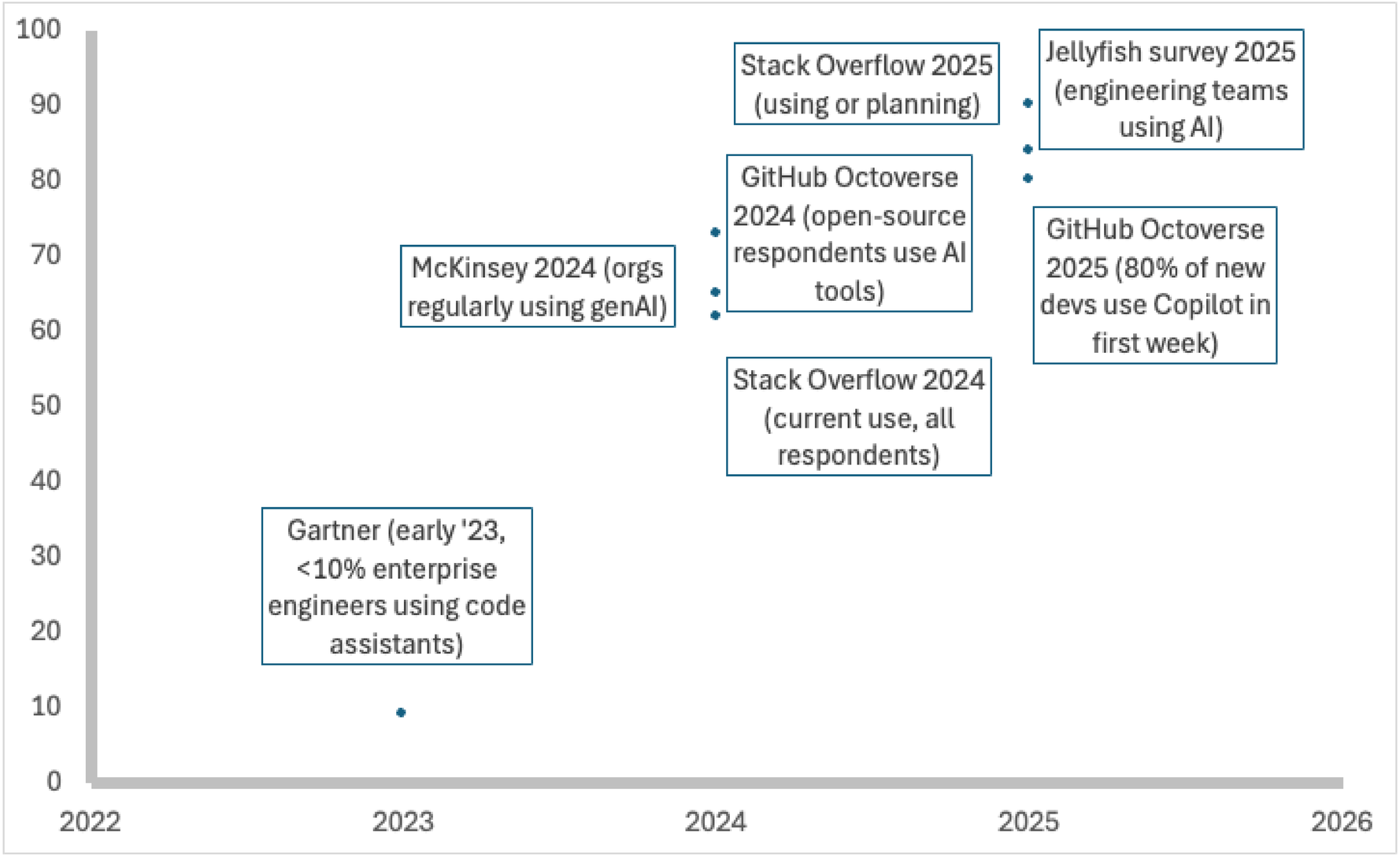
AI for Hardware Design: From Fine-Tuned Models to Autonomous Agents

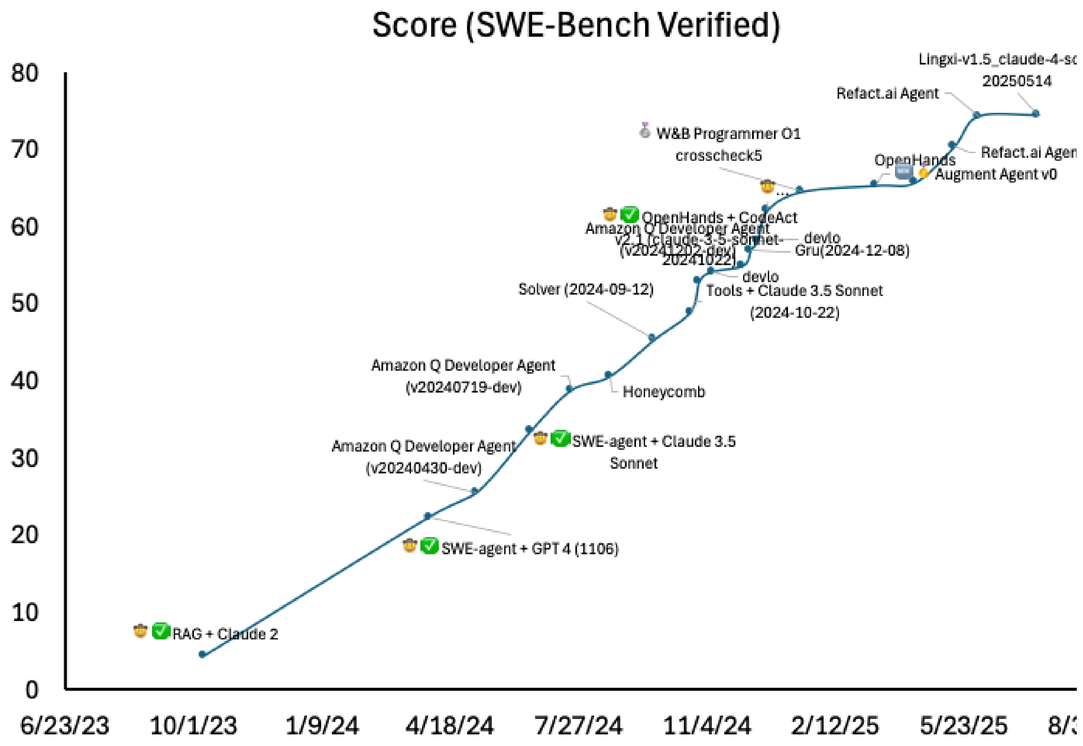
Mark Ren, Director of Design Automation Research

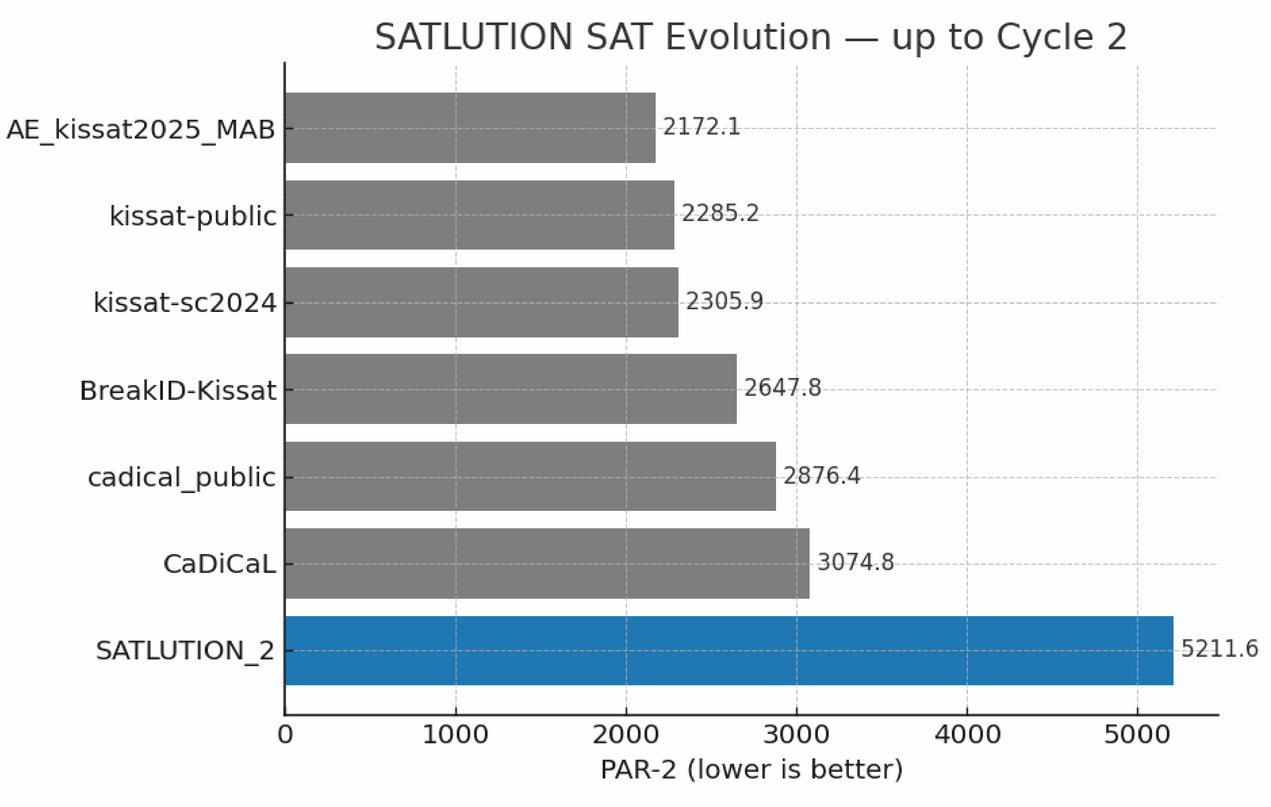
Wide Adoption of AI in Software Engineering



AI for Software Engineering

From CoPilot to Debug and Optimization





Solving GitHub repository debug issues

Coding IDEs and CLIs adopted by the world

Start from the winning solvers in 2004 SAT competition
Improved performance by 18% in a month of evolution

Leverage Cursor and Claude models

AI for RTL Coding Today

Question Prompt:
Implement the Verilog module based on the following description. Assume that signals are positive clock/clk edge triggered unless otherwise stated.

Problem Description:
Given an 8-bit input vector [7:0], reverse its bit ordering.

```
module top_module (
        input [7:0] in,
        output [7:0] out
);
```

Canonical Solution:

```
  assign {out[0],out[1],out[2],out[3],out[4],
  ↪  out[5],out[6],out[7]} = in;
endmodule
```

VerilogEval (2023)

