

Special Day—A Design Blueprint for Scalable Multi-Agent Architectures in Complex EDA Workflows

Valerio Tenace[§], Pierre-Emmanuel Gaillardon^{‡§}

[§]PrimisAI, West Jordan, UT, USA

[‡]University of Utah, Salt Lake City, UT, USA

Abstract—*Electronic Design Automation (EDA) workflows involve complex, tightly coupled tools and artifacts that require high reliability and traceability. Recent advances in Large Language Models (LLMs) have opened new avenues for AI-driven automation through Multi-Agent Systems (MASs), which can decompose complex tasks into manageable subtasks. However, deploying MASs in EDA remains challenging due to weak coordination, unstructured communication, and limited reproducibility. In this paper, we propose a design blueprint for scalable, LLM-based, MASs tailored to EDA workflows, emphasizing hierarchical orchestration, explicit task interfaces, tool-grounded execution, structured communication, modular memory management, and observability with recovery paths. We also introduce Nexus, an open-source Software Development Kit (SDK) that implements these principles, thus enabling low-code workflow specification and robust agent interactions. We validate our approach on open-source benchmarks, achieving 100% functional accuracy on RTL generation (VerilogEval-Human), up to 98.78% functional pass rate on HumanEval, and timing closure with 26.64% average LUT reduction and almost 30% lower total power on VTR designs.*

Index Terms—Large language models, Multi-agent systems, Generative AI, Electronic design automation

I. INTRODUCTION

Electronic Design Automation (EDA) workflows entail a complex and tightly coupled ecosystem of specialized tools that span specification analysis, design implementation, simulation, formal verification, optimization, and several other downstream back-end tasks. Each of these tools exposes heterogeneous interfaces, produces artifacts at multiple levels of abstraction, and is highly sensitive to small inconsistencies that can propagate across stages [1], [2]. As a result, even minor errors in early steps can cascade into costly downstream failures, making reliability and traceability critical concerns.

In an attempt to address these challenges, recent progress in *Large Language Models (LLMs)* has paved the way for a new class of AI-enhanced automation, where *Multi-Agent Systems (MASs)* [3] hold significant promise for effective task automation. By decomposing a complex objective into fine-grained tasks, multi-agent architectures can combine reasoning, planning, and tool execution to navigate EDA workflows that are otherwise difficult to address with a single model invocation. Although MAS-based automation may appear to be a straightforward game-changer on paper, it remains challenging to deploy reliably in practice: when agents interact over unstructured artifacts and heterogeneous tool interfaces, unconstrained communication and weak orchestration can amplify

small mistakes, triggering cascading failures that would eventually shift the effort from design work to *post-hoc* debugging of AI-produced artifacts that must be audited and corrected. Despite encouraging early attempts, building MASs that are reliable and scalable for EDA remains challenging. Prior work has shown that weak coordination policies and unconstrained inter-agent communication can lead to compounding failure modes, unreliable execution traces, and limited reproducibility in multi-agent reasoning systems [4], [5]. In practice, many multi-agent prototypes exhibit limited self-awareness, brittle task decomposition, unbounded context growth, inter-agent misalignment, and poor observability [6]. As a result, systems may perform well on simple benchmarks but degrade significantly when exposed to realistic designs, heterogeneous (and often proprietary) toolchains, or organizational constraints such as reproducibility, auditability, and extensibility within existing workflows. While some limitations are intrinsic to LLMs themselves (e.g., finite context length and stochastic generation, to name a few), a key reason for this gap lies in the architecture of multi-agent systems: the lack of a concrete, standard, and reusable architectural blueprint tailored to the requirements of production-grade EDA automation. Existing work often emphasizes prompting strategies or isolated agent behaviors, while leaving system-level concerns under-specified, e.g., how agent orchestration should be structured, how artifacts should be represented and validated, how tool outputs should be grounded and tracked, how failures should be detected and recovered, and how performance should be evaluated beyond end-task success.

In this paper, we propose a design blueprint for scalable multi-agent architectures tailored to EDA workflows and related applications. By leveraging this blueprint, we also introduce Nexus, an open-source *Software Development Kit (SDK)* for agentic EDA, which distills architectural principles and concrete design requirements that emphasize reliability, modularity, and extensibility. The contributions of this paper can be summarized as follows:

- 1) We propose a design blueprint for building reliable and scalable multi-agent architectures for complex EDA workflows, distilling architectural principles and concrete system-level requirements.
- 2) We introduce a flexible multi-supervisor hierarchy for efficient task delegation, combining a single root *Supervisor* with distributed *Task Supervisors* and specialized *Worker*

agents to support divide-and-conquer execution.

