2024
DESIGN AND VERIFICATION™
DVCON
CONFERENCE AND EXHIBITION
UNITED STATES
SAN JOSE, CA, USA
MARCH 4-7, 2024

Large Language Model for Verification: A Review and Its Application in Data Augmentation

Dan Yu, Eman El Mandouh, Waseem Raslan, Harry Foster,
Tom Fitzpatrick - Siemens Industry Software Inc.

SIEMENS

accellera
SYSTEMS INITIATIVE

Motivation

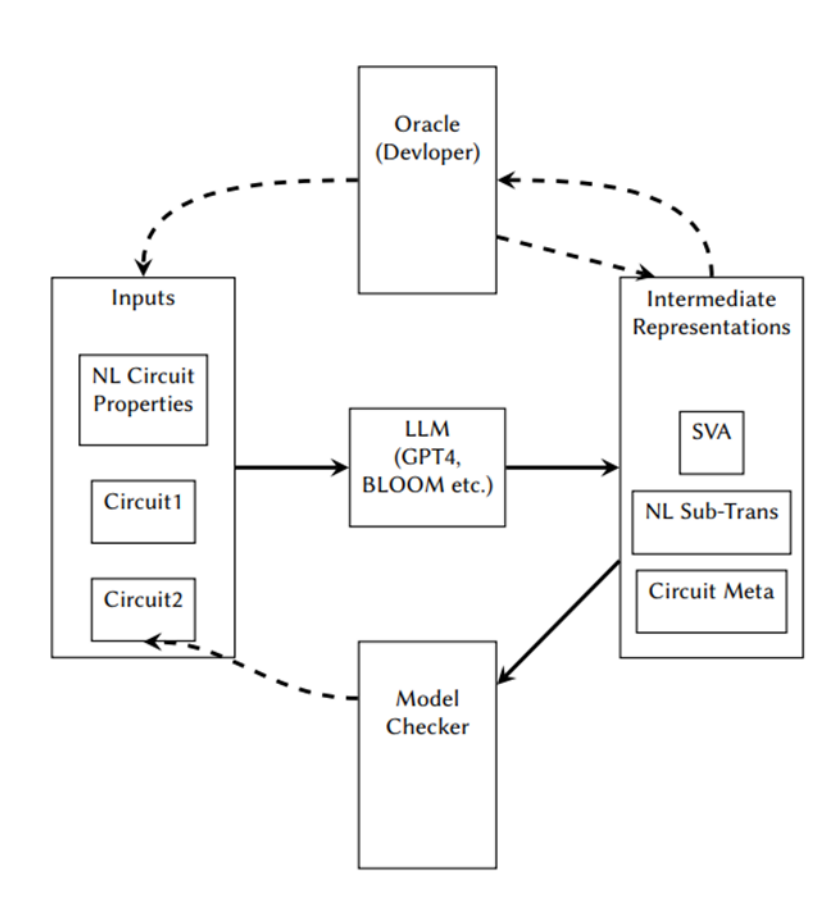
Newest development of ML in the last 2 years
Small application area w/ scarcity of research results
Transfer pretrained knowledge to EDA verification
Augment data for EDA ML works
Paradigms to apply LLM to improve quality for production

LLM Applications in Verification

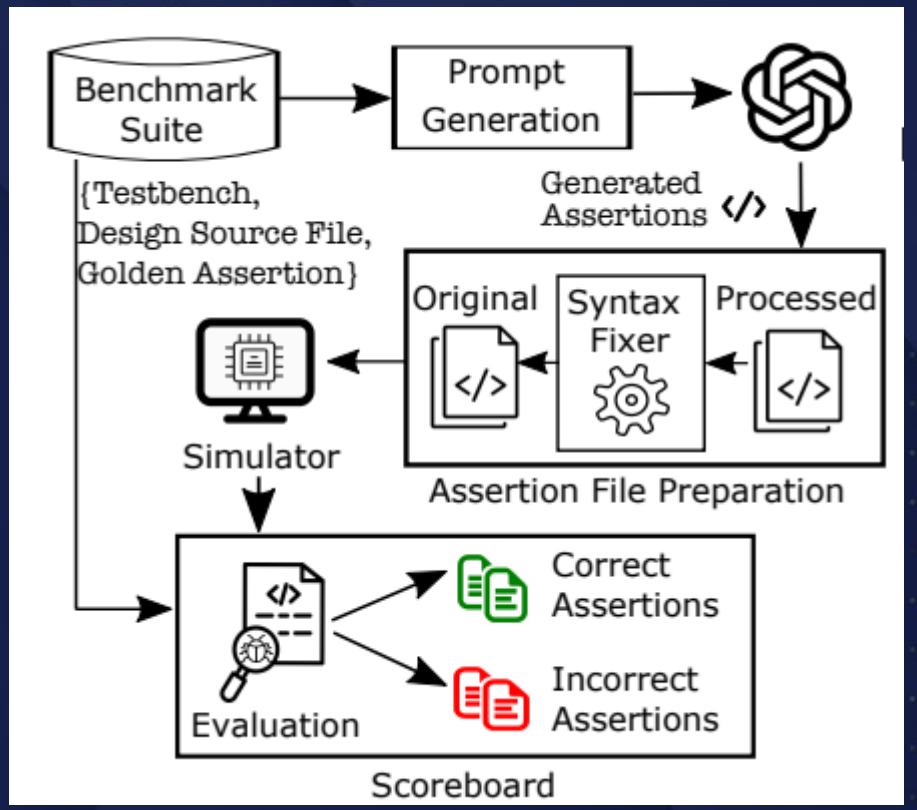
- Assertion generation
- Coverage closure
- Formal verification
- Debugging
- Test stimulus generation
- Functional safety and security
- Code generation & completion

accellera
SYSTEMS INITIATIVE

Review: Assertion Generation



Sun, Chuyue, et al.. "Towards Improving Verification Productivity with Circuit-Aware Translation of Natural Language to SystemVerilog Assertions." *First International Workshop on Deep Learning-aided Verification*. 2023.