70+% pass rate

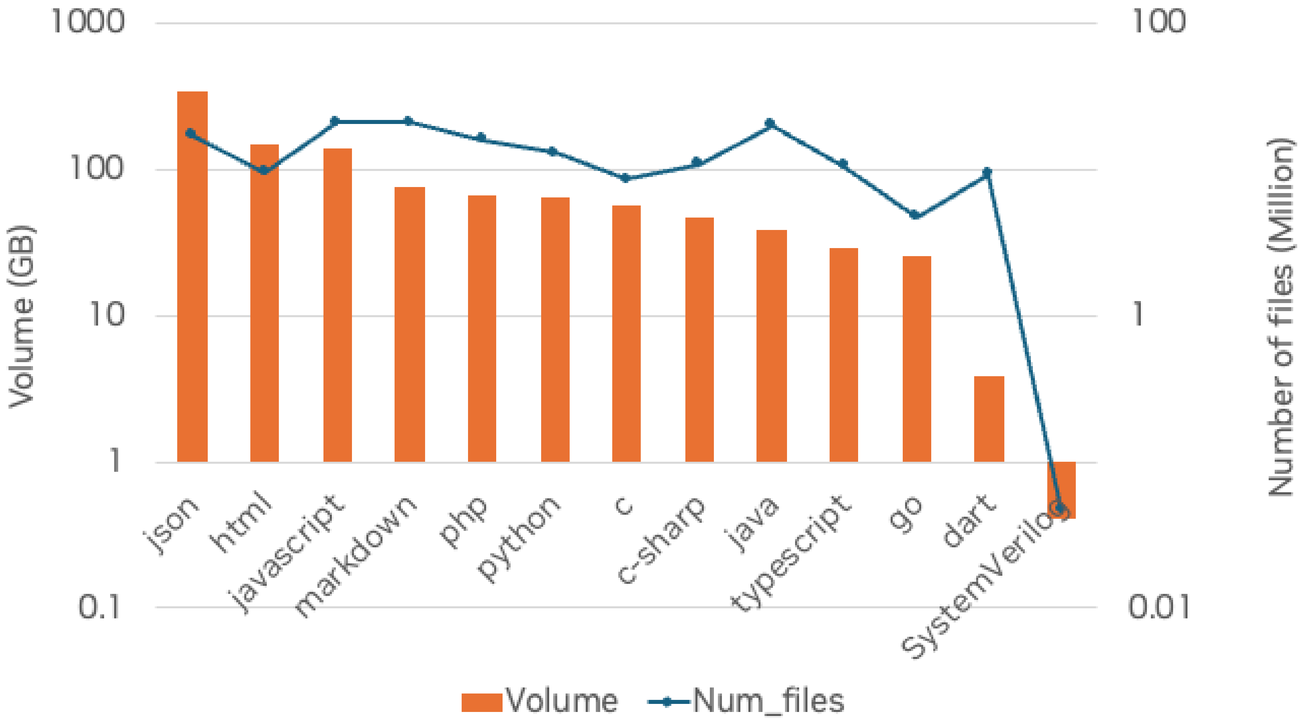
ID	Category Description	Volume		Tokens Mean/Max (k)	
		Non-Agentic	Agentic	Non-Agentic	Agentic
Code Generation					
cid02	RTL - Code Completion	94	0	1.5/4.5	–
cid03	RTL - Natural Language Spec to Code	78	37	1.2/6.9	2.7/7.9
cid04	RTL - Code Modification	56	26	2.0/4.6	5.7/19.5
cid05	RTL - Spec to RTL (Module Reuse)	0	26	–	7.4/28.5
cid07	RTL - Code Improvement (Linting/QoR)	41	0	1.9/5.9	–
cid12	Design Verification - Testbench Stimulus Gen.	68	18	1.4/6.2	2.1/4.6
cid13	Design Verification - Testbench Checker Gen.	53	18	2.8/7.3	4.5/10.7
cid14	Design Verification - Assertion Generation	68	30	2.6/7.5	4.8/14.6
cid16	Design Verification - Debugging / Bug Fixing	36	11	2.3/6.5	3.9/14.5
Code Comprehension					
cid06	Correspondence - RTL to/from Specification	34	0	1.6/5.5	–
cid08	Correspondence - Testbench to/from Test Plan	29	0	3.1/6.1	–
cid09	Question & Answer - RTL	34	0	1.1/5.0	–
cid10	Question & Answer - Testbench	26	0	3.6/4.8	–
	Total # of Problems	**617**	**166**		

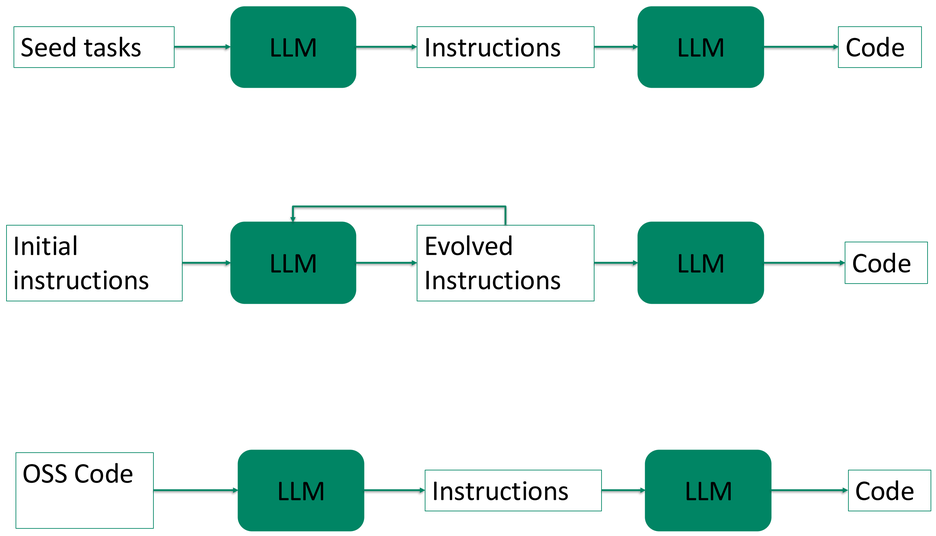
CVDP (2025)

10%-40% pass rate

RTL Code Data Inefficiency

Synthetic Data Generation to the Rescue





StarCoder: may the source be with you!

Use Frontier models to generate both instructions and code for training

CraftRTL: Crafting SDG for VerilogEval

Conventional SDG:

80.1K x [Instruction → code]

Generate correct-by-construction data for non-textual problems such as : Karnaugh map, FSM and waveform:

28.5K x [non-textual problem instruction → code]

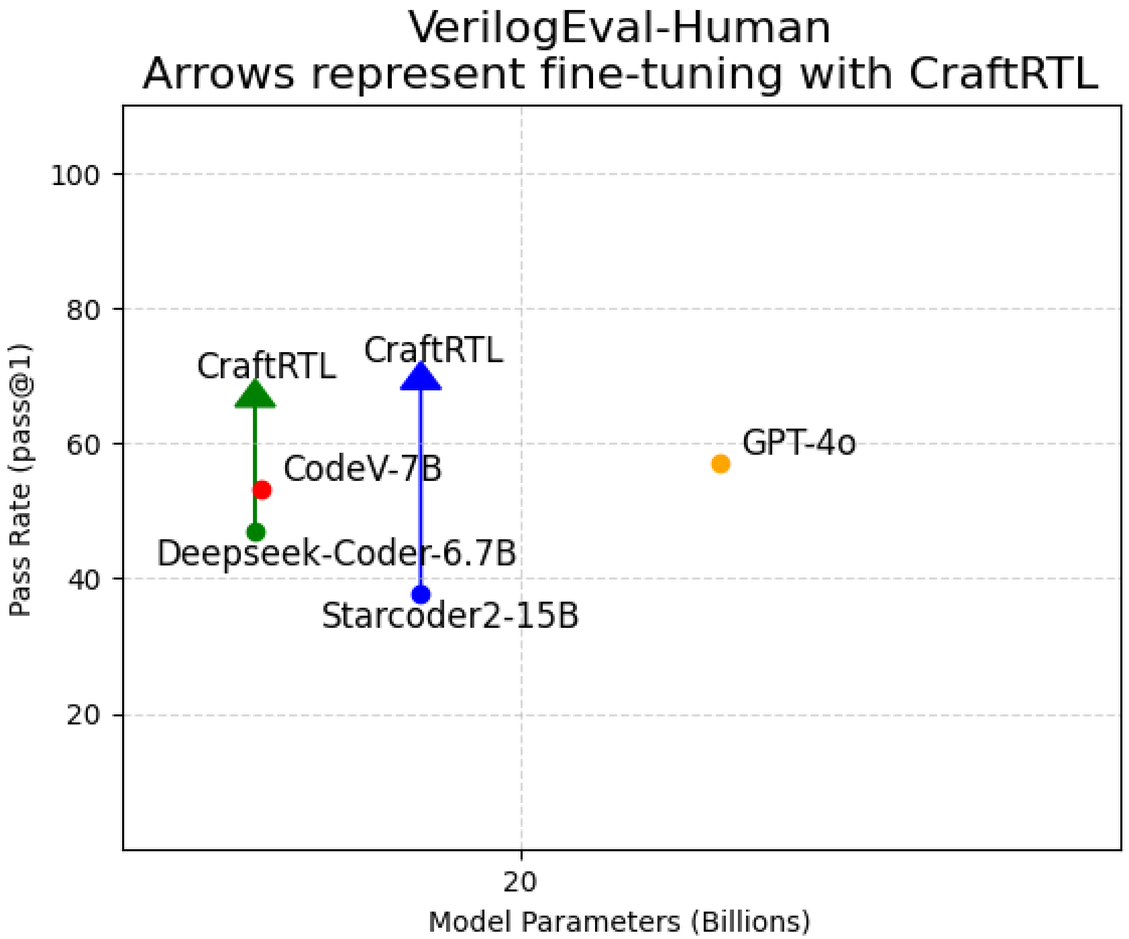
Generate error correction data by automatically analyzing common generation errors and injecting these errors to open-source code snippets:

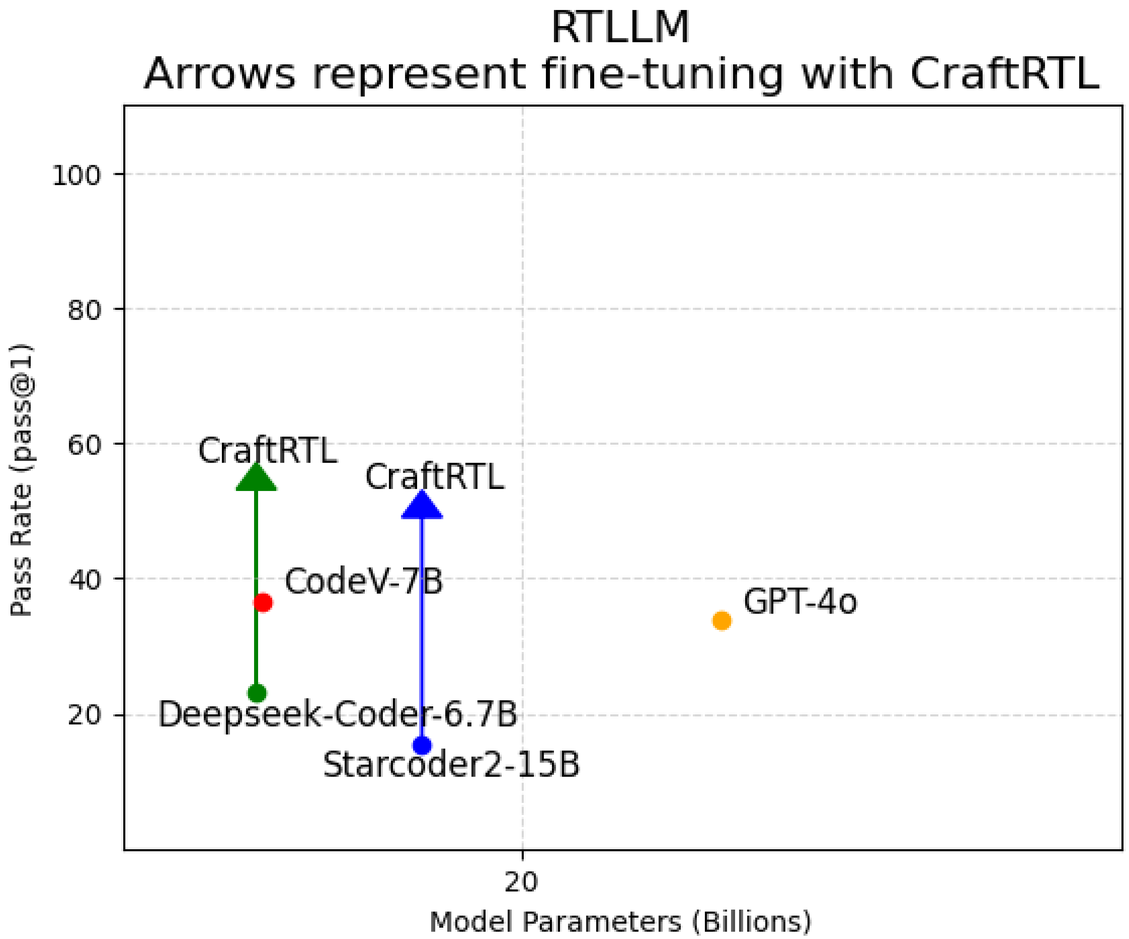
1.4K x [instruction, wrong code → correct code]

Use NVIDIA nemotron-4-340b-instruct as LLM for SDG

Mingjie Liu and Yun-Da Tsai et al., CraftRTL: High-quality Synthetic Data Generation for Verilog Code Models with Correct-by-Construction Non-Textual Representations and Targeted Code Repair