- 3) We present Nexus, a novel open-source SDK that implements the proposed blueprint, featuring explicit task and artifact interfaces, tool-grounded execution, and structured inter-agent communication.
- 4) We validate the proposed approach on open-source benchmarks, achieving state-of-the-art results across multiple EDA tasks. MASs built with the Nexus SDK achieve 100% accuracy on RTL generation tasks on the VerilogEval-Human benchmark suite [7], outperforming recent reasoning language models such as o3-mini and DeepSeek-R1. Moreover, we show that lightweight MAS hierarchies can effectively handle multi-objective optimization by addressing challenging timing-closure tasks on designs from the VTR benchmark suite [8], while delivering, on average, nearly 30% power savings.
- 5) We release the source code of this project on GitHub at <https://github.com/PrimisAI/nexus> under a permissive open-source license.

The remainder of the paper is organized as follows. Section II reviews related work and positions our contribution within the existing literature. Section III details the proposed architectural blueprint and its core components. Section IV presents experimental results on representative EDA workloads and benchmarks. Finally, Section V concludes the paper.

II. BACKGROUND & RELATED WORK

MASs have long provided a powerful paradigm for decomposing complex tasks into autonomous, interacting agents [9], [10]. Traditional MAS architectures relied heavily on rule-based coordination and heuristic protocols for communication and task allocation, which often limited their adaptability and scalability in dynamic environments. Recent advances in LLMs have sparked a new generation of MASs in which agents leverage near-human capabilities for reasoning, planning, and natural-language interaction [11], [12]. This shift enables MASs to tackle more sophisticated, open-ended problems that were previously intractable with hand-engineered rules.

Contemporary solutions such as AutoGPT [13] and HuggingGPT [14] provide turnkey pipelines for task decomposition and tool invocation, but they are typically optimized for single-agent settings and do not natively support fully decentralized or hierarchical architectures. Other frameworks, including LangGraph [15], AutoGen [16], and crewAI [17], offer greater multi-agent flexibility, but often at the cost of substantial engineering effort. Furthermore, many commercial *no-code* or *low-code* platforms simplify deployment but tend to obscure internal workflows and restrict extensibility, limiting researchers' and developers' ability to experiment and customize.

Finally, recent contributions on MASs for EDA often rely on one-off architectural designs that are difficult to reproduce or extend [18], [19]. These systems typically lack modularity, standardized interfaces, and robust coordination mechanisms, making them brittle when deployed in real-world EDA workflows involving heterogeneous tools, complex artifacts, and stringent reliability requirements.

The proposed blueprint has been specifically conceived to address these gaps, providing a platform that balances scal-

ability, transparency, and ease of use. By combining these elements, the proposed solution enables rapid prototyping of robust, scalable LLM-based MASs without sacrificing transparency, robustness, or extensibility. In Section III, we detail the Nexus SDK implementation which leverages the proposed design principles to translate high-level user intents into fully instantiated, executable, EDA-driven workflows.

III. DESIGN BLUEPRINT: PRINCIPLES, REQUIREMENTS, AND IMPLEMENTATION

We define a *design blueprint* as a set of architectural principles and system-level requirements that constrain how agents, artifacts, tools, and memory interact in real-world EDA automation. Concretely, the blueprint requires: (i) **hierarchical orchestration** for scalable task decomposition and delegation; (ii) **explicit task and role assignments** to standardize responsibilities and enable validation; (iii) **tool-grounded execution** with tracked tool calls and smart log distillation strategies; (iv) **structured inter-agent communication** that preserves context and metadata; (v) **modular memory and context management** with controlled access and stable state; and (vi) **observability and recovery paths** to detect failures, reproduce executions, and enable retries or reassignment. In the remainder of this section, we describe Nexus as a reference implementation and highlight how each component implements these requirements.

A. Nexus SDK: Core Components and Structure

The core components of the Nexus SDK implement the blueprint requirements on hierarchical orchestration, explicit interfaces, tool-grounded execution, structured communication, and modular memory. In fact, Nexus architectures rely on a single root *Supervisor* that mediates interactions between the user and the network of agents. Its primary responsibilities include: (i) **task decomposition**, i.e., breaking high-level prompts into actionable subtasks; (ii) **agent selection**, where subtasks are delegated to the most appropriate *Worker* agent (or an intermediate *Task Supervisor*, when instantiated) based on each agent's specialization; and (iii) **result aggregation**, i.e., collecting outputs from delegated subtasks and synthesizing them into a cohesive final response.

Worker agents are specialized problem solvers tasked with executing the subtasks assigned by their supervisor. Each *Worker* operates in an isolated environment and is configured with a distinct *specialization* (via its system message) together with an associated set of tools, environment variables, and *ad hoc* functions. Their key capabilities include: (i) invoking dedicated tools (e.g., web search, shell commands, file manipulation) or knowledge bases to perform domain-specific operations; (ii) iteratively refining intermediate results through interactions with these tools or external data sources; and (iii) returning structured, task-specific outputs to the appropriate supervisor upon completion.

