

# CHIA: An open-source framework for principled, agentic AI-driven hardware/software co-design research

Angela Cui\*, Ferran Hermida-Rivera\*, Jack Toubes\*,

Raghav Gupta, Jim Fang, Chengyi Lux Zhang, Ella Schwarz, Junha Kim,  
Yakun Sophia Shao, Borivoje Nikolić, Christopher W. Fletcher, Sagar Karandikar

University of California, Berkeley

{angelacui,ferran,jtoubes,raghavgupta,yf238,iansseijelly,schwarzem,junhak}@berkeley.edu  
{ysshao,bora,cwfletcher,sagark}@eecs.berkeley.edu

## Abstract

Agentic artificial intelligence shows great promise for radically improving the pace of innovation in hardware/software co-design research across computer architecture, systems, compilers, and VLSI. Thus far, however, applications of AI in these contexts have generally been demonstrated in isolated settings on small-scale problems, due to the difficulty of designing and deploying complex AI-infused hardware and software development workflows.

This paper introduces CHIA, an open-source hardware/software co-design framework for agile and principled research on the application of AI to co-design. CHIA treats the productive construction and scalable deployment of the co-design flow itself as a first-class objective. In CHIA, agentic AI-driven hardware and software design flows are expressed as *CHIA loops*: directed cyclic graphs whose nodes execute various system-on-chip design tools, microarchitectural simulators, software build systems, AI models, evolutionary coding agents, and more. The *CHIA library* provides node implementations for many popular tools, including Chipyard, gem5, ChampSim, FireSim, Hammer (thus several commercial ASIC CAD tools), Vivado, AlphaEvolve, AdaEvolve, and many others.

CHIA also provides a broad set of features to conduct principled science around these flows. These include isolation between AI models and hardware tools, profiling mechanisms, fault-tolerant execution, and reliability at scale across hundreds of heterogeneous systems (CPUs, FPGAs, GPUs, etc., across public cloud/on-prem.).

To showcase CHIA, we present five CHIA loops as case studies: (1) automatic RTL-to-gem5 simulator alignment, (2) LLM-driven implementation of microarchitectural features in RTL, (3) agentic, IPC-aware critical path optimization, (4) evolutionary architectural discovery, and (5) maintainer-friendly agentic GitHub issue fixing.

Across the case studies, the agents operating within CHIA loops are able to meet the high standards of verification and validation that the loops enforce. For example, the RTL implemented by agents delivers substantial performance improvements while successfully executing all 25+ trillion instructions of the SPEC CPU2006 reference suite within an out-of-order superscalar microprocessor in a RISC-V system-on-chip. These agent-written implementations improve performance while meeting or improving frequency and area constraints in open-source/commercial ASIC PDKs.

These case studies are not an exhaustive demonstration of CHIA’s capabilities. Rather, we hope they serve as an instructive set of examples for the community to build on.

## 1 Introduction

The design of modern computing systems is a challenging and labor-intensive endeavor. Architects, system designers, compiler engineers, and VLSI specialists must navigate enormous design spaces while reasoning across several abstraction layers, including software, architecture, microarchitecture, and physical implementation. Pushing the limits of co-design across these layers remains an important challenge and opportunity.

Recent advances in agentic artificial intelligence offer an opportunity to accelerate the pace of innovation in both hardware design and at the hardware/software boundary. For example, in November 2025, Gupta et. al. [8, 43] demonstrated the ability to harness AI to automatically generate new state-of-the-art microarchitectural components (cache replacement policies) using the context of established design championships. As another case-in-point, Krishna, et. al. [98, 99] recently demonstrated the generation of a 5-stage RISC-V microprocessor design from scratch using agentic AI. Computer architecture has clearly entered a new era, wherein humans, agents, and existing hardware design tools must all smoothly operate together [8, 13, 15, 18, 24, 28, 29, 42, 63, 91, 93, 97, 99, 100, 105, 107, 110]. This is a tall order, considering that constructively composing hardware design tools was a challenge even before the application of AI. Moreover, while the aforementioned studies show interesting achievements from applying AI to hardware design and HW/SW co-design problems, they present early and individual points in the space of potential AI-driven design flows. *How* to design the most effective AI-driven design flows remains a critical open question.

Naively deploying existing hardware design workflows is not sufficient to tackle this question. The quick-and-dirty approach of writing bespoke scripts that glue together the output of one tool into the input of the next tool (or AI model) breaks easily and does not scale well to large AI-driven design flows on hundreds of heterogeneous machines (CPUs, FPGAs, GPUs, etc.). Shifting from script writing to building a distributed and scalable implementation of each bespoke AI-driven design flow is error-prone and imposes immense engineering cost, even with AI-assistance. AI agents can also orchestrate design flows themselves, sometimes desirably so. However, relying strictly on agents to orchestrate the entire design flow (e.g., letting a coding agent “run wild” inside

\*Equal Contribution.



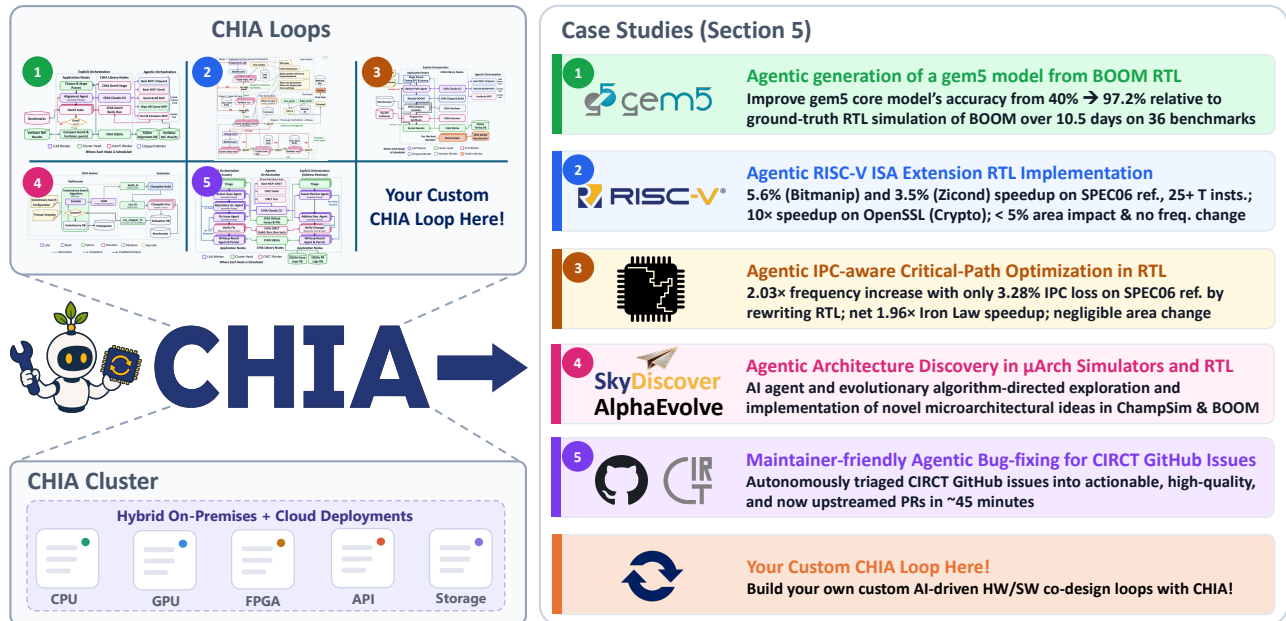


Figure 1: Executive summary of this work.

system-on-chip design frameworks [6, 14, 64, 88] or microarchitectural simulators [16, 21, 38, 61, 71, 90]) makes it challenging to enforce verification and validation requirements that are critical to successful hardware design while minimizing human review overhead. Though several design flows have been proposed (e.g., evolutionary coding agents, multi-agent collaborative environments, etc.), they still prescribe a single design pattern, rather than enabling the expression of arbitrary AI-driven hardware design flows. Altogether, there is no good option for productively expressing and reliably deploying new agentic AI-driven hardware/software co-design flows, which hinders the progress of the field.

We built CHIA<sup>1</sup> to address this problem. CHIA is an open-source, agent-forward hardware/software co-design framework built on the premise that the productive construction and scalable deployment of the co-design flow itself should be a first-class objective. CHIA enables agile and principled research on the application of AI to co-design by making complex AI-infused workflows easy to express, deploy, and study.

In CHIA, the expression of a design flow is called a “CHIA loop”<sup>2</sup>, which is a directed cyclic graph whose nodes execute various system-on-chip design tools, microarchitectural simulators, software build systems, AI models, evolutionary coding agents, and more. The CHIA library includes node implementations for many widely used tools, including Chipyard, gem5, ChampSim, FireSim, Hammer (and thus several commercial ASIC CAD tools), Vivado, AlphaEvolve, AdaEvolve, and many others, allowing researchers

to compose sophisticated flows from reusable components rather than from scratch.

Beyond composition, CHIA provides the features needed to conduct principled science around these flows. These include isolation between AI models and hardware tools, profiling mechanisms, and fault tolerance and reliability at scale across hundreds of heterogeneous systems. Together, these capabilities allow co-design experiments to be run reproducibly and at a scale that makes meaningful evaluation of agent capabilities possible.

To showcase CHIA, we present five example CHIA loops as case studies: (1) automatic, agentic generation of a representative gem5 model for BOOM; (2) LLM-driven implementation of microarchitectural features in RTL in an out-of-order superscalar RISC-V microprocessor, BOOM [111]; (3) agentic, IPC-aware critical path optimization; (4) evolutionary architectural discovery flows (à la [8, 18, 43]); and (5) maintainer-friendly automatic GitHub-issue fixing in the CIRCT compiler project [30]. Throughout, we hold the agents to a high standard of verification and validation. For example, when RTL changes are made, we show that RTL implemented by the agents delivers substantial performance improvements while successfully executing the entire 25+ trillion instructions of SPEC06-Ref within a complete out-of-order superscalar microprocessor (BOOM) in a RISC-V system-on-chip (Chipyard) [6]. These gains are achieved while meeting or improving frequency and area constraints in both open-source and commercial ASIC PDKs.

These case studies are not an exhaustive demonstration of what CHIA can do. Rather, we offer them as an instructive set of examples that illustrate the framework’s capabilities and that we hope the community will build upon. CHIA will be open sourced in the coming days and we are excited to see what the community builds upon it.

<sup>1</sup>CHIA originally stood for Chipyard Intelligent Agents (or: the Chipyard Intelligence Agency), but was retroactively renamed as we realized its reach was far beyond a single RTL/system-on-chip (SoC) design framework. CHIA now stands for Co-designing Hardware/software with Intelligent Agents, and supports many common design flows.

<sup>2</sup>CHIA Project Website: <https://chialoops.ai>

## 2 CHIA Design Goals

In this section we highlight the capabilities we want CHIA to provide to its users, to enable them to build productive, agile, AI-driven design workflows.

In an agentic loop targeting a task like the hardware/software co-design of a domain-specific accelerator, agents can significantly reduce the time to perform many relevant tasks, from profiling to RTL implementation. Moreover, agents, unlike humans, are able to perform these tasks at huge scale and can implement tens and even hundreds of designs in parallel. Such massive parallelism is what many LLM-guided search and optimization algorithms, including evolutionary algorithms, leverage. At such a large scale, brute force scripting or bespoke construction of an entire distributed design flow for each new approach becomes infeasible. **CHIA’s graph-based loop abstraction must allow users to build parameterized, massive-scale design pipelines simply.**

At this scale, classic distributed computing challenges such as task scheduling become important. Tasks in hardware design loops occupy a wide range of latencies, from milliseconds to days, and have diverse and strict resource requirements, such as FPGAs for simulation or GPUs for model serving or ASIC CAD tools. Similarly, node failures will impact jobs more frequently at this scale. **The nodes in CHIA loops should be flexibly and fault-tolerantly scheduled onto a dynamic and heterogeneous substrate.**

As a project progresses, an architect may use many different tools, each with different input and output shapes, and organize them in many different ways. A naive approach that pieces together different tools using bespoke scripts or programs becomes brittle at scale and increasingly difficult to extend as a workflow grows more complex. **CHIA should be modular, and should provide plug-and-play support for many different co-design tools with various input/output formats.**

Additionally, control flow in a design loop is not always easy to define statically. In an agentic loop, this may manifest as explicit conditional logic guiding the flow. But, as the capabilities of agents improve, their own ability to guide flows can be leveraged to adapt a workflow on the fly. **CHIA must be able to express programmatic and agentic decision making, and as agents become more capable and potentially more trustworthy, the framework must be able to tune the agent’s decision making control.**

AI-driven design brings new opportunities, with radically reduced time from idea to design, but also new challenges. A growing body of work investigates the capabilities of various agentic loops to tackle hardware/software design problems. However, we need principled methods to evaluate these loops. Metrics should be built into agentic platforms for trustworthy and simple data collection. In addition to introspection to improve the loop itself, data collection is also invaluable for evaluation of the design generated by the flow. **CHIA needs data collection and profiling of both results and of the loop itself built in as first-class citizens.**

Together, these requirements enable us to perform agile and productive experimentation on agentic loops targeting co-design.

## 3 The CHIA Workflow Abstraction

A CHIA project is called a workflow, and is composed of a CHIA cluster and a CHIA loop. *Clusters* are physical machines and software environments where work is executed, and *loops* are programs which *orchestrate* a pipeline of hardware and software design tasks. In this section, we illustrate the CHIA design abstractions which constitute loops and clusters, with the help of a running example.

### 3.1 Running Example

Figure 2 shows a simple example of one way an architect may use CHIA. Specifically, this workflow uses an agent to write a new hardware module in RTL which achieves higher performance than an existing reference implementation. The steps of the workflow are as follows.

- (1) **Prepare inputs:** The architect starts with a reference implementation whose performance they want to beat without losing functionality. Next, they write a prompt with the specification of the hardware module, and prepare a suite of unit tests that can be used to verify the functionality of the module.
- (2) **Agentic implementation:** The loop takes the user’s prompt and passes it to an agent. The agent attempts to write hardware matching the specification.
- (3) **Simulation:** The loop then compiles a Verilator simulator from the implementation, which is used to run the user’s tests. To maximize parallelism, Verilator runs are split across three workers, two of which run on public cloud machines. Some longer benchmarks are run in FPGA-accelerated simulations using FireSim [48].
- (4) **Evaluation:** An evaluator node examines the results of the simulations and determines if the performance threshold is met. If it is not, then the loop restarts. Feedback is provided to the agent, and it continues iterating until the evaluator determines that a stop condition has been reached.

We will use this example in the next few sections to understand CHIA loops and clusters more concretely.

### 3.2 Clusters

Each step in a CHIA workflow may require different physical resources, such as different amounts of memory and different types of accelerators. Similarly, each step may require different software resources, like credentials, dependencies, and isolation, necessitating different environments. CHIA clusters provide the functionality to support these diverse requirements simply and efficiently.

**3.2.1 Physical Machines and Logical Workers.** A cluster starts with a collection of physical machines onto which users plan to distribute the tasks in their CHIA loop. In Figure 2, these machines are numbered CPU1 through CPU3, FPGA1, and AWS1 and AWS2. These machines provide heterogeneous physical resources, including different amounts of memory, different processors, and in this example, one machine has an FPGA while the others do not.