only 9.29% of the generated assertions are correct

Kande, Rahul, et al. "LLM-assisted Generation of Hardware Assertions." *arXiv preprint arXiv:2306.14027* (2023).

accellera SYSTEMS INITIATIVE

Review: Coverage Closure

Focal Method $\{m\}$

```
public String foo(int x){
    if(x == 0){
        return "zero";
    } else if(x > 0){
        return "positive";
    } else {
        return "negative";
    }
    return "impossible";}
```

Test Case $\{t\}$

```
public void testFoo() {
    String res = foo(2);
    Assert.isEqual("positive", res);}
```

Coverage-Annotated Method $\{\text{cov}(m,t)\}$

```
>  public String foo(int x){
>      if(x == 0){
!          return "zero";
>      } else if(x > 0){
>          return "positive";
!      } else {
!          return "negative";
!      }
-      return "impossible";}
```

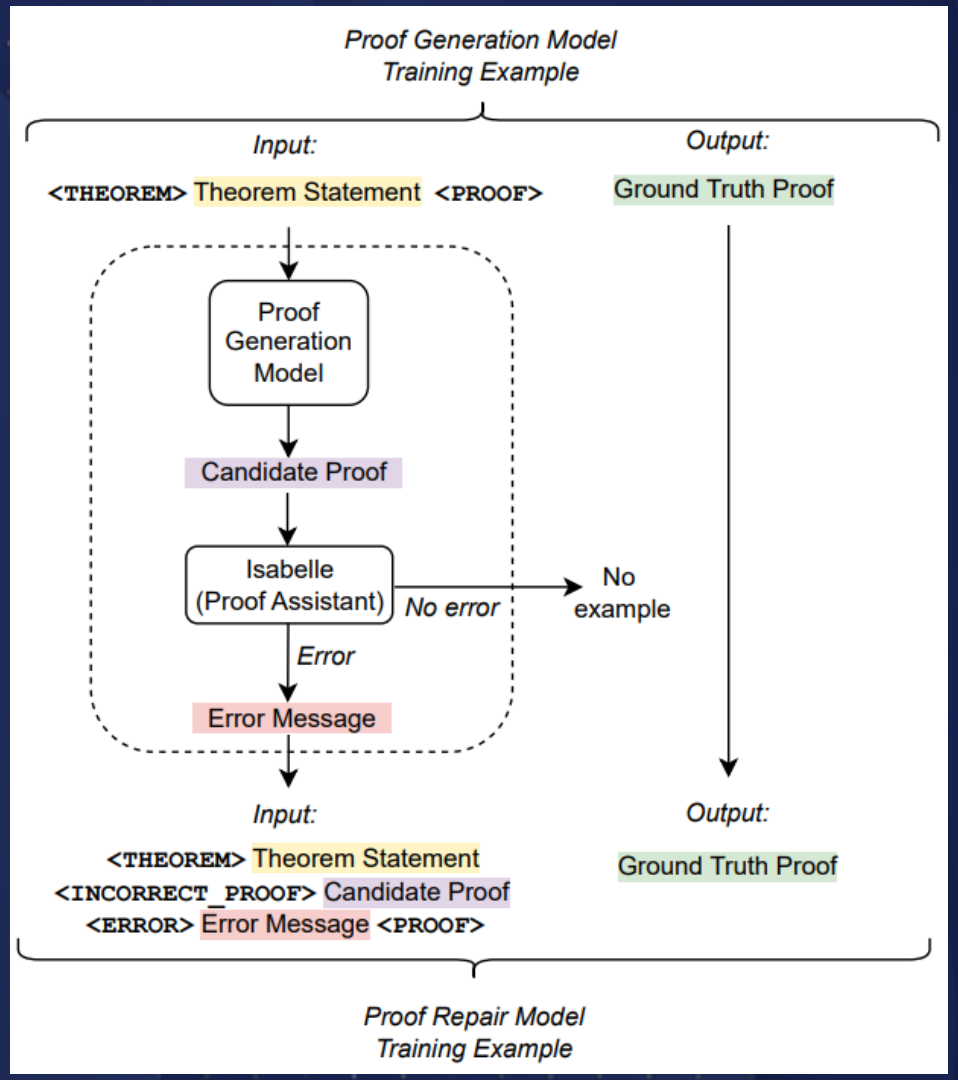
Not EDA, but applicable

Model	zero-shot			one-shot			multi-shot		
	Match	Stmt	Branch	Match	Stmt	Branch	Match	Stmt	Branch
OpenAI GPT-4 (gpt-4)	**25.75**	**84.47**	**20.16**	**22.85**	**90.71**	**22.65**	**30.04**	**90.5**	**22.5**
OpenAI GPT-3.5 (gpt-3.5-turbo)	0	39.87	8.33	8.17	76.53	17.17	11.03	82.29	17.9
Google BARD (text-bison-001)	0	81.27	17.21	1.87	86.93	19.63	21.56	85.66	20.52
Anthropic Claude (claude-1.3)	3.9	84.47	20.07	4.83	83.21	19.16	6.88	55.7	12.23