In addition to these agents, Nexus incorporates a global *Memory* mechanism along with a set of external *Tools*. The *Memory* serves as a centralized repository for partial results, metadata, and relevant instructions, ensuring that all agents maintain an up-to-date view of task progress. Although the *Memory* component is shared, we enforce role-based access

control: the *Supervisor* has global access, a *Worker* is confined to its own event history, and a *Task Supervisor* can access all memory locations associated with its assigned agents. External *Tools* provide specialized functionality (e.g., data processing pipelines or access to external computational resources), enabling tool-grounded execution while preserving transparency and traceability.

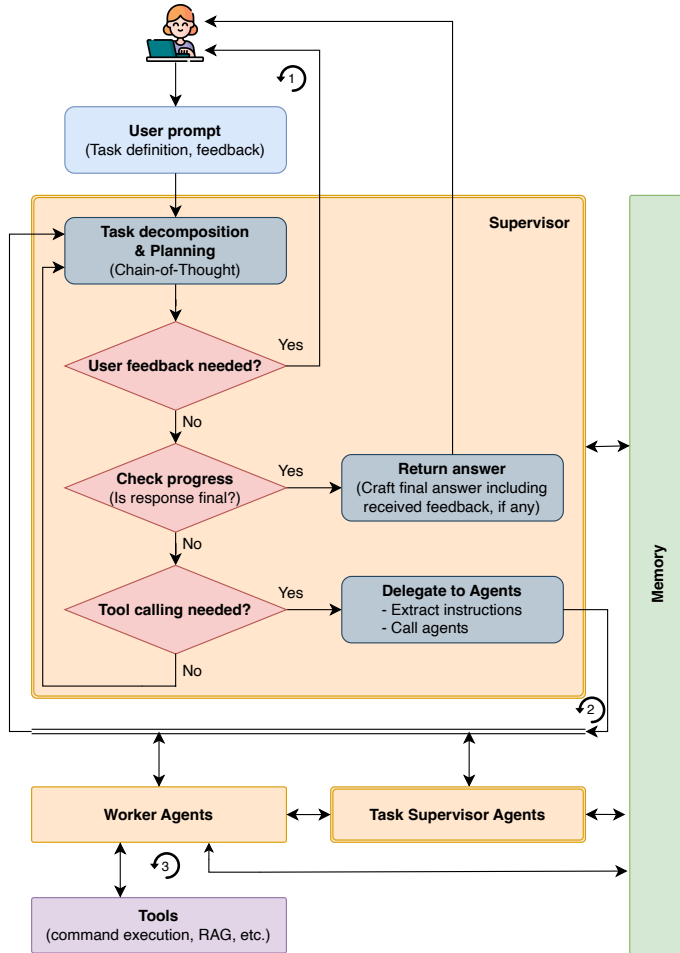


Fig. 1. Overview of the Nexus architecture. A root *Supervisor* receives user prompts and decides whether to finalize the solution or delegate its execution. Tasks of moderate complexity can be handled by specialized *Worker* agents, while particularly intricate tasks can be coordinated by intermediate *Task Supervisors*. *Memory* maintains a synchronized record of partial outputs and distilled relevant context. Circled markers denote the three main loops that are entailed in the proposed workflow.

Figure 1 depicts the overall architecture and workflow of Nexus. More formally, the architecture is modeled as a rooted directed graph, denoted by

$$\Gamma = (V, E), \quad (1)$$

which captures the relationships between various agents and components within the overall structure. The vertex set V is partitioned into the following three disjoint subsets:

$$V = S \cup T \cup W, \quad (2)$$

where S represents the set of *Supervisor* agents with the

unique root node $s \in S$, T denotes the set of *Task Supervisor* agents, and W corresponds to the set of *Worker* agents. Based on these assumptions, the edge set

$$E \subseteq V \times V \quad (3)$$

captures all relationships among agents defined within any given Γ . A critical element of this structure is the hierarchical relationship between two or more elements, which can be formalized by introducing a parent function defined as

$$\varphi: V \setminus \{s\} \rightarrow V. \quad (4)$$

For every node $v \in V \setminus \{s\}$, the directed edge

$$(\varphi(v), v) \in E \quad (5)$$

identifies the immediate supervisor-to-agent relationship between node v and its parent node $\varphi(v)$. This relationship adheres to the following constraints: first, if $v \in T$ (a *Task Supervisor*), then its parent $\varphi(v)$ must belong to S ; second, if $v \in W$ (a *Worker*), then $\varphi(v)$ must belong to $T \cup S$. Consequently, this design guarantees that every agent, other than the root, has a unique predecessor, thus ensuring that there exists a unique directed path from the root s to any node $v \in V$. In addition to these hierarchical relationships, the architecture also allows the inclusion of extra edges that capture non-hierarchical interactions between components and the shared memory. Although these communication edges do not necessarily conform to the strict parent-child relationship, they are incorporated in such a way as to maintain the overall hierarchical integrity of Γ .

B. Multi-Loop Runtime Execution

Each Nexus-based architecture follows an iterative workflow for task decomposition and execution, organized into three primary interaction loops (Figure 1, circled markers). These loops implement the blueprint requirements on hierarchical supervision, tool-grounded execution, and observability with recovery paths.