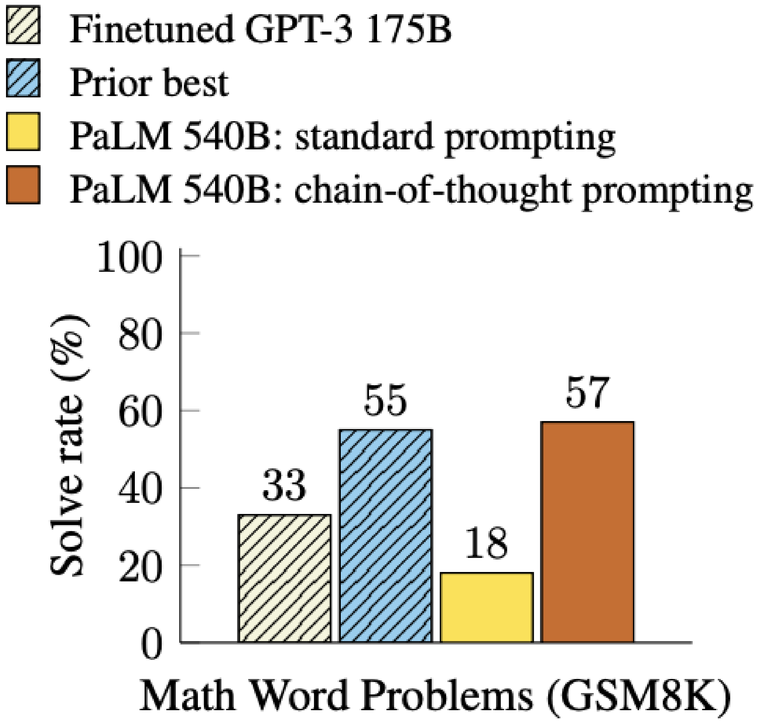
CraftRTL Models (7B/15B) were SOTA when published





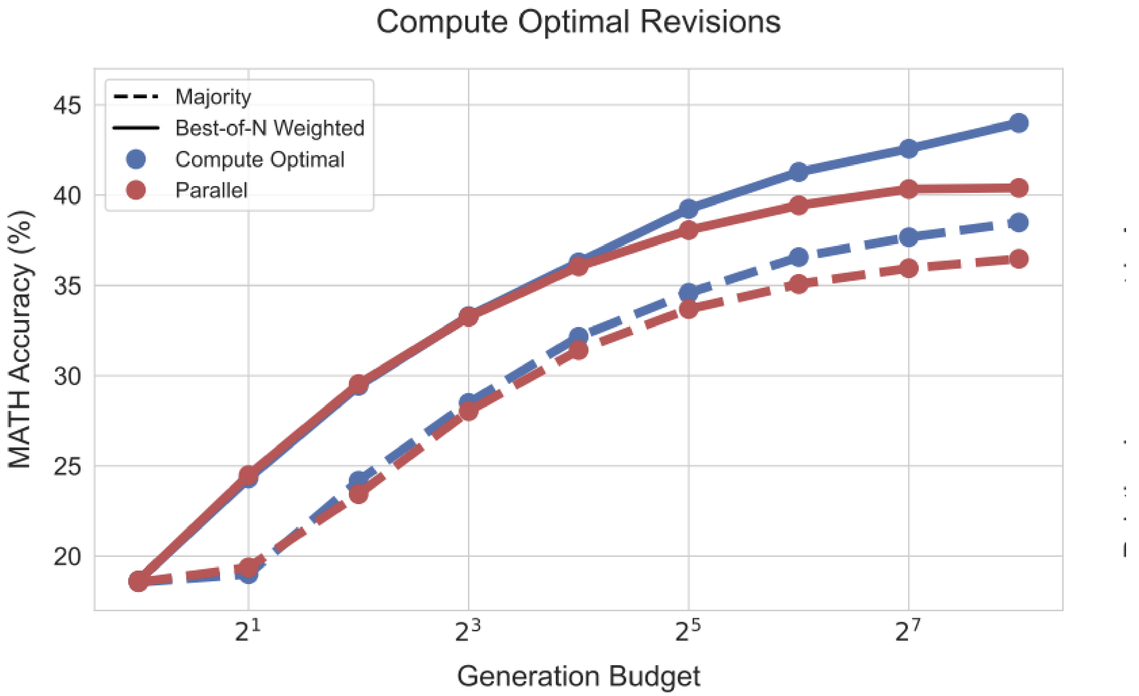
Two Ideas

Reasoning



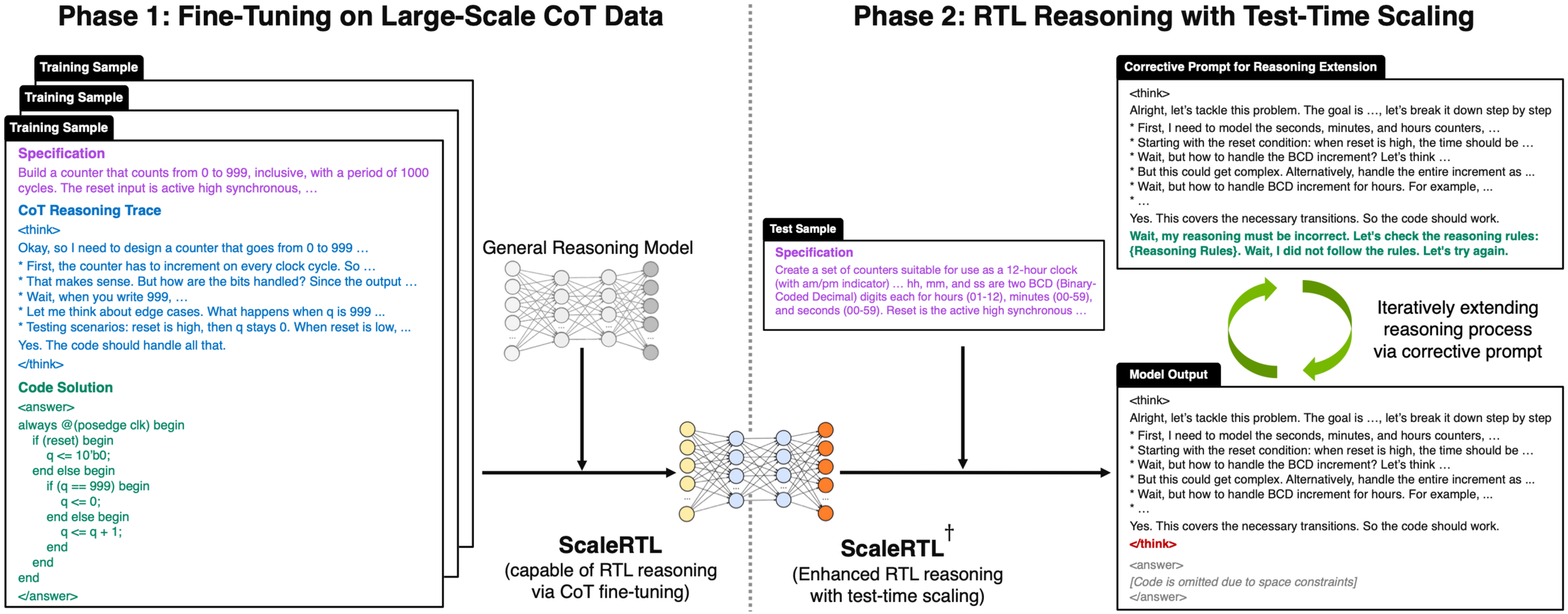
Chain-of-Thought Prompting Elicits Reasoning in Large Language Models

Test time compute



Scaling LLM Test-Time Compute Optimally can be More Effective than Scaling Model Parameters

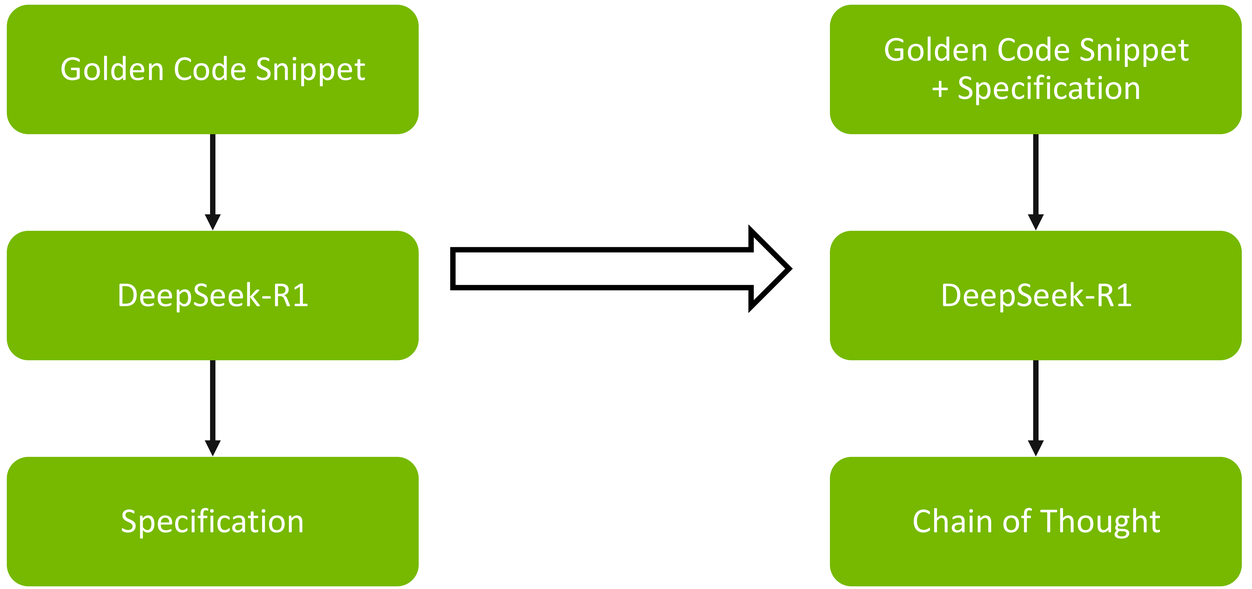
ScaleRTL: Scaling LLMs with Reasoning Data and Test Time Compute



Chain-of-Thought (CoT) Reasoning Data Curation

- High-quality RTL scripts curation
  - Collect 5M RTL scripts from public web sources
  - Obtain 62K high-quality scripts after extensive filtering
  - **Ensure no benchmark data contamination**

- Chain-of-thought (CoT) data distilled from DeepSeek-R1
  - 62K CoT data samples, with each sample containing ~56K tokens