On top of these physical machines, users specify a set of *logical workers*. A logical worker’s definition includes a set of *resources* which are virtualized representations of the hardware and software environment which a logical worker provides. Nodes in the CHIA

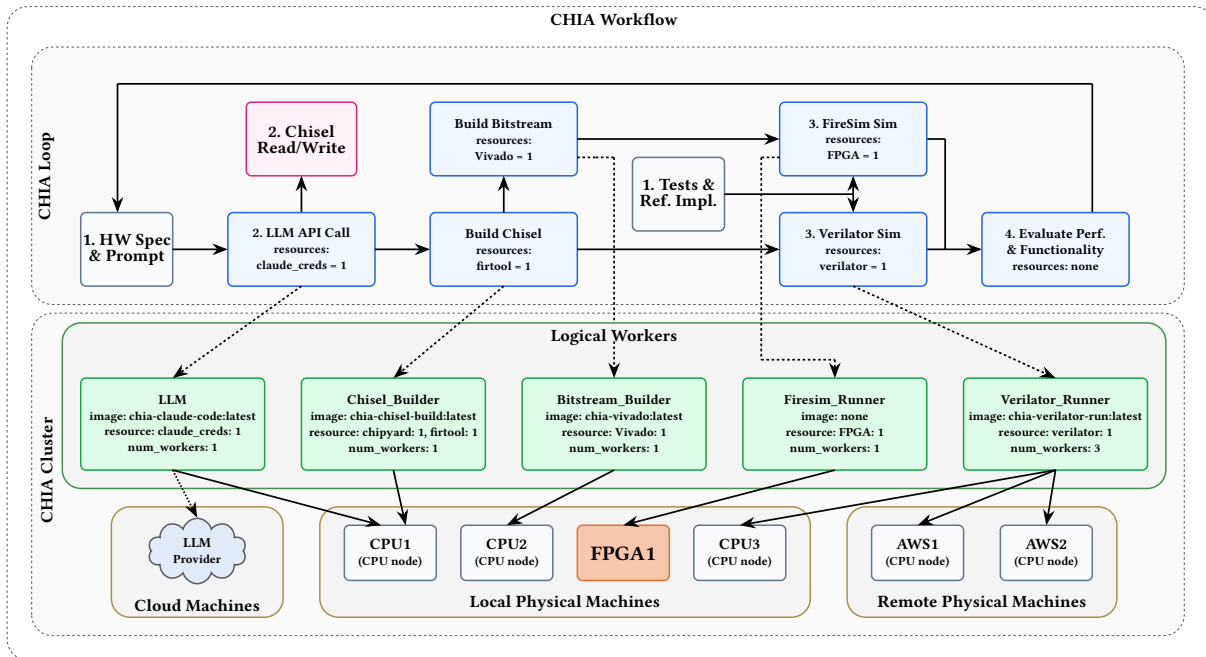


Figure 2: Simple example CHIA workflow for turning a specification into an RTL description of hardware. The CHIA loop takes a specification for a hardware block and in a loop uses an LLM to design the block, then programmatically evaluates its functionality and performance. Numbers correspond to enumerated steps in Section 3.1

loop specify the types and quantities of the resources that they need to run, and, using Ray [68], the CHIA runtime schedules these resource requests to logical workers in the CHIA cluster. This abstraction of resources allows users to abstract certain machine-specific information from the CHIA loop.

Concretely, the `Firesim_Runner` logical worker in our example workflow defines a resource called “FPGA”, which the `FireSim Sim` node requests. Similarly, the `LLM` logical worker defines a resource called “claude\_code”, virtualizing a Claude Code binary and credentials, which the `LLM API Call` node requests.

**3.2.2 Containerization.** CHIA supports and encourages placing logical workers inside of containers. Containerizing workers provides a number of benefits to a workflow. Most importantly for an AI agent-forward platform, containers provide isolation between different workers. An LLM finding and copying from the example workflow’s reference implementation of the hardware module would violate the integrity of an experiment. Even worse, an LLM operating in an insecure environment could access privileged information like proprietary PDKs. Assuming a non-adversarial, but potentially erroneous or unreliable LLM, containerization provides this isolation. We do not account for malicious LLMs in our threat model.

Additionally, containerization streamlines first-time cluster setup. Containers front-load and automate a large portion of the one-time setup cost, allowing clusters to be brought up much faster than they otherwise would. For instance, setting up a fresh Chipyard

environment for the `Chisel Builder` worker in our example workflow would normally take nearly an hour, but CHIA’s `Chipyard` container takes seconds to setup.

Containers also make clusters portable, so CHIA clusters can be deployed on a diverse set of physical machines. For instance, not all machines will have Vivado installed, but the example’s `Bitstream Builder` worker can build a bitstream in a container with Vivado on any machine with sufficient computing resources.

**3.2.3 Public Cloud Machines.** CHIA clusters can be split across on-premises/owned computing resources and remote/rented computing resources from the public cloud. This way users can leverage the compute resources they own, while still scaling their experiments. In our example loop, we leverage AWS to run Verilator on more tests than we otherwise could with just on-premises machines.

### 3.3 CHIA Loops

A CHIA loop is a Python program which orchestrates a sequence of tasks required to achieve some hardware or software design-flow related goal. We represent a CHIA loop as a graph, where nodes are the tasks that need to be performed and edges are the data and control flow between tasks. The loop for our running example is described by the graph in Figure 2.

**3.3.1 Nodes.** A CHIA node is a Python function that performs a step in a hardware/software design pipeline, often by invoking some underlying hardware or software design tool. Each node in our example flow executes an application and performs some postprocessing. For instance, the `bitstream build` node runs Vivado and collects build logs and results.

When a node is defined, it is tagged with the resources that it requires from the worker it executes on. Nodes are scheduled and dispatched when all of their inputs are ready and there is a logical worker with sufficient resources to satisfy the node’s request. Once the node can be scheduled, its arguments are serialized and sent to the worker where the node will execute.

Nodes can return control immediately to their callers with a future which can be used to collect their results later. This makes it easy to parallelize work and can be seen in our example workflow, where we can start building a FireSim bitstream at the same time as kicking off multiple Verilator simulations.

Nodes are the basic unit for retry, caching, bypass, subprocess tracking, and profiling, as described later in Section 4.3.

**3.3.2 Edges.** Edges represent data and/or control flow in a CHIA graph. Depending on whether the caller of a node is an AI agent or a line of code in the CHIA loop’s Python orchestration program, an edge can either be agentic or programmatic. Any data/control flow that is explicitly specified in the orchestration program of the loop is termed a *programmatic edge*, such as the edge between the Build Chisel and Verilator Sim nodes in our example. An agent using a tool call to run a node, instead of the node being explicitly initiated by the orchestration program, is called an *agentic edge*. In our example workflow, the edge between the LLM and Chisel implementation nodes is an agentic edge. CHIA provides first-class primitives for executing nodes from both programmatic and agentic edges, and users can define any node to be run both programmatically and agentially, or only one or the other. By supporting these two types of edges, CHIA generalizes across a broad spectrum of agentic design patterns that vary in the amount of control and decision-making entrusted to the AI agent.

## 4 Design

In this section we detail the design of the interfaces for creating CHIA clusters and loops, and the features of the CHIA runtime which runs a user-defined CHIA workflow.

### 4.1 Cluster Interface

Listing 1 shows part of a configuration file used in CHIA to specify the constituents of the cluster for our running example. Logical workers are defined, along with their exposed resources, container images, and the physical machines they can run on, under the `available_node_types` mapping. The set of physical machines in the cluster is the union of all IP addresses and hostnames specified for all of the logical workers’ `compatible_ips` keys. The cluster’s head machine is specified under the `provider` mapping. Any cloud instances are specified under a mapping corresponding to the cloud provider (e.g. `aws_nodes`).

Clusters are managed from a command line interface (CLI). For example, the cluster in Listing 1 is launched with the command `chia up Spec2RTL.yaml` which first spawns any public cloud node instances specified in the yaml, assigns workers to machines, creates containers, and then sets up the workers to listen for nodes. The cluster can be brought down with `chia down Spec2RTL.yaml`.

Clusters can also be expanded dynamically with the CHIA CLI by adding workers to the cluster configuration yaml, and using

```
# Spec2RTL.yaml

# AWS instances to spawn
aws_nodes:
  verilator_aws:
    InstanceType: c5.9xlarge
    count: 2
    ImageId: ami-0ec10929233384c7f

provider:
  head_ip: CPU1

# Authentication
auth:
  ssh_user: ${USER}
  ssh_private_key: ~/.ssh/key

# Logical Workers
available_node_types:
  Verilator_Runner:
    docker:
      image: "chia-verilator-run:latest"
      resources: {"verilator": 1}
    min_workers: 3
    max_workers: 3
    compatible_ips: [CPU3 \
                     @verilator_aws:0, \
                     @verilator_aws:1]
    worker_setup_commands: ["source ~/.bashrc"]
...

```

Listing 1: CHIA cluster yaml configuration fragment. Logical workers are specified under the “`available_node_types`” mapping, and physical machines are specified by the union of all logical workers’ “`compatible_ips`”.

`chia up Spec2RTL.yaml --add`, which brings up any workers specified in the configuration file which are not already running. This command can also be used without changing the configuration to attempt to restart dead workers.

Commands required for worker environment setup can be specified per worker in the `worker_setup_commands` key. Credentials for logging in to each of the machines are specified under the `auth` mapping, and can be overwritten per physical machine.

### 4.2 Loop Interface

Listing 2 shows a fragment of a Python file that defines parts of the CHIA loop for our example workflow. The `@ChiaFunction` is the core primitive in the CHIA loop. Functions annotated with `@ChiaFunction`, like `def verilate` in Listing 2 can be scheduled onto the cluster.

**4.2.1 Programmatic Nodes and Edges.** Nodes designed to be run on the cluster specify their resources in their `@ChiaFunction` annotation as seen in Listing 2: `@ChiaFunction(resources="...")`. A programmatic edge to a node can be created by calling a `@ChiaFunction` with `fn_name.chia_remote(args)`. A function marked as a `@ChiaFunction` can also be run directly like a normal Python function (without `chia_remote()`), and it will run on the same machine in the same process as its caller.

```

# Spec2RTL.py

# Define node for agentic control flow
@ChiaFunction
def read_chisel_src() -> str:
    ...

@ChiaFunction
def write_chisel_src(contents):
    ...

class rw_src_tool(ChiaTool):
    def setup(self):
        self.mcp.add_tool(read_chisel_src)
        self.mcp.add_tool(write_chisel_src)

# Prompt the LLM with tools
chisel_impl_Tool = rw_src_tool(
    task_options={"resources": {"firtool": 1.0}})

llm_future = claude_code_cli.prompt.chia_remote(
    prompt, tools=[chisel_impl_Tool])

...

# Define node for programmatic control flow
@ChiaFunction(resources={"verilator": 1.0})
def verilate(sim_binary, test_binaries)
    -> List(test_result):
    ...

# Schedule the node explicitly
test_results_future =
    (verilate.chia_remote(sim_bin, test_bins),
     firesim.chia_remote(bitstream, test_bins))

# Pass futures
evaluate.chia_remote(test_results_future)
# or collect them explicitly
test_results = get(test_results_future)

```

**Listing 2: CHIA loop fragment for our running example.**

Nodes called with `node.chia_remote(args)` return control immediately to their callers, enabling asynchronous computing. In our example this makes it easy to build a FireSim bitstream and run Verilator simulations in parallel. As shown at the bottom of Listing 2, these functions return futures which can be blocked on to collect the eventual result using `get()`, or can be passed along other explicit edges directly as arguments to other nodes.

**4.2.2 Agentic Nodes and Edges.** As discussed in Section 3, nodes can also be designed to be run by agents as MCP tools. These nodes must be registered with `ChiaTool` objects, in a special function which `ChiaTools` must override called `def setup(...)` as shown in Listing 2. Nodes are defined as functions and attached to a `ChiaTool` with `ChiaTool.mcp.add_tool()`. The `ChiaTool` object itself sets up an MCP server which hosts all registered tools. This server runs on a worker chosen according to the `task_options` argument in its constructor, including resource requirements, specified when initializing the `ChiaTool` object.

The `ChiaTool.mcp.add_tool()` function takes a bare Python function as input, and by default it executes on the worker that hosts the tool server. This is why we don’t need to specify resources for `read/write_chisel_src`, since we specify the resources they require when they run when we construct the `rw_src_tool`. To register a `@ChiaFunction` as a tool, and have it be scheduled remotely at runtime according to its own specified resources, a user would pass `ChiaTool.mcp.add_tool(fn.chia_remote_blocking)`. In this way, a tool server can be hosted on one worker and its tools can execute on another.

All LLM and agent nodes in CHIA expose a prompt method which accepts a list of `ChiaTools` which it provides to the agent. The docstring of the registered function becomes the description passed to the LLM. By providing tools for reading and writing Chisel, the LLM in our example loop can decide when in its reasoning it wants to read from and write to the Chisel code.

### 4.3 Runtime Features

CHIA’s runtime maps a CHIA loop onto a CHIA cluster and handles runtime features like fault tolerance and profiling transparently to the user. In this subsection we discuss some of the features of CHIA’s runtime.

**4.3.1 Mapping Workers to Physical Machines.** Logical workers are mapped onto to physical machines when the cluster is brought online. Multiple workers can share an underlying physical machine (like CPU1 in Figure 2), and a single type of worker can be instantiated onto many different physical machines (like our example’s Verilator Runner worker).

Worker definitions specify what machines they are capable of running on. For containerized workers, this is based on what hardware resources the physical machine provides. For bare workers (no container), this also implies that the physical machine has the necessary software resources. The CHIA runtime offers load balancing globally and on a worker by worker basis when allocating workers to compatible physical machines.

**4.3.2 Node Scheduling.** CHIA relies on Ray’s scalable distributed scheduling policy to handle the dynamic and sometimes non-deterministic nature of an agentic CHIA loop[68]. In particular, non-deterministic agents make it impossible to know what nodes will actually run in any given execution of a CHIA loop. At runtime, nodes are scheduled dynamically when all of their inputs are ready and a logical worker with sufficient resources is available.

**4.3.3 Fault Tolerance.** CHIA leverages Ray’s fault-tolerance capabilities in order to reschedule nodes that fail due to system failures [68]. When a logical worker or a physical machine goes down, that crash is detected and no more work is scheduled on that worker/machine. All tasks which were actively being run by that worker/machine are automatically queued to be rerun if there is another worker/machine with the requested set of resources for the task. Clusters can be expanded dynamically and reintegrate dead nodes with the CHIA CLI using ‘`chia up --add`’, which allows users to add resources to an already running cluster.

**4.3.4 Subprocess Tracking.** CHIA tracks all processes spawned by nodes. When a node or a loop is stopped/cancelled, CHIA stops

all spawned processes that were not spawned into a new process group, preventing resources from leaking. If the example workflow was stopped during a long-running bitstream build, this feature would prevent a stray Vivado process from consuming memory for hours after its results stopped being relevant.

**4.3.5 Profiling.** Data collection and loop analytics are built into CHIA through its profiling mechanism. All functions annotated with `@ChiaFunction` are automatically profiled, no matter how they are run, and the profiled results are automatically logged to disk. Results include start time, wall-clock time, worker identification, and additional user-specified information. The profiler also reconstructs the task graph by tracking object identity across different node executions.

**4.3.6 Caching and Bypassing.** CHIA supports bypassing nodes by injecting data into the CHIA loop to stand in for the node’s output. Bypassed nodes can still be scheduled according to their resource requirements, enabling easy testing of the CHIA loop without needing to actually run long-running nodes. CHIA also supports a complementary caching feature, where the results of user-specified nodes are persisted on disk. When bypass and caching are used together, CHIA loops can be restarted quickly after crashing by using the cached results to bypass actual node executions. Additionally, caching and bypass can be used to skip over non-deterministic nodes, such as the LLM API call in our example workflow, in order to reproduce an experiment’s results.