Loop 1: User-Supervisor Interaction. In the first loop, the user provides a high-level prompt to the *Supervisor*. The *Supervisor* interprets the request and proposes an initial execution plan, optionally eliciting clarifications or constraints to ensure that the plan remains aligned with the user's objectives. This exchange continues until the *Supervisor* either finalizes a solution or delegates one or more subtasks.

Loop 2: Supervisor-Agent Coordination. In the second loop, the *Supervisor* (or a *Task Supervisor*, when instantiated) assigns well-defined subtasks to *Worker* agents based on their specialization and available tools. *Worker* agents produce intermediate outputs by invoking tools and operating on shared artifacts (via *Memory*). If an execution attempt fails, yields low-quality results, or encounters a bottleneck, the supervisor refines the subtask specification, adjusts constraints, or reassigns the task to another agent. This coordination loop repeats until the subtasks collectively satisfy the prescribed completion criteria. In case of failure, the supervisor can also re-delegate the task to a different agent or escalate it to a higher-level supervisor (in

case this happens with the root *Supervisor* the user represents the ultimate authority) for further analysis and guidance.

Loop 3: Worker Internal Iteration. The third loop runs locally within each *Worker*'s execution environment. After receiving a subtask, a *Worker* iteratively invokes relevant tools, consults local data structures or external knowledge bases, and updates intermediate artifacts until it reaches a satisfactory solution. The final output is then returned to its supervisor for aggregation and synthesis.

In summary, Nexus combines a hierarchical execution graph with iterative feedback loops to support diverse interaction patterns across agents and tools. This design equips the framework with three fundamental properties:

- **Scalability:** The framework can incorporate new *Worker* agents or supervisory nodes as task complexity increases.
- **Modularity:** *Worker* agents operate independently, enabling parallelism and straightforward integration or replacement of domain-specific capabilities.
- **Robustness:** Hierarchical delegation and iterative refinement reduce the impact of individual agent failures by enabling retries, reassignment, and incremental plan updates.

IV. RESULTS

In this section, we present the experimental evaluation of the Nexus SDK. Our objective is to assess both performance and robustness across representative EDA workloads and benchmarks. We organize the discussion into two case studies: (i) timing-closure and multi-objective optimization on designs from the VTR benchmark suite, and (ii) code generation tasks on HumanEval and VerilogEval-Human, evaluating both self-verifying and non-self-verifying workflows.

A. Methodology

In all experiments, except those reported in Section IV-B, performance is evaluated using the *pass rate* (denoted by \mathcal{A}), which is defined as the ratio between the number of samples that pass all checks and the total number of samples in the benchmark suite. In coding-related tasks, we distinguish between \mathcal{A}_s , i.e., the success rate of solutions passing all syntax checks, and \mathcal{A}_f , which reflects the success rate of designs that are not only syntactically correct but also functionally accurate. Notably, \mathcal{A}_f was determined by executing the tests provided in the benchmark suite, ensuring a comprehensive validation of the proposed approach. For the agents, we employed Claude 3.5 Sonnet v2¹ [20] configured with a *temperature* of 0.7 and a *top_p* of 1.

B. Case Study I: Timing Closure

Achieving timing closure and optimizing resource utilization are fundamental challenges in hardware design. In practice, synthesis, placement, and routing strategies are routinely employed to balance stringent timing constraints with efficient hardware resource usage, a balance that is critical for the successful deployment of complex applications on modern

computer architectures. To assess the efficacy of our proposed solution, we conducted extensive experiments using benchmark designs from the well-established VTR benchmark suite [8]. These benchmarks span a diverse range of application domains, including computer vision (`stereovision0`, `stereovision1`), signal processing (`diffeq1`, `diffeq2`), cryptography (`sha`), and various encoding-decoding applications. In our experimental setup, we leverage the Xilinx Vivado 2020 Design Suite (hereafter, Vivado) for synthesis, placement, and routing, with all implementations targeting the Xilinx Alveo U200 card.

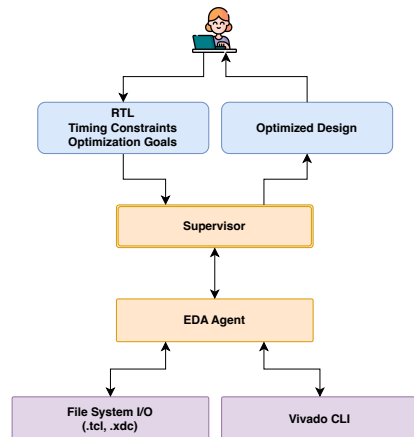


Fig. 2. Proposed architecture for achieving timing closure and design optimization in EDA.