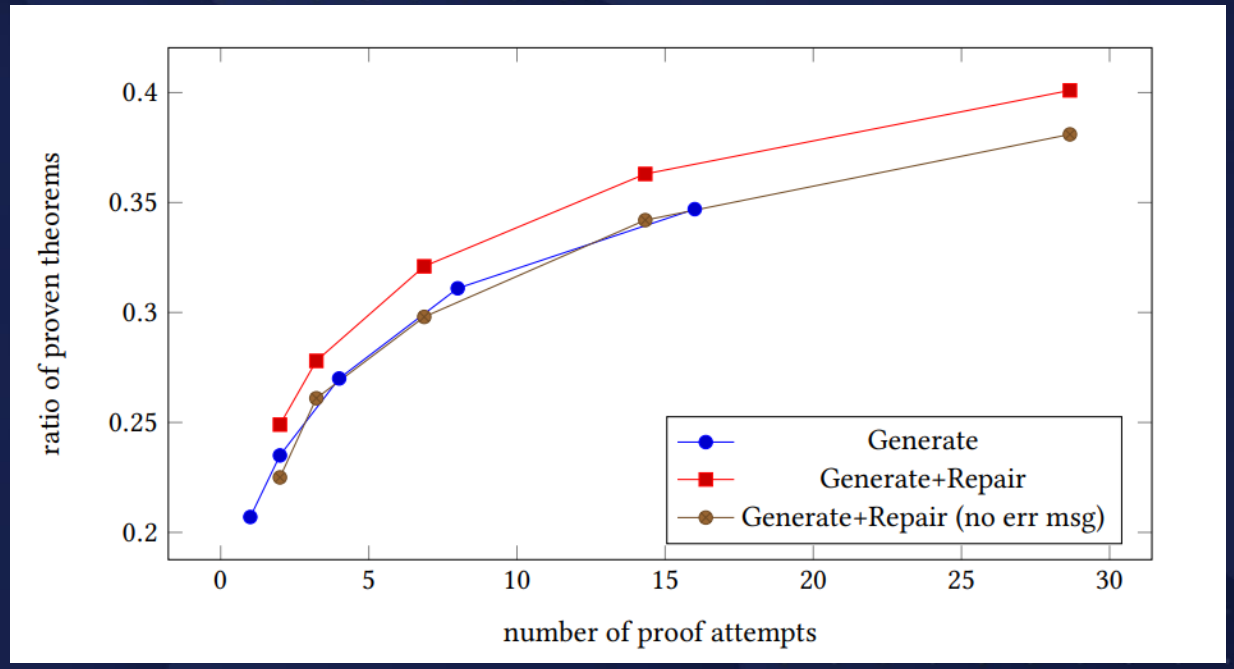
Tufano, Michele, et al. "Predicting Code Coverage without Execution." *arXiv preprint arXiv:2307.13383* (2023).

accellera SYSTEMS INITIATIVE

Review: Formal Verification



LLM w/ 700m parameters fine tuned on proofs
Combine with classic automatic proving tools 67.5%

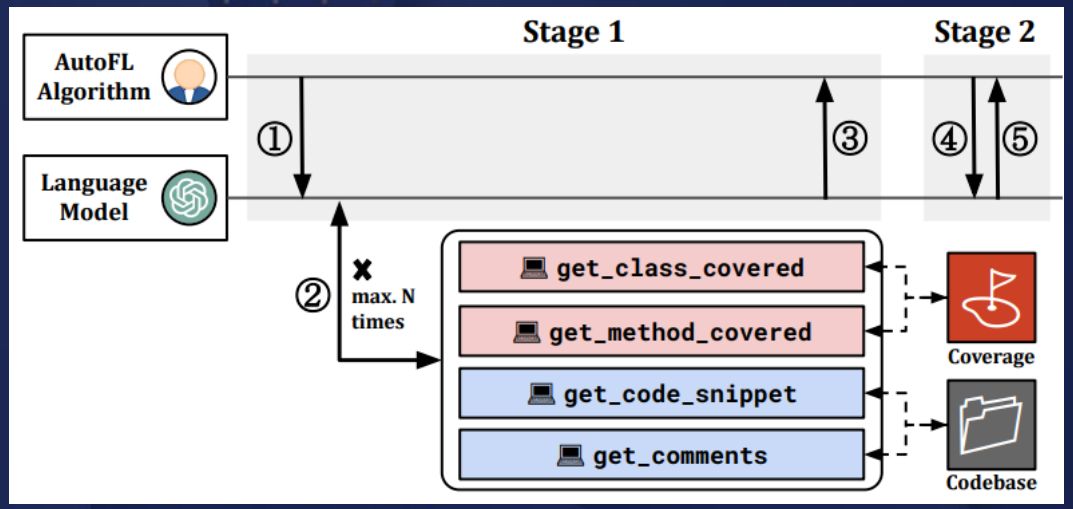


First, Emily, et al. "Baldur: whole-proof generation and repair with large language models." *arXiv preprint arXiv:2303.04910* (2023).