## 4.4 The CHIA Library

CHIA seamlessly pieces together a diverse set of platforms to provide feature-rich support for a wide range of workflows. As evidence of this, CHIA provides a library of reusable CHIA nodes and supporting containers. We will be continuously adding new features to the CHIA library and, in open-source spirit, welcome contributions from the broader community. The following list contains some of the nodes we already provide as part of the CHIA library:

- **LLM providers, servers, and agents:** Vertex AI/Gemini Enterprise Agent Platform [36], Antigravity [39], AWS Bedrock [11], Claude Code [9], Codex [73], Ollama [72], OpenCode [7], vLLM [51], FireworksAI [31], Groq [41], OpenRouter [75]
- **SoC Design:** Chipyard [6]
- **Hardware Compilation:** CIRCT [30], Scala FIRRTL Compiler [46, 54]
- **Simulators:** Verilator [94], FireSim [48], Gem5 [16], ChampSim [38]
- **Verification:** RISC-V torture [53], Spike co-simulation [86]
- **VLSI:** Hammer (and therefore support for various commercial CAD tools [56])
- **Databases and Cloud Storage:** Postgres [96], SQLite [32], AWS S3 [12]
- **Repository Management:** GitHub [37]

## 4.5 Implementation Stack

To provide this flexible and feature-rich environment, we build CHIA on top of a curated set of existing tools and libraries. The Ray

distributed computing platform [68] serves as the substrate underlying CHIA’s loop, cluster, and runtime implementation. Ray was chosen over other orchestration frameworks like LangGraph [52], Microsoft Agent Framework [67], and Apache Airflow [10] because Ray has expressive control flow semantics, fine-grained flexible scheduling, distributed execution support, and distributed fault tolerance support.

Other projects play significant roles in CHIA’s implementation. We leverage Docker for containerization of software resources [65]. CHIA MCP tools rely on PrefectHQ’s fastmcp [81], and our cloud integration relies on AWS’ Boto3 [82] and GCP’s Python Cloud Client Libraries [40]. Our profiling and visualization mechanisms are built on TensorBoard [2], Weights and Biases [102], and GraphViz [34].

## 5 Case Studies

We demonstrate CHIA’s capabilities through a series of case studies that showcase the flexibility and productivity advantages of the framework. These case studies cover a wide variety of workflows that are critical to research and development in computer architecture and systems. These include:

- (1) Generating a representative gem5 microarchitectural simulator from RISC-V BOOM RTL.
- (2) Implementing RISC-V ISA extensions in an OoO core
- (3) Automated critical path optimization in an OoO core
- (4) Agentic discovery of microarchitectural structures with evolutionary coding agents
- (5) Addressing GitHub issues in the CIRCT compiler in a maintainer friendly way

### 5.1 Automatically generating a representative gem5 microarchitectural simulator from RISC-V BOOM RTL

High-level microarchitectural simulators like gem5 [16] are widely-used tools for exploring architectural ideas. Unfortunately, there is no easy way to accurately map a full RTL description of a processor to a microarchitectural simulator. Large teams dedicate many engineer-hours to keeping simulators up-to-date and validating alignment with their microprocessor implementations [17, 20, 89, 97]. For small teams and academia, this maintenance is not feasible, and microarchitectural simulators either fall out of alignment, or do not exist for RTL implementations.

We developed a CHIA loop for deriving a gem5-based microarchitectural simulator from an RTL description of a processor, as described in Figure 3. Our pipeline gives an agent (Claude Code with Opus 4.6) access to the source of a gem5 processor core and a gem5 configuration file (where parameters and constants are specified for the core). In each iteration, the agent is asked to modify the architecture and microarchitecture of the gem5 core (not just parameter tuning) to align its performance more closely with a target processor’s RTL.

The agent’s updated gem5 implementation is executed, and its results are compared to RTL simulation results collected in Verilator. Results are saved into a database, and the parent for the next iteration is chosen based on the most aligned entries in the database. In each iteration, the agent is provided access to this database, including pipeline traces and performance counter results

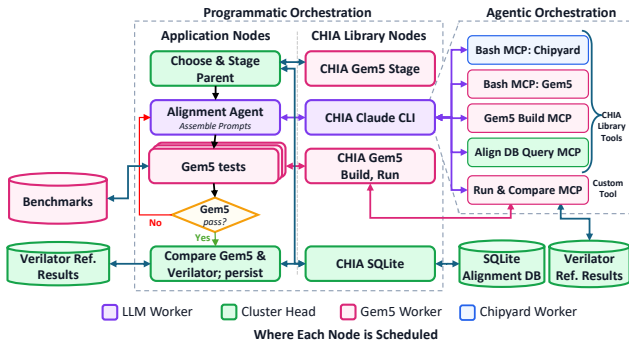


Figure 3: CHIA loop for automatically generating a representative gem5 microarchitectural simulator from RISC-V BOOM RTL.

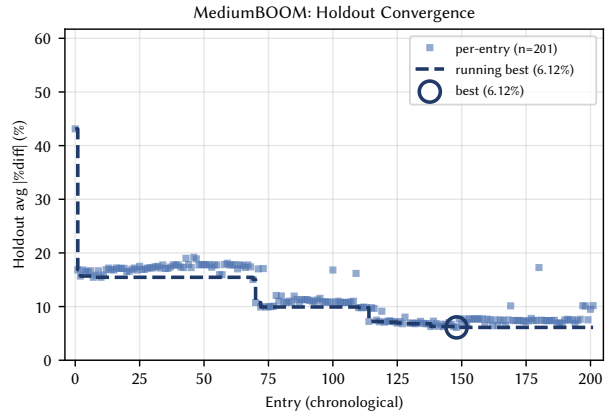


Figure 5: Average difference in gem5 and Verilator cycle counts for each iteration of the CHIA loop across withheld Embench benchmarks. Lower is more aligned. Alignment generally improves as the loop progresses.

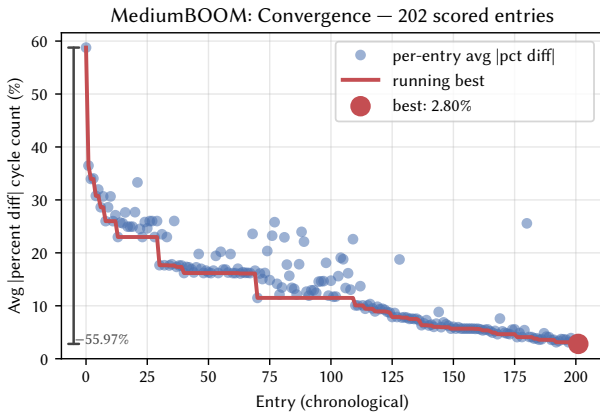


Figure 4: Average difference in gem5 and Verilator simulation cycle counts for each iteration of the CHIA loop across a suite of small benchmarks. Lower is more aligned. Alignment generally improves as the loop progresses.

from previous gem5 runs and RTL simulations. We include in our prompt instructions for how to analyze these pipeline traces and performance counters the way a human architect would.

As shown in Figure 4, over 202 iterations (about 10.5 days of wall-clock time) our CHIA loop modeled a Chipyard SoC [6] with a 2-wide MediumBOOM (Berkeley Out of Order Machine) core [22, 111] in gem5 to within 3% cycle count error on a training suite of 36 benchmarks taken from the Microbench suite [70], each designed to stress different parts of the core. We hypothesized that a large suite of small benchmarks requires a more accurate alignment to reach performance parity than a small suite of large benchmarks, since performance overestimates and underestimates cannot cancel each other out. In reading the LLM’s chain of thought, we find that its success arose from comparing pipeline traces and performance counters between the Verilator results and the gem5 results. It was able to use the diverging metrics to correctly identify misaligned structures and align them. The final aligned core has a diff of 1520 lines of code (excluding comments and blank lines) relative to the baseline gem5 RISC-V OoO core.

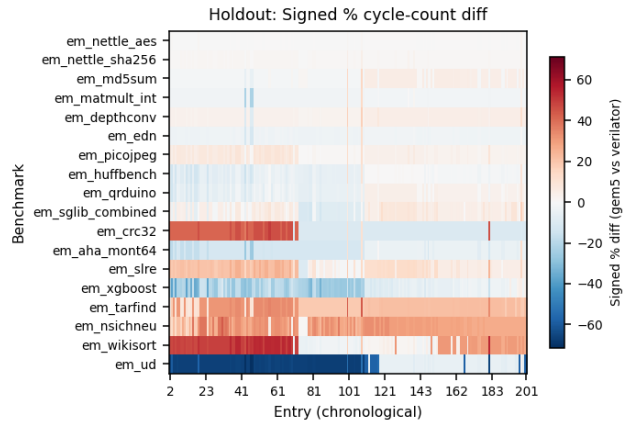


Figure 6: Per-benchmark and iteration results on the withheld suite. Lighter color is more aligned on a given benchmark in a given iteration. Alignment generally improves on most benchmarks for later iterations.

We used the Embench IOT 2.0 benchmark suite [79] as a withheld and hidden test set to evaluate the quality of our alignment—the agent in the loop does not know Embench is used, cannot see the Embench suite’s source, and cannot see its Embench results. In Figure 5, we show that average misalignment is less than 7% in our best iteration on the holdout benchmarks, though there are a few outlier benchmarks.

We attribute these outliers primarily to two effects. First, we see instances of overfitting to some of the training set of microbenchmarks. For example, one of the later iterations introduced a parameter `kSuppressThreshold` which suppresses a branch prediction penalty in the gem5 core after a certain number of times the penalty is applied. The agent explicitly stated that the parameter was tuned “with a threshold calibrated to [training benchmark]’s inner loop,”

Metric	All Iterations		Run & Compare tool		No Run & Compare tool	
	Avg	Total	Avg	Total	Avg	Total
Input Tokens	151	30.3K	267	22.4K	67	7.89K
Cache Creation Tokens	300K*	60M*	405K	34.0M	200K*	25M*
Cache Read Tokens	14M*	3B*	21.0M	1.76B	10M*	1B*
Output Tokens	119K	23.9M	189K	15.9M	68.4K	8.0M
Cost (\$)	11.72	2,355.98	19.84	1,666.48	5.89	689.50

**Table 1: Average and total LLM token usage and cost for medium BOOM alignment. Split by LLM access to the Run & Compare tool which allowed the agent to test its changes. Data with asterisks are estimates, as we did not collect this data until the 113th iteration of the loop. The vast majority of tokens input to the model are consumed as “Cache Creation Tokens” and “Cache Read Tokens”, and not as “Input Tokens”.**

to more accurately align that one benchmark. This is the same iteration which introduced a sharp degradation in alignment for the Embench Wikisort benchmark, as can be seen in the bottom right of Figure 6. The second effect is that our suite of training benchmarks does not capture all potential states of the core. For example, only one of our benchmarks exercises the core with a working set of code larger than the instruction cache. We plan to explore different benchmarking strategies to improve alignment on withheld benchmarks.<sup>3</sup>

On average, each iteration of our loop cost \$11.72 in Claude API credits, as seen in Table 1. We split reporting according to whether the agent could use the “Run & Compare MCP” tool as described below, as the token usage changes significantly based on whether that tool was used.

**The Power of CHIA:** We see the power of CHIA around entry 110 in Figure 4. At this point in the flow, we were not satisfied with the amount of iterations which were regressing overall alignment. To remedy this, we gave the agent the “Run & Compare MCP” tool, as shown in Figure 3, which gave it the ability to test its gem5 changes on its choice of our training benchmarks. This tool, which was implemented in a single hour, allowed the agent to empirically determine whether its changes were helping and adapt on the fly, which dramatically increased the number of iterations that improved alignment. Not visible in the final results, however, is how easy it was to try many different remedies to address the plateauing results in parallel. In the course of a single day, CHIA’s flexibility allowed us to try multiple different strategies for parent selection, prompts for the alignment agent with different selections of information, LLM effort levels, new MCP tools (the “gem5 Build MCP” was introduced shortly before the “Run & Compare MCP”), and more, before we introduced the “Run & Compare MCP”, which ended up being the most effective.

## 5.2 Automatically implementing RISC-V ISA extensions in an RTL OOO superscalar core

Proposing and evaluating microarchitectural improvements is a fundamental part of computer architecture research. Cross-stack

<sup>3</sup>We do not include the SPECint suite for this case study because it is difficult to execute the full SPECint suite in gem5 without SimPoints. In other case studies we are able to do so because we can easily run the full suite on 12 FPGAs in parallel in FireSim.

approaches often propose new ISA extensions backed by additional microarchitectural features. In either case, implementing these features in RTL and evaluating them on substantial benchmarks in FPGA simulation while collecting ASIC quality-of-result data is the gold standard for evaluation in architecture research. However, despite recent advances in agile hardware design [6], realizing this evaluation loop can be challenging for both small chip design teams and pathfinding teams in large companies.

To tackle this challenge, we develop a CHIA loop to implement ISA extensions in RTL for the 4-wide MegaBOOM core in the context of a complete Chipyard SoC. We target the RISC-V Bitmanip [83], Crypto [84], and Zicond [85] extensions.

First, we demonstrate that in each case, the loop produces a correct implementation, as validated by successfully running substantial benchmarks.

For Bitmanip and Zicond, this includes all 25.5 trillion instructions of the SPEC06 [1] reference suite, while demonstrating substantial speedups (5% and 3.5% respectively). For Crypto, we demonstrate 10x speedups on OpenSSL benchmarks (200 billion instructions). We also demonstrate that these implementations do not cause critical path regressions in either the SkyWater 130nm process [49] or a commercial 16nm process, and add only modest silicon area in both cases.

The CHIA loop takes the specification of a given extension and a formatted list of all of the new instructions and produces a functional and verified implementation with area and timing results. CHIA allows us to make the loop’s target highly customizable: a user could implement any extension specification on any core. To do so, the loop only needs to be provided with two pieces of verification collateral: a golden reference functional ISA-simulator implementation (e.g., Spike for RISC-V) for co-simulation and the set of instructions that the core can execute, including the ones to add. The latter collateral will be used by a random test generator to produce a set of random mixes of instructions for functional verification. This collateral is straightforward for the user to implement and provides the basis for verifying the generated RTL.

It is important to note that, within the CHIA loop, verification is as rigorous as in any other open source or academic flow. While the LLM is in charge of implementing and debugging the RTL, the user is still responsible for providing the golden references. CHIA lets us completely isolate the LLM from the verification infrastructure, preventing the agent from modifying the verification infrastructure in any way that would be considered cheating. In fact, the LLM knows nothing about this infrastructure, it only receives the necessary traces and information to debug the RTL.

The loop logic is divided into different stages as depicted in Figure 7.

**5.2.1 Stage 1.** The first stage consists of a loop with an LLM node in charge of producing the main RTL implementation of a given extension. The LLM uses the user specification to iterate on the RTL. A small set of generated tests that covers the entire repertoire of new instructions is run before moving to the next stage. In parallel, the loop initiates generation of random tests and synthesis of the initial RTL implementation. These random tests consist of long mixes of instructions generated by riscv-dv [26] and constitute the basis of Stage 3 verification.

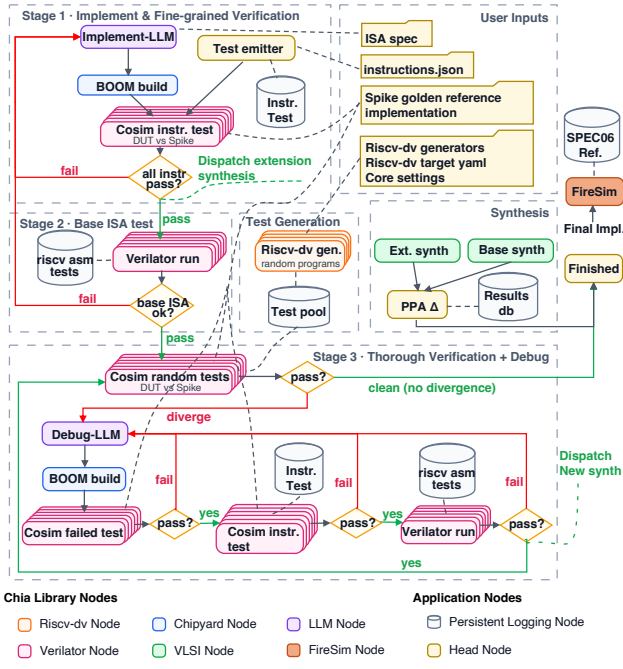


Figure 7: Architecture of the RISC-V core extension RTL implementation CHIA loop.