For the baseline, the designs were synthesized and mapped by specifying only its target timing constraints, without employing additional optimization techniques. On the other hand, Figure 2 illustrates the adopted architecture developed with the Nexus SDK, in which all agents are powered by Claude 3.5 v2. The workflow is summarized as follows:

- **User Input:** The process begins with the user providing the RTL code along with a prompt that details the timing constraints and optimization goals, i.e., minimum area and power.
- **Analysis:** The *Supervisor* processes these inputs to generate design constraints and commands specifically tailored to meet the timing requirements using Vivado.
- **Execution:** The generated constraints and commands are forwarded to the *EDA* agent, which writes them to tool-specific files (e.g., `.xcd` for constraints and `.tcl` for commands) and interfaces directly with the Vivado tool environment. The *EDA* agent then issues these commands and retrieves reports that provide detailed information on resource utilization, power consumption, timing metrics, and critical paths.
- **Feedback:** The *Supervisor* reviews the reports and iteratively refines the optimization strategy until timing closure is achieved or no further improvements are possible.

Table I reports the figures of merit for each benchmark. In particular, the column *Frequency* reports the target frequency for each benchmark, columns *LUTs* and *FFs* report resource utilization, while column *WSN* reports the worst negative slack

¹Model accessed through AWS Bedrock with identifier anthropic.claude-3-5-sonnet-20241022-v2:0.

TABLE I
RESULTS FOR TIMING CLOSURE TASKS. SYMBOLS IN COLUMN WNS INDICATE WHETHER TIMING CONSTRAINTS ARE MET (✓) OR NOT (✗).

Design	Frequency (MHz)	LUTs		FFs		WNS (ns)		Power (W)	
		Baseline	Nexus	Baseline	Nexus	Baseline	Nexus	Baseline	Nexus
diffeq1	150	357	345	209	209	-0.187 ✗	0.022 ✓	2.634	2.6
blob_merge	200	5400	5227	575	575	0.402 ✓	0.0384 ✓	2.512	2.51
stereovision0	333	3959	3176	10290	7595	0.5 ✓	0.313 ✓	2.995	3
stereovision1	200	13321	1281	11843	6186	1.269 ✓	1.42 ✓	2.963	2.9
diffeq2	167	229	232	111	111	0.011 ✓	0.032 ✓	2.618	2.621
sha	300	1031	998	895	895	0.3 ✓	0.298 ✓	2.577	2.551
stereovision2	154	9862	8062	13589	17619	0.403 ✓	0.8 ✓	3.268	3.14
stereovision3	500	57	78	99	144	0.9 ✓	0.806 ✓	2.493	0.61
mkPktMerge	500	12	16	16	16	0.389 ✓	0.174 ✓	3.904	2.01
mkSMAadapter4B	200	910	885	859	865	1.363 ✓	0.92 ✓	2.612	2.62
LU8PEEng	65	14581	13740	5703	3569	-11.409 ✗	0.24 ✓	2.905	0.98
bgm	200	10801	10292	5063	6150	0.257 ✓	0.133 ✓	2.947	0.98
boundtop	770	221	218	205	444	0.459 ✓	0.279 ✓	2.5	0.685
ch_intrinsics	1250	25	25	90	122	-0.029 ✗	0.018 ✓	2.68	0.768
Δ Avg.			-26.64%		-10.19%				-29.37%

and the last column reports the total power. As can be noted, across all benchmarks, the proposed MAS architecture not only achieved timing closure at the target frequencies, but it also delivered significant improvements in key metrics, with an average LUT reduction of 26.64% and a power reduction of nearly 30%.

Among the benchmarks, the LU8PEEng stands out as a compelling demonstration of our proposed architecture’s optimization capabilities. This design posed significant challenges for the baseline strategy, with a WNS of -11.409ns observed at 65MHz. In contrast, our architecture achieved a positive slack of 0.24ns while maintaining the target frequency. Additionally, the framework improved resource utilization: LUT usage decreased by 5.77% (from 14,581 to 13,740) and flip-flop utilization dropped by 37.42 (from 5,703 to 3,569). This dual optimization of timing and resources highlights the framework’s ability to effectively navigate complex design trade-offs. Furthermore, the architecture strategically reallocated block RAM resources, increasing BRAM utilization from 42 to 71 units while maintaining DSP usage at 16 units, to achieve optimal implementation. Notably, these improvements were accompanied by a substantial 3× reduction in power consumption, lowering it from 2.9W to 0.98W. This significant power optimization, alongside enhanced timing and maintained functionality, underscores the architecture’s ability to efficiently exploit advanced features in professional-grade EDA tools, thereby enabling the simultaneous achievement of multiple competing optimization objectives without human intervention.

C. Case Study II: Coding Tasks

In this section, we assess the effectiveness of the Nexus-based MASs in addressing programming-related tasks. Our evaluation encompasses two benchmark families: HumanEval [21], a suite of 164 problems focused on Python code generation, and VerilogEval-Human [7], which comprises 156 challenges involving Verilog code generation and verification. Notably, our approach leverages a single, unified Nexus architecture that is consistently applied across both sets of coding challenges.

