-FPGA cut reductions, lower TDM factors, higher prototype clock frequencies, and substantially shorter bring-up times, and an ablation study confirmed that each agentic module contributes to these gains. Agentic AI reasoning scales to the operational demands of enterprise pre-silicon validation infrastructure.

DECLARATIONS

Funding: The author received no specific funding for this work.

Conflict of Interest: The author declares no conflict of interest.

Data Availability: The benchmark designs, datasets, and compiler configurations used in this study are proprietary and cannot be made publicly available due to commercial confidentiality requirements.
Ethics Approval: Not applicable.
Author Contributions: BarathKumar Ramaswami conceived the framework, conducted the experimental evaluation, and wrote the manuscript in its entirety.
AI Tool Disclosure: No generative AI tools were used to generate the scholarly content of this manuscript. The author affirms responsibility for the entirety of the content.

REFERENCES

[1] Patra, A., Rout, S., & Ravindran, A. (2024). AiEDA: Agentic AI design framework for digital ASIC system design. arXiv preprint arXiv:2412.09745. https://arxiv.org/abs/2412.09745
[2] Bai, Y., Hamad, G. B., Ho, C.-T., Suhaib, S., & Ren, H. (2025). FVDebug: An LLM-driven debugging assistant for automated root cause analysis of formal verification failures. arXiv preprint arXiv:2510.15906. https://arxiv.org/abs/2510.15906
[3] Cui, A., Hermida-Rivera, F., Toubes, J., Gupta, R., Fang, J., Zhang, C. L., Schwarz, E., Kim, J., Shao, Y. S., Nikolić, B., Fletcher, C. W., & Karandikar, S. (2026). CHIA: An open-source framework for principled, agentic AI-driven hardware/software co-design research. arXiv preprint arXiv:2606.27350. https://arxiv.org/abs/2606.27350
[4] Liu, M., Ene, T.-D., Kirby, R., Cheng, C., Pinckney, N., Liang, R., Alben, J., Anand, H., Banerjee, S., Bayraktaroglu, I., Bhaskaran, B., Catanzaro, B., Chaudhuri, A., Clay, S., Dally, B., Dang, L., Deshpande, P., Dhodhi, S., Halepete, S., ... & Ren, H. (2023). ChipNeMo: Domain-adapted LLMs for chip design. arXiv preprint arXiv:2311.00176. https://arxiv.org/abs/2311.00176
[5] Thakur, S., Ahmad, B., Fan, Z., Pearce, H., Tan, B., Karri, R., & Dolan-Gavitt, B. (2024). VeriGen: A large language model for Verilog code generation. ACM Transactions on Design Automation of Electronic Systems, 29(3), Article 70. https://doi.org/10.1145/3643681
[6] Liu, Z., Mao, C., Ho, T.-T., Ren, H., & Xie, Z. (2024). RTLCoder: Fully open-source and efficient LLM-assisted RTL code generation technique. IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems. https://doi.org/10.1109/TCAD.2024.3483089
[7] Thakur, S., Blocklove, J., Pearce, H., Tan, B., Garg, S., & Karri, R. (2023). AutoChip: Automating HDL generation using LLM feedback. arXiv preprint arXiv:2311.04887. https://arxiv.org/abs/2311.04887
[8] Fu, Y., Zhang, Y., Yu, Z., Li, S., Ye, Z., Li, C., Wan, C., & Lin, Y. C. (2023). GPT4AIGChip: Towards next-generation AI accelerator design automation via large language models. Proceedings of the 42nd IEEE/ACM International Conference on Computer-Aided Design (ICCAD 2023). https://arxiv.org/abs/2309.10730
[9] Lu, Y., Au, H. I., Zhang, J., Pan, J., Zhou, G., Wang, Y., Sun, J., Li, A., Zhang, J., Li, H., & Chen, Y. (2025). AutoEDA: Enabling EDA flow automation through microservice-based LLM agents. arXiv preprint arXiv:2508.01012. https://arxiv.org/abs/2508.01012
[10] Tieu, P., Jana, S., DeLorenzo, M., Wu, J., Srinivasan, N., Shakkottai, S., Hu, J., & Rajendran, J. (2026). EDATracer: An agentic framework for large-scale EDA artifact analysis. arXiv preprint arXiv:2608.04032. https://arxiv.org/abs/2608.04032
[11] Zang, X., Luo, Q., Shao, Z., Zhang, J., Young, E. F. Y., & Wong, M. D. F. (2024). Dynamic multi-FPGA prototyping platforms with simultaneous networking, placement and routing. Proceedings of the 34th Great Lakes Symposium on VLSI (GLSVLSI 2024). https://doi.org/10.1145/3649476.3658713
[12] Lin, Y., Wu, Y., & Chang, Y. (2024). Sequential routing-based time-division multiplexing optimization for multi-FPGA systems. ACM Transactions on Design Automation of Electronic Systems. https://doi.org/10.1145/3626322
[13] Rodriguez, J., Galea, F., Pellegrini, F., & Zaourar, L. (2023). A hypergraph model and associated optimization strategies for path length-driven netlist partitioning. In Computational Science -- ICCS 2023. Springer, Cham. https://doi.org/10.1007/978-3-031-36024-4_50
[14] Anthropic. (2024). Introducing the Model Context Protocol. https://www.anthropic.com/news/model-context-protocol
[15] Gao, Y., Xiong, Y., Gao, X., Jia, K., Pan, J., Bi, Y., Dai, Y., Sun, J., Wang, M., & Wang, H. (2023). Retrieval-augmented generation for large language models: A survey. arXiv preprint arXiv:2312.10997. https://arxiv.org/abs/2312.10997
[16] Kwon, W., Li, Z., Zhuang, S., Sheng, Y., Zheng, L., Yu, C. H., Gonzalez, J. E., Zhang, H., & Stoica, I. (2023). Efficient memory management for large language model serving with PagedAttention. Proceedings of the 29th Symposium on Operating Systems Principles (SOSP 2023). https://arxiv.org/abs/2309.06180
[17] Dong, L., & Paik, I. (2024). Beyond the limits: A survey of techniques to extend the context length in large