accellera SYSTEMS INITIATIVE

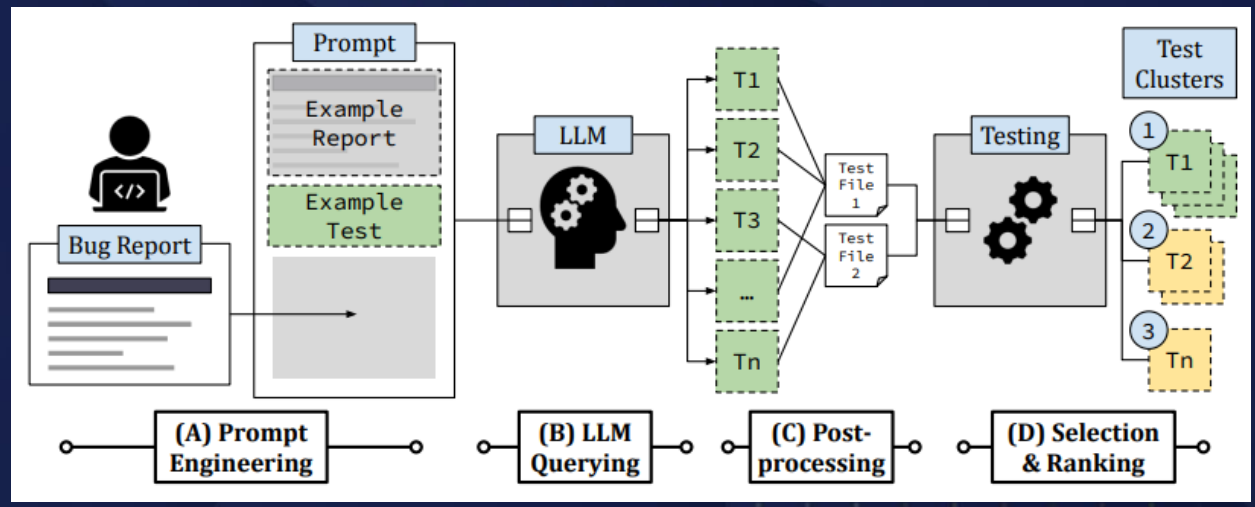
Review: Debugging

40% more new bugs compared to the best classic approaches at acc@1 on Defects4J



Kang, Sungmin, et al. "A Preliminary Evaluation of LLM-Based Fault Localization." *arXiv preprint arXiv:2308.05487* (2023).

successfully reproduce 33.5% of all bugs in the benchmark dataset



Kang, Sungmin, et al. "Large language models are few-shot testers: Exploring llm-based general bug reproduction." 2023 *IEEE/ACM 45th International Conference on Software Engineering (ICSE)*. IEEE, 2023.

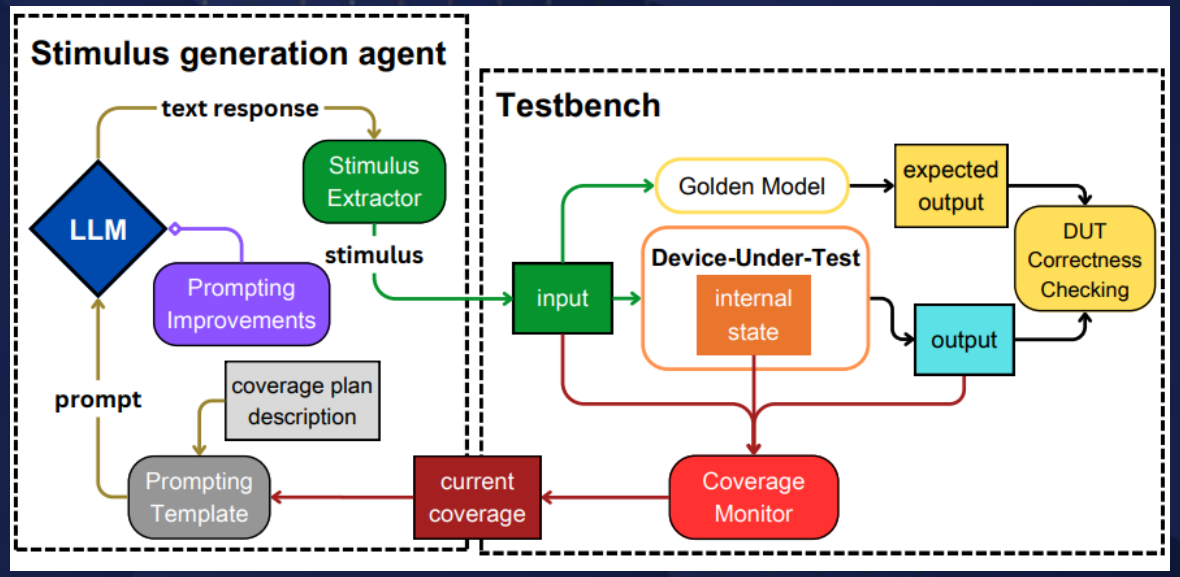
accellera SYSTEMS INITIATIVE

Review: Test Stimulus Generation

Primitive Data Prefetcher Core
- Code coverage: 98.94%

Ibex CPU Instruction Decoder
- Increased Complexity, Decreased Results
- Code coverage: 86.19%

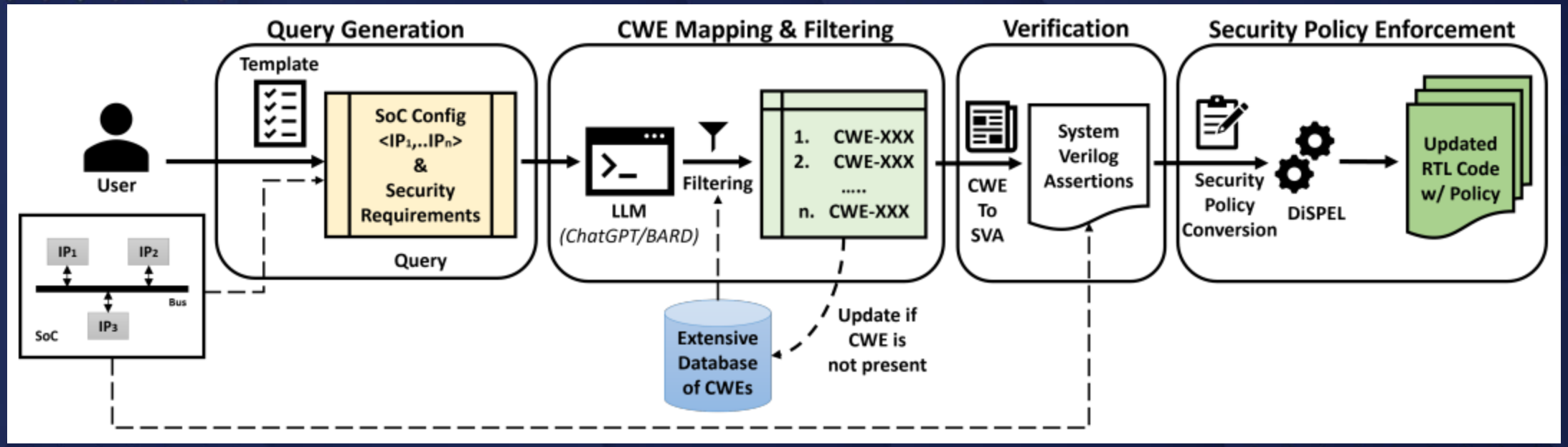
Ibex CPU Design
- Code coverage only 5.61%