Specification-Code Generation

CoT Data Generation

ScaleRTL (32B) is the SOTA small LLM on VerilogEval

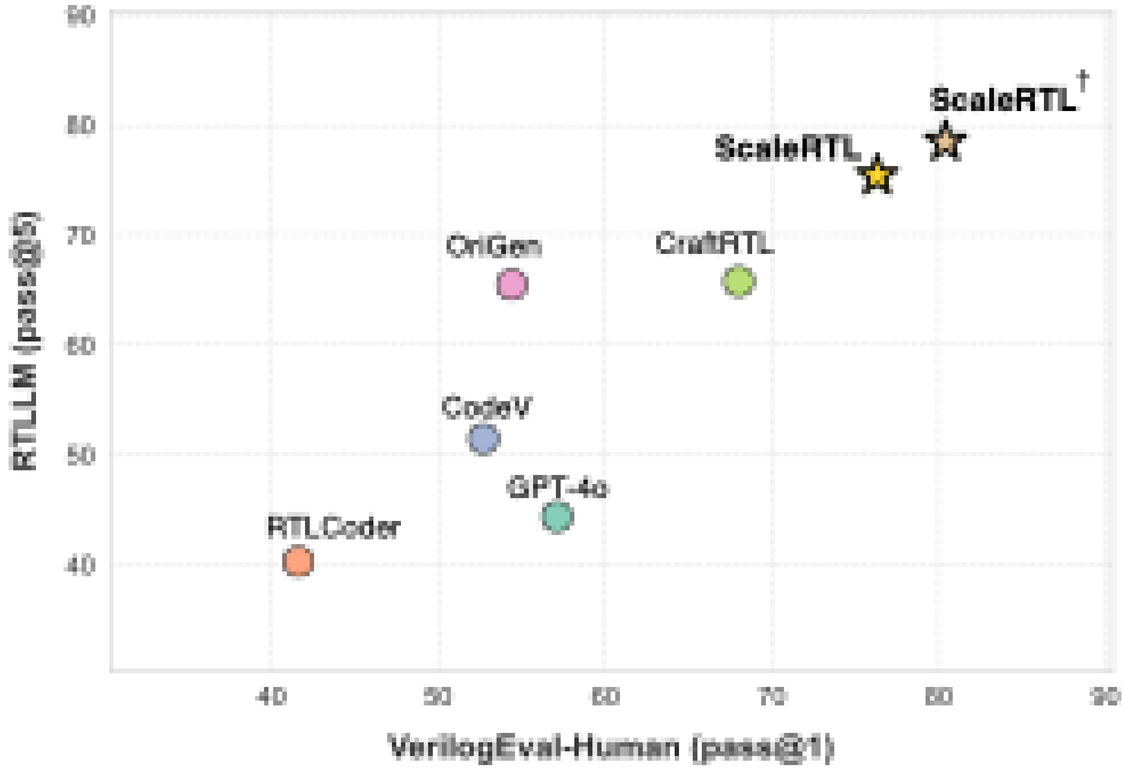
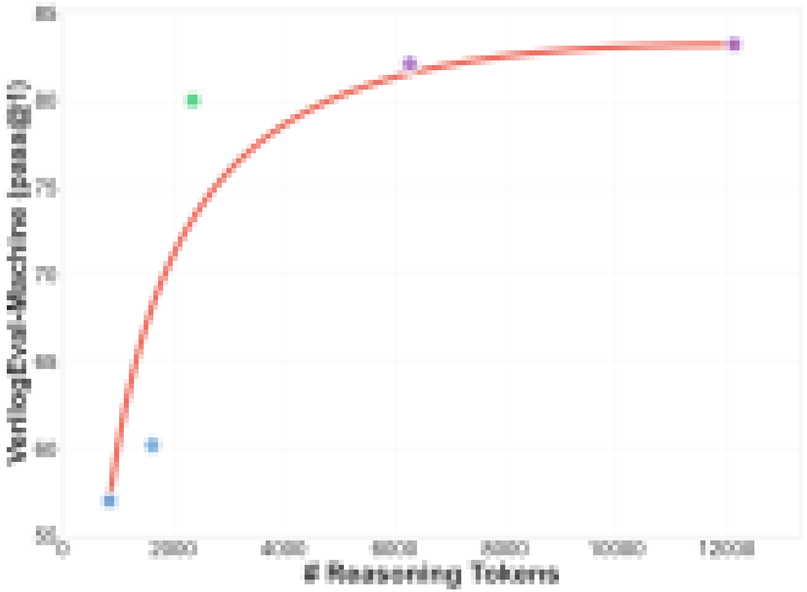


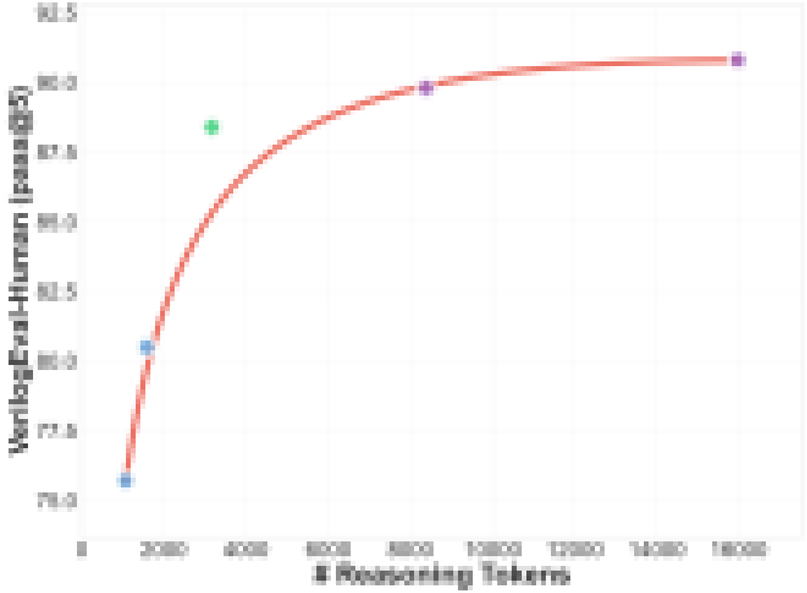
Figure 1: Comparison of LLMs on RTL Coding benchmarks — ScaleRTL† refers to the test-time scaling variant of ScaleRTL.

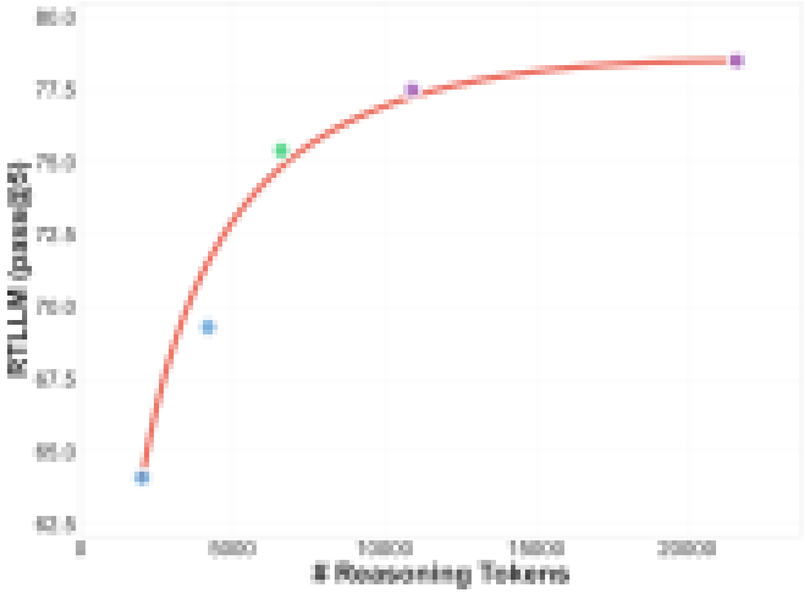
Test-Time Scaling Law for RTL Coding

Pass rate improves with more reasoning tokens

- **Blue dots**: ScaleRTL^{-} with truncated reasoning traces
- **Green dot**: ScaleRTL
- **Purple dots**: ScaleRTL† with extended reasoning traces







5 Levels of AI according to OpenAI

Stage Level 1: **Chatbots**, AI with conversational language

Stage Level 2: **Reasoners**, human-level problem solving

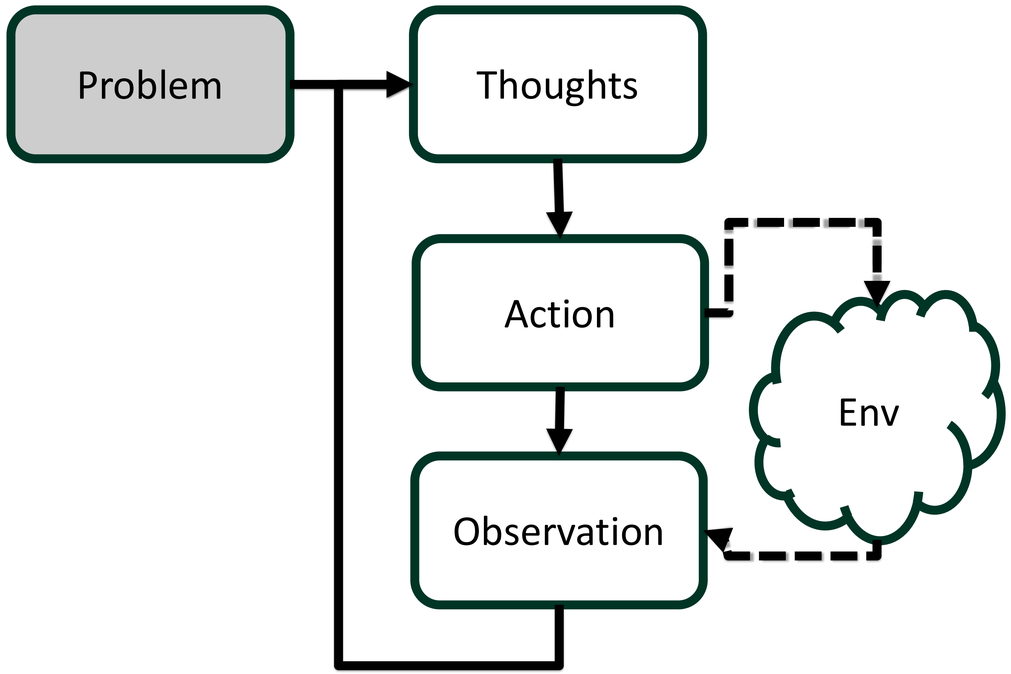
Stage Level 3: **Agents**, systems that can take actions