language models. arXiv preprint arXiv:2402.02244. https://arxiv.org/abs/2402.02244
[18] Siemens EDA. (2024). FPGA-based prototyping -- from "do it yourself" to an essential SoC verification and system validation tool. Siemens Hardware Assisted Verification Blog. https://blogs.sw.siemens.com/hardware-assisted-verification/2024/10/16
[19] Wang, J., Guo, X., Lin, C., Xu, J., & Huang, T. (2024). An effective and comprehensive hypergraph partitioner for VLSI designs. IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems. https://ieeexplore.ieee.org/document/11126419/
[20] Zheng, S., Zhang, J., & Young, E. F. Y. (2021). Multi-FPGA co-optimization: Hybrid routing and competitive-based time division multiplexing assignment. Proceedings of the 58th ACM/IEEE Design Automation Conference (DAC 2021). https://doi.org/10.1145/3394885.3431565
[21] S2C Inc. (2024). Prototypical II: The practice of FPGA-based prototyping for SoC design. S2C Technical Report. https://www.s2cinc.com/resources/lit/en/ebooks/s2c-prototypical-the-practice-of-fpga-based-prototyping-for-soc
-design.pdf
[22] Chen, Z., Yin, H., & Jiang, Y. (2025). RTMF: Routing based on TDM for multi-FPGA system. ACM Transactions on Design Automation of Electronic Systems. https://doi.org/10.1145/3760778
[23] Uddandarao, D. P. (2026). Statistics and the science of causal economics: A new paradigm for data science. Deep Science Publishing.
[24] Uddandarao, D. P. (2026). The reality of an AI implementation. Communications of the ACM, 69(8), 8–9. https://doi.org/10.1145/3821443
[25] Kang, A. S., Basem, A., Kumar, A., et al. (2025). Enhancing biocompatibility: Heat-treated Al_2O_3- and ZrO_2-reinforced hydroxyapatite coatings on metallic implants. The International Journal of Advanced Manufacturing Technology, 141, 6195–6208. https://doi.org/10.1007/s00170-025-17033-1
[26] Varanasi, B. S. S. (2025). AI-enabled continuous authority-to-operate (ATO) frameworks for information systems. Enterprise Development and Microfinance, 35(2), 748–757. http://www.papjournals.com/index.php/edm/article/view/1201]
[27] Tokas, D. (2026). Cloud-native architectures for large-scale remote sensing and geospatial data platforms. Computer Fraud & Security, 2026(1), 650–658.
[28] Gaggar, H. (2026). MemGov: Policy-Governed Memory Lifecycle Management for Enterprise LLM Agents. Journal of Intelligent Decision Making and Information Science, 3(11s), 2199-2214.
[29] . Vytla, S. K. (2024). POLICY-ML: Policy-as-code enforcement for compliance, auditability, and trust across data and machine learning pipelines. International Journal of Computational and Experimental Science and Engineering, 10(4). https://doi.org/10.22399/ijcesen.5393