Zhang, Zixi, et al. "LLM4DV: Using Large Language Models for Hardware Test Stimuli Generation." *arXiv preprint arXiv:2310.04535* (2023).

accellera SYSTEMS INITIATIVE

Review: Functional Safety and Security



40%-55% in generating relevant CWEs

Paria, Sudipta, et al. "DIVAS: An LLM-based End-to-End Framework for SoC Security Analysis and Policy-based Protection." *arXiv preprint arXiv:2308.06932* (2023).

accellera SYSTEMS INITIATIVE

Review: Code Generation and Completion

Model	Size	Python		Java		JavaScript	
		PSM	SPM	PSM	SPM	PSM	SPM
InCoder	6B		31.0%		49.0%		51.0%
SantaCoder	1.1B		44.0%		62.0%		60.0%
StarCoder	15.5B		62.0%		73.0%		74.0%
Code Llama	7B	67.6%	72.7%	74.3%	77.6%	80.2%	82.6%
	13B	**68.3%**	**74.5%**	**77.6%**	**80.0%**	**80.7%**	**85.0%**

For general code generation
Not specifically tailored for RTL code

Roziere, Baptiste, et al. "Code llama: Open foundation models for code." *arXiv preprint arXiv:2308.12950* (2023).

accellera SYSTEMS INITIATIVE

LLM Mutation-Based Testing Framework

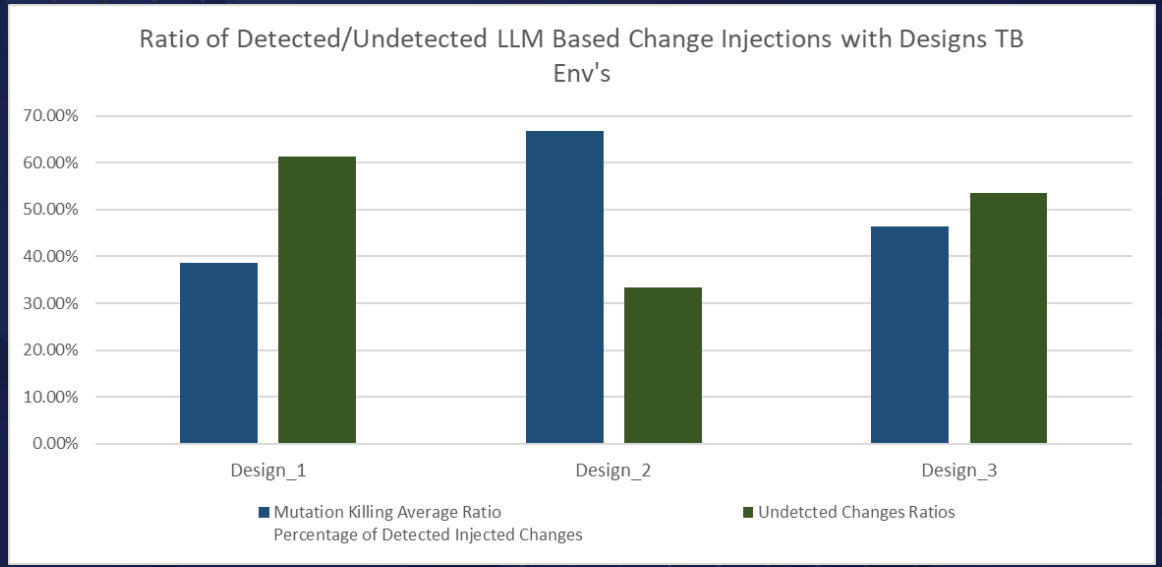
- **Mutation Testing:** identifies weakness/holes that may be unnoticed in Design TB.
- LLM injects “Mutation”: an **artificial modification** in the tested design that changes its behavior.
- Design Test set should be ***mutation-adequate:*** detect as many as induced mutations to prove robustness.
- The proposed LLM-based mutation testing framework starts by extracting **design signals and design scope** as fault injection candidates.
- The LLM is directed by a **set of prompts** to generate a set of faulty versions of the original DUT.
- **A compilation step** is followed to assure that the injected changes are syntactically correct.
- The “**Mutation Killing Ratio**” is used as a criteria to measure how many of the injected faults have been detected by the design test set.

accellera SYSTEMS INITIATIVE

LLM Mutation-Based Testing Framework

Experimental Results

- ✓ Our Experiment is based on 3 In-House Designs.
- ✓ 12 variations have been created for every design.
- ✓ No of Mutations Injected (Design_1 : 63, Design_2 : 69, Design_3: 105)
- ✓ Among all LLM-generated changes, although 75% compile without errors, 25% still requires manual fixes.
- ✓ Design TB's can detect an average of 50.59% of injected changes only.
- ✓ 49.41% were undetected by the current testbench highlights coverage gaps in TB's
- ✓ These results prove the power of LLM in a mutation-injection methodology to automate testbench hole identification.