Stage Level 4: **Innovators**, AI that can aid in invention

Stage Level 5: **Organizations**: AI that can do the work of an organization

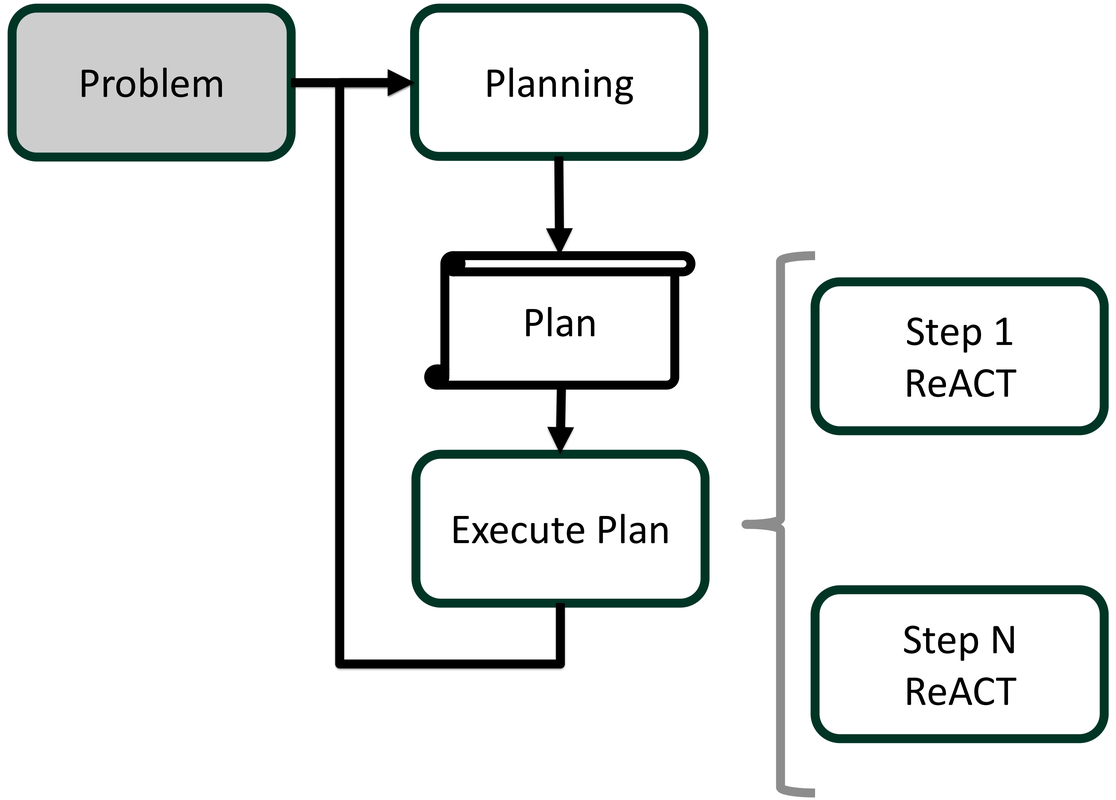
Two Types of Agents

ReACT – System 1



Think while acting - adaptive, exploratory

Plan and Execute – System 2



Think before acting – structure and efficiency

RTLFixer: fix RTL syntax errors generated by LLM with RAG

Leverage Compiler Feedback and Rules

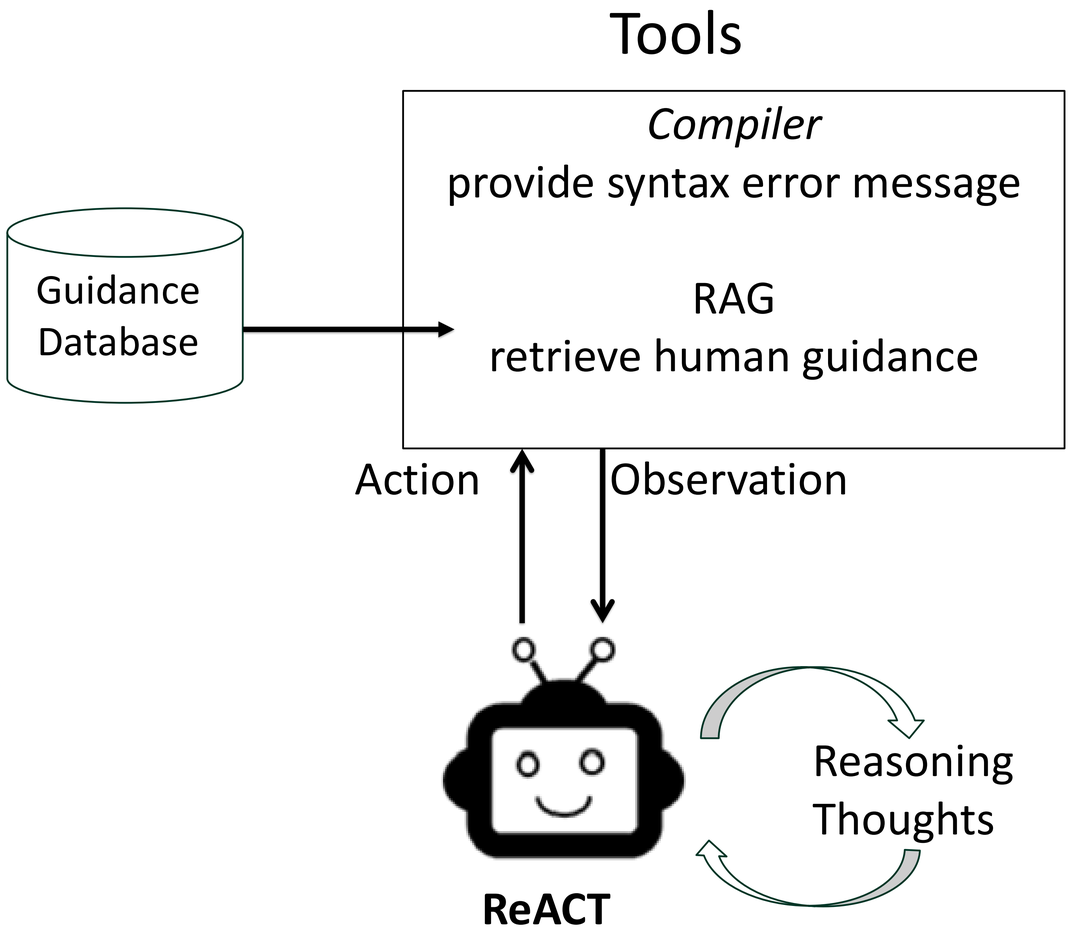
55% of GPT-3.5 Verilog generation errors are Syntax errors

RTLFixer Agent can fix 99% of Syntax Errors

Leverage **ReACT** prompting

Get feedback/error messages from Verilog compiler

Retrieve human guidance for each syntax error with **RAG**



Compiler Logs:
Object 'clk' is not declared. Verify the object name is correct. If the name is correct, declare the object.
Human Expert Guidance:
Check if 'clk' is an input. If not, and if 'clk' is used within the module, make sure the name is correct. If it's meant to trigger an 'always' block, replace 'posedge clk' with '*'.

Yun-Da Tsai, et al, RTLFixer: Automatically Fixing RTL Syntax Errors with Large Language Models

VerilogCoder

Leverage Planning and Multiple Tools

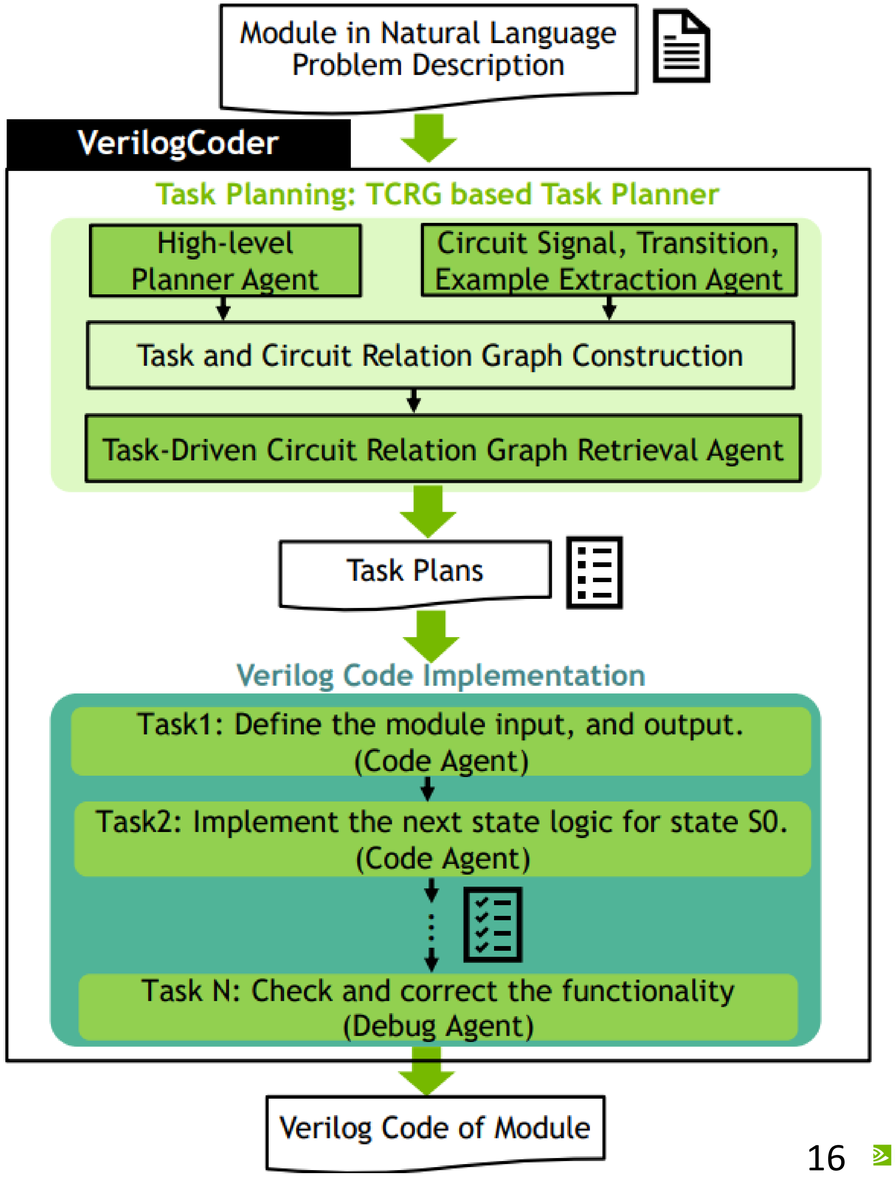
The task planning stage generates a task plan

The code implementation stage writes code for each planned task

Leverage special knowledge base and tools

- Task Planning
  - Task-Driven Circuit Relation Graph (TCRG)
- Code Implementation
  - AST-guided waveform debugging tool

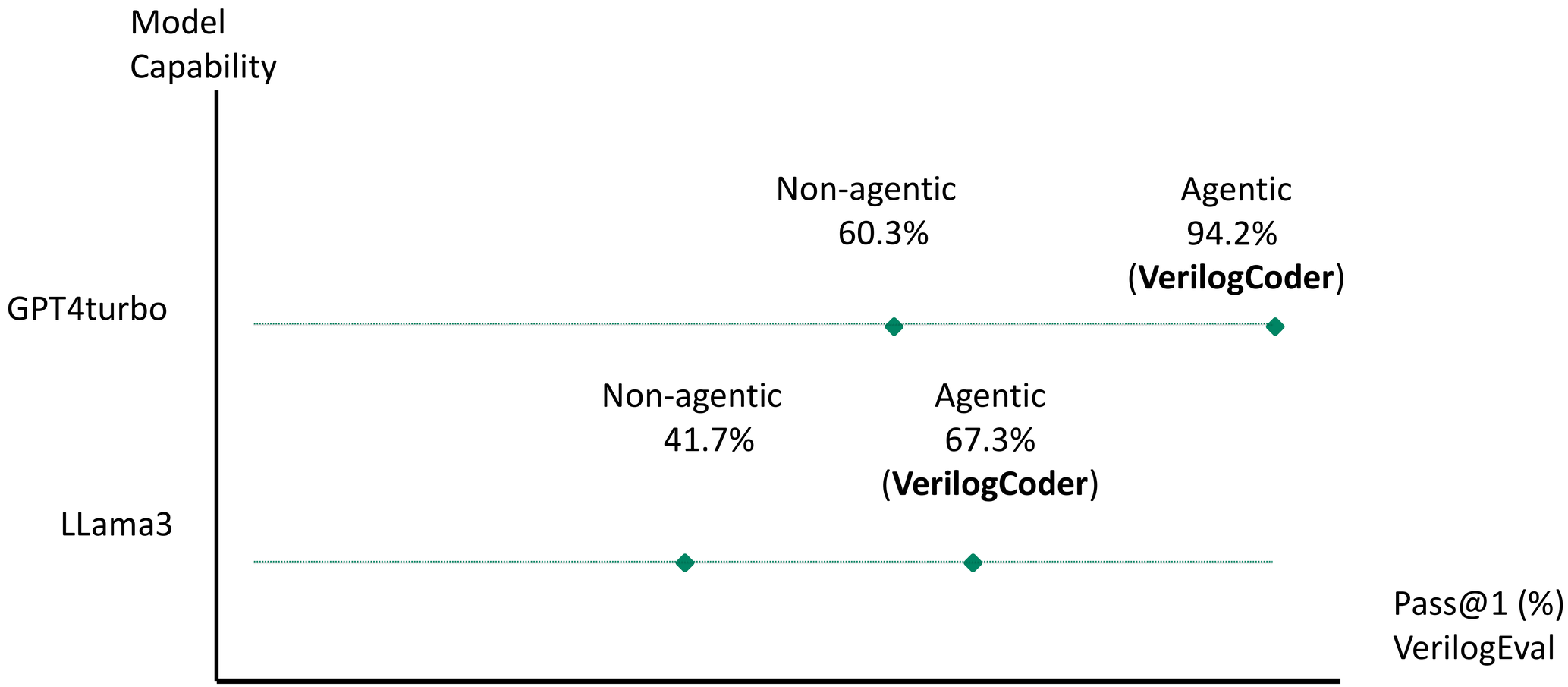
Dynamic task plan and knowledge base



Chia-Tung Ho, et al, VerilogCoder: Autonomous Verilog Coding Agents with Graph-based Planning and Abstract Syntax Tree (AST)-based Waveform Tracing Tool