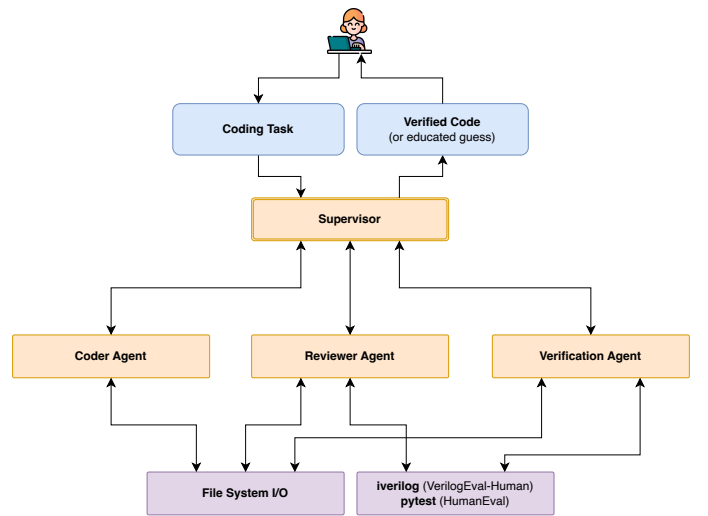


Fig. 3. Unified Nexus-based MAS architecture for solving code-related tasks.

As depicted in Figure 3, the implemented MAS architecture comprises the following core agents: (i) a *Coder* agent, responsible for generating code solutions (in either Python or Verilog) along with corresponding unit tests or testbenches; (ii) a *Reviewer* agent, tasked with reviewing and refining code to identify and correct syntax or runtime issues; and (iii) a *Verification* agent, which executes tests or simulations to assess functionality. As mentioned earlier, the overall structure remains consistent for both benchmark suites, with the specific tools employed (i.e., `pytest` or `iverilog`) chosen to suit the target domain. The workflow proceeds as follows:

- 1) **Planning & Task Delegation:** The *Supervisor* receives the user’s prompt, decomposes the problem into multiple tasks, and assigns the *Coder* agent to initiate the first iteration.
- 2) **Code Generation:** The *Coder* agent produces the Verilog (or Python) solution along with a testbench (or a set of unit tests), storing the output via the `save_code` tool, a utility function defined as part of the *File System I/O* interface.

TABLE II
COMPARISON OF THE PROPOSED *self-verifying* AND *non-self-verifying* MASS DESIGNED WITH THE NEXUS SDK VERSUS RELEVANT EXISTING SOLUTIONS. HUMAN EVAL NUMBERS ARE GATHERED FROM THE PAPERS WITH CODE LEADERBOARD [22].

Benchmark Suite	Technology	Self Verifying	Model	\mathcal{A}_f
HumanEval	L2MAC [23]	Yes	GPT-4	90.2
	MapCoder [24]	Yes	GPT-4	93.9
	AgentCoder [25]	Yes	GPT-4	96.3
	LLMDebugger [26]	Yes	GPT-4o	98.2
	LPW [27]	Yes	GPT-4o	98.2
	QualityFlow [28]	Yes	Claude 3.5	98.8
	Nexus (this work)	Yes	Claude 3.5	98.8
LLMDebugger [26]	Yes	o1	99.4	
VerilogEval-Human	RTLFixer [19]	Yes	GPT-3.5	36.8
	VeriAssist* [18]	Yes	GPT-4	48.3
	Alvri1 [29]	Yes	Claude 3.5	67.3
	Alvri2 [30]	Yes	Claude 3.5	77
	Nexus (this work)	Yes	Claude 3.5	85.9
	VeriAssist* [18]	No	GPT-4	50.5
	VerilogCoder [31]	No	GPT-4	94.2
	MAGE [32]	No	Claude 3.5	94.8
Nexus (this work)	No	Claude 3.5	100	

* This framework employs a dual-mode verification mechanism.

- 3) **Syntax Adjustments:** The *Reviewer* analyzes the proposed implementation by leveraging the `get_code` tool and attempts to execute it using `iverilog` (or the Python interpreter). If syntax errors are detected, corrective prompts are issued and sent back to the *Coder*, which iterates on the previous version to generate a revised solution.
- 4) **Functional Adjustments:** Once the loop between the *Coder* and *Reviewer* concludes and the candidate solution is syntactically correct, the *Supervisor* delegates the *Verification* agent to run the unit tests using `iverilog` (or `pytest`) to assess functionality and correctness. In this stage, any errors are analyzed and communicated back to the *Supervisor* agent for further refinement.
- 5) **Wrap-up:** The *Supervisor* collects the verified output and returns the final solution to the user.

It is important to note that this workflow is considered *self-verifying*, meaning that the overall system autonomously devises tests without any external input.

1) *Comparison with State-of-the-Art Approaches:* Applying LLMs to solve coding challenges is an emerging topic that has already captured significant attention, as these models are rapidly transforming how developers approach programming tasks on a daily basis [33]. Table II compares the performance of the proposed workflow with the most relevant solutions in this fast-evolving application field.