	Design_1	Design_2	Design_3	Average Summary
No of LLM Generated Design Versions	12	12	12	12
No of LLM RTL Changes Injected	69	63	105	79
Mutation Killing Average Ratio "Percentage of Detected Injected Changes"	38.66%	66.67%	46.43%	50.59%
Undetcted Changes Ratios "Need to Fix TB"	61.34%	33.33%	53.57%	49.41%

accellera SYSTEMS INITIATIVE

Bottom-up Design Creation

- “Chain of thought”, divide into smaller tasks
- Keep enhancing results
- Ask to refactor
- Make incremental changes
- Instruct to follow a coding style

Instructions to LLM

Create a multadd Verilog module that is a two-stage pipeline. It accepts four 8-bit numbers (a, b, c, d) multiplies "a" and "b" and multiplies "c" and "d" in the first cycle. It returns the sum of the two multiplications in one 17-bit on output "prodsum" port on the second clock cycle.
Good start, add a new input signal, pipeline_ready, that indicates when the data is ready on the input ports (a-d)
OK, good. Add another output port that indicates when the summation data on prodsum is ready.
Great, now I need to create a simple ALU that utilizes this multadd_pipeline module you have created.
Add a new operation to this ALU to give the summation of the four input data ports [a-d].
Not there yet, we need to add a new port, operation, to the ALU that identifies when the ALU should do the multadd operation and when it should do the summation operation.
OK, refactor the summation operation in a new Verilog module and use that module in the ALU with the multadd_pipeline.

accellera SYSTEMS INITIATIVE

LLM Generated Code

```
module simple_alu(
    input wire clk,          // Clock input
    input wire reset,        // Reset input
    input wire pipeline_ready,  // Data ready signal for multadd_pipeli
    input wire [1:0] operation,  // Input operation (0 for multadd, 1 f
    input wire [7:0] a,       // Input a (8 bits)
    input wire [7:0] b,       // Input b (8 bits)
    input wire [7:0] c,       // Input c (8 bits)
    input wire [7:0] d,       // Input d (8 bits)
    output wire [16:0] result,  // Output result (17 bits)
    output reg result_ready   // Output result_ready signal
);
    wire [16:0] multadd_result;  // Internal wire for multadd_pipeline
    wire [31:0] summation_result;  // Internal wire for summation_unit

    // Instantiate the multadd_pipeline module
    multadd_pipeline multadd_inst (
        .clk(clk),
        .reset(reset),
        .pipeline_ready(pipeline_ready),
        .a(a),
        .b(b),
        .c(c),
        .d(d),
        .prodsum(multadd_result),
        .result_ready(result_ready)
    );

    // Instantiate the summation_unit module
    summation_unit summation_inst (
```

```
    // Instantiate the summation_unit module
    summation_unit summation_inst (
        .a(a),
        .b(b),
        .c(c),
        .d(d),
        .result(summation_result)
    );

    always @(posedge clk or posedge reset) begin
        if (reset) begin
            result <= 0;
            result_ready <= 0;
        end
        else begin
            // Check operation and select operation accordingly
            case(operation)
                2'b00: begin  // Multadd operation
                    if (pipeline_ready) begin
                        result <= multadd_result;  // Output from multa
                        result_ready <= 1;  // Set result_ready when mu
                    end
                end
                2'b01: begin  // Summation operation
                    result <= summation_result;  // Output from summati
                    result_ready <= 1;  // Set result_ready when summat
                end
                default: begin
                    result_ready <= 0;  // Reset result_ready if invali
                end
            endcase
        end
    end
endmodule
```

```
module multadd_pipeline(
    input wire clk,          // Clock input
    input wire reset,        // Reset input
    input wire pipeline_ready,  // Data ready signal
    input wire [7:0] a,       // Input a (8 bits)
    input wire [7:0] b,       // Input b (8 bits)
    input wire [7:0] c,       // Input c (8 bits)
    input wire [7:0] d,       // Input d (8 bits)
    output reg [16:0] prodsum,  // Output prodsum (17 bits)
    output reg result_ready   // Output result_ready signal
);
    reg [16:0] mult1_result, mult2_result;  // Internal registers to st

    always @(posedge clk or posedge reset) begin
        if (reset) begin
            mult1_result <= 0;
            mult2_result <= 0;
        end
        else begin
            if (pipeline_ready) begin
                mult1_result <= a * b;  // Multiply a and b
                mult2_result <= c * d;  // Multiply c and d
            end
        end
    end

    always @(posedge clk or posedge reset) begin
        if (reset) begin
            prodsum <= 0;
        end
        else begin
            prodsum <= mult1_result + mult2_result;  // Sum of multipli
        end
    end
endmodule
```

Top-down Design Creation

- Use of “Flipped Interaction Pattern”
- Instruct LLM to ask you questions to narrow down requirements
- Refine smaller components by providing more details

LLM Generated Questions

What is the word size of this CPU (e.g., 8-bit, 16-bit, 32-bit)?
What are the main components of the CPU (e.g., ALU, register file, control unit, etc.)?
Are there any specific instructions that this CPU will support?
Do you have any specific clock frequency or timing requirements in mind?
Are there any specific addressing modes that you would like to support in this CPU?