VerilogCoder Agent Solves most of problems in VerilogEval

Agentic system significantly boost base model performance



Spec2RTL Agent – Complex RTL Generation

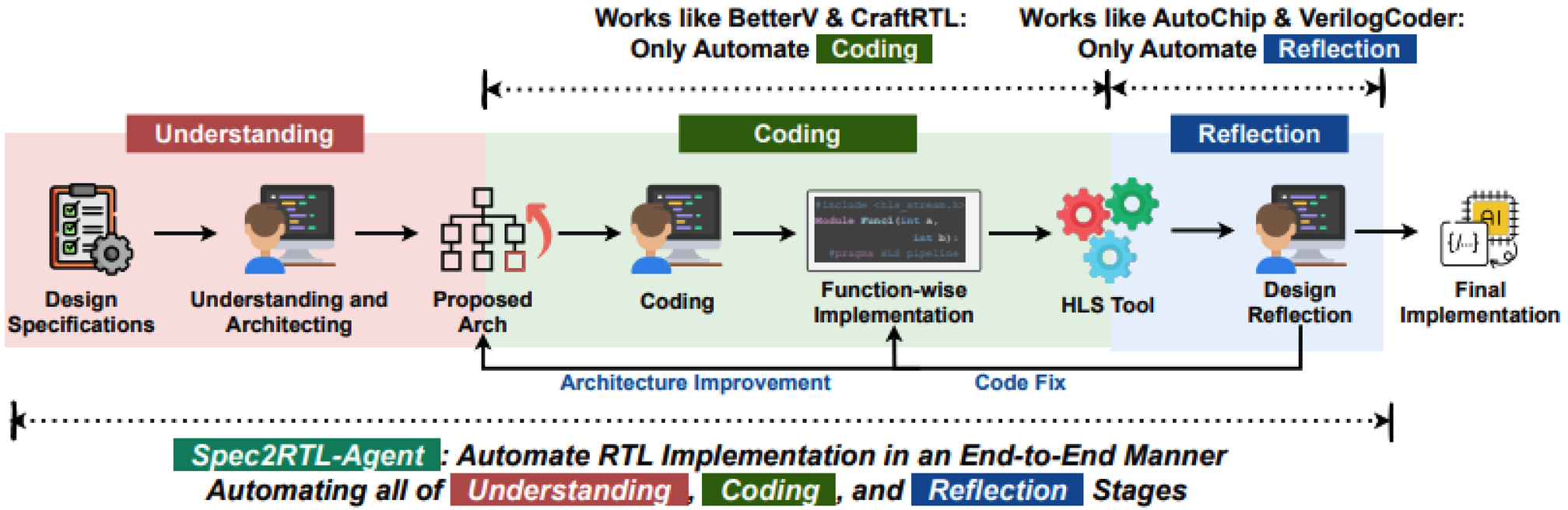
Generate RTL from AES Specification (38 pages of PDF)

A multi-agent system (Spec2RTL-Agent) that emulates the human flow: understand → code → reflect.

Progressively Code: Don’t emit RTL directly; generate Python then synthesizable C++ first, then convert via HLS for higher correctness/compatibility.

Only need 3-4 times human interactions to generate functionally correct RTL for specification such as AES, DSS, HMAC

Key is to have verifiable tests



Zhongzhi Yu, et al, Spec2RTL-Agent: Automated Hardware Code Generation from Complex Specifications Using LLM Agent Systems

Verification is a Key Task for Agentic AI

AI for Chip Design: find where most of manual design efforts are

1:3 Design and Verification Engineer Ratio

Most verification effort spent on debug

Debug effort = **bug count** * **debug time per bug**

90% effort here



Develop reference model, testbench, etc

Nightly
Directed tests for features
Random tests for corner cases

Weekly
Coverage analysis

Can AI agent to reduce bug counts and shorten debug time ?

FVDebug: LLM assist debug

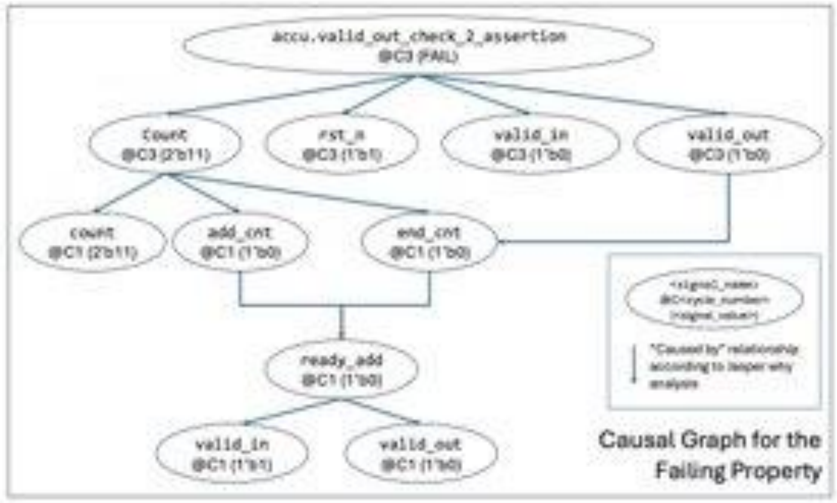
Bug localization with causal graph

Debug with causal graph generated by EDA tools : Jasper, Verdi

Agent traverse graph to locate RTL and reasoning about the bug

Support both FV and DV bugs

Solve most of bugs in the open-source datasets, and works on internal bugs we tested



Case Study on accumulator

Buggy Line

```
assign ready_add = valid_out | !valid_in;
```

Golden Answer

Fixed Line

```
assign ready_add = !valid_out | valid_in;
```

FVDebug Output

🎯 **Ranked Hypotheses**

Hypothesis 1: Ready_add Logic Issue Preventing valid_out Assertion

Confidence: 48.0%

1 Hypothesis

The valid_out signal remains low because the ready_add condition (`ready_add = valid_out OR !valid_in`) does not become true when count reaches 3, causing end_cnt to never trigger a valid_out high; this indicates an RTL bug where the handshake logic for ready_add is flawed.

2 Evidence

Supporting Evidence:

- The RTL code `'assign end_cnt = ready_add && (count == 'd3)'` implies that even if count reaches 3, end_cnt depends on ready_add being true. This supports the hypothesis if ready_add remains low, preventing valid_out from becoming high.

...

🌳 Causal Chain Timeline

...

3. **Cycle 1** 🔴

At cycle 1, with valid_in high, the node 'ready_add' evaluates to 1'b0 because it is computed as `valid_out OR !valid_in`. Since valid_in is high, !valid_in is false and valid_out is low, which blocks accumulation and consequently prevents the triggering of end_cnt and the high assertion of valid_out.

Yunsheng Bai, et al, FVDebug: An LLM-Driven Debugging Assistant for Automated Root Cause Analysis of Formal Verification Failures

PPA Optimization is a Key task for Agentic AI

AI for Chip Design: Improve PPA is the ultimate goal

- Other than verification, most design efforts in PPA optimization
- PPA optimization is another kind of debug – performance debug



Run Simulation, run hand-coded scripts, EDA tools,
Perform log analysis, spreadsheet analysis, various checks
Manual, Tedious

Modify architecture, micro-architecture and RTL
Synthesis and PD configuration tuning
Rule-based heuristic
Manual, try-and-error

Can AI agent automate and systematically optimize the PPA?

ASPEN: LLM-assisted RTL Optimization

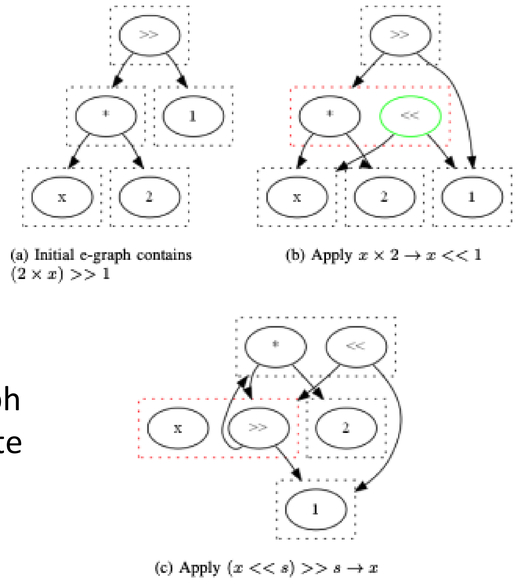
Leverage e-graph for correct-by-construction RTL rewriting

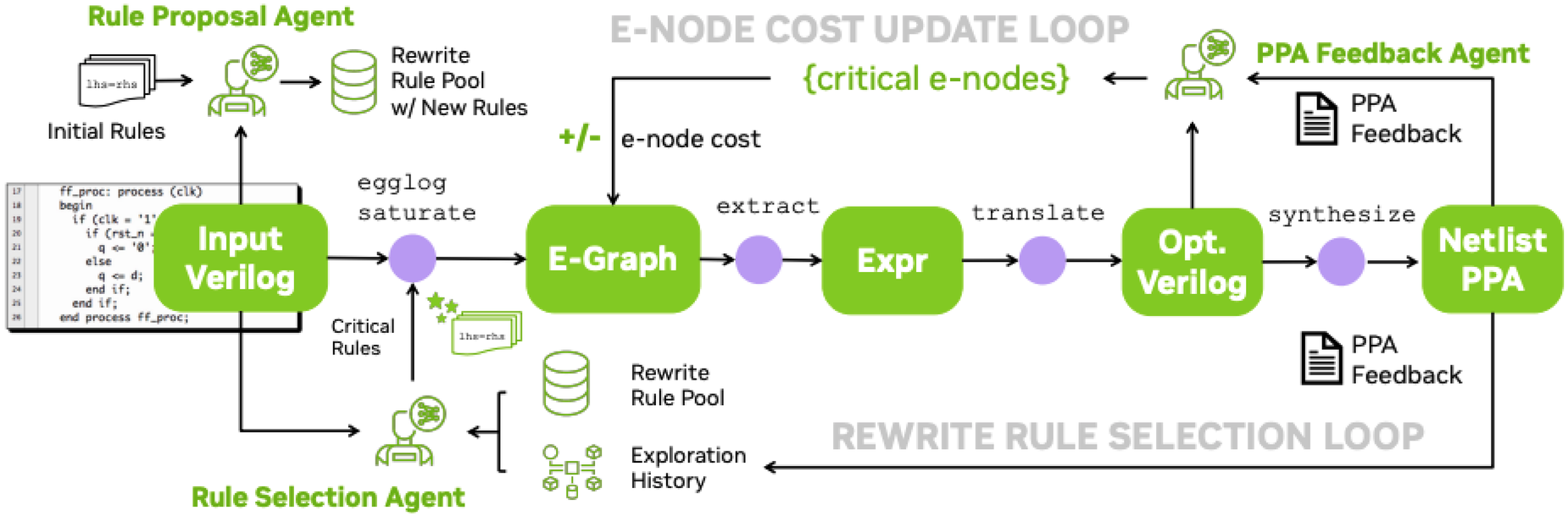
LLM assisted PPA feedback

LLM assisted rewrite rule selection and generation

Results: –24.23% area / –12.43% delay vs baseline

E-Graph Rewrite





Niansong Zhang et al, ASPEN: LLM-Guided E-Graph Rewriting for RTL Datapath Optimization

Agent for Chip Design Applications

Coding Assistance → Generating design and verification code (RTL, SVA, testbenches, EDA scripts, tools scripts, and configs)

- Generate code from specifications
- Generate code for auxiliary design tasks such as assertions, comments, etc.
- Generate lower-level programs from higher-level descriptions
- Generate scripts for specific tasks (VLSI, Verification)
- Transform code for more efficient implementation

Know-how Assistance → : Generating insights, knowledge, and ideas. Answer questions about designs, infrastructures, tools, flows, HW domains, etc.

Analysis Assistance → Summarization, report/log analysis, visualization of design and related data, check rule violations, etc.

Debug Assistance → Triage and fix a design problem, e.g. regression failures for logic design and timing failures for physical design.

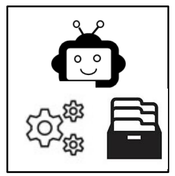
Optimization Assistance → Optimize for design and verification: coverage closure, recipe optimization, architecture exploration, etc.