For a fair comparison, we divide the results into three main categories: (i) self-verifying MASs designed to solve Python-related programming tasks, as measured by the HumanEval benchmark suite; (ii) self-verifying MASs intended to produce and autonomously verify Verilog solutions to help hardware engineers meet their requirements; and (iii) non-self-verifying MASs that, in addition to the user prompt, incorporate guidance on how to verify the RTL produced by the LLM. For the latter, we introduce a slightly revised workflow compared to the one presented earlier in this section. Instead of generating

the testbench autonomously, this approach uses the testbench provided in the benchmark suite as a blueprint to create its own version, following a principle similar to the one adopted in previous works [18], [31], [32]. As the numbers indicate, on HumanEval the proposed workflow ranks *ex aequo* in second place behind LLMDebugger [26]. This result is particularly noteworthy given that the same workflow, when applied to a completely different programming language, and thus operating sub-optimally relative to its intended domain, remains effective despite being orthogonal to the programming language stack. As a result, when considering the VerilogEval-Human benchmark suite, both the *self-verifying* and *non-self-verifying* workflows outperform existing solutions. Notably, the *non-self-verifying* version achieves a remarkable 100% accuracy, a feat that, to the best of the authors' knowledge, has never been achieved before.

V. CONCLUSIONS

This paper introduced a design blueprint for reliable, scalable multi-agent architectures in complex EDA workflows, formalizing often-missing system requirements such as hierarchical supervision, explicit task/artifact interfaces, tool-grounded execution, structured communication, modular memory, and observability. We also presented Nexus, an open-source SDK that implements these principles in a reusable framework.

Experiments on two case studies support the effectiveness of the proposed blueprint. On VTR timing-closure and multi-objective optimization, MASs developed with the Nexus SDK achieved timing closure at target frequencies while reducing LUTs by 26.64% on average and total power by 29.37%. On coding benchmarks, the self-verifying workflow outperformed existing solutions, while the non-self-verifying workflow reached 100% accuracy on VerilogEval-Human.

Overall, these results validate the proposed design blueprint and the Nexus SDK as effective tools for building robust, scalable LLM-based multi-agent systems in EDA.