5.2.2 *Stage 2.* In this stage the loop checks for any major regression of the base ISA. It relies on the upstream RISC-V ISA tests [87] and in case of an error it goes back to Stage 1 with a trace of the failing test.

5.2.3 *Stage 3.* Only when all instructions execute correctly does the loop move to stage three. It relies on numerous random tests, which were generated in parallel to Stage 1 and 2, that stress the new instructions in combination with the existing ones. In case of divergence it goes back to Stage 1 with a different debugging prompt and the failing test to iterate on an RTL fix. The loop also launches ASIC synthesis of the RTL implementation in this stage.

With this CHIA loop we automatically implemented and evaluated three new RISC-V extensions in MegaBOOM [22, 111]; Bitmanip (Zba\_Zbb\_Zbc\_Zbs), Crypto (Zbkb\_Zbkc\_Zbkc\_Zknd\_Zkne\_Zknh) and Integer Conditional (Zicond). Figure 8 represents a summary of the loop evolution constructed using data collected with the CHIA profiler. The timelines show how different nodes were launched and executed while running the loop, and the pie charts on the right show the share of total machine-time per node. Note that some nodes such as simulation and generation run on many machines, while others like synthesis and the LLM run on just one.

Leveraging CHIA’s scalability, the loop runs on several heterogeneous machines and enables us to harness as much compute as we need from the cloud, in addition to our local cluster. Bitmanip and Crypto have been implemented on a shared cluster that contained 28 machines (a total of 1024 cores) distributed across both our machines and AWS. While the LLM implementation phase is lightweight, proper functional verification requires significant compute power to run all the spike co-simulation nodes. In fact, as

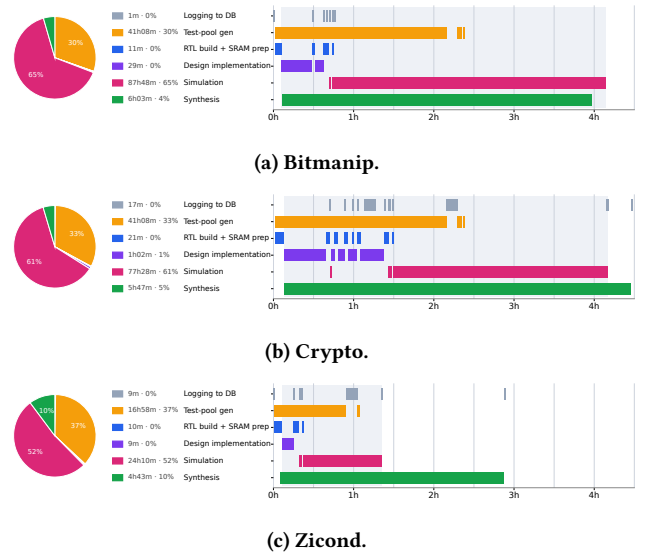


Figure 8: Wall-clock execution profile of the agentic RISC-V extension-generation loop. Each panel pairs a Gantt chart of pipeline stages (right) with the share of machine-time per stage (left). Nodes with a total machine time below one minute are not shown (Stage 1’s test emitter and PPA collection).

Benchmark	Bitmanip (GCC 13.2)			Zicond (GCC 16.1)		
	Baseline (s)	Bitmanip (s)	Speedup	Baseline (s)	Zicond (s)	Speedup
400.perlbench	1501.02	1507.84	0.995	1542.59	1461.49	1.056
401.bzip2	1593.04	1528.20	1.042	1579.95	1570.06	1.006
403.gcc	1237.20	1209.68	1.023	1207.76	1216.58	0.993
429.mcf	997.88	997.20	1.001	974.47	974.44	1.000
445.gobmk	1317.51	1291.24	1.020	1321.21	1316.23	1.004
456.hammer	1972.58	1190.03	1.658	1924.84	1359.93	1.415
458.sjeng	1737.12	1662.04	1.045	1711.17	1702.92	1.005
462.libquantum	2197.98	2169.19	1.013	2234.69	2234.48	1.000
464.h264ref	2577.43	2557.88	1.008	2509.29	2537.91	0.989
471.omnetpp	830.17	819.39	1.013	815.68	801.21	1.018
473.astar	1029.62	1005.49	1.024	1041.58	1045.25	0.997
483.xalancbmk	798.15	823.26	0.969	764.04	767.93	0.995
geomean			1.056			1.035

Table 2: SPEC06 ref. (25+ trillion instructions) runtime comparison between the baseline core and the Bitmanip-enabled core, as well as with the Zicond-enabled core.

we can see in Figure 8, co-simulation represents between 50% and 66% of total machine usage and test generation between a quarter and a third. Zicond was implemented on a smaller cluster of 12 machines and 256 cores, yet its implementation and simulation run considerably faster given that it is the simplest of the three extensions.

The loop synthesizes both the MegaBOOM [111] baseline and the LLM implementations on top of the baseline using the same CHIA synthesis nodes as the loop in Section 5.3. The area overhead when synthesizing in SkyWater 130nm [49] is around 2% for Bitmanip, less than 5% for Crypto, and is negligible for Zicond. When using a commercial 16nm PDK, the area overhead for Bitmanip is 3%, for Crypto is 6%, and for Zicond is 0.8%. We see no regressions in timing with either PDK.

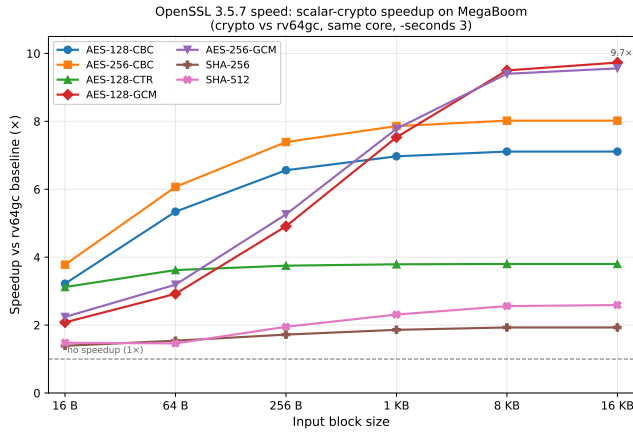


Figure 9: OpenSSL speedup when executed with the Crypto extension automatically implemented in BOOM using CHIA.

Case	LLM Iters	Input Tokens	Cache Creation.	Cache Read.	Output	Cost (\$)
Bitmanip	2	136	956,922	22,448,288	110,887	23.57
Crypto	6	205	2,044,094	37,802,171	194,255	46.24
Zicond	1	80	167,106	5,329,628	30,774	5.11

Table 3: LLM (Claude Opus 4.7) usage and cost of the CHIA loop implementing Bitmanip, Crypto, and Zicond.

We have evaluated these extensions on different benchmarks, all of them booting Linux and running in FireSim[48].

- (1) Bitmanip: We evaluate Bitmanip’s implementation on the entire SPEC06 reference suite [1], executing over 25 trillion instructions. Compiling SPEC with gcc 13.2 we obtained an overall speedup of 5.6% over the baseline, as seen in Table 2.
- (2) Crypto: Crypto is evaluated using OpenSSL Speed [76], a benchmark designed to test the performance of different cryptographic algorithms’ performance. The metric used in these benchmarks is throughput—given an input size, it measures the amount of times it can execute a given algorithm. Figure 9 shows the speedup over baseline for different input sizes (x-axis) for different OpenSSL algorithms. The reason for us not to use SPEC06 reference [1] is that it contains no Crypto instructions.
- (3) Zicond is evaluated against SPEC06 ref.[1] too, compiled with gcc 16.1 provided that previous versions do not emit Zicond instructions. Table 2 shows the per-benchmark breakdown improvement and an overall geomean speedup of 3.5%.

Finally, we have also collected the cost in dollars and tokens of running the loop for the three implemented specifications. The results can be found in the Table 3.

**The Power of CHIA:** CHIA enables us to leverage LLMs for specific tasks (processing the specification, making an implementation or debugging) without losing any control in phases where we cannot tolerate the LLM making mistakes (performing verification and performance evaluation, or collecting synthesis results). CHIA also allows each stage of the workflow to be designed using the most

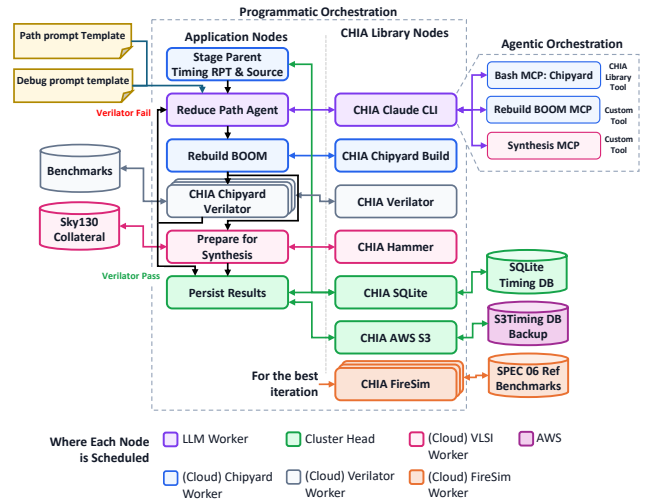


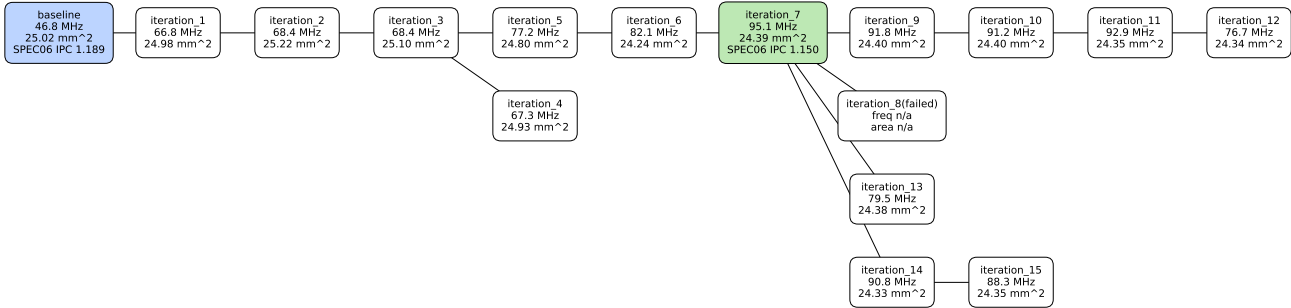
Figure 10: Architecture of the BOOM critical path optimization CHIA loop. In each iteration, an LLM is asked to reduce the critical path latency of the BOOM core based on timing reports from running gate level Synthesis with the Sky130 PDK [49].

suited approach. An example of this can be found in our 3-stage division (see Figure 7) where each stage has been constructed with different goals and principles. Still, they all integrate seamlessly in a single functional workflow. Another aspect of CHIA that has made this loop feasible is its ability to harness compute resources from across different physical clusters. In this specific case we have used a mix of on-premises and cloud machines. Verification is computationally intensive, as seen in Figure 8, so we have divided it across clusters, while other tasks depend on specific licenses that only exist on-premises. CHIA handles the complexity of making these clusters appear as a single pool of logical resources that the user can employ to run the nodes of any loop.

### 5.3 IPC-aware automated critical path optimization in an out-of-order superscalar RISC-V core

A processor’s maximum operating frequency, and thus its maximum performance, is most directly limited by the longest latency (critical) path which must be traversed by a signal within a single clock cycle. For this reason, optimizing a processor’s RTL to minimize the length of this critical path is an important engineering problem. However, breaking down a single long chain of logic without affecting the functionality of the processor can take days of focused human effort, requiring a complex understanding of the signals involved and how they interact with the processor at large.

We developed a CHIA loop to take on this task for the 4-wide MegaBOOM core [22, 111] in Chipyard [6]. Our loop is described in Figure 10. The loop takes a timing report from running a parent iteration’s final BOOM core through a commercial gate-level synthesis tool on the open-source Skywater 130nm PDK [49], and provides it to the Claude Code CHIA library node using Opus 4.6 as the underlying model. For the first four iterations we asked the



**Figure 11: Timing optimization tree of results, showing maximum achievable frequency and area of each iteration when synthesized in Sky130 [49]. Our best iteration yielded a 2.03x improvement in frequency, with an IPC loss of only 3% compared to the baseline, for a net Iron-Law speedup of 1.97x. Silicon area changes by a negligible amount.**

	Input Tokens	Cache Creation	Cache Read	Output	Cost (\$)
Average	106	177K	4.69M	53.4K	14.43
Total (n=14)	1,478	2.48M	65.7M	747KM	202.03

**Table 4: Average and total LLM token usage and cost for our MegaBOOM timing optimization loop. We achieved an iron-law speedup near 2x for \$202 in API costs. Failed iteration 8 is excluded, though its costs were on the same order of magnitude as the other iterations. The vast majority of tokens input to the model are consumed as “Cache Creation Tokens” and “Cache Read Tokens”, and not as “Input Tokens”**

agent to reduce critical path with no impact on IPC, but after results plateaued, we changed the focus to iron-law performance (asking the agent to reduce the critical path with as small an IPC reduction as possible). This meant writing a new prompt and turning on IPC collection in the CHIA Verilator nodes, only a few lines of code in CHIA. The LLM in our loop is provided a bash shell in the Chipyard codebase for exploring and editing BOOM, a tool to confirm that BOOM rebuilds correctly, and a tool to run synthesis on small modules within BOOM as a way to test its ideas. Once the LLM is satisfied with its changes, the loop (without AI involved) rebuilds BOOM, and runs it through a large suite of benchmarks in Verilator to verify functionality is unchanged, and to estimate impact on IPC. In parallel, we synthesize the modified BOOM core for area and timing estimates. At the end of each iteration, we persist our results in a SQLite database.

We ran our loop until it reached a frequency plateau (less than 5 active days, at less than 8 hours per iteration on average), and during that time it more than doubled the frequency at which we could run the MegaBOOM core in Sky130 [49], from around 47MHz to around 95MHz in our best iteration (iteration 7, see Figure 11).

For our final measurement of how much IPC was lost, we compare the IPC of iteration 7 with our baseline on all 25 trillion instructions of the SPECint 2006 reference benchmark suite using FireSim [48] (not shown in Figure 10).

The IPC reduction on the SPEC06 ref. suite between the baseline and iteration 7 was 3.28%, which is very similar to the IPC reduction we see while running the flow with smaller benchmarks in Verilator (3.20%). **This yields a net Iron-Law speedup of 1.97x, nearly a doubling in overall performance.** Additionally, we use the SPEC06 runs (over 25 trillion instructions) as validation that the optimizations implemented by the agent did not degrade the functionality of the core. Table 4 shows the token usage and cost for running this loop per iteration and in total.

After iteration 7, we see that results generally were flat or regressed. Analyzing the synthesis reports, we find that this plateauing occurs because at this frequency the core is somewhat balanced, with many distinct paths with the same latency, with endpoints spread across the frontend, the load-store unit, and the data cache. On the other hand, we find that our two large regressions share a common critical path, which is unrelated to the agent’s changes, and likely emerged because certain synthesis heuristics, for example SRAM macro placement, gave significantly different outputs due to small changes in the inputs.

Invisibly to the agent, we also ran each of our designs through synthesis in a commercial 16nm process. The first of the agent’s timing optimizations (restructuring linear-depth structures in the issue queues to log-depth) improved the critical path in the commercial process by about 70%. However, that was the only improvement in the commercial process from our loop’s changes. Put another way, our Sky130 results do not generalize to the advanced commercial node beyond the first optimization. As the 16nm process is proprietary, we could not use it in this case study, since information from the PDK cannot be given to public frontier models. In section 7, we will discuss this problem in more detail, and its implications for assessing QoR in AI-guided hardware design.