Evolution of Custom Agents for Hardware Design

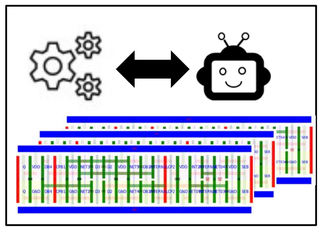
Learnings: Custom Agent Development Not Scalable

Agents

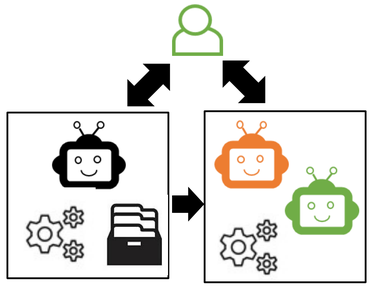
RTLFixer [DAC'24]
ReAct+ Compiler + RAG



LLM for Standard Cell Design Optimization [LAD'24 Best Paper]
ReAct+ Netlist Tools Reasoning

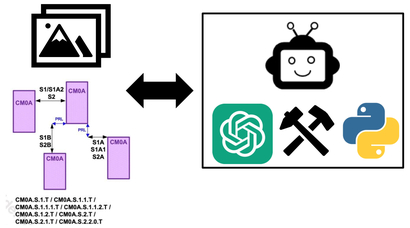


STA Report Summary Agent
Multi-LLMs + Tools Reasoning + Human Interactive

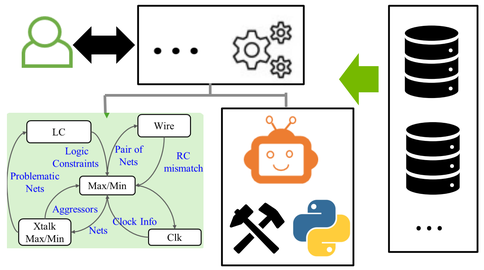


≈ 60× Speedups

DRC-Coder [ISPD 2025]
Multi-Modality & Multi-LLMs + Tools + Coding
NTECH'24

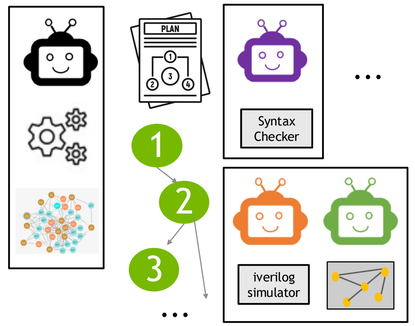


NanoTime Agent
Task Graph + Multi-LLMs + Agentic Retrieval



Reduce hours to mins

VerilogCoder [AAAI 2025]
Multi-LLMs + Dynamic Task Graph + Customized Debug Tools



More Agent for Hardware Design

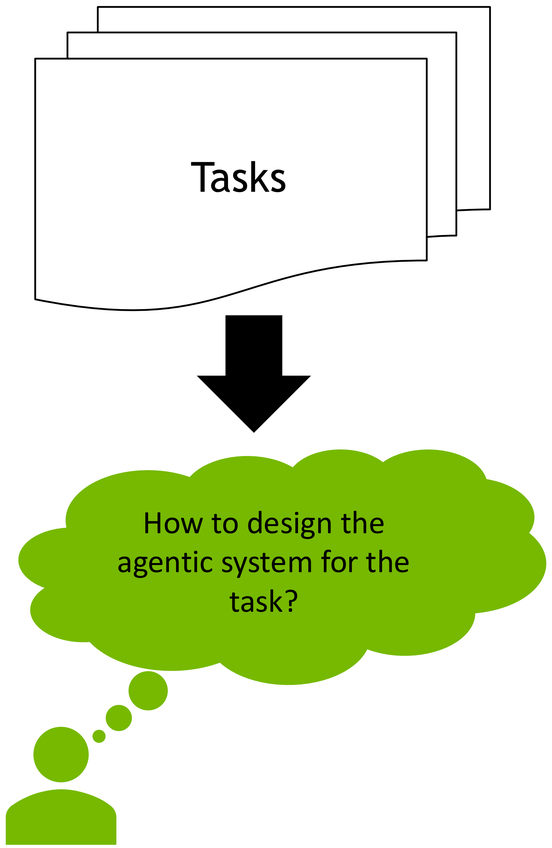
SPICE Verification Agent
ChipVQA Agent
ERC Check Agent
AssertionForge Agent [ICLAD 2025]
SPEC2RTL Agent [ICLAD 2025]
⋮

Sept, 2023

April, 2024

July, 2024

Scale Agent for a Diverse Set of Tasks



Diverse tasks: ChatBot, RTL coding, Optimization, Analysis, Debugging, etc

Multiple domains: Digital Design, Analog Design, PCB design, etc

Ad-hoc tasks

Ideally: a generalist agent to solve all the tasks

How ?

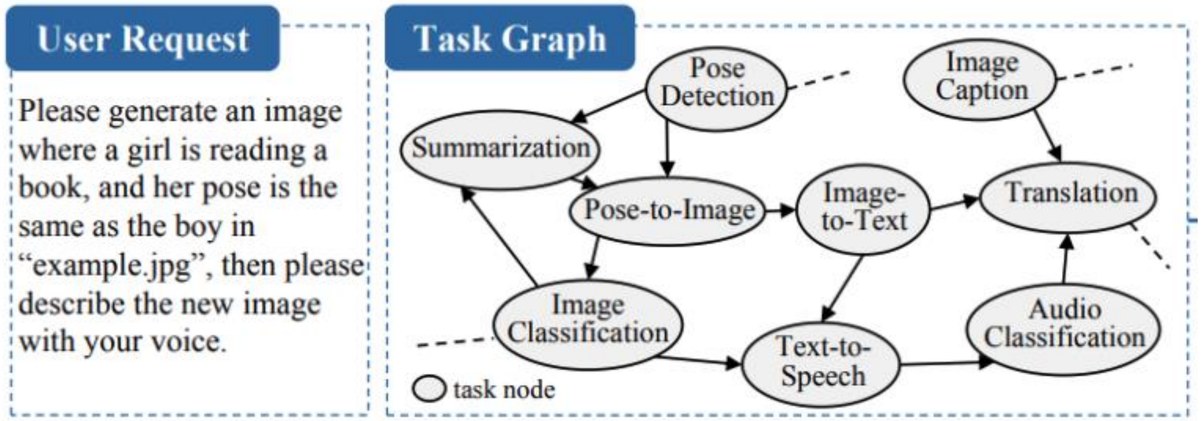
Leverage LLM **planning → How to plan ?**

Leverage LLM **optimization** → requires verification → how to **verify**

Make Good Plans

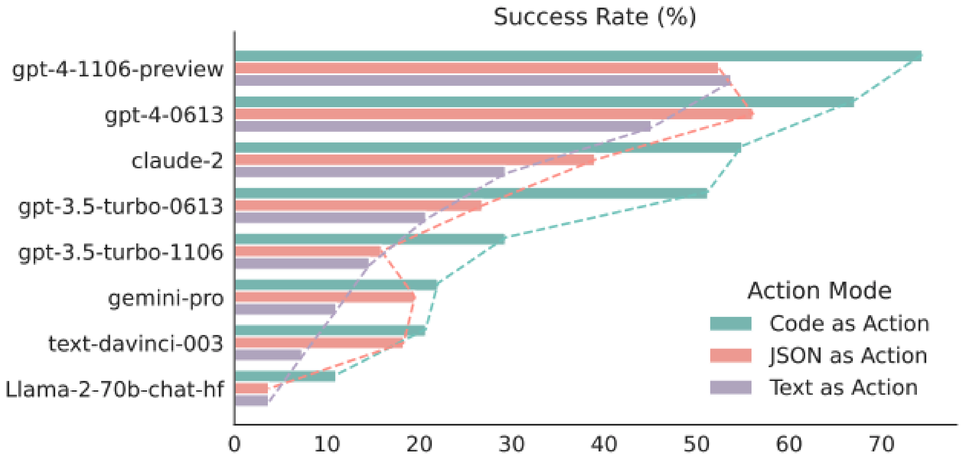
Favors structured plans

Graph as Plan



HuggingGPT

Code as Plan

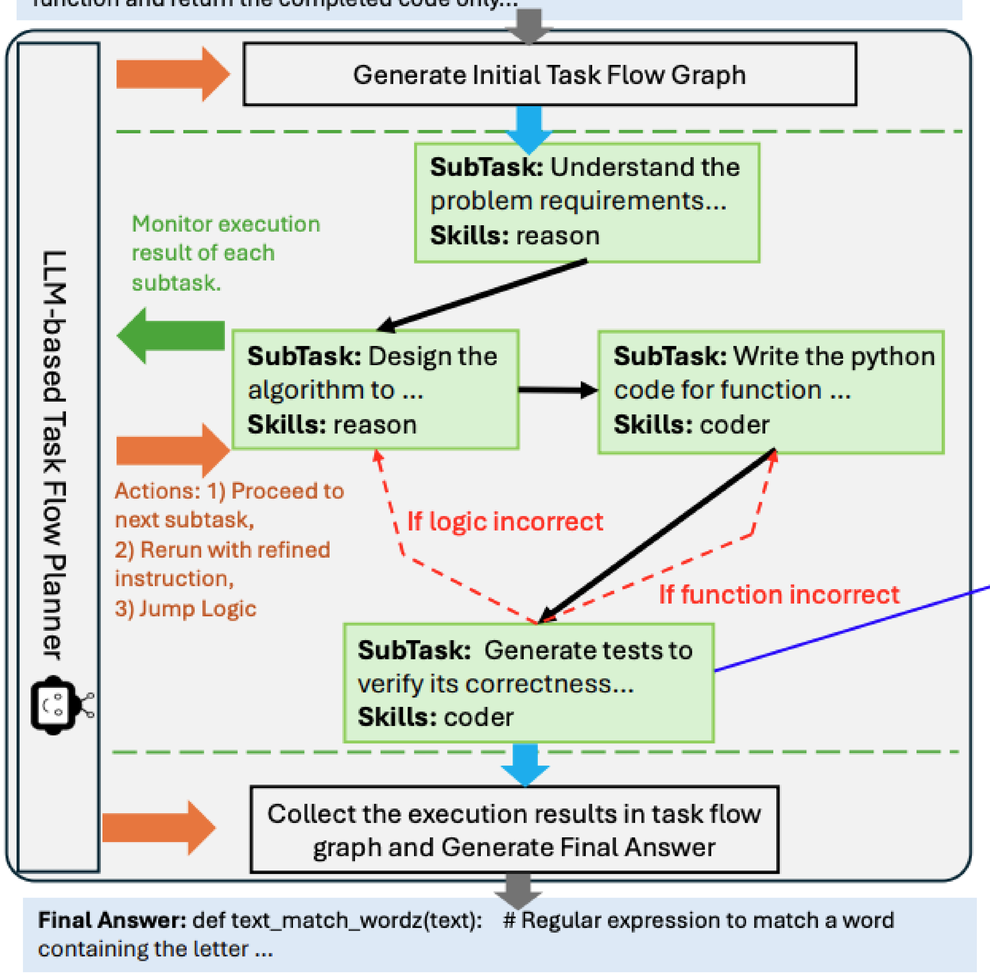


CodeAct

Polymath Agent

Hierarchical Planning with graph and code plans

(a) An example of task flow graph solving



(b) An example of subtask code-represented workflow

SubTask: Generate tests to verify its correctness...

```
def workflow(instruction: str):
    instruction = "### Instruction Background ###\n" + instruction + "\n"
<<<<<<< SEARCH
    ... # your designed workflow here.
=======
    import json

    # Task_3: Test the text_match_wordz function
    # Test inputs for the function
    test_cases = [
        "apple",
        "zebra",
        "buzz",
        "fizz",
        "hello"
    ]
    # Expected outputs: [False, True, True, True, False]
    expected_outputs = [False, True, True, True, False]

    # Chat with coder to run tests on function
    coder_instruction = f"Test the text_match_wordz function with inputs {test_cases}.\\n"
    coder_instruction += "Ensure the function returns the correct boolean values indicating the presence of 'z' in words.\\n"
    coder_instruction += "Return True for words containing 'z' and False otherwise.\\n"
    response = coder_assistant.initiate_chat(message=instruction + coder_instruction)

    # Output the test results
>>>>>>> REPLACE
    return response
```

Chia-Tung Ho et al, Polymath: A Self-Optimizing Agent with Dynamic Hierarchical Workflow

“

The Verification Principle: An AI system can create and maintain knowledge only to the extent that it can verify that knowledge itself.

Verification: The Key to AI -- Rich Sutton, 2001

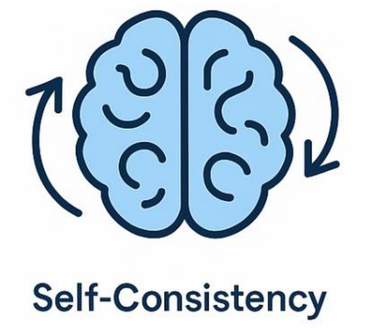
”

Verification Methods

How to verify without ground truth?