REFERENCES

- [1] R. Bahar, A. K. Jones, S. Katkooi, P. H. Madden, D. Marculescu, and I. L. Markov, "Workshops on extreme scale design automation (esda) challenges and opportunities for 2025 and beyond," *arXiv preprint arXiv:2005.01588*, 2020.
- [2] J. Jung, A. B. Kahng, S. Kim, and R. Varadarajan, "Metrics2. 1 and flow tuning in the ieee ceda robust design flow and openroad iccad special session paper," in *2021 IEEE/ACM International Conference On Computer Aided Design (ICCAD)*. IEEE, 2021, pp. 1–9.
- [3] T. Guo, X. Chen, Y. Wang, R. Chang, S. Pei, N. V. Chawla, O. Wiest, and X. Zhang, "Large language model based multi-agents: A survey of progress and challenges," *arXiv preprint arXiv:2402.01680*, 2024.
- [4] Z. He, Y. Pu, H. Wu, T. Qiu, and B. Yu, "Large language models for eda: Future or mirage?" *ACM Transactions on Design Automation of Electronic Systems*, vol. 30, no. 6, pp. 1–53, 2025.
- [5] H. Wu, H. Zheng, Z. He, and B. Yu, "Divergent thoughts toward one goal: Llm-based multi-agent collaboration system for electronic design automation," *arXiv preprint arXiv:2502.10857*, 2025.
- [6] M. Cemri, M. Z. Pan, S. Yang, L. A. Agrawal, B. Chopra, R. Tiwari, K. Keutzer, A. Parameswaran, D. Klein, K. Ramchandran *et al.*, "Why do multi-agent llm systems fail?" *arXiv preprint arXiv:2503.13657*, 2025.
- [7] M. Liu, N. Pinckney, B. Khailany, and H. Ren, "VerilogEval: evaluating large language models for verilog code generation," in *2023 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*, 2023.
- [8] K. E. Murray, O. Petelin, S. Zhong, J. M. Wang, M. Eldafrawy, J.-P. Legault, E. Sha, A. G. Graham, J. Wu, M. J. Walker *et al.*, "Vtr 8: High-performance cad and customizable fpga architecture modelling," *ACM Transactions on Reconfigurable Technology and Systems (TRETS)*, vol. 13, no. 2, pp. 1–55, 2020.
- [9] M. Wooldridge and N. R. Jennings, "Intelligent agents: Theory and practice," *The knowledge engineering review*, vol. 10, no. 2, pp. 115–152, 1995.
- [10] P. Stone and M. Veloso, "Multiagent systems: A survey from a machine learning perspective," *Autonomous Robots*, vol. 8, pp. 345–383, 2000.
- [11] J. S. Park, J. O'Brien, C. J. Cai, M. R. Morris, P. Liang, and M. S. Bernstein, "Generative agents: Interactive simulacra of human behavior," in *Proceedings of the 36th annual acm symposium on user interface software and technology*, 2023, pp. 1–22.
- [12] S. Yao, J. Zhao, D. Yu, N. Du, I. Shafran, K. Narasimhan, and Y. Cao, "ReAct: Synergizing reasoning and acting in language models," *arXiv preprint arXiv:2210.03629*, 2022.
- [13] AutoGPT. (2025) AutoGPT: Build, Deploy, and Run AI Agents. [Online]. Available: <https://github.com/Significant-Gravitas/AutoGPT>
- [14] Y. Shen, K. Song, X. Tan, D. Li, W. Lu, and Y. Zhuang, "Hugginggpt: Solving ai tasks with chatgpt and its friends in hugging face," *Advances in Neural Information Processing Systems*, vol. 36, 2024.
- [15] LangChain. (2025) LangGraph. [Online]. Available: <https://github.com/langchain-ai/langgraph>
- [16] Q. Wu, G. Bansal, J. Zhang, Y. Wu, S. Zhang, E. Zhu, B. Li, L. Jiang, X. Zhang, and C. Wang, "Autogen: Enabling next-gen llm applications via multi-agent conversation framework," *arXiv preprint arXiv:2308.08155*, 2023.
- [17] CrewAI. (2025) CrewAI: Production-grade framework for orchestrating sophisticated AI agent systems. [Online]. Available: <https://github.com/crewAIInc/crewAI>
- [18] H. Huang, Z. Lin, Z. Wang, X. Chen, K. Ding, and J. Zhao, "Towards LLM-Powered Verilog RTL Assistant: Self-Verification and Self-Correction," *arXiv preprint arXiv:2406.00115*, 2024.
- [19] Y. Tsai, M. Liu, and H. Ren, "RTLFixer: Automatically fixing RTL syntax errors with large language model," in *Proceedings of the 61st ACM/IEEE Design Automation Conference*, 2024, pp. 1–6.
- [20] Anthropic, "Claude 3.5 Sonnet," <https://www.anthropic.com/news/claude-3-5-sonnet>, 2025, accessed: 2025-07-15.
- [21] M. Chen, J. Tworek, H. Jun, Q. Yuan, H. P. D. O. Pinto, J. Kaplan, H. Edwards, Y. Burda, N. Joseph, G. Brockman *et al.*, "Evaluating large language models trained on code," *arXiv preprint arXiv:2107.03374*, 2021.
- [22] (2025) Code Generation on HumanEval. [Online]. Available: <https://paperswithcode.com/sota/code-generation-on-humaneval>
- [23] S. Holt, M. R. Luyten, and M. van der Schaar, "L2mac: Large language model automatic computer for unbounded code generation," *arXiv preprint arXiv:2310.02003*, 2023.
- [24] M. A. Islam, M. E. Ali, and M. R. Parvez, "MapCoder: Multi-Agent Code Generation for Competitive Problem Solving," *arXiv preprint arXiv:2405.11403*, 2024.
- [25] D. Huang, Q. Bu, J. M. Zhang, M. Luck, and H. Cui, "Agentcoder: Multi-agent-based code generation with iterative testing and optimisation," *arXiv preprint arXiv:2312.13010*, 2023.
- [26] L. Zhong, Z. Wang, and J. Shang, "Debug like a human: A large language model debugger via verifying runtime execution step by step," *arXiv preprint arXiv:2402.16906*, 2024.
- [27] C. Lei, Y. Chang, N. Lipovetzky, and K. A. Ehinger, "Planning-driven programming: A large language model programming workflow," *arXiv preprint arXiv:2411.14503*, 2024.
- [28] Y. Hu, Q. Zhou, Q. Chen, X. Li, L. Liu, D. Zhang, A. Kachroo, T. Oz, and O. Tripp, "Qualityflow: An agentic workflow for program synthesis controlled by llm quality checks," *arXiv preprint arXiv:2501.17167*, 2025.
- [29] M. ul Islam, H. Sami, P.-E. Gaillardon, and V. Tenace, "AIvritl: AI-Driven RTL Generation With Verification In-The-Loop," *arXiv preprint arXiv:2409.11411*, 2024.
- [30] —, "EDA-Aware RTL Generation with Large Language Models," *arXiv preprint arXiv:2412.04485*, 2024.
- [31] C.-T. Ho, H. Ren, and B. Khailany, "Verilogcoder: Autonomous verilog coding agents with graph-based planning and abstract syntax tree (ast)-based waveform tracing tool," *arXiv preprint arXiv:2408.08927*, 2024.
- [32] Y. Zhao, H. Zhang, H. Huang, Z. Yu, and J. Zhao, "MAGE: A Multi-Agent Engine for Automated RTL Code Generation," *arXiv preprint arXiv:2412.07822*, 2024.
- [33] D. Etsenake and M. Nagappan, "Understanding the Human-LLM Dynamic: A Literature Survey of LLM Use in Programming Tasks," *arXiv preprint arXiv:2410.01026*, 2024.