**The Power of CHIA:** This case study heavily leverages the reusability of CHIA’s library nodes, and the ability to carve out pieces of one workflow and incorporate them into another. Many of the nodes used in this case study, including building BOOM’s Chisel, simulation in Verilator and FireSim, and gate-level Synthesis, rely on CHIA library nodes. Furthermore, this case study was able to share many of its application nodes with the previous case study

(implementing new RISC-V ISA extensions in the BOOM core) because the two are operating on the same core, meaning much of the pre- and post-processing required for each of the library nodes is the same, despite the goals of the two case studies being very different.

This case study was also able to heavily leverage CHIA’s support for public cloud services. In particular, we used cloud machines to run Chipyard Build, Verilator, Hammer/Synthesis, and FireSim, and we also used CHIA’s AWS S3 [12] support to keep our database backed up in the cloud.

## 5.4 Agentic Architectural Discovery with Evolutionary Coding Agents

AI techniques such as evolutionary coding agents have shown significant promise in the discovery of novel ideas and approaches across scientific domains including the computing stack [25, 35, 69].

Within computer architecture, ArchAgent [8, 43] combined AlphaEvolve with a highly distributed simulation cluster to discover novel cache replacement policies. These generated policies outperformed prior human-designed state-of-the-art policies in a competition setting while far outpacing their estimated development times.

*5.4.1 Challenges in Using Agentic Discovery Flows.* While we have observed success in applying agentic discovery flows to problems in computer architecture and computer systems, significant challenges still remain. These challenges can be divided into three broad categories:

- (1) **Low Fidelity and Low Performance Evaluations:** AI agents are prone to reward hacking. In an early experiment in ArchAgent, AlphaEvolve was observed to reward hack an improperly enforced assertion to drop memory traffic and claim performance improvements.

More broadly, evolutionary coding agents are sensitive to evaluation fidelity - the quality and accuracy of feedback significantly affects the utility of generated designs. For instance, microarchitectural simulators such as ChampSim are unable to provide reliable feedback on power, area, and critical paths. This brings into question the hardware realizability of designs generated with feedback from such models. Substantial human effort is required to reason about and verify large volumes of such AI-generated designs. This defeats the purpose of deploying an automated flow in the first place.

For any given time frame, an evolutionary coding agent’s ability to explore a design space is fundamentally limited by the latency of evaluating proposed candidates. Thus, feedback from fast and high-fidelity simulators is essential for microarchitecture discovery. However, ChampSim required days to simulate running a single billion-instruction length SimPoint of a SPEC06 workload on a quad-core processor [43]. One could easily compile a larger design onto an FPGA-accelerated RTL simulator such as FireSim (as shown in our other case studies) and execute much longer workloads in lesser time while simultaneously gaining much higher fidelity feedback.

- (2) **Unergonomic Flow Development:** Researchers report that designing and delivering a successful discovery flow is extremely challenging, especially in the presence of long-running evaluations (often observed in hardware/software co-design research) that can cause errors to materialize at timescales of hours to days. Machine errors and downtime also become an increasingly relevant concern, and flow interruptions can discard hours to days of valuable intermediate results. Similarly, it can take hours to days to observe the effects of a single change. Experimenting with a sequence of such individual changes on flows with fixed configuration parameters or heuristics (such as those for population sampling or commit) costs extensive amounts of time and effort.
- (3) **Limited Flow Exploration:** Relatedly, existing microarchitecture discovery flows as seen in [19, 43] offer point solutions in the design space of agentic approaches to microarchitecture discovery. The broader composition of agents for problems in computer architecture (as shown in our other case studies) remains underexplored in the context of evolutionary discovery. For instance, current evolutionary discovery flows have an evolutionary coding agent as the topmost flow node. However, this is not necessary and an evolutionary coding agent could instead be more effective as a piece in a larger agentic flow. We believe such hypotheses remain untested in part due to the churn and tedium experienced in developing long-running discovery flows.

Rather than showing a specific result in this section, we demonstrate how existing flows like ArchAgent can be expressed in CHIA, how CHIA enables addressing many of the challenges faced in conventional evolutionary coding agent flows, and how CHIA facilitates further research towards groundbreaking discovery flows.

*5.4.2 “Classic” evolutionary discovery flows in CHIA.* First, we look at the design of an illustrative CHIA loop implementing an evolutionary flow similar to the one seen in ArchAgent. In order to do this, we provide ChampSim nodes and an AlphaEvolve node in CHIA, organizing them as shown in Figure 12.

The user selects configurations for the ChampSim nodes, configuring the simulated SoC, and the AlphaEvolve node, configuring various evolutionary search parameters. The user also provides a few lightweight Python functions. The function `build_fn` is used to define how binaries are built on a ChampSim Build node and built binaries are passed further into evaluation. In the case of evolution feedback, for example, ChampSim Run nodes could be set up to execute a selection of workload traces from a workloads database executing for a short number of warmup and detailed simulation instructions. This configuration is defined in the function `run_fn`. Metrics such as instructions-per-cycle, misses-per-kilo-instruction, prefetcher coverage/accuracy can be aggregated and provided as feedback using the function `run_mapper_fn`.

*5.4.3 Advanced high-fidelity evolutionary discovery flows in CHIA.* CHIA facilitates designing much more powerful agentic discovery flows by providing extensible interfaces and a rich library of tool nodes that enable incorporating comprehensive and actionable feedback into the evolutionary search:

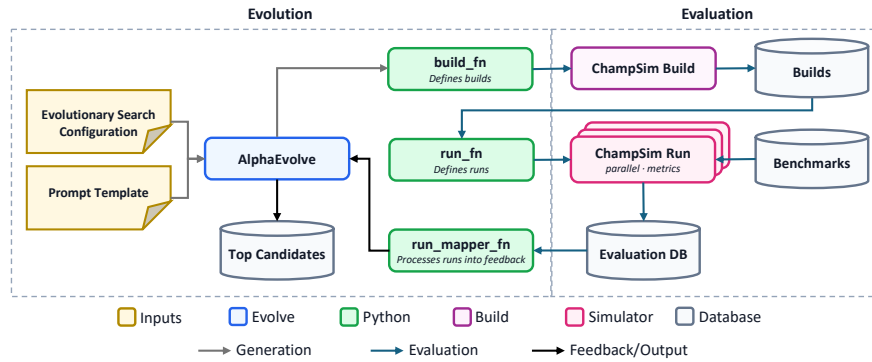


Figure 12: Architecture of an illustrative agentic discovery CHIA loop mirroring the simpler flows shown in prior work.

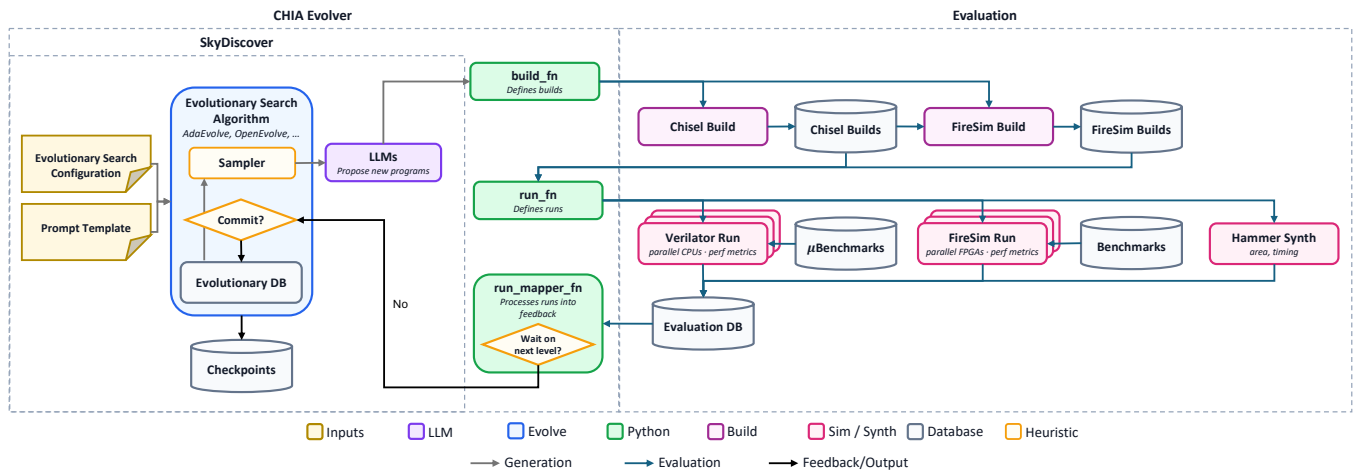


Figure 13: Architecture of an RTL discovery CHIA loop featuring the ability to choose evolutionary coding agents along with a cascade of evaluators including Verilator, FireSim, and commercial synthesis tools via Hammer. In practice, we would expect to add CHIA LLM nodes throughout the flow to act upon the comprehensive data generated by the cascade of evaluators.

- **Evaluator Nodes:** CHIA enables the composition of diverse evaluation backends such as functional models (Spike), microarchitectural simulators (gem5, ChampSim), RTL simulators (Verilator) and emulators (FireSim), and physical design tools (via Hammer). These tools can be used to provide high-fidelity performance (frequency and cycle-by-cycle), power, and area metrics to evolutionary coding agents.
- **Evolver Node:** SkyDiscover [58] is a flexible framework for AI-driven scientific and algorithmic discovery with support for widely-used large language models, the Claude code coding agents, and a variety of evolutionary coding agents such as AdaEvolve [23], EvoX [57], GEPA [4], and OpenEvolve [92]. CHIA ships with an Evolver node providing flexible interfaces around SkyDiscover and which allow it to interact with the rest of the framework. We provide support for AlphaEvolve [69] via the AlphaEvolve Early Access Program API [101] as an external backend to SkyDiscover inside the CHIA Evolver node (referred to as AlphaEvolve node above).

Figure 13 shows an example of a more data-rich and flexible RTL discovery flow that lets the user pick between various evolutionary coding agents and features a cascade of evaluators providing Verilator, FireSim, and Hammer ASIC Quality-of-Results data for feedback, all productively composed with CHIA.

Traditionally, building and debugging a flow like this would be much more difficult than constructing one using a single kind of evaluator that executes purely in software (such as a microarchitectural simulator). CHIA not only provides a library of battle-tested tools, it also makes much-needed improvements to the development flow. In an early experiment with AdaEvolve, we made a typographical error in configuring its guide model. This error disabled paradigm generation, one of AdaEvolve’s most valuable features that counters flow stagnation. Paradigm generation errors were first observed 12 iterations and 4 hours into the experiment and significantly slowed down its rate of progress. In the absence of CHIA, our options were to restart from scratch or wait for the next checkpoint at 25 iterations. In such a situation, CHIA’s cache and bypass feature allows us to pause the flow, fix the error, fast-forward

to the previous error point using cached CHIA function outputs, and resume normal execution. CHIA can also flexibly manage its clusters such that we would not have to bring down or restart our entire cluster to correct this error.

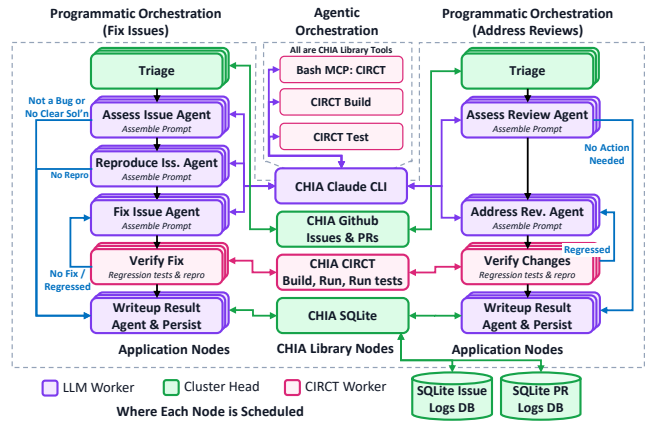
Similarly, CHIA allows further composition of evolutionary flows into broader agentic flows (similar to the ones described in previous case studies), targeting discovery across the computing stack with the ability to span multiple levels of representation such as evolving software programs, microarchitecture models, RTL implementations, and circuit designs. In combination with improvements to flow development ergonomics, we believe CHIA enables researchers to push the frontier of agentic discovery flow design patterns and explore open questions in the design and use of such flows for hardware-software co-design.

**The Power of CHIA:** This case study showcases CHIA’s ability to compose powerful discovery flows from reusable parts: its extensible node library lets us assemble cascades of high-fidelity evaluators—from functional models and microarchitectural simulators to RTL simulators, emulators, and physical design tools—and freely swap between evolutionary coding agents, turning what were once bespoke, monolithic flows into building blocks for discovery across the computing stack. Just as importantly, CHIA relieves the churn and tedium of developing long-running discovery flows, with features like cache-and-bypass and flexible cluster management that help researchers recover from the errors and interruptions that inevitably surface over hours- to days-long evaluations. Together, these capabilities let researchers move past today’s point solutions and explore the broader, largely untested design space of agentic discovery flows.

### 5.5 Automatically addressing GitHub issues in the CIRCT compiler in a project maintainer friendly way

AI tools have enabled an increase in outside contributions to large open source GitHub repositories, including new issue reports and pull requests. As noted in the LLVM project’s AI tool use policy, when these contributions are done entirely by AI in an unprincipled way and without human review, this “extracts work from [maintainers] in the form of design and code review” [60]. While CHIA cannot force a human contributor to review the work of their agent, it can be used to force the agent to develop contributions in a more principled way. To demonstrate this, we built a CHIA loop, as seen in Figure 14, to solve issues for LLVM’s CIRCT project, which compiles high-level hardware description languages into SystemVerilog and RTL simulators. Notably, CIRCT is used in Chipyard for lowering its Chisel-based generators down to System Verilog. To reduce maintainer burden rather than increase it, we focus our efforts only on issues that demonstrate bugs, and where the conversation surrounding the issue has left no ambiguity about the correct resolution for the bug.

The loop works as follows. First, a GitHub issue is triaged heuristically, filtering out issues with no code blocks, and those with an open PR referencing the issue (which implies a solution to the issue has already been proposed). Additionally, since we limit our scope to bugs and not feature requests, we avoid any issues labeled as feature requests. An issue is then assessed by an LLM to determine



**Figure 14: Architecture of the CIRCT GitHub issue fixing and PR response CHIA loops.** Each loop uses a sequence of prompts to an agent to assess an issue or comment and take the correct action in a principled way.

whether (1) the issue is actually a bug and (2) there is a single clear solution to the issue. We do this to make sure that any changes align with our narrow scope of bugs with a single, clear resolution. For issues where this is the case, an agent then tries to reproduce the bug (outputting a bash file which, when run, reproduces the bug) and if successful, another agent tries to fix the bug. These agents are given tools to modify and build CIRCT source, and run tests using CIRCT binaries to validate their changes. After this, we take control back from the agents, and without any AI involvement, the loop verifies the fix by running CIRCT’s full suite of regression tests, and the reproduction bash file, to ensure the bug is fixed.

While we do have the LLM write up a PR for the issue, to minimize extractive work and to comply with LLVM’s AI policy, we had a human review each change in detail to ensure quality and hand-write pull requests to submit to the upstream CIRCT repository. It took a graduate student with novice-level familiarity with CIRCT an hour per issue to review the agents’ changes and ensure they were high enough quality to submit pull requests. It took about 10 more minutes per issue to prepare the pull requests. Additionally, in compliance with CIRCT’s AI usage policy, we did not submit pull requests for any issues marked as “good first issues”. We also built a loop for addressing reviewer comments, which follows a similar pattern, first assessing comments for action items, and taking action where necessary, before validating in an environment with no agents. We used this loop to privately address comments we expected to receive, and to address actual comments from maintainers.