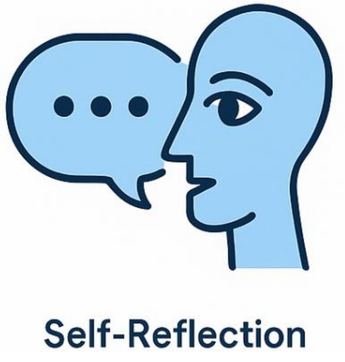
Strength: High reliability
Limitation: Needs tool access / external sources.



Strength: Easy to implement; improves reliability in math, logic, QA.
Limitation: Wasteful if all samples share the same flaw.



Strength: Domain-agnostic; scalable.
Limitation: Critic may share biases with generator.

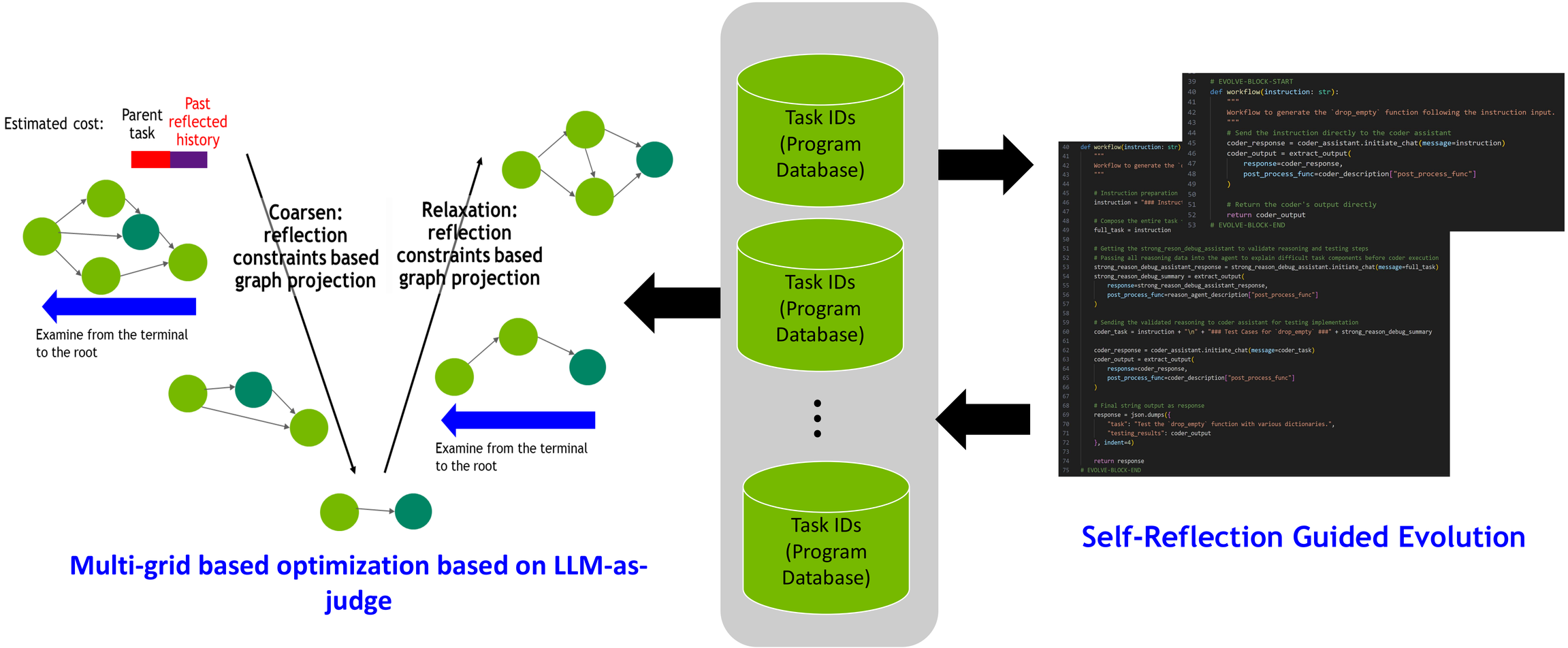


Strength: Agents improve across attempts; lightweight adaptation.
Limitation: Effectiveness depends on quality of feedback signal.

Optimize the Plan with Verification

Top-level
Divide & Conquer: Manageable Subtask

Subtask/Coding-level (Evolutionary Alg.)
Optimize subtask execution & performance




```python
# EVOLVE-BLOCK-START
def workflow(instruction: str):
    """
    Workflow to generate the `drop_empty` function following the instruction input.
    """
    # Send the instruction directly to the coder assistant
    coder_response = coder_assistant.initiate_chat(message=instruction)
    coder_output = extract_output(
        response=coder_response,
        post_process_func=coder_description["post_process_func"]
    )

    # Return the coder's output directly
    return coder_output
# EVOLVE-BLOCK-END
```

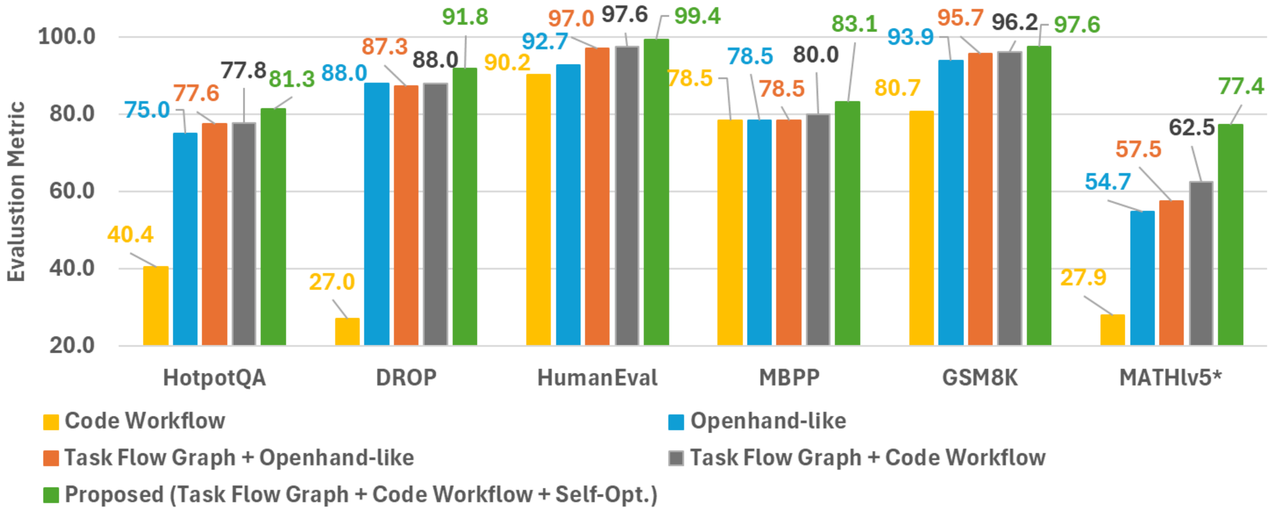
Multi-grid based optimization based on LLM-as-judge

Self-Reflection Guided Evolution

Performance Comparison

Method	Multi-Turn QA		Coding		Math		Avg.
	HotpotQA	DROP	HumanEval	MBPP	GSM8K	MATH	
gpt-4o (Vanilla)	75.0	64.7	91.5	74.9	85.5	48.2	73.3
o1 model (Vanilla)	70.6	84.9	89.0	74.5	94.6	67.1	80.1
CoT (Wei et al. 2022)	67.9	78.5	88.6	71.8	92.4	48.8	74.7
CoT SC (5-shot) (Wang et al. 2022)	68.9	78.8	91.6	73.6	92.7	50.4	76.0
MultiPersona (Wang et al. 2023)	69.2	74.4	89.3	73.6	92.8	50.8	75.1
Self Refine (Madaan et al. 2023)	60.8	70.2	87.8	69.8	89.6	46.1	70.7
ADAS (Hu et al. 2024)	64.5	76.6	82.4	53.4	90.8	35.4	67.2
AFlow (Zhang et al. 2024a)	73.5	80.6	94.7	83.4	93.5	56.2	80.3
Proposed	81.3	91.8	99.4	83.1	97.6	77.4	88.4

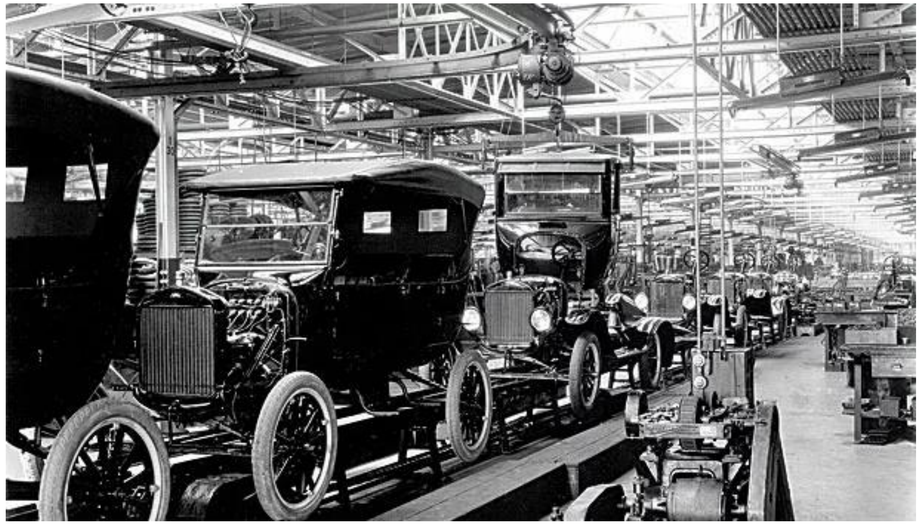
We use GPT-4o as the core model in Polymath, supported by a set of assistants: **a coder assistant** (GPT-4o), a **reasoning assistant** (o1), **an image reader** (GPT-4o), and **a file reader** (GPT-4o)

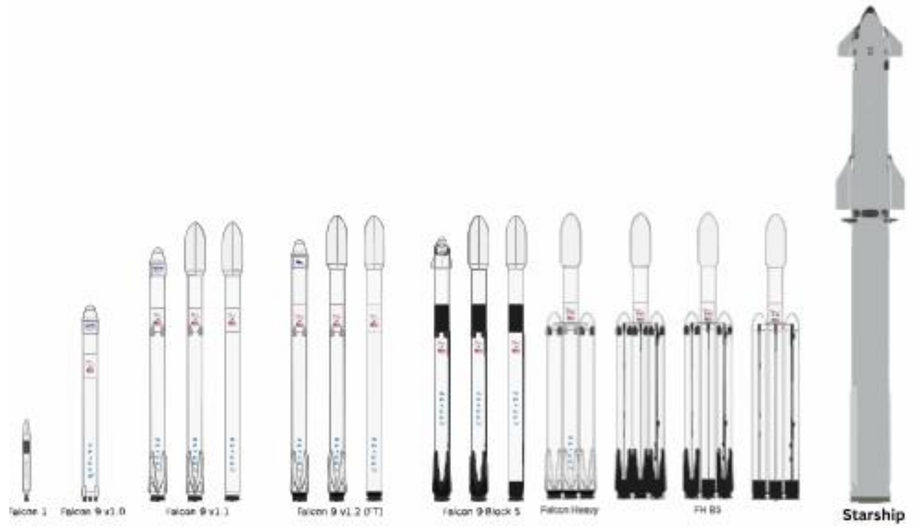


QA, Coding and Math Task

Challenges and Opportunities

- How to cope with large legacy code base?
- How to re-define arch/micro-arch spec for AI to generate code
- How to effectively program domain-specific languages used in hardware design?
- How to effectively teach LLM/Agent domain knowledge?
- How to effectively use existing EDA tools – Agentic EDA
- Should EDA tools change since users are Agents instead of human?
- How to modify design methodology to make it suitable for AI instead of Human?
- How to effectively organize teams to cowork with Agents





Takeaways

AI for hardware is lagging behind AI for software

SDG, reasoning, test time compute help fine-tune LLM for RTL code generation

Common agent types and how agents improve RTL generation accuracy over foundation models

LLM helps understand long specifications and progressively generate Python and C++ code for HLS

LLM can assist collateral generation for RTL optimization with PPA feedback

LLM can assist debug with signal tracing by EDA tools

Generalist agent with self-optimization capabilities can be applied to a diverse set of tasks

Verification techniques for agent self-optimization

AI for hardware design future is bright and will happen fast