LLM Generated Code

```
module simple_16bit_cpu (
    input clk,
    input rst,
    input [15:0] instruction,
    input [15:0] data_input,
    output reg [15:0] data_output
);

    // Internal signals
    reg [15:0] pc;
    reg [15:0] ir;
    reg [15:0] reg_file_read_data_a, reg_file_read_data_b;
    reg [15:0] alu_result;
    reg [15:0] data_mem_read_data, data_mem_write_data;
    reg [15:0] next_pc;

    // Instantiate components
    alu_16bit alu_inst (
        .operand_a(reg_file_read_data_a),
        .operand_b(reg_file_read_data_b),
        .alu_op(ir[3:0]),
        .result(alu_result)
    );

    register_file_16bit reg_file_inst (
        .read_addr_a(ir[7:4]),
        .read_addr_b(ir[11:8]),
        .write_addr(ir[11:8]),
        .write_data(data_mem_read_data),
        .write_en(ir[15]),
        .read_data_a(reg_file_read_data_a),
        .read_data_b(reg_file_read_data_b)
    );

    control_unit control_unit_inst (
        .instruction(ir),
        .alu_op(alu_inst.alu_op),
        .reg_write_addr(reg_file_inst.write_addr),
        .reg_write_en(reg_file_inst.write_en),
        .next_pc(next_pc)
    );

    program_counter pc_inst (
        .clk(clk),
        .rst(rst),
        .next_pc(next_pc),
        .pc_out(pc)
    );

    instruction_memory instruction_mem_inst (
        .address(pc),
        .read_data(ir)
    );

    data_memory data_mem_inst (
        .address(alu_result),
        .read_data(data_mem_read_data),
        .write_data(data_mem_write_data),
        .write_en(control_unit_inst.mem_write_en)
    );

    // Data path connections
    always @* begin
        data_mem_write_data = data_input;
        reg_file_read_data_a = reg_file_inst.read_data_a;
        reg_file_read_data_b = reg_file_inst.read_data_b;
    end

    always @* begin
        data_output = data_mem_read_data;
    end

endmodule
```

accellera
SYSTEMS INITIATIVE

Design Understanding

The `control_operation` module appears to be a finite state machine (FSM) designed to control the operation of a system based on certain input conditions. Below is a description of the key elements and functionality of the `control_operation` module based on the provided Verilog code:

Inputs:

- **clk:** Clock signal to synchronize the operations within the module.
- **clr_int_en:** Input signal used for clearing interrupt enables.
- **cs:** Input signal for chip select.
- **nrw:** Input signal indicating whether the operation is a write operation (1) or a read operation (0).
- **xmitdt_en:** Input signal indicating whether data transmission is enabled.

Outputs:

- **clear_flags:** Output signal indicating whether flags should be cleared.
- **enable_write:** Output signal indicating whether a write operation should be enabled.
- **start_xmit:** Output signal indicating the start of data transmission.

Internal State Encoding:

- **idle (3'd0):** The initial state where the module is waiting for an operation to be triggered.
- **reading_from_reg (3'd1):** State where the module is reading data from a register.

Describe control_operation Module

Current State	Inputs	Next State
idle	nrw = 1, ~cs	writing_to_reg
idle	~nrw, ~cs	reading_from_reg
idle	otherwise	idle
reading_from_reg	cs = 1	idle
reading_from_reg	~nrw, clr_int_en = 1	clearing_flags
reading_from_reg	otherwise	reading_from_reg
clearing_flags	cs = 1	idle
clearing_flags	otherwise	clearing_flags
writing_to_reg	cs = 1	idle
writing_to_reg	nrw = 1, xmitdt_en = 1	xmitting
writing_to_reg	otherwise	writing_to_reg
xmitting	cs = 1	idle
xmitting	otherwise	xmitting
default	anything	idle

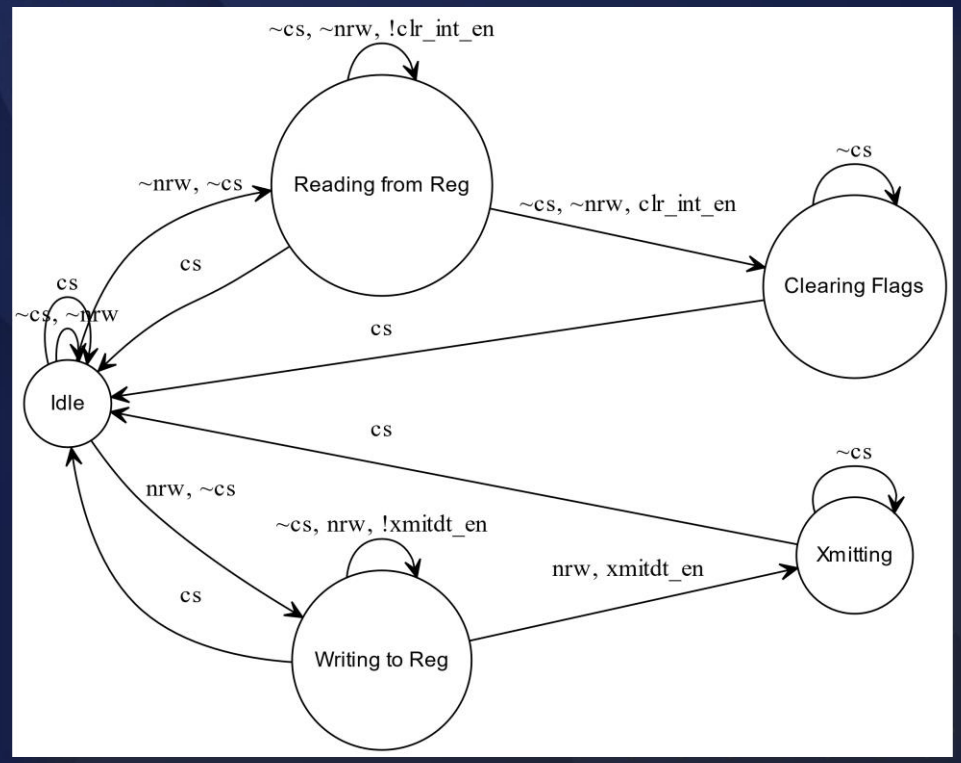
Write state-transition Table of control_operation

accellera SYSTEMS INITIATIVE

Design Understanding

```
digraph control_operation_fsm {
    rankdir=LR;
    node [shape=circle, fontsize=12, fontname="Helvetica", width=0.7];
    edge [arrowhead=vee];

    idle [label="Idle"];
    reading_from_reg [label="Reading from Reg"];
    clearing_flags [label="Clearing Flags"];
    writing_to_reg [label="Writing to Reg"];
    xmitting [label="Xmitting"];

    idle -> idle [label="~cs, ~nrw"];
    idle -> writing_to_reg [label="nrw, ~cs"];
    idle -> reading_from_reg [label="~nrw, ~cs"];
    idle -> idle [label="cs"];

    reading_from_reg -> reading_from_reg [label="~cs, ~nrw, !clr_int_en"];
    reading_from_reg -> idle [label="cs"];
    reading_from_reg -> clearing_flags [label="~cs, ~nrw, clr_int_en"];

    clearing_flags -> clearing_flags [label="~cs"];
    clearing_flags -> idle [label="cs"];

    writing_to_reg -> writing_to_reg [label="~cs, nrw, !xmitdt_en"];
    writing_to_reg -> idle [label="cs"];
    writing_to_reg -> xmitting [label="nrw, xmitdt_en"];

    xmitting -> xmitting [label="~cs"];
    xmitting -> idle [label="cs"];
}
```

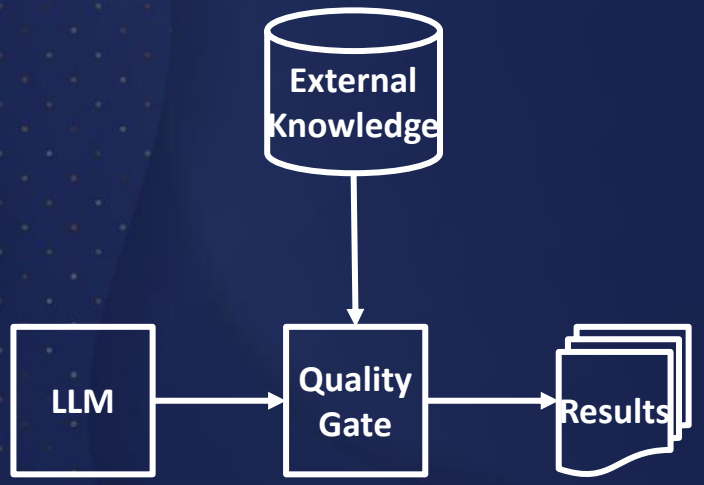
LLM-generated GraphViz DOT Notation of a Given Verilog Module



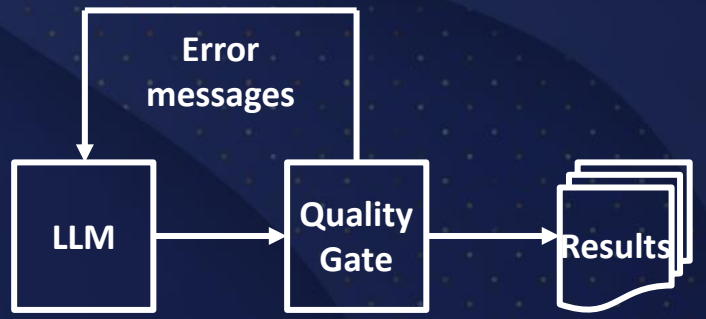
GraphViz Visualization of Recognized FSM

accellera SYSTEMS INITIATIVE

LLM Application Paradigms



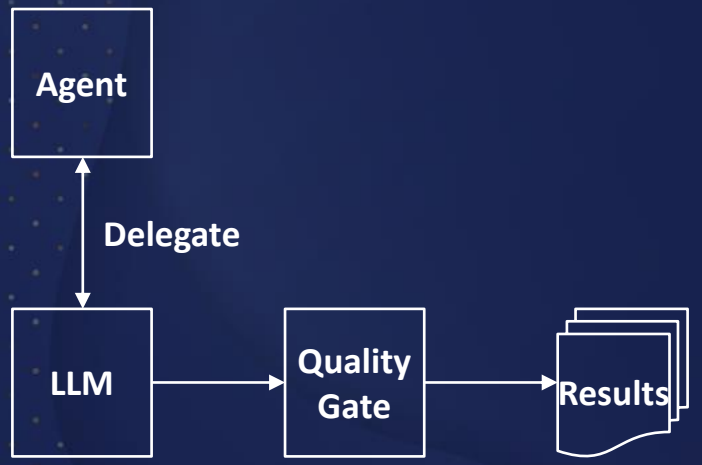
Quality Gate/Guardrail



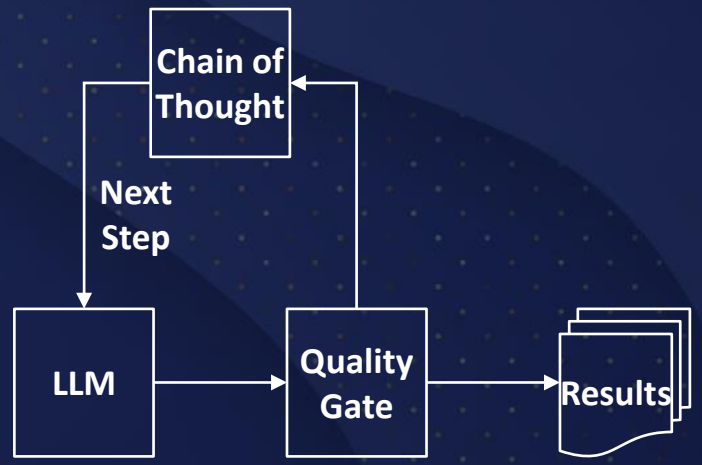
Self-check Feedback Loop

accellera SYSTEMS INITIATIVE

LLM Application Paradigms



External Agent



Chain-of-Thought

accellera SYSTEMS INITIATIVE

Thanks

Q&A

accellera
SYSTEMS INITIATIVE