To minimize extractive work, we discussed closely with maintainers of the project to ensure our contributions were of a reasonable quantity and quality. In the end, we ran the loop on 16 randomly selected GitHub issues, and its results are outlined in Table 5. Of the 16 issues the loop examined, 11 were determined to be bugs, four of which did not have unambiguous solutions, leaving seven issues entering the reproduction phase. In this stage, two of the seven could not be reproduced, indicating that they had already been fixed in upstream CIRCT. Our loop successfully fixed the remaining

Issue No.	Assessed As	Reproducible	Fixed	Pull Request Submitted
#2266	Not a bug	-	-	-
#2669	Not a bug	-	-	-
#4354	Bug	Yes	Yes	No (Good First Issue)
#4396	Not a bug	-	-	-
#4649	Fix unclear	-	-	-
#5626	Fix unclear	-	-	-
#5789	Bug	Already fixed	-	-
#6226	Bug	Already fixed	-	-
#6740	Bug	Yes	Yes	No (Good First Issue)
#7127	Not a bug	-	-	-
#7388	Bug	Yes	Yes	Yes (Merged)
#7531	Fix unclear	-	-	-
#7949	Bug	Yes	Yes	Yes (Merged)
#8508	Fix unclear	-	-	-
#10104	Bug	Yes	Yes	Yes (Merged)
#10571	Not a bug	-	-	-

**Table 5: Characterization of the 16 randomly selected CIRCT GitHub issues at which we targeted our loop. Our workflow correctly categorized nine issues as not bugs or bugs with unclear solutions and two issues as already fixed, and successfully fixed the remaining five issues.**

Metric	Assess	Reproduce	Fix	Writeup
Input Tokens	7	27	68	2
Cache Creation Tokens	10.5K	21.5K	74.6K	9.4K
Cache Read Tokens	95.8K	648K	3.58M	8.1K
Output Tokens	3.2K	8.8K	30.0K	1.2K
Cost (\$)	0.40	0.68	3.01	0.09

**Table 6: Average LLM token usage and cost for each agent stage in our issue fixing loop. The vast majority of tokens input to the model are consumed as “Cache Creation Tokens” and “Cache Read Tokens”, and not as “Input Tokens”.**

five bugs, and we submitted pull requests for the three of them that were not labeled as good first issues. Table 6 shows the token consumption and cost of each phase of our loop for solving CIRCT issues. Our loop considered all 16 issues in parallel, leveraging CHIA’s scalability, and finished in less than 45 minutes.

Each of these three PRs received feedback from reviewers, and we used our loop for addressing reviewer comments to handle their requested changes. All three of our PRs have been merged into the main branch of the CIRCT repository.

We agree with the loop’s conclusions for each of the 16 issues. However, as we were prototyping the loop, we came across a situation where our assessor agent determined that issue #8508 had a clear solution, and our fixer agent successfully fixed the issue, but the assessor and the fixer disagreed about the correct solution. To remedy this, in our final loop, the assessor was required to enumerate all good solutions for a given issue and, if more than one could be described, declare that issue did not have a clear and unambiguous solution. After making this change, the assessor agent reliably declared issue #8508 had no clear solution.

**The Power of CHIA:** We see the power of CHIA in a few major ways for this workflow. Firstly, we were able to prototype our flow at a very small scale, with only a single agent and CIRCT instances. Then, only changing 3 lines of code across the cluster configuration and Python parameters, we were instantly able to run as many issues in parallel as we wanted. We also see, in this case study, CHIA’s ability to balance agentic orchestration with programmatic orchestration. In each agentic step, the agent is given runway to orchestrate complicated sequences of tasks in order to make decisions about the issue it is fixing. But, to ensure fixes are principled, programmatic orchestration is used to connect the narrowly-scoped agents together, and more importantly, to verify that the changes work as well as the agents claim.

## 6 Related Work

Applying artificial intelligence in the context of hardware design is a booming research area. In particular, many papers have explored using agents to generate RTL [29, 33, 59, 98, 99, 103, 105, 109, 112, 113]. Strategies for this vary greatly, from e-graph rewriting [109] to multi-level summarization [112]. Each of these works uses a bespoke design loop, making their work difficult to compare and impossible to interoperate. Performing these experiments in CHIA would be easier, allow for fairer comparison, and streamline merging successful ideas together.

Discovery of novel architectural ideas is also a subject of growing interest in the architecture community [8, 18, 43, 91]. Some recent works used evolutionary algorithms to discover novel microarchitectural techniques to optimize performance in out-of-order superscalar cores [8, 18, 43]. Similarly, one work explored GPU architecture design space exploration by composing DSE rules with LLMs [110]. One work sought to develop a unified sandbox for connecting search algorithms to architectural simulators, in order to streamline ML-aided design space exploration [50]. Another work uses an LLM to guide multi-objective black-box optimization towards the Pareto-frontier, significantly reducing time wasted exploring infeasible points [78].

Multiple papers have demonstrated the strong capabilities of LLMs to generate high-performance kernels for GPUs and accelerators [45, 106]. Likewise, LLMs have been used to provide humans with better understandings of kernel [28] and CPU cache performance [66]. LLM-based oracles like this can easily be built in CHIA.

Building off of the success of benchmark suites evaluating the ability of LLMs to generate software [47, 77], and noticing the proliferation of work targeting LLMs for hardware design, many hardware focused benchmark suites have been developed. Some of these suites emphasize evaluating agentic co-design [5, 100], RTL generation [42, 62, 80, 108] and high-level synthesis [3]. Building these benchmark suites on top of CHIA would streamline the process of comparing different design loops, and we are interested in integrating these suites into CHIA.

Recent work has focused on LLM guided usage of EDA tools [24, 44, 55, 104]. These ideas range from MCP EDA tool servers [104] to standardized execution frameworks for EDA [24]. We also hope to encapsulate tools like this into CHIA.

A growing body of work targets general agentic frameworks. CHIA is built on top of Ray [68], and adds customizable profiling as

well as fault tolerance and testing features through the bypassing and caching capabilities. CHIA also adds easy setup for heterogeneous clusters that have multiple different workers per machine as well as clusters split across on-premises and cloud compute. Other distributed agent frameworks lack expressive control flow semantics that encompass cyclic workflows [10], distributed fault tolerance support [67], or resource-aware scheduling [67]. Agents and agent platforms designed for a single machine generally lack support for distributed execution [7, 9, 39, 52, 74].

The CHIA framework is the first of its kind. As we note, CHIA synergizes well with many of these new tools like benchmark suites and EDA frameworks, by providing a substrate for easily developing loops which interoperate these tools with others in complex ways. Furthermore, CHIA provides a simple, efficient, and streamlined substrate for running the experiments performed in much of the work discussed above.

## 7 Discussion and Future Work

While developing and using CHIA, it has become clear that the bottlenecks in hardware design pipelines have shifted. As the process of turning an idea into a preliminary implementation speeds up, evaluating the performance and QoR of that idea as well as verifying the idea takes the bulk of the time. CHIA on its own attacks this problem with parallelism, but Amdahl's law tells us that parallelism alone is not enough. Rapid, high-fidelity evaluation and verification of hardware designs will be key to taking full advantage of the capabilities of agentic AI. This means we need shorter representative benchmarks, faster simulation methods and PPA estimation tools, and rapid verification methods without sacrificing fidelity and reliability.

Our field also faces challenges with privileged information. Since proprietary data usually cannot be fed to public frontier model-based agents, open PDKs [27, 49, 95] are used in CHIA loops in this paper to demonstrate the power of having PDK data in-the-loop in an agentic flow. Unsurprisingly, these PDKs do not always serve as predictive guides when the ultimate goal is to produce a chip in an advanced process. Therefore, it is critical to develop methodologies to sufficiently abstract quality-of-result and other relevant data before passing it to the largest frontier models. Alternatively, having capable, small, locally hosted models could solve this problem, since sensitive data remains private. CHIA already supports local model hosting and will continue to improve support for it moving forward.

Finally, we are actively dogfooding CHIA across a variety of projects and we plan to continue developing and supporting CHIA in the open-source.

## 8 Conclusion

CHIA reframes the central challenge of AI-driven hardware/software co-design: rather than treating each application of AI as a bespoke, one-off experiment, it makes the construction, deployment, and rigorous evaluation of the co-design flow itself a first-class concern. By expressing agentic design flows as CHIA loops and backing them with infrastructure for isolation, profiling, evaluation, and reliable execution across hundreds of heterogeneous systems, CHIA lowers the barrier to conducting principled, reproducible science

at scale. The five case studies presented here demonstrate the capabilities of CHIA, spanning evolutionary architectural discovery, LLM-driven RTL implementation, automatic simulator alignment, IPC-aware critical path optimization, and automated GitHub-issue fixing. Across the case studies, we hold the agents operating within the CHIA loops to a high standard of verification and validation. For example, the RTL implemented by agents delivers substantial performance improvements while successfully executing the entire 25+ trillion instructions of SPEC06-Ref in the context of a complete out-of-order superscalar microprocessor in a RISC-V system-on-chip. These agent-written implementations achieve their improvements while meeting or improving frequency and area constraints in open-source and commercial ASIC PDKs.

We emphasize that these case studies are illustrative rather than exhaustive. CHIA is designed as a foundation: our goal is to give the architecture, systems, compiler, and VLSI communities a shared, open-source substrate on which to build, compare, and extend AI-infused co-design workflows. As agentic AI continues to mature, we believe frameworks like CHIA will be essential to translating that progress into durable, verifiable advances in hardware/software co-design, and we hope that CHIA serves as a useful platform for the community upon which to build new AI-driven co-design flows.

## 9 Acknowledgments

We thank Schuyler Eldridge and the CIRCT code reviewers for their help in shepherding the merging of our CIRCT pull requests.

The information, data, or work presented herein was funded in part by the Jean and Hing Wong Foundation Faculty Fellowship, Samsung Advanced Institute of Technology (SAIT), and SLICE Lab industrial sponsors and affiliates. Any opinions, findings, conclusions, or recommendations in this paper are solely those of the authors and do not necessarily reflect the position or the policy of the sponsors. This material is also based in part upon work supported by the National Science Foundation Graduate Research Fellowship Program under Grant No. DGE 2146752. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the National Science Foundation.

## References

- [1] 2006. SPEC CPU 2006 benchmark. [spec.org. https://www.spec.org/cpu2006/](https://www.spec.org/cpu2006/)
- [2] Martin Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard, Manjunath Kudlur, Josh Levenberg, Rajat Monga, Sherry Moore, Derek G. Murray, Benoit Steiner, Paul Tucker, Vijay Vasudevan, Pete Warden, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. 2016. TensorFlow: A system for large-scale machine learning. [arXiv:1605.08695 \[cs.DC\] https://arxiv.org/abs/1605.08695](https://arxiv.org/abs/1605.08695)
- [3] Stefan Abi-Karam and Cong Hao. 2025. HLS-Eval: A Benchmark and Framework for Evaluating LLMs on High-Level Synthesis Design Tasks. In *2025 IEEE International Conference on LLM-Aided Design (ICLAD)*. 219–226. doi:10.1109/ICLAD65226.2025.00021
- [4] Lakshya A Agrawal, Shangyin Tan, Dilara Soylu, Noah Ziemis, Rishi Khare, Krista Opsahl-Ong, Arnab Singhvi, Herumb Shandilya, Michael J Ryan, Meng Jiang, Christopher Potts, Koushik Sen, Alexandros G. Dimakis, Ion Stoica, Dan Klein, Matei Zaharia, and Omar Khattab. 2026. GEPA: Reflective Prompt Evolution Can Outperform Reinforcement Learning. [arXiv:2507.19457 \[cs.CL\] https://arxiv.org/abs/2507.19457](https://arxiv.org/abs/2507.19457)
- [5] Elisavet Lydia Alvanaki, Kevin Lee, and Luca P. Carloni. 2025. SLDB: An End-To-End Heterogeneous System-on-Chip Benchmark Suite for LLM-Aided Design. [arXiv:2507.06376 \[cs.AR\] https://arxiv.org/abs/2507.06376](https://arxiv.org/abs/2507.06376)
- [6] Alon Amid, David Biancolin, Abraham Gonzalez, Daniel Grubb, Sagar Karandikar, Harrison Liew, Albert Magyar, Howard Mao, Albert Ou, Nathan

- Pemberton, Paul Rigge, Colin Schmidt, John Wright, Jerry Zhao, Yakun Sophia Shao, Krste Asanović, and Borivoje Nikolić. 2020. Chipyard: Integrated Design, Simulation, and Implementation Framework for Custom SoCs. *IEEE Micro* 40, 4 (July 2020), 10–21. doi:10.1109/MM.2020.2996616
- [7] anomalyco. 2025. *opencode*. <https://github.com/anomalyco/opencode>
- [8] Anonymous. 2025. ArchAgent: Agentic AI-driven Computer Architecture Discovery. OpenReview Anonymous Preprint. <https://openreview.net/forum?id=hcXN9l6zqZ> Submission Number 714.
- [9] Anthropic. [n. d.]. *Claude Code*. <https://www.anthropic.com/claude-code>
- [10] apache. 2015. *airflow*. <https://github.com/apache/airflow>
- [11] AWS. 2026. Amazon Bedrock. <https://aws.amazon.com/bedrock/> Accessed: 2026-06-23.
- [12] AWS. 2026. S3. <https://aws.amazon.com/s3/idc-server-side-test/awswt-956-v2-template-s3/variant/> Accessed: 2026-06-23.
- [13] Yunsheng Bai, Ghaiith Bany Hamad, Syed Suhaib, and Haoxing Ren. 2025. AssertionForge: Enhancing Formal Verification Assertion Generation with Structured Representation of Specifications and RTL. In *2025 IEEE International Conference on LLM-Aided Design (ICLAD)*. 85–92. doi:10.1109/ICLAD65226.2025.00009
- [14] Jonathan Balkind, Michael McKeown, Yaosheng Fu, Tri Nguyen, Yanqi Zhou, Alexey Lavrov, Mohammad Shahrad, Adi Fuchs, Samuel Payne, Xiaohua Liang, Matthew Matl, and David Wentzlaff. 2016. OpenPiton: An Open Source Many-core Research Framework. In *Proceedings of the Twenty-First International Conference on Architectural Support for Programming Languages and Operating Systems (Atlanta, Georgia, USA) (ASPLOS '16)*. Association for Computing Machinery, New York, NY, USA, 217–232. doi:10.1145/2872362.2872414
- [15] Abhishek Bhandwadar, Mihir Choudhury, Ruchir Puri, and Akash Srivastava. 2026. Agent Factories for High Level Synthesis: How Far Can General-Purpose Coding Agents Go in Hardware Optimization? arXiv:2603.25719 [cs.AI] <https://arxiv.org/abs/2603.25719>
- [16] Nathan Binkert, Bradford Beckmann, Gabriel Black, Steven K. Reinhardt, Ali Saidi, Arkaprava Basu, Joel Hestness, Derek R. Hower, Tushar Krishna, Somayeh Sardashti, Rathijit Sen, Korey Sewell, Muhammad Shoaib, Nilay Vaish, Mark D. Hill, and David A. Wood. 2011. The gem5 simulator. *SIGARCH Comput. Archit. News* 39, 2 (Aug. 2011), 1–7. doi:10.1145/2024716.2024718
- [17] B. Black and J.P. Shen. 1998. Calibration of microprocessor performance models. *Computer* 31, 5 (1998), 59–65. doi:10.1109/2.675637
- [18] Alexander Blasberg, Vasilis Kypriotis, and Dimitrios Skarlatos. 2026. Agentic Architect: An Agentic AI Framework for Architecture Design Exploration and Optimization. arXiv:2604.25083 [cs.AI] <https://arxiv.org/abs/2604.25083>
- [19] Alexander Blasberg, Vasilis Kypriotis, and Dimitrios Skarlatos. 2026. Agentic Architect: An Agentic AI Framework for Architecture Design Exploration and Optimization. arXiv:2604.25083 [cs.AI] <https://arxiv.org/abs/2604.25083>
- [20] Sriyash Caculo, Mahesh Madhav, and Jeff Baxter. 2025. Memory Access Vectors: Improving Sampling Fidelity for CPU Performance Simulations. arXiv:2506.02344 [cs.AR] <https://arxiv.org/abs/2506.02344>
- [21] Trevor E. Carlson, Wim Heirman, and Lieven Eeckhout. 2011. Sniper: Exploring the level of abstraction for scalable and accurate parallel multi-core simulation. In *SC '11: Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis*. 1–12. doi:10.1145/2063384.2063454
- [22] Christopher Celio. 2018. *A Highly Productive Implementation of an Out-of-Order Processor Generator*. Ph.D. Dissertation. EECS Department, University of California, Berkeley. <http://www2.eecs.berkeley.edu/Pubs/TechRpts/2018/EECS-2018-151.html>
- [23] Mert Cemri, Shubham Agrawal, Akshat Gupta, Shu Liu, Audrey Cheng, Qiuyang Mang, Ashwin Naren, Lutfi Eren Erdogan, Koushik Sen, Matei Zaharia, Alex Dimakis, and Ion Stoica. 2026. AdaEvolve: Adaptive LLM Driven Zeroth-Order Optimization. arXiv:2602.20133 [cs.NE] <https://arxiv.org/abs/2602.20133>
- [24] Zhengrui Chen, Zixuan Song, Yu Li, Qi Sun, and Cheng Zhuo. 2026. FluxEDA: A Unified Execution Infrastructure for Stateful Agentic EDA. arXiv:2603.25243 [cs.AR] <https://arxiv.org/abs/2603.25243>
- [25] Audrey Cheng, Shu Liu, Melissa Pan, Zhifei Li, Bowen Wang, Alex Krentsel, Tian Xia, Mert Cemri, Jongseok Park, Shuo Yang, Jeff Chen, Lakshya Agrawal, Aditya Desai, Jiarong Xing, Koushik Sen, Matei Zaharia, and Ion Stoica. 2025. Barbarians at the Gate: How AI is Upending Systems Research. (2025). arXiv:2510.06189 [cs.AI] <https://arxiv.org/abs/2510.06189>
- [26] CHIPS Alliance. 2026. riscv-dv: Random Instruction Generator for RISC-V Processor Verification. <https://github.com/chipsalliance/riscv-dv>. Accessed: 2026-06-09.
- [27] Lawrence T. Clark, Vinay Vashishtha, Lucian Shifren, Aditya Gujja, Saurabh Sinha, Brian Cline, Chandarasekaran Ramamurthy, and Greg Yeric. 2016. ASAP7: A 7-nm finFET predictive process design kit. *Microelectronics Journal* 53 (2016), 105–115. doi:10.1016/j.mejo.2016.04.006
- [28] Joshua H. Davis, Klaudiusz Rydz, Srinivasan Ramesh, Aadit Nilay, Daniel Nichols, Swapna Raj, Nikhil Jain, and Abhinav Bhatete. 2026. KEET: Explaining Performance of GPU Kernels Using LLM Agents. arXiv:2605.04467 [cs.PF] <https://arxiv.org/abs/2605.04467>
- [29] Chenhui Deng, Zhongzhi Yu, Guan-Ting Liu, Nathaniel Pinckney, Bruce Khailany, and Haoxing Ren. 2026. ACE-RTL: When Agentic Context Evolution Meets RTL-Specialized LLMs. arXiv:2602.10218 [cs.AR] <https://arxiv.org/abs/2602.10218>
- [30] Schuyler Eldridge, Prithayan Barua, Aliaksei Chapyzenka, Adam Izraelevitz, Jack Koenig, Chris Lattner, Andrew Lenhardt, George Leontiev, Fabian Schuiki, Ram Sunder, Andrew Young, and Richard Xia. 2021. MLIR as hardware compiler infrastructure. In *Workshop on Open-Source EDA Technology (WOSET)*, Vol. 3.
- [31] FireworksAI. 2026. FireworksAI. <https://fireworks.ai/> Accessed: 2026-06-23.
- [32] Kevin P. Gaffney, Martin Prammer, Larry Brasfield, D. Richard Hipp, Dan Kennedy, and Jignesh M. Patel. 2022. SQLite: past, present, and future. *Proc. VLDB Endow.* 15, 12 (Aug. 2022), 3535–3547. doi:10.14778/3554821.3554842
- [33] Jiahao Gai, Hao Chen, Zhican Wang, Hongyu Zhou, Wanru Zhao, Nicholas Lane, and Hongxiang Fan. 2025. Exploring Code Language Models for Automated HLS-based Hardware Generation: Benchmark, Infrastructure and Analysis. In *Proceedings of the 30th Asia and South Pacific Design Automation Conference (Tokyo, Japan) (ASPDAC '25)*. Association for Computing Machinery, New York, NY, USA, 988–994. doi:10.1145/3658617.3697616
- [34] Emden R. Gansner and Stephen C. North. 2000. An open graph visualization system and its applications to software engineering. *Softw. Pract. Exper.* 30, 11 (Sept. 2000), 1203–1233.
- [35] Bogdan Georgiev, Javier Gómez-Serrano, Terence Tao, and Adam Zsolt Wagner. 2025. Mathematical exploration and discovery at scale. *arXiv preprint arXiv:2511.02864* (2025).
- [36] Michael Gerstenhaber and Michael Bachman. 2026. Introducing Gemini Enterprise Agent Platform, powering the next wave of agents. <https://cloud.google.com/blog/products/ai-machine-learning/introducing-gemini-enterprise-agent-platform>
- [37] Github. [n. d.]. GitHub REST API documentation. <https://docs.github.com/en/rest> Accessed: 2026-06-23.
- [38] Nathan Gober, Gino Chacon, Lei Wang, Paul V. Gratz, Daniel A. Jimenez, Elvira Teran, Seth Pugsley, and Jinchun Kim. 2022. The Championship Simulator: Architectural Simulation for Education and Competition. arXiv:2210.14324 [cs.AR] <https://arxiv.org/abs/2210.14324>
- [39] Google. 2026. Google Antigravity. <https://antigravity.google/> Accessed: 2026-06-23.
- [40] Google googleapis. 2014. *google-cloud-python*. <https://github.com/googleapis/google-cloud-python>
- [41] Groq. 2026. Groq. <https://groq.com/> Accessed: 2026-06-23.
- [42] Ce Guo and Tong Zhao. 2025. ResBench: A Resource-Aware Benchmark for LLM-Generated FPGA Designs. In *Proceedings of the 15th International Symposium on Highly Efficient Accelerators and Reconfigurable Technologies (HEART '25)*. Association for Computing Machinery, New York, NY, USA, 25–34. doi:10.1145/3728179.3728192
- [43] Raghav Gupta, Akanksha Jain, Abraham Gonzalez, Alexander Novikov, Po-Sen Huang, Matej Balog, Marvin Eisenberger, Sergey Shirobokov, Ngán Vù, Martin Dixon, Borivoje Nikolić, Parthasarathy Ranganathan, and Sagar Karandikar. 2026. ArchAgent: Agentic AI-driven Computer Architecture Discovery. arXiv:2602.22425 [cs.AI] <https://arxiv.org/abs/2602.22425>
- [44] Zhuolun He, Haoyuan Wu, Xinyun Zhang, Xufeng Yao, Su Zheng, Haisheng Zheng, and Bei Yu. 2023. ChatEDA: A Large Language Model Powered Autonomous Agent for EDA. In *2023 ACM/IEEE 5th Workshop on Machine Learning for CAD (MLCAD)*. 1–6. doi:10.1109/MLCAD58807.2023.10299852
- [45] Charles Hong, Sahil Bhatia, Alvin Cheung, and Yakun Sophia Shao. 2025. Autocomp: A Powerful and Portable Code Optimizer for Tensor Accelerators. arXiv:2505.18574 [cs.PL] <https://arxiv.org/abs/2505.18574>
- [46] A. Izraelevitz, J. Koenig, P. Li, R. Lin, A. Wang, A. Magyar, D. Kim, C. Schmidt, C. Markley, J. Lawson, and J. Bachrach. 2017. Reusability is FIRRTL ground: Hardware construction languages, compiler frameworks, and transformations. In *2017 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*. 209–216. doi:10.1109/ICCAD.2017.8203780
- [47] Carlos E. Jimenez, John Yang, Alexander Wettig, Shunyu Yao, Kexin Pei, Ofir Press, and Karthik Narasimhan. 2024. SWE-bench: Can Language Models Resolve Real-World GitHub Issues? arXiv:2310.06770 [cs.CL] <https://arxiv.org/abs/2310.06770>
- [48] Sagar Karandikar, Howard Mao, Donggyu Kim, David Biancolin, Alon Amid, Dayeol Lee, Nathan Pemberton, Emmanuel Amaro, Colin Schmidt, Aditya Chopra, Qijing Huang, Kyle Kovacs, Borivoje Nikolic, Randy Katz, Jonathan Bachrach, and Krste Asanović. 2018. FireSim: FPGA-accelerated Cycle-exact Scale-out System Simulation in the Public Cloud. In *Proceedings of the 45th Annual International Symposium on Computer Architecture (Los Angeles, California) (ISCA '18)*. IEEE Press, Piscataway, NJ, USA, 29–42. doi:10.1109/ISCA.2018.00014
- [49] Steve Kosier. 2023. SKY's the Limit with the SKY130 Open-Source PDK. <https://www.skywatertechnology.com/sky130-open-source-pdk/> Accessed: 2026-06-30.
- [50] Srivatsan Krishnan, Amir Yazdanbakhsh, Shvetank Prakash, Jason Jabour, Ikechukwu Uchendu, Susobhan Ghosh, Behzad Bouroujrdian, Daniel Richins, Devashree Tripathy, Aleksandra Faust, and Vijay Janapa Reddi. 2023. ArchGym:

- An Open-Source Gymnasium for Machine Learning Assisted Architecture Design. In *Proceedings of the 50th Annual International Symposium on Computer Architecture* (Orlando, FL, USA) (ISCA '23). Association for Computing Machinery, New York, NY, USA, Article 14, 16 pages. doi:10.1145/3579371.3589049
- [51] Woosuk Kwon, Zhuohan Li, Siyuan Zhuang, Ying Sheng, Lianmin Zheng, Cody Hao Yu, Joseph E. Gonzalez, Hao Zhang, and Ion Stoica. 2023. Efficient Memory Management for Large Language Model Serving with PagedAttention. arXiv:2309.06180 [cs.LG] <https://arxiv.org/abs/2309.06180>
- [52] langchain ai. 2023. *langgraph*. <https://github.com/langchain-ai/langgraph>
- [53] Yunsup Lee and Henry Cook. 2012. *riscv-torture*. <https://github.com/ucbar/riscv-torture>
- [54] Patrick S. Li, Adam M. Izraelevitz, and Jonathan Bachrach. 2016. *Specification for the FIRRTL Language*. Technical Report UCB/EECS-2016-9. EECS Department, University of California, Berkeley. <http://www2.eecs.berkeley.edu/Pubs/TechRpts/2016/EECS-2016-9.html>
- [55] Xingquan Li, Simin Tao, Zengrong Huang, Shijian Chen, Zhisheng Zeng, Liwei Ni, Zhipeng Huang, Chunan Zhuang, Hongxi Wu, Weiguo Li1, Xueyan Zhao, He Liu, Shuaiying Long, Wei He, Bojun Liu, Sifeng Gan, Zihao Yu, Tong Liu, Yuchi Miao, Zhiyuan Yan, Hao Wang, Jie Zhao, Yifan Li, Ruizhi Liu, Xiaozhe Lin, Bo Yang, Zhen Xue, Fuxing Huang, Zonglin Yang, Zhenggang Wu, Jiangkai Li, Yuezuo Liu, Ming Peng, Yihang Qiu, Wenrui Wu, Zheqing Shao, Kai Mo, Jikang Liu, Yuyao Liang, Mingzhe Zhang, Zhuang Ma, Xiang Cong, Daxiang Huang, Guojie Luo, Huawei Li, Haihua Shen, Mingyu Chen, Dongbo Bu, Wenxing Zhu, Ye Cai, Xiaoming Xiong, Ying Jiang, Yi Heng, Peng Zhang, Biwei Xie, and Yungang Bao. 2023. iEDA: An Open-Source Intelligent Physical Implementation Toolkit and Library. arXiv:2308.01857 [cs.AR] <https://arxiv.org/abs/2308.01857>
- [56] Harrison Liew, Daniel Grubb, John Wright, Colin Schmidt, Nayiri Krzysztofowicz, Adam Izraelevitz, Edward Wang, Krste Asanović, Jonathan Bachrach, and Borivoje Nikolić. 2022. Hammer: a modular and reusable physical design flow tool: invited. In *Proceedings of the 59th ACM/IEEE Design Automation Conference* (San Francisco, California) (DAC '22). Association for Computing Machinery, New York, NY, USA, 1335–1338. doi:10.1145/3489517.3530672
- [57] Shu Liu, Shubham Agarwal, Monishwaran Maheswaran, Mert Cemri, Zhifei Li, Qiuyang Mang, Ashwin Naren, Ethan Boneh, Audrey Cheng, Melissa Z. Pan, Alexander Du, Kurt Keutzer, Alvin Cheung, Alexandros G. Dimakis, Koushik Sen, Matei Zaharia, and Ion Stoica. 2026. EvoX: Meta-Evolution for Automated Discovery. arXiv:2602.23413 [cs.LG] <https://arxiv.org/abs/2602.23413>
- [58] Shu Liu, Mert Cemri, Shubham Agarwal, Alexander Krentsel, Ashwin Naren, Qiuyang Mang, Zhifei Li, Akshat Gupta, Monishwaran Maheswaran, Audrey Cheng, Melissa Pan, Ethan Boneh, Kannan Ramchandran, Koushik Sen, Matei Zaharia, Alexandros G. Dimakis, and Ion Stoica. 2026. *SkyDiscover: A Flexible, Adaptive Framework for AI-Driven Scientific and Algorithmic Discovery*. Association for Computing Machinery, New York, NY, USA, 1223–1227. <https://doi.org/10.1145/3786335.3813221>
- [59] Shang Liu, Wenji Fang, Yao Lu, Qijun Zhang, Hongce Zhang, and Zhiyao Xie. 2024. RTLcoder: Outperforming GPT-3.5 in Design RTL Generation with Our Open-Source Dataset and Lightweight Solution. In *2024 IEEE LLM Aided Design Workshop (LAD)*. IEEE, 1–5. doi:10.1109/lad62341.2024.10691788
- [60] LLVM. 2026. LLVM AI Tool Use Policy. <https://llvm.org/docs/AIToolPolicy.html> Accessed: 2026-06-19.
- [61] Jason Lowe-Power, Abdul Mutaal Ahmad, Ayaz Akram, Mohammad Alian, Rico Amslinger, Matteo Andreozzi, Adrià Armejach, Nils Asmussen, Brad Beckmann, Srikant Bharadwaj, Gabe Black, Gedare Bloom, Bobby R. Bruce, Daniel Rodrigues Carvalho, Jeronimo Castrillon, Lizhong Chen, Nicolas Derumigny, Stephan Diestelhorst, Wendy Elsasser, Carlos Escuin, Marjan Fariborz, Amin Farmahini-Farahani, Pouya Fotouhi, Ryan Gambord, Jayneel Gandhi, Dibakar Gope, Thomas Grass, Anthony Gutierrez, Bagus Hanindhito, Andreas Hansson, Swapnil Haria, Austin Harris, Timothy Hayes, Adrian Herrera, Matthew Horsnell, Syed Ali Raza Jafri, Radhika Jagtap, Hanhui Jang, Reiley Jayapaul, Timothy M. Jones, Matthias Jung, Subash Kannoth, Hamidreza Khaleghzadeh, Yuetsu Kodama, Tushar Krishna, Tommaso Marinelli, Christian Menard, Andrea Mondelli, Miquel Moreto, Tiago Mück, Omar Naji, Krishendra Nathella, Hoa Nguyen, Nikos Nikolieris, Lena E. Olson, Marc Orr, Binh Pham, Pablo Prieto, Trivikram Reddy, Alec Roelke, Mahyar Samani, Andreas Sandberg, Javier Setoain, Boris Shingarov, Matthew D. Sinclair, Tuan Ta, Rahul Thakur, Giacomo Travaglino, Michael Upton, Nilay Vaish, Ilias Vougioukas, William Wang, Zhengrong Wang, Norbert Wehn, Christian Weis, David A. Wood, Hongil Yoon, and Eder F. Zulian. 2020. The gem5 Simulator: Version 20.0+. arXiv:2007.03152 [cs.AR] <https://arxiv.org/abs/2007.03152>
- [62] Yao Lu, Shang Liu, Qijun Zhang, and Zhiyao Xie. 2024. RTLML: An Open-Source Benchmark for Design RTL Generation with Large Language Model. In *2024 29th Asia and South Pacific Design Automation Conference (ASP-DAC)*. 722–727. doi:10.1109/ASP-DAC58780.2024.10473904
- [63] Yi-Chen Lu, Hao-Hsiang Hsiao, and Haoxing Ren. 2025. Invited Paper: LLM-Enhanced GPU-Optimized Physical Design at Scale. In *2025 IEEE/ACM International Conference On Computer Aided Design (ICCAD)*. 1–7. doi:10.1109/ICCAD666269.2025.11240986
- [64] Paolo Mantovani, Davide Giri, Giuseppe Di Guglielmo, Luca Piccolboni, Joseph Zuckerman, Emilio G. Cota, Michele Petracca, Christian Pilato, and Luca P. Carloni. 2020. Agile SoC development with open ESP. In *Proceedings of the 39th International Conference on Computer-Aided Design* (Virtual Event, USA) (ICCAD '20). Association for Computing Machinery, New York, NY, USA, Article 96, 9 pages. doi:10.1145/3400302.3415753
- [65] Dirk Merkel. 2014. Docker: lightweight Linux containers for consistent development and deployment. *Linux J*. 2014, 239, Article 2 (March 2014).
- [66] Kaushal Mhapsekar, Azam Ghanbari, Bitu Aslrousta, and Samira Mirbagher-Ajorpaz. 2026. CacheMind: From Miss Rates to Why - Natural-Language, Trace-Grounded Reasoning for Cache Replacement. In *Proceedings of the 31st ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2* (USA) (ASPLOS '26). Association for Computing Machinery, New York, NY, USA, 307–322. doi:10.1145/3779212.3790136
- [67] Microsoft. 2025. *agent-framework*. <https://github.com/microsoft/agent-framework>
- [68] Philipp Moritz, Robert Nishihara, Stephanie Wang, Alexey Tumanov, Richard Liaw, Eric Liang, Melih Elibol, Zongheng Yang, William Paul, Michael I. Jordan, and Ion Stoica. 2018. Ray: a distributed framework for emerging AI applications. In *Proceedings of the 13th USENIX Conference on Operating Systems Design and Implementation* (Carlsbad, CA, USA) (OSDI'18). USENIX Association, USA, 561–577.
- [69] Alexander Novikov, Ngàn Vù, Marvin Eisenberger, Emilien Dupont, Po-Sen Huang, Adam Zsolt Wagner, Sergey Shirobokov, Borislav Kozlovskii, Francisco J. R. Ruiz, Abbas Mehrabian, M. Pawan Kumar, Abigail See, Swarat Chaudhuri, George Holland, Alex Davies, Sebastian Nowozin, Pushmeet Kohli, and Matej Balog. 2025. AlphaEvolve: A coding agent for scientific and algorithmic discovery. (2025). arXiv:2506.13131 [cs.AI] <https://arxiv.org/abs/2506.13131>
- [70] Tony Nowatzki. 2015. *Microbench*. <https://github.com/VerticalResearchGroup/microbench>
- [71] Surim Oh, Mingsheng Xu, Tanvir Ahmed Khan, Baris Kasikci, and Heiner Litz. 2024. UDP: Utility-Driven Fetch Directed Instruction Prefetching. In *Proceedings of the 51st International Symposium on Computer Architecture* (ISCA) (ISCA 2024).
- [72] ollama. 2023. *ollama*. <https://github.com/ollama/ollama>
- [73] OpenAI. 2026. Codex. <https://chatgpt.com/codex/> Accessed: 2026-06-23.
- [74] openclaw. 2026. *openclaw*. <https://github.com/openclaw/openclaw>
- [75] OpenRouter. 2026. OpenRouter. <https://openrouter.ai/> Accessed: 2026-06-23.
- [76] openssl. 2013. *openssl*. <https://github.com/openssl/openssl>
- [77] Anne Ouyang, Simon Guo, Simran Arora, Alex L Zhang, William Hu, Christopher Re, and Azalia Mirhoseini. 2025. KernelBench: Can LLMs Write Efficient GPU Kernels?. In *Forty-second International Conference on Machine Learning*. <https://openreview.net/forum?id=yeoNliQ1T1x>
- [78] Sujay Pandit, Akanksha Jain, Rami Cohen, Zhijie Deng, Sagar Karandikar, Sagi Perel, Anand Raghunathan, and Parthasarathy Ranganathan. 2025. LLM-Box : An Agentic Framework for Guided Black-Box Optimization in Mapping LLMs onto Specialized Hardware Accelerators. In *Machine Learning for Systems 2025*. <https://openreview.net/forum?id=W2htGuo9de>
- [79] David Patterson, Jeremy Bennett, Mary Bennett, Hélène Chelin, David Harris, Jennifer Hellar, William Jones, Konrad Moron, Paolo Savini, Roger Shepherd, Ray Simar, Zachary Susskind, and Stefan Wallentowitz. 2025. Embench IOT 2.0 and DSP 1.0: Modern Embedded Computing Benchmarks. *Computer* 58, 5 (May 2025), 37–47. doi:10.1109/MC.2024.3511352
- [80] Nathaniel Pinckney, Chenhui Deng, Chia-Tung Ho, Yun-Da Tsai, Mingjie Liu, Wenfei Zhou, Bruce Khailany, and Haoxing Ren. 2025. Comprehensive Verilog Design Problems: A Next-Generation Benchmark Dataset for Evaluating Large Language Models and Agents on RTL Design and Verification. arXiv:2506.14074 [cs.LG] <https://arxiv.org/abs/2506.14074>
- [81] PrefectHQ. 2024. *fastmcp*. <https://github.com/PrefectHQ/fastmcp>
- [82] AWS Boto project. 2014. *boto3*. <https://github.com/boto/boto3>
- [83] RISC-V International. 2026. Bit Manipulation Extensions. <https://docs.riscv.org/reference/isa/v20260120/unpriv/b-st-ext.html>. Accessed: 2026-06-23.
- [84] RISC-V International. 2026. RISC-V Cryptography Extensions Volume I. [https://docs.riscv.org/reference/isa/extensions/crypto-scalar/\\_attachments/riscv-crypto-spec-scalar.pdf](https://docs.riscv.org/reference/isa/extensions/crypto-scalar/_attachments/riscv-crypto-spec-scalar.pdf). Accessed: 2026-06-23.
- [85] RISC-V International. 2026. Zicond Extension for Integer Conditional Operations. <https://docs.riscv.org/reference/isa/v20260120/unpriv/zicond.html>. Accessed: 2026-06-23.
- [86] riscv-software src. 2011. *riscv-isa-sim*. <https://github.com/riscv-software-src/riscv-isa-sim>
- [87] riscv-software src. 2013. *riscv-tests*. <https://github.com/riscv-software-src/riscv-tests>
- [88] Davide Rossi, Francesco Conti, Andrea Marongiu, Antonio Pullini, Igor Loi, Michael Gautschi, Giuseppe Tagliavini, Alessandro Capotondi, Philippe Flatters, and Luca Benini. 2015. PULP: A parallel ultra low power platform for next generation IoT applications. In *2015 IEEE Hot Chips 27 Symposium (HCS)*. 1–39. doi:10.1109/HOTCHIPS.2015.7477325
- [89] Hemant Rotthor. 2013. A Taxonomy of Performance Assurance Methodologies and its Application in High Performance Computer Architectures. *International*

- Journal of Software Engineering & Applications* 4, 2 (March 2013), 1–17. doi:10.5121/jsea.2013.4201
- [90] Daniel Sanchez and Christos Kozyrakis. 2013. ZSim: fast and accurate microarchitectural simulation of thousand-core systems. In *Proceedings of the 40th Annual International Symposium on Computer Architecture* (Tel-Aviv, Israel) (ISCA '13). Association for Computing Machinery, New York, NY, USA, 475–486. doi:10.1145/2485922.2485963
- [91] Karthikeyan Sankaralingam. 2026. Computer Architecture’s AlphaZero Moment: Automated Discovery in an Encircled World. arXiv:2604.03312 [cs.AR] <https://arxiv.org/abs/2604.03312>
- [92] Asankhaya Sharma. 2025. *OpenEvolve: an open-source evolutionary coding agent*. <https://github.com/algorithmsuperintelligence/openevolve>
- [93] Vinamra Sharma, Xingjian Fu, Jude Haris, and José Cano. 2026. LLM-Driven Design Space Exploration of FPGA-based Accelerators. arXiv:2605.05920 [cs.AR] <https://arxiv.org/abs/2605.05920>
- [94] Wilson Snyder, Paul Wasson, Duane Galbi, Geza Lore, and et al. [n. d.]. Verilator. <https://verilator.org>
- [95] James E. Stine, Ivan Castellanos, Michael Wood, Jeff Henson, Fred Love, W. Rhett Davis, Paul D. Franzon, Michael Bucher, Sunil Basavarajiah, Julie Oh, and Ravi Jenkal. 2007. FreePDK: An Open-Source Variation-Aware Design Kit. In *2007 IEEE International Conference on Microelectronic Systems Education (MSE'07)*. 173–174. doi:10.1109/MSE.2007.44
- [96] Michael Stonebraker and Lawrence A. Rowe. 1986. The design of POSTGRES. In *Proceedings of the 1986 ACM SIGMOD International Conference on Management of Data* (Washington, D.C., USA) (SIGMOD '86). Association for Computing Machinery, New York, NY, USA, 340–355. doi:10.1145/16894.16888
- [97] S. Surya, P. Bose, and J.A. Abraham. 1994. Architectural performance verification: PowerPC processors. In *Proceedings 1994 IEEE International Conference on Computer Design: VLSI in Computers and Processors*. 344–347. doi:10.1109/ICCD.1994.331922
- [98] The Verkor Team, Ravi Krishna, Suresh Krishna, and David Chin. 2026. Design Conductor 2.0: An agent builds a TurboQuant inference accelerator in 80 hours. arXiv:2605.05170 [cs.AR] <https://arxiv.org/abs/2605.05170>
- [99] The Verkor Team, Ravi Krishna, Suresh Krishna, and David Chin. 2026. Design Conductor: An agent autonomously builds a 1.5 GHz Linux-capable RISC-V CPU. arXiv:2603.08716 [cs.AR] <https://arxiv.org/abs/2603.08716>
- [100] Pei-Huan Tsai, Kuan-Lin Chiu, William Baisi, Pin-Yu Chen, and Luca P. Carloni. 2026. HSCO-Bench: An Agent-Driven End-to-End Hardware-Software Co-design Benchmark for Systems-on-Chip. arXiv:2605.19399 [cs.AR] <https://arxiv.org/abs/2605.19399>
- [101] Vladimir Vuskovic and Anant Nawalgaria. 2025. *AlphaEvolve on Google Cloud: AI for agentic discovery and optimization*. <https://cloud.google.com/blog/products/ai-machine-learning/alphaevolve-on-google-cloud> Google Cloud Blog.
- [102] wandb. 2017. *wandb*. <https://github.com/wandb/wandb>
- [103] Yiting Wang, Wanghao Ye, Ping Guo, Yexiao He, Ziyao Wang, Bowei Tian, Shwai He, Guoheng Sun, Zheyu Shen, Sihan Chen, Ankur Srivastava, Qingfu Zhang, Gang Qu, and Ang Li. 2025. SymRTL0: Enhancing RTL Code Optimization with LLMs and Neuron-Inspired Symbolic Reasoning. arXiv:2504.10369 [cs.AR] <https://arxiv.org/abs/2504.10369>
- [104] Yiting Wang, Wanghao Ye, Yexiao He, Yiran Chen, Gang Qu, and Ang Li. 2025. MCP4EDA: LLM-Powered Model Context Protocol RTL-to-GDSII Automation with Backend Aware Synthesis Optimization. arXiv:2507.19570 [cs.AR] <https://arxiv.org/abs/2507.19570>
- [105] Yuheng Wu, Berk Gokmen, Zhouhua Xie, Peijing Li, Caroline Trippel, Priyanka Raina, and Thierry Tambe. 2026. LLM-FSM: Scaling Large Language Models for Finite-State Reasoning in RTL Code Generation. arXiv:2602.07032 [cs.AI] <https://arxiv.org/abs/2602.07032>
- [106] Bing Xu, Terry Chen, Fengzhe Zhou, Tianqi Chen, Yangqing Jia, Vinod Grover, Haicheng Wu, Wei Liu, Craig Wittenbrink, Wen mei Hwu, Roger Bringmann, Ming-Yu Liu, Luis Ceze, Michael Lightstone, and Humphrey Shi. 2026. VibeTensor: System Software for Deep Learning, Fully Generated by AI Agents. arXiv:2601.16238 [cs.SE] <https://arxiv.org/abs/2601.16238>
- [107] Zhongzhi Yu, Mingjie Liu, Michael Zimmer, Yingyan Celine, Yong Liu, and Haoxing Ren. 2025. Spec2RTL-Agent: Automated Hardware Code Generation from Complex Specifications Using LLM Agent Systems. In *2025 IEEE International Conference on LLM-Aided Design (ICLAD)*. 37–43. doi:10.1109/ICLAD652.26.2025.00013
- [108] Zhongkai Yu, Chenyang Zhou, Yichen Lin, Hejia Zhang, Haotian Ye, Junxia Cui, Zaifeng Pan, Jishen Zhao, and Yufei Ding. 2026. ChipBench: A Next-Step Benchmark for Evaluating LLM Performance in AI-Aided Chip Design. arXiv:2601.21448 [cs.AI] <https://arxiv.org/abs/2601.21448>
- [109] Niansong Zhang, Chenhui Deng, Johannes Maximilian Kuehn, Chia-Tung Ho, Cunxi Yu, Zhiru Zhang, and Haoxing Ren. 2025. ASPEN: LLM-Guided E-Graph Rewriting for RTL Datapath Optimization. In *2025 ACM/IEEE 7th Symposium on Machine Learning for CAD (MLCAD)* (Santa Cruz, CA, USA). IEEE Press, 1–9. doi:10.1109/MLCAD65511.2025.11189222
- [110] Tao Zhang, Rui Ma, Shuotao Xu, Yongqiang Xiong, and Peng Cheng. 2026. LUMINA: LLM-Guided GPU Architecture Exploration via Bottleneck Analysis. arXiv:2603.05904 [cs.AR] <https://arxiv.org/abs/2603.05904>
- [111] Jerry Zhao and Abraham Gonzalez. 2020. Sonic BOOM: The 3rd Generation Berkeley Out-of-Order Machine. In *Proceedings of the Fourth Workshop on Computer Architecture Research with RISC-V (CARRV 2020)*. <https://api.semanticscholar.org/CorpusID:221212196>
- [112] Yang Zhao, Di Huang, Chongxiao Li, Pengwei Jin, Muxin Song, Yinan Xu, Ziyuan Nan, Mingju Gao, Tianyun Ma, Lei Qi, Yansong Pan, Zhenxing Zhang, Rui Zhang, Xishan Zhang, Zidong Du, Qi Guo, and Xing Hu. 2025. CodeV: Empowering LLMs with HDL Generation through Multi-Level Summarization. arXiv:2407.10424 [cs.PL] <https://arxiv.org/abs/2407.10424>
- [113] Yujie Zhao, Hejia Zhang, Hanxian Huang, Zhongming Yu, and Jishen Zhao. 2024. MAGE: A Multi-Agent Engine for Automated RTL Code Generation. arXiv:2412.07822 [cs.AR] <https://arxiv.org/abs/2412.